Online Shared Memory Dependence Reduction via Bisectional Coordination

Yanyan Jiang*, Chang Xu*, Du Li†, Xiaoxing Ma*, Jian Lu*
*State Key Lab for Novel Software Technology, Nanjing University, China
Department of Computer Science and Technology, Nanjing University, China
†School of Computer Science, Carnegie Mellon University, United States
jiangyy@outlook.com, changxu@nju.edu.cn, duli@cs.cmu.edu, {xxm,lj}@nju.edu.cn

ABSTRACT

Order of shared memory accesses, known as the shared memory dependence, is the cornerstone of dynamic analyses of concurrent programs. In this paper, we study the problem of reducing shared memory dependences. We present the first online software-only algorithm to reduce shared memory dependences without vector clock maintenance, opening a new direction to a broad range of applications (e.g., deterministic replay and data race detection). Our algorithm exploits a simple yet effective observation, that adaptive variable grouping can recognize and match spatial locality in shared memory accesses, to reduce shared memory dependences. We designed and implemented the bisectional coordination protocol, which dynamically maintains a partition of the program's address space without its prior knowledge, such that shared variables in each partitioned interval have consistent thread and spatial locality properties. Evaluation on a set of real-world programs showed that by paying a 0–54.7% (median 21%) slowdown, bisectional coordination reduced 0.95–97% (median 55%) and 16–99.99% (median 99%) shared memory dependences compared with RWTrace and LEAP, respectively.

CCS Concepts

•**Software and its engineering → Concurrent programming languages; Runtime environments;**

Keywords

Concurrency, shared memory dependence reduction, dynamic analysis

1. INTRODUCTION

Shared memory concurrent systems are notoriously difficult to program, to test, and to debug due to the non-deterministic nature of thread scheduling and shared memory accesses. A fundamental approach to understanding the behavior of such systems is to keep track of a partial order between conflicting shared memory accesses, i.e., *shared memory dependences*, which facilitates not only faithful reproduction of a past program execution [9, 17, 18, 35] but also predictive analyses for disclosing software defects (e.g., data races [23], atomicity violations [22], and more general defect patterns [10]).

Shared memory dependences are the basis of further analyses. Online algorithms [4, 8, 28] check a dependence immediately after it is captured, while post-mortem algorithms [9, 10, 14, 22] parse the dependence log after the program termination. Efficiency of these algorithms is tightly coupled with the amount of shared memory dependences. Therefore, shared memory dependence reduction algorithms are proposed, either by offline parsing [13] or online analysis with customized hardware [31]. However, it is still an open problem to efficiently perform such reduction *online* by *software-only* instrumentation, as transitivity maintenance by clock incurs large overhead [8]. We discuss the background on shared memory dependence and its reduction in Section 2.

In this paper, we present the first online software-only shared memory dependence reduction algorithm, based on the interplay of both *thread* and *spatial locality* of shared memory accesses. Thread locality indicates that consecutive accesses to a shared variable tend to happen in the same thread. Spatial locality indicates that a successive range of shared memory locations tends to be exclusively accessed by a single thread in a time period. When they are combined together, we observed that the entire address space can be partitioned into a relatively small amount of segments. Each segment can be virtually treated as a single unit in the inter-thread communication. Such segment grouping can achieve the effect of existing transitive reduction algorithms [21, 31]. We discuss locality in Section 2.3 and expand our discussion of how locality facilitates shared memory dependence reduction in Section 3 with an empirical study.

Our major technical contribution is extending existing shared memory dependence tracing algorithms to *dynamically* and *adaptively* maintaining an interval partition of the address space in which each interval simultaneously exhibits thread and spatial locality. Our algorithm starts with the initial partition that contains a single group that covers all memory locations. As shared memory dependences are obtained during the program's execution, a large interval that violates the locality assumption (detected by "false dependences") is split into smaller ones. Such splits always happen at the midpoint of an interval so we name this algorithm "*bisectional coordination*". We discuss the technical details of the bisectional coordination protocol in Section 4.

Bisectional coordination maintains a program's thread and spatial locality view at runtime for shared memory dependence reduction. This property facilitates more efficient dynamic analyses of

[1] Chang Xu and Xiaoxing Ma are the corresponding authors.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950326

concurrent programs. Particularly, we study how to adapt three typical applications to utilize the benefits of bisectional coordination: deterministic replay, data race detection, and false sharing detection in Section 5.

We implemented the bisectional coordination protocol as an extension to our optimistic shared memory dependence tracing tool RWTrace [15]. Particularly, we use a globally shared cache to efficiently determine each variable's corresponding interval and use bloom filters to keep approximate shared memory access log for false dependence detection. These techniques are crucial to facilitating an efficient implementation and are discussed in Section 6.

Evaluation results in Section 7 showed that for 12 evaluated real-world benchmark programs under practical workloads, we paid 0–54.7% (median 21%) overhead upon RWTrace to achieve a significant shared memory dependence reduction: 0.95–97% (median 55%) dependences are reduced compared with RWTrace and 16–99.99% (median 99%) compared with LEAP. This result also outperformed the data reported in CARE [14] (11–99.6%, median 86%) and CLAP [12] (72–97.7%, median 91%) that used LEAP as a baseline for comparison.

Finally, we discuss related work and conclude the paper in Section 8 and Section 9, respectively.

2. BACKGROUND

2.1 Shared Memory Dependence

Concurrent programming becomes increasingly popular since the adoption of multicore computers in which inter-thread communication is mainly realized by shared memory for best efficiency. To analyze such a system's behavior, it is crucial to keep track of the order of shared memory accesses (i.e., *shared memory dependences*) because different orders yield diverge execution paths.

A shared memory dependence is the chronological order between two shared memory access events in different threads performed on the same variable and at least one of them is a write. Once such dependences are available, a past execution can be deterministically replayed. Based on the logged execution trace, further active testing or analyses [23] can be conducted.

Tracing shared memory dependences is challenging because such dependences are massive in quantity and tracing them requires program modifications that inevitably slow down the program execution. A straightforward approach, LEAP [9], is to serialize all accesses performed on the same shared variable using a lightweight lock. Observing that most shared memory accesses are thread-local, optimistic shared memory dependence tracing algorithms [6, 14, 35] are proposed to reduce runtime cost and log size. More recent work [18] takes a different approach of merely logging write-after-read dependences and the other dependences are synthesized by an offline constraint solving.

2.2 Transitive Reduction

Regarding the efficiency of shared memory dependence tracing, runtime overhead is not the only concern. There can be a huge amount of shared memory dependences to be logged, incurring slowdown of the runtime system as well as the follow-up analyses. Fortunately, not all dependences are necessary and simplification algorithms are carried out to compress the dependence log. Simplification can either be performed on the fly along with the program execution [21, 31] or be conducted offline as a combinatorial optimization problem for even more compact logs [13]. In this paper, we focus on the online algorithms because online analyses (e.g., data race detection) check a dependence immediately after its occurrence, which can only benefit from online shared memory dependence reduction.

A straightforward approach to shared memory dependence reduction is *transitive reduction* (TR) [21]. For events a, b, and c, if both $a \to b$, $b \to c$, and $a \to c$ hold, $a \to c$ is redundant according to the transitivity law. Every partial order set has a unique minimum transitively reduced representation [21] and online transitive reduction can be achieved by maintaining vector clocks of events, implemented either at software- [21] or hardware-level [30].

However, even transitively minimum logs can be further reduced. The seminal work *regulated transitive reduction* (RTR) [31] extends TR by an ingenious observation that parallel dependences can be replaced by a stricter one. Particularly, when a shared memory dependence $a \to b$ is captured, it tries to find an earlier event $b' \to b$ such that (1) b and b' are in the same thread and (2) dependence $a \to b'$ does not introduce cycles in the dependence log. Because $a \to b'$ implies $a \to b$ by transitivity, RTR logs $a \to b'$ rather than $a \to b$ and removes any dependence that is transitively reducible from $a \to b'$.

2.3 Locality of References

Locality of references (*locality* for short) is a phenomenon frequently being exploited in the design and implementation of computer systems. The classical definition of locality is that for relatively extended periods of time, a (single-threaded) program references only some relatively small subsets of its owned resources [20]. Resources can be memory locations, disk blocks, network hosts, to name but a few.

In a multiprocessor concurrent system, shared memory accesses also exhibit strong *thread locality*: consecutive accesses of a shared memory location tend to be performed by the same thread. In shared memory dependence tracing, thread local shared memory accesses can easily be deduced by optimistic algorithms [6, 14, 35].

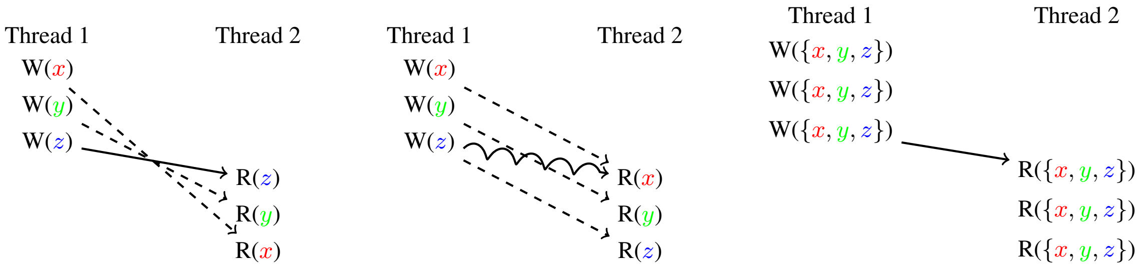
The design of multiprocessor architecture, especially the cache coherence protocol [25], also incorporates the *spatial locality* of shared memory accesses: nearby memory addresses are more likely to be referenced in the same time period. The interplay of thread and spatial locality facilitates the design of multiprocessor cache coherence protocols that a processor exclusively owns the entire cache line if any variable in it has been written. Such interplay is also the key insight of our bisectional coordination algorithm.

3. MOTIVATION

This section introduces the key factors that motivated our work. We first discuss the limitation of transitive reduction algorithms in Section 3.1. Then, in Section 3.2, we present our key observation that locality of references facilitates shared memory dependence reduction with an empirical study. Finally, the challenge of detecting spatial locality is discussed in Section 3.3.

3.1 Limitation of TR and RTR

Both TR and RTR are effective in shared memory dependence reduction. However, it is challenging to determine whether a dependence is transitively reducible. The best known approach is Lamport's vector clock [16] designed to capture causal relationship in asynchronous systems. Vector clock operations, however, are $O(T)$ in time and space complexity where T is the number of threads. The original RTR is implemented by cache coherence hardware customization [31] to avoid performance problems. Even the best known software-only implementation, FastTrack [8], has much higher overhead than an optimistic shared memory dependence tracing algorithm [6, 15].



(a) Transitive reduction (b) Regulated transitive reduction (c) Grouping $\{x, y, z\}$

Figure 1: Illustrations of the shared memory dependence reduction effects of TR, RTR, and variable grouping.

Therefore, a natural question is how to reduce shared memory dependences without maintaining vector clock. Our work is motivated by the observation that the *interplay of thread and spatial locality can achieve the similar effect of both TR and RTR* and this insight is described as follows.

3.2 Locality and Transitive Reduction

A shared memory dependence orders two shared memory accesses performed on the same variable. Shared memory dependence tracing algorithms therefore only capture dependences between *consecutive* inter-thread accesses separately for each shared variable [6, 9, 14, 17, 18, 32, 35]. If we consider the dependence log for just one variable, it is already transitively irreducible. Both TR and RTR reduce dependences by analyzing dependences among multiple variables.

Following such intuition, we found that TR and RTR actually identified two orthogonal shared memory dependence reduction patterns across different shared variables: TR removes *transitively redundant dependences* (Figure 1a)[1] while RTR replaces *parallel dependences* by a stricter one (Figure 1b).

To reduce dependences without order maintenance, we found that if variables are properly grouped to be a single entity, the effect of both TR and RTR can be achieved, as illustrated in Figure 1c. In this example, we treat variables x, y, z as a single bundle $\{x, y, z\}$ and the bundle's successive accesses in the same thread become thread-local. Both cases (Figures 1a and 1b) can be reduced by grouping $\{x, y, z\}$. There might be shared memory accesses performed on other variables in between, however, once we group $\{x, y, z\}$, the reduction effect can be achieved. Dependences reduced by variable grouping is *not* strictly equivalent to TR/RTR. We further discuss this issue in Section 4.4.

Thread and spatial locality provides hints to what variables should be grouped together for transitive reduction. Spatial locality implies that x, y, and z in Figure 1c are usually adjacent in memory address and thread locality suggests that the bundle $\{x, y, z\}$ should be exclusively accessed by a thread for an extended time period. Therefore, we should group variables that have both thread and spatial locality and use existing algorithms [6, 9, 15] to reduce thread local dependences.

To validate the existence of thread and spatial locality as well as find guidance for variable grouping, we conducted an empirical study using the similar methodology in [26]. For different values of k, we partition the address space into interval groups of k memory words (similar to adjusting the cache line size in [26] but we studied much larger values of k that are infeasible in a hardware setting) and trace shared memory dependences between groups (i.e., memory words in a group are treated as a single variable).

The amount of dependences correlates to k, the degree of spatial locality. A too small k underestimates the spatial locality and there would be transitively reducible dependences (dashed dependences in Figures 1a and 1b). A too large k groups independent variables together, yielding unnecessary false dependences (Figure 4, where a boxed variable is the one actually accessed). The amount of dependences also correlates to the existence of thread locality because the shared memory dependence reduction effect in Figure 1c could not exist without thread locality.

We present the study results in Figure 2. For each subject program, we normalize the number of dependences by the minimum number among all settings of k. Aligned with the conclusion of [26], increasing k (the group size) benefits from spatial locality for small values of k. When k exceeds a subject-dependent value, the amount of dependences starts to increase.

There is no single optimal k for all programs. For example, qsort achieved the minimum dependence log at $k = 2^6$, x264 at $k = 2^{12}$, radix at $k = 2^{14}$, and water at $k = 2^{20}$. Furthermore, an improperly chosen k may lead to logging tens to hundreds times of more dependences. The best k is dependent on both the program and its inputs. Therefore, even if we have confirmed the existence of both thread and spatial locality, it is still challenging to find a universal variable grouping algorithm.

3.3 Challenge: Detecting Spatial Locality

The previous study of variable grouping, though demonstrated the potential of reducing shared memory dependences via locality of references, is difficult to realize in practice because:

1. The fixed-size grouping assumes that all groups have similar shared memory access pattern, which may not be the case in practice, yielding redundant dependences created by improper groups.
2. Even if such grouping is feasible, it is difficult to determine the optimal group size k that can best reduce dependences. Furthermore, the optimal k may be different for each memory region and time period.

Therefore, to leverage thread and spatial locality for shared memory dependence reduction, the key challenge is how to keep a dynamic and flexible view of the shared memory that reflects such locality properties. The main technical contribution of this paper, the bisectional coordination protocol, exactly addresses this challenge. Bisectional coordination maintains a dynamic partition of

[1] For visual convention, we use R and W to denote a shared memory read and write, respectively. An arrow denotes a shared memory dependence. Solid arrows are captured and actually logged dependences, dashed ones are reduced dependences, and coiled ones are synthesized dependences by RTR.

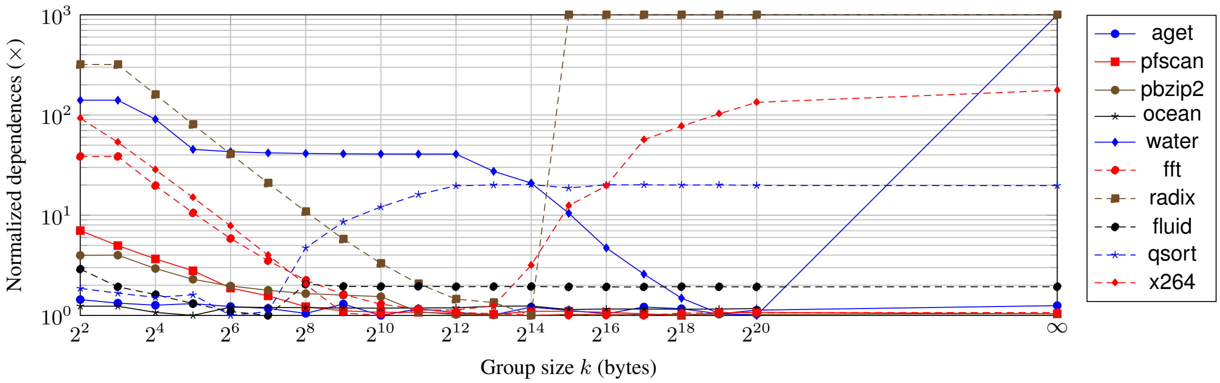


Figure 2: Impact of the variable group size k: an empirical study.

the address space such that variables in the same partition exhibit both thread and spatial locality and have similar access pattern.

4. BISECTIONAL COORDINATION

This section presents the bisectional coordination protocol. After introducing the basic notations and definitions in Section 4.1, we discuss the limitation of static variable grouping and the basic idea of dynamic interval partition in Section 4.2. In Section 4.3, we describe the bisectional coordination protocol followed by a qualitative analysis of effectiveness in Section 4.4.

4.1 Notations and Definitions

For simplicity, we assume that a program consists of statically allocated threads and each thread executes a stream of shared memory access events. The shared memory consists of variables that each has a unique integral address $x \in \mathcal{M}$ where $\mathcal{M} = [0, M)$. We use the notation $\langle t{:}e_i \rangle$ to denote the i-th event performed by thread t, which is either a read or write event defined as follows:

1. $\mathsf{R}_t[x] = v$, for reading value v from variable x;
2. $\mathsf{W}_t[x] = v$, for writing value v to variable x.

Order of events can be obtained either by system modification [6, 7], program instrumentation [9, 15], hardware customization [30], or constraint solving [12]. The following dependences are of particular interest:

1. Program order $\xrightarrow{po}$ that orders the events in the same thread (for each $\langle t{:}e_i \rangle$ and $\langle t{:}e_j \rangle$ where $i < j$, $\langle t{:}e_i \rangle \xrightarrow{po} \langle t{:}e_j \rangle$ holds). Program order, though essential in understanding the behavior of concurrent systems, does not have to be logged because it can be trivially obtained by executing the program.
2. Shared memory dependence $\langle t_1{:}e_i \rangle \xrightarrow{smd} \langle t_2{:}e_j \rangle$ where $t_1 \neq t_2$, both events are performed on the same variable, and at least one is a write.

Typically, only dependences of consecutive conflicting shared memory accesses are logged. According to the types of e_i and e_j, a shared memory dependence can be read-after-write (RAW), write-after-read (WAR) or write-after-write (WAW). WAR dependences can be derived by RAW and WAW dependences [14, 35] and the transitive closure of program order and shared memory dependences, $\xrightarrow{\star} = tr(\xrightarrow{po} \cup \xrightarrow{smd})$, is sufficient for eliminating nondeterminism in a concurrent program's execution.

4.2 Adaptive Variable Grouping

Threads access the shared memory by reading or writing shared variables. However, as discussed in Section 3.2, we can treat a set of variables to be a group and shared memory dependences are traced in terms of groups, as shown in Figure 1c. Formally, we can partition the address space $\mathcal{M}$ into k disjoint groups $\mathcal{P} = \{G_1, G_2, \dots G_k\}$ and use the lookup function $f : \mathcal{M} \to [k]$ to map an address x to its corresponding group $G_{f(x)}$. When a thread accesses x, we account it for accessing $G_{f(x)}$. In other words, if we treat $G_1, G_2, \dots, G_k$ as shared variables, $\mathsf{R}_t[x] = v$ is considered as $\mathsf{R}_t[G_{f(x)}] = v$ and $\mathsf{W}_t[x] = v$ is considered as $\mathsf{W}_t[G_{f(x)}] = v$ in shared memory dependence tracing. Static variable grouping, i.e., using fixed $\mathcal{P}$ and f along the program execution, is already adopted in existing shared memory dependence tracing algorithms [6, 7, 9, 15, 32, 35].

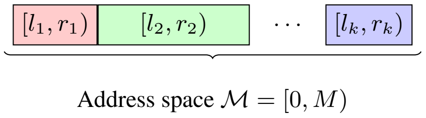


Figure 3: Illustration of the dynamic interval grouping. The address space is partitioned into consecutive segments that subject to change over time.

A pre-designed static partition cannot exploit the full strength of spatial locality. A typical example is the treatment of arrays. Existing algorithms either treat the entire array as a single entity or separately trace dependences for each individual array cell. However, fine-grained sharing of arrays is widely adopted in concurrent data structures and scientific programs: static variable grouping clearly falls short in such cases.

Realizing the limitation of static variable grouping, the major innovation of our algorithm is to allow the partition $\mathcal{P}$ to be dynamic and adaptive at runtime. To leverage spatial locality in the variable grouping, we maintain a dynamic *interval* partition of $\mathcal{M}$: $\mathcal{P} = \{G_1, G_2, \dots G_k\}$ (Figure 3) such that $\cup_i G_i = \mathcal{M}$ and $\mathcal{P}$ can be changed at runtime, plus the following interval constraints:

1. Each group contains variables of consecutive addresses, i.e., $G_i = [l_i, r_i)$;
2. Each interval's end point meets the next one's starting point, i.e., $r_i = l_{i+1}$.

We chose the interval partition because it is natural and simple to compute. It always groups consecutive addresses and therefore reflects spatial locality and $f(x)$ can be efficiently computed.

We expect $\mathcal{P}$ to meet the requirement that variables in the same interval have both thread and spatial locality at runtime. Variables in an interval should either be exclusive accessed by a specific

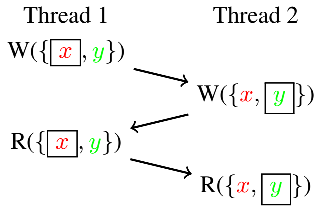


Figure 4: Improper grouping and false dependences. The boxed variable indicates the one actually being accessed. All shared memory accesses are thread-local, however, improper grouping yields three unnecessary dependences.

thread or be read-shared among threads in an extended time period. We use the following bisectional coordination protocol to maintain this dynamic property at runtime.

4.3 The Bisectional Coordination Protocol

4.3.1 The Initial Partition

Initially, we assume that the program is single-threaded, i.e., only one thread exclusively owns all shared variables. Therefore, we start with the trivial partition $\mathcal{P} = \{[0, M)\}$ (in practice, the initial partition consists of 2^{20}-byte intervals to avoid unnecessary bisections at program start).

Along with the program execution, such an assumption would break because threads can simultaneously access the shared memory. The single-threaded assumption thus leads to unnecessary false dependences, suggesting that a large group is improper and should be corrected.

4.3.2 Detecting Improper Groupings

Improper grouping, as its name suggests, denotes a group of shared variables that do not have thread and spatial locality. Figure 4 displays an example of the improper group of $\{x, y\}$. If x and y are not in the same group, there would be no dependence at all. We call a shared memory dependence caused by improper grouping a *false dependence*. Particularly, $\langle t_1{:}e_p\rangle \overset{smd}{\to} \langle t_2{:}e_q\rangle$ created by group G_i is a false dependence if: (1) $\langle t_2{:}e_q\rangle$ is $\mathsf{R}_{t_2}[x] = v$ or $\mathsf{W}_{t_2}[x] = v$ and (2) there is no write event to the same variable (i.e., $\mathsf{W}_{t_1}[x] = v$) since the last dependence of G_i is logged in thread t_1. In other words, if a dependence of G_i does not account for any read-after-write or write-after-write dependence of variable x, we consider it as a false dependence.

False dependences are an indicator of the need of correcting an improper variable group. If accumulated false dependences of a group exceed a threshold, we carry out the bisectional coordination protocol to fix the issue, which is described in the following. The implementation of false dependence detection is further discussed in Section 6.

4.3.3 Resolving Improper Groupings

Suppose that we have confirmed that $G_i = [l_i, r_i)$ is improperly grouped due to false dependences: variables in G_i do not consistently have thread and spatial locality. Therefore, we should split G_i into smaller sub-intervals such that variables causing false dependences are separated in different sub-intervals.

To best reduce runtime overhead, we take a simple approach of bisecting $G_i = [l_i, r_i)$ into two smaller subgroups $[l_i, m)$ and $[m, r_i)$ where $m = (l_i + r_i)/2$ is the midpoint of G_i and this is exactly why our algorithm is named *bisectional coordination*. We do not split the interval to separate the variables in a false dependence (e.g., x and y in Figure 4) because improper grouping is triggered by a series of false dependences and it is difficult to decide the precise split point.

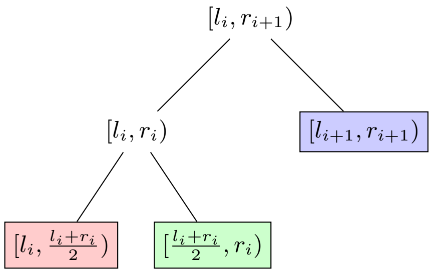


Figure 5: Illustration of bisectional coordination. The interval $[l_i, r_i)$ is bisected at the midpoint.

After bisecting $[l_i, r_i)$, the partition $\mathcal{P}$ is updated to be $\mathcal{P}' = \{G'_1 = G_1, \dots, G'_i = [l_i, m), G'_{i+1} = [m, r_i), \dots, G'_{k+1} = G_k\}$. The bisection procedure is illustrated in Figure 5: the red and green intervals denote the result of bisecting G_i. After a series of bisections, the bisection history forms a binary tree in which leaf nodes are the dynamic partition.

This treatment of improper grouping is simple and can be efficiently implemented. However, in the worst case that two variables causing false dependences are close in memory address, this algorithm may take $\Theta(\log M)$ bisections to separate them. Fortunately, as demonstrated in our evaluation, such case is rare and the bisectional coordination protocol performed surprisingly well for real-world benchmarks.

4.3.4 Dealing with Fragmented Groups

There remains a final problem: our algorithm only breaks an interval into smaller ones. For long-running programs, the partition may be fragmented and cannot reflect the spatial locality of shared memory accesses. We propose a simple solution to this issue: if the production rate of shared memory dependence has increased for a time period, we merge all intervals together, backtracking to the initial setting of $\mathcal{P} = [0, M)$. According to the evaluation results in Section 7, the number of intervals (i.e., $|\mathcal{P}|$) is always small (0–954 with median 31, indicating that programs often have steady spatial locality) and such restart strategy will have negligible performance impact in practice.

4.4 Bisectional Coordination for Shared Memory Dependence Reduction

The bisectional coordination protocol is a cautious optimistic algorithm to leverage locality of references. It optimistically assume that variables in a large interval have both thread and spatial locality and such assumption is repeatedly validated through the program execution. Unless we find evidence (i.e., false dependences) to contradict this assumption, we treat variables in the interval as a single entity, achieving transitive reduction effects in Figures 1a and 1b.

Bisectional coordination only reduces a subset of transitively reducible dependences: TR [21] and RTR [31] can eliminate a transitively reducible dependence by cascading dependences. However, such reduction incurs the heavy cost of maintaining vector clock at runtime. In RTR, most shared memory dependences are reduced by "vectorization": using one stricter dependence to represent many dependences between two threads. Bisectional coordination performs a similar vectorization using locality sensitive variable grouping instead of vector clocks, striking a balance between the runtime cost and the reduction effectiveness.

To qualitatively illustrate the effectiveness of bisectional coordination, we apply it to several typical array use cases:

1. *Read-shared* for the entire array written once by a master thread and all subsequent accesses are shared read. Static lookup tables belong to this category.
2. *Partitioned* for the array being partitioned into consecutive sub-arrays that are exclusively accessed by the threads and the results are finally aggregated by a master thread. This pattern is frequently exhibited in scientific and data analytic programs.
3. *Random-access* for all cells being randomly accessed by all threads. Concurrent data structures (e.g., hash tables or trees) usually belong to this category.

For static variable grouping [6, 7, 9, 15, 32, 35], if we group the entire array [6], it would create a large amount of false dependences in Cases 2 and 3 because all writes to the array are serialized. Alternatively, if we trace shared memory dependence for each individual array cell [24], unnecessary parallel dependences are introduced for Cases 1 and 2.

Bisectional coordination does not suffer from such issues because variable grouping is adaptive and dynamic. For Case 1, there is no false dependence and there would be no bisection at all. In this case, bisectional coordination acts as if it groups all cells of the array and only one dependence per thread is logged. For Case 2, the bisection procedure will proceed until no interval contains cells from distinct sub-arrays and consecutive cells in the same sub-array are still grouped together. For Case 3, false dependences would guide the partition to finally separate each individual array cell, falling back to the fine-grained variable grouping.

These justifications for arrays also apply for any consecutive memory layouts: fields in an object, static variables, etc. Bisectional coordination protocol can detect the shared memory access pattern and adapt to a better partition that reflects locality. Quantitative study of effectiveness is presented in Section 7.

5. APPLICATIONS

5.1 Deterministic Replay

A past concurrent program execution can be deterministically reproduced by re-executing the program with the shared memory dependence order [7, 9, 15, 32, 35].

To deterministically replay a past program execution, the shared memory dependence log must determine the order of all conflicting shared memory access pairs. Such soundness is obvious for existing work that uses static variable grouping (using a pre-defined fixed partition $\mathcal{P} = \{G_1, G_2, \dots G_k\}$ and lookup function f). The soundness of bisectional coordination is proved in the following:

THEOREM 1. *For any pair of conflicting shared memory accesses $\langle t_1{:}e_i \rangle$ and $\langle t_2{:}e_j \rangle$, they are ordered in $\overset{\star}{\rightarrow}$.*

PROOF. (sketch) $\langle t_1{:}e_i \rangle$ and $\langle t_2{:}e_j \rangle$ are conflicting if $t_1 \neq t_2$, at least one of them is a write, and both are performed on the same variable x. Suppose that $\langle t_1{:}e_i \rangle$ happened first (the other case is symmetric) and its corresponding variable group is $G = [l, r)$. If G is not bisected, shared memory dependences are traced like static grouping and all conflicting shared memory accesses of x are ordered in $\overset{\star}{\rightarrow}$.

Later, a false dependence is detected at event $\langle t'{:}e_p \rangle$ and G is bisected, yielding subgroups $G_l = [l, m)$ and $G_r = [m, r)$. The false dependence first ensures $\langle t_1{:}e_i \rangle \overset{\star}{\rightarrow} \langle t'{:}e_p \rangle$. Furthermore, either $x \in G_l$ or $x \in G_r$ holds because $x \in G = G_l \cup G_r$. If any other thread $t'' \neq t'$ accessed group G_l or G_r and yield an event $\langle t''{:}e_q \rangle$, we have $\langle t'{:}e_p \rangle \overset{\star}{\rightarrow} \langle t''{:}e_q \rangle$. The transitivity rule implies $\langle t_1{:}e_i \rangle \overset{\star}{\rightarrow} \langle t''{:}e_q \rangle$. An induction on the bisection count shows that this property holds after an arbitrary amount of bisections, i.e., $\langle t_1{:}e_i \rangle \overset{\star}{\rightarrow} \langle t_2{:}e_j \rangle$. □

Though the result seems to be obvious, the proof reveals that a pair of conflicting shared memory accesses can be connected by a *chain* of dependences[2], which does not affect the soundness of deterministic replay but complicates data-race detection using bisectional coordination in Section 5.2.

The shared memory dependence reduction effect of bisectional coordination facilitates more efficient deterministic replay: less dependences to store and less meta-data to maintain at runtime. Less dependences to analyze also facilitates more efficient replay, especially for the search-based algorithms. Seminal research work Light [18] synthesizes shared memory dependences only using read-after-write dependences by constraint solving that is believed to be exponential-time in the worst case. Bisectional coordination greatly simplifies the constraints to be solved.

5.2 Data Race Detection

Data race, caused by two threads simultaneously accessing a same variable (at least one is a write) without synchronization, is a major cause of concurrency bugs. Data race can be detected by dynamic happens-before race detectors [4, 8, 28] that keep track of the partial order $\overset{syn}{\rightarrow}$, the causality of synchronization operations. Any pair of conflicting shared memory accesses not ordered by $\overset{syn}{\rightarrow}$ is a data race.

Such race detectors pay large runtime cost in meta-data bookkeeping (e.g., updating epoch or vector clock) for each individual shared variable at every access, causing tens of times of slowdown. Furthermore, data race detection is not compatible with variable grouping because a false dependence (Figure 4) not ordered by $\overset{syn}{\rightarrow}$ does not imply a data race.

Fortunately, the opposite of this argument is true: if all shared memory dependences (with variable grouping) are ordered in $\overset{syn}{\rightarrow}$, there is no data race in the execution (Theorem 1). We therefore propose an adaptive tracking algorithm for efficient race detection[3]. We first use bisectional coordination to trace shared memory dependences between variable groups and variables in the same group share one copy of meta-data. We name this strategy "coarse-grained checking". As long as group dependences are ordered in $\overset{syn}{\rightarrow}$, the program execution is free of data race.

If a dependence $\langle t_1{:}e_i \rangle \overset{smd}{\rightarrow} \langle t_2{:}e_j \rangle$ (on variable group G_i) is unordered in $\overset{syn}{\rightarrow}$, we fall back to the fine-grained checking [8] that separately maintains each variable's metadata in G_i, to ensure correctness of race detection. Later, at another dependence created by G_i, if all previous accesses of G_i were happened before the current event in $\overset{syn}{\rightarrow}$, we switch back to coarse-grained checking for improved efficiency. At bisection, a sub-group inherits its parent group's checking granularity.

In the coarse-grained checking, we keep each variable's access log in the thread-local storage in $O(1)$ time without updating its meta-data. The log is used when a group switches to the fine-grained checking: each variable's epoch/clock information is re-

[2] Similarly, conflicting accesses can be connected by a chain of false dependences in static variable grouping.

[3] The term adaptive tracking is first proposed in [33] to switch between field- and object-granularity, however, our algorithm is entirely different.

stored by parsing the log. Parsing each log entry costs exactly one epoch/clock operation.

In the fine-grained checking, we keep a queue of events whose corresponding group dependences are not ordered in $\overset{syn}{\to}$. Once such an event is later ordered in $\overset{syn}{\to}$, it is removed from the queue. When the queue becomes empty, the group is switched back to the coarse-grained checking for recovered efficiency. Queue operations can also be performed in $O(1)$ amortized epoch/clock operations per group dependence.

Whenever there is a data race, it must not be ordered in $\overset{syn}{\to}$ and we guarantee to catch it in the fine-grained checking. Assume that (1) group dependences are magnitudes less than shared memory accesses; and (2) most group dependences are ordered in $\overset{syn}{\to}$ and never fallback, efficiency of data race detection can be improved. Assumption (1) is validated in our evaluation and assumption (2) is expected to be valid because after a series of bisections, variables in a group exhibit both thread and spatial locality, which are very likely to be properly synchronized.

The above analyses reveal the potential of bisectional coordination in efficient data race detection. However, this algorithm is non-trivial and requires tuning and enhancements for a practically efficient implementation. As we focus on the fundamental problem of online shared memory dependence reduction, these details are to be studied in the future work.

5.3 False Sharing Detection

False sharing occurs when non-shared variables are accidentally placed in a same hardware cache line [26]. False-sharing variables cause the cache line to bounce between the processor cores, which may severely degrade the performance.

False sharing can be detected by either hardware profiling or dynamic analysis via program instrumentation [19, 34]. A typical false sharing detection algorithm monitors memory addresses that may potentially happen in the same cache line. This resembles a static variable grouping that has cache-line-size groups. A variable group is suspicious to false sharing if it frequently incurs false dependences. Such an approach requires a shadow memory to keep track of sharing status for each variable group, incurring significant time and space overhead.

Bisectional coordination provides a mean to alleviate the problem because its amount of intervals (i.e., meta-data) is magnitudes smaller than a concurrent program's memory footprint. By setting its minimum interval size to be the hardware cache line size, cache-line-size intervals are not bisected even if false dependences are accumulated[4]. All such false dependences should be checked against false sharing. If false sharing does exist and is frequent enough to be noticeable, a large interval will eventually be decomposed into cache-line-size intervals and false sharing will be detected.

6. IMPLEMENTATION

Though the basic idea of bisectional coordination is straightforward, there still remain implementation challenges. We implemented bisectional coordination protocol as a deterministic replay extension to our previous work RWTrace [15] based on the LLVM toolchain [1] for C/C++ programs. Section 6.1 describes implementation details of the bisectional coordination protocol, followed by the deterministic replay implementation in Section 6.2.

6.1 Bisectional Coordination

The first implementation challenge is efficient computation of $f(x)$, i.e., finding the variable group $[l_i, r_i)$ containing a given memory address x. The bisection history has a binary tree structure (Figure 5). A naive binary tree lookup algorithm starting from the root costs an unaffordable $O(\log M)$ time.

We accelerate the lookup by a globally shared cache (i.e., a concurrent hash table with LRU replacement) that buffers each variable's last-time accessed interval node in the binary tree. Each time before the program accesses a shared variable, the validity of its cached tree node is checked. Only when the cache misses or the node is invalid (the node is bisected), the $O(\log M)$ lookup is performed. Furthermore, cache lookups are not synchronized. We use the same memory ordering technique in RWTrace [15] to check if a cached tree node is still valid after the thread-local read check, otherwise the read operation is retried with synchronization (so does the cache lookup). This treatment retains the $O(1)$ wait-free and barrier-free thread-local read fast path[5] of RWTrace, reducing the runtime overhead.

The second challenge is efficient identification of false dependences. Suppose that a dependence is captured for $G_i = [l_i, r_i)$ when accessing variable x. To determine whether it is a false dependence, we should know whether x has been written since the last dependence of G_i being logged. However, it costs too much time and space to maintain each group's write set. Instead, we associate each variable group G_i with a 256-bit bloom filter [5] to keep its *approximate* write set, achieving $O(1)$ overhead for each shared memory access. If there is a false dependence, it is guaranteed to be detected by the bloom filter.

Our experiments show that without these two implementation techniques, bisectional coordination would incur 5–10× slowdown to RWTrace for our evaluated benchmarks, making it interesting in theory but useless in practice.

Furthermore, setting the initial partition to be $\mathcal{P} = \{[0, M)\}$ would cause the "cold start" problem: false dependences and bisections will be frequent at the period of program start. In practice, we set the initial partition that decomposes the address space into relatively large intervals to alleviate this issue. We set the initial partition size to be 2^m ($m = 20$, 1MB interval) according to the empirical study results in Figure 2. We also do not bisect an interval of cache-line-size (64 bytes) because false dependences on it indicates false sharing, which is not expected in practice.

Finally, we set the bisection threshold $\delta = 20$ (i.e., we bisect an interval if $\delta = 20$ false dependences are captured on it) such that not too many false dependences are captured. As shown in the evaluation, the bisection counts are small for all evaluated subjects. Adjusting the value of δ would have minor impact on the amount of dependences being logged.

6.2 Deterministic Replay

In practice, logging only shared memory dependences is insufficient for deterministically replaying a past execution. We should also log dependences between synchronization operations (e.g., mutex lock/unlock and conditional variable wait/signal). In theory, this treatment is unnecessary because the order of shared memory accesses implies the synchronization order. However, in practice, our instrumentation cannot reach the shared memory accesses in dynamically linked library functions (e.g., memcpy). Thread-safety of such library functions is usually managed by synchronization operations. We therefore replace functions with a pthread_

[4] Such intervals always aligns with the cache line boundary because bisection is always at the midpoint of an interval whose length is a power of 2.

[5] Like [15], the barrier-freeness is achieved for the x86-TSO memory model. Barriers are required for weaker memory models.

Table 1: List of benchmark programs and evaluation settings.

Subject		*Description*	*LOC*	*Evaluation Setting*
desktop	aget	parallel data fetch	2.5K	16 threads, 64MB file
	pfscan	parallel file scan	1.1K	16 threads, 640MB file
	pbzip2	parallel compression	1.9K	16 threads, 64MB file
scientific	ocean	ocean simulation	9.1K	16 threads, 1026 × 1026 grid
	water	water simulation	3.6K	16 threads, 1000 molecules, 10 steps
	fft	fast fourier transformation	1.4K	16 threads, 2^{24} data points
	radix	radix sort	1.9K	16 threads, 2^{24} elements
	fluid	fluid simulation	1.2K	16 threads, 10 frames, 100KB input
	qsort	quick sort	0.9K	16 threads, 2^{24} elements
	x264	video encoding	37K	16 threads, 128 frames, 640 × 360 resolution
server	knot	HTTP server	1.2K	16 threads, 2^{14} requests
	apache	HTTP server	339K	16 threads, 2^{14} requests

Table 2: Effectiveness evaluation results. In Column 2, numbers in the brackets indicate the reduction percentages compared with RWTrace and LEAP. Column 5 shows the amount of bisections. Column 6 shows the amount of synchronization dependences. Column 7 shows the shared memory access count (collected by separated profile runs).

Subject	*# Shared memory dependences*			*# Bisect*	*# Sync.*	*# Mem.*
	Bisect (RWT / LEAP)	*RWT*	*LEAP*			
aget	7.40K (↓9.0% / ↓16%)	8.13K	8.82K	0	8.10K	39.9K
pfscan	116K (↓34% / N.A.)	175K	Overflow[6]	12	323K	9.82G
pbzip2	0.30K (↓55% / ↓76%)	0.67K	1.25K	28	0.54K	5.21K
ocean	27.1K (↓5.2% / ↓99%)	28.6K	57.4M	60	24.3K	138M
water	53.8K (↓97% / ↓99%)	1.69M	78.1M	52	99.0K	112M
fft	0.23K (↓90% / ↓99%)	2.35K	29.9K	4	0.23K	40.0M
radix	0.16K (↓93% / ↓99%)	2.34K	9.19M	34	0.56K	112M
fluid	9.52K (↓39% / ↓99%)	15.7K	170M	16	813K	463M
qsort	319K (↓55% / ↓79%)	706K	1.50M	72	206K	15.3M
x264	1.63M (↓91% / ↓98%)	18.9M	95.5M	954	20.3K	6.80G
knot	37.6K (↓0.5% / ↓45%)	37.8K	68.2K	18	0.03K	159K
apache	44.2K (↓79% / ↓98%)	214K	1.96M	89	32.3K	6.64M

prefix by our instrumented ones such that dependences between synchronization operations are logged.

Furthermore, we should also log each thread's I/O read operations so that file and socket readings are faithfully logged. Rather than instrumenting the library functions, we place kernel probes at system call returns using systemtap [2] to collect I/O read logs separately for each thread and flush these logs at program exit.

With these logs, the replay implementation is relatively straightforward. We elide all synchronization operations in the program to avoid deadlock and run the program in respect with the logged dependences (each operation waits until all its predecessors are finished) as well as feeding each thread with its corresponding I/O read log by the kernel probe.

7. EVALUATION

7.1 Experimental Settings

We conducted evaluation of bisectional coordination to answer the following two questions:

1. (*Effectiveness*) Can bisection coordination effectively reduce shared memory dependences compared to existing conservative static variable grouping techniques?
2. (*Efficiency*) Maintaining a dynamic address space partition brings extra costs. Is such overhead affordable?

We evaluated our bisectional coordination implementation on a set of real-world benchmark programs and practical workloads listed in Table 1. The same set of programs were also used in our evaluation of RWTrace [15]. The subject programs contain desktop, scientific, and server programs, which are selected from PARSEC [3], SPLASH-2 [29], and previous research work [17].

The effectiveness of bisectional coordination is quantified by the amount of shared memory dependences reduced. We compared bisectional coordination with RWTrace and LEAP [9], two state-of-the-art shared memory dependence tracing algorithms that can be used for deterministic replay. Both RWTrace and LEAP implementations use a 64-byte static variable grouping. This value is equal to the hardware L1 cache line size and is the best static variable grouping strategy according to our study in Section 3.2.

The efficiency of bisectional coordination is quantified by the slowdown against the unmodified RWTrace implementation. Evaluation results of RWTrace [15] showed that it outperforms the traditional lock-based techniques and scales well. We evaluated whether the implementation techniques in Section 6 have reduced the runtime overhead to an acceptable level.

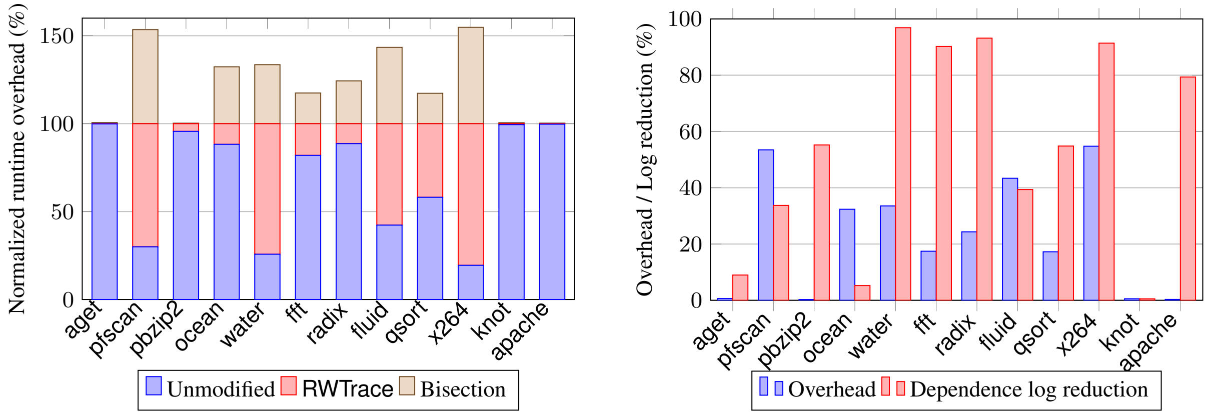
All experiments were conducted on a server with 4×6-core Intel Xeon X7460 processors (24 cores in total) and 64GB RAM. All evaluation results were averaged over 10 program runs.

7.2 Evaluation Results

7.2.1 Effectiveness

The effectiveness evaluation results are shown in Table 2. We first observed that there are only a few amount of bisections (0–

[6] LEAP exceeded our buffer size limit of 4G dependences.



(a) Normalized runtime overhead. RWTrace (static variable grouping) is the baseline (100%).

(b) Trade-off: runtime overhead v.s. dependence reduction. Values are normalized to RWTrace.

Figure 6: Efficiency evaluation results.

954, median 31) during the program execution. Recall that the initial size of an interval is 1MB and an interval is bisected when it accumulates $\delta = 20$ false dependences (Section 6.1). These results validate our assumption that most shared memory accesses exhibit both thread and spatial locality.

In comparison to RWTrace that uses static variable grouping, bisectional coordination reduced 0.5–97% (median 55%) of shared memory dependences. Compared with LEAP, 16–99.99% (median 99%) of dependences were reduced. CARE [14] and CLAP [12] can also reduce shared memory dependences and both use LEAP as the baseline. According to the evaluation results in their papers[7], CARE reduced 11–99.6% (median 86%) of dependences and CLAP [12] reduced 72–97.7% (median 91%), which are both lower than ours.

For scientific programs that partition data into blocks (water and fft) or perform passes of linear scan on data (radix), bisectional coordination reduced more than 90% dependences. For data-intensive programs of diverse shared memory access patterns: qsort (recursive sorting), x264 (pipelined consumer-producer), and apache (producer-consumer), we achieved 55%–91% reduction. For desktop applications (aget, pfscan, and pbzip2) that do not use data-parallelism, we also achieved 9.0%–55% reduction. The two exceptions are ocean and knot. The producer-consumer communications in ocean only exchange a small amount of data, yielding transitively irreducible dependences. The similar reason applies for knot because most of its dependences are incurred by a single racy shared counter.

Finally, by comparing the shared memory dependence count and the synchronization operation count (Columns 2 and 6), we found that our reduced shared memory dependences are similar in *magnitude* to synchronization operations for 10 out of 12 subjects, except for x264 (due to the dynamic pipeline) and knot (due to data races). We believe that with bisectional coordination, the amount of shared memory dependences would no longer be a bottleneck for understanding the behavior of many practical concurrent systems.

7.2.2 Efficiency

The efficiency evaluation study results are presented in Figure 6a. The runtime data is normalized such that the runtime of RWTrace (static variable grouping) is one unit. The runtime overhead of bisectional coordination is 0–54.7% (median 21%) over RWTrace.

A closer examination shows that the runtime overhead of bisectional coordination, like RWTrace, was higher for those programs who have more shared memory accesses. The worst runtime overhead was around 55% for pfscan and x264 because both programs perform more than 10^9 memory accesses per second (Column 7 in Table 2). Even only a few wait-free checks are performed for most of the shared memory accesses, the overhead is magnified by the large quantity. We believe that if such shared memory accesses can be proven to be race-free in advance (e.g., by a static analysis [27]), the runtime overhead would be significantly reduced if we do not instrument them.

Bisectional coordination presents a trade-off between the runtime overhead and the shared memory dependence reduction effect. Figure 6b demonstrates such results. In most cases, bisectional coordination achieved significant dependence reduction with little or even negligible overhead. Overall, we believe that a 0–54.7% (median 21%) of overhead upon an optimistic shared memory dependence tracing algorithm is beneficial for dynamic analyses, considering it reduced up to 97% of the shared memory dependences.

8. RELATED WORK

The order between shared memory accesses, i.e., shared memory dependences, can be used to test and debug concurrent programs. The most fundamental application of shared memory dependences is deterministically replaying a past concurrent execution, which enables further trace analyses [10, 22, 23].

Obtaining shared memory dependences is the basis of all deterministic replay techniques, no matter such dependences are obtained online (LEAP [9] and CARE [14]) or synthesized offline (CLAP [12] and Light [18]). How to efficiently obtain shared memory dependences is also studied as a standalone problem [6, 15] or as a basic technique for dynamic analysis [8].

Grouping spatially adjacent variables can reduce the amount of shared memory dependences (Section 3.2). This intuition has already been adopted in the shared memory dependence tracing algorithms, also for reducing the overhead of meta-data bookkeeping. For example, it is natural to group fields of an object in Java [6, 32] or group variables in a hardware cache line [15]. Grouping variables in the same hardware page is also a possible choice [7].

[7] We did not evaluate these techniques because they use different language platforms and difficult to be re-implemented without losing their dedicated optimizations.

Experiments showed that non-consecutive grouping is also efficient in practice [9, 35]. The hybrid of static analysis and dynamic profile information are used to establish a more efficient static variable grouping [17].

The idea of dynamically changing the granularity of a variable group is first studied in RaceTrack [33], which adaptively switches between field and object granularity. SlimState [28] adaptively compresses an array's meta-data to adapt to its shared memory access patterns. Bisectional coordination differs from these two approaches because it maintains a spatial and thread locality view of the entire address space, which is more efficient in reducing shared memory dependences.

A straightforward approach to reducing shared memory dependences is offline analysis, which solves a combinatorial optimization problem [13, 11] to obtain a more compact yet equivalent shared memory dependence log. Such offline analyses are effective but barely scale when the trace becomes huge. Therefore, it would be better to reduce shared memory dependences before they are logged. However, all existing online shared memory dependence reduction algorithms use vector clock for (regulated) transitive reduction [21, 31], which is difficult to be efficiently realized without hardware customization.

9. CONCLUSION AND FUTURE WORK

In this paper, we addressed the fundamental problem of online reducing shared memory dependences by software-only program instrumentation. We present the bisectional coordination protocol that adaptively maintains a dynamic interval partitioning of the address space that reflects the locality of shared memory accesses. Evaluation results on real-world benchmarks show that by paying an extra 0–55% runtime overhead, bisectional coordination reduced up to 97%/99.99% shared memory dependences compared with RWTrace [15]/LEAP [9].

An interesting (and challenging) future direction is to allow an arbitrary set of variables to be grouped as well as to allow online merging of variable groups. Furthermore, applications of bisectional coordination are also worth studying. Data race detection, false sharing detection, and software transactional memory are all promising future directions.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for helpful comments and suggestions. This work was supported in part by National Basic Research 973 Program (Grant #2015CB352202), National Natural Science Foundation (Grant #61472174, #91318301, #61321491) of China, the program for Outstanding PhD candidate of Nanjing University, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

11. REFERENCES

[1] *The LLVM compiler infrastructure*. http://llvm.org/.

[2] *SystemTap*. https://sourceware.org/systemtap/.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, software-only region conflict exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 241–259, 2015.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and controlling cross-thread dependences efficiently. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA, pages 693–712, 2013.

[7] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE, pages 121–130, 2008.

[8] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 121–133, 2009.

[9] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE, pages 207–216, 2010.

[10] J. Huang, Q. Luo, and G. Rosu. GPredict: Generic predictive concurrency analysis. In *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 847–857, 2015.

[11] J. Huang and C. Zhang. An efficient static trace simplification technique for debugging concurrent programs. In *Static Analysis*, volume 6887 of *Lecture Notes in Computer Science*, pages 163–179. Springer Berlin Heidelberg, 2011.

[12] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pages 141–152, 2013.

[13] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE, pages 57–66, 2010.

[14] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu. CARE: Cache guided deterministic replay for concurrent Java programs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE, pages 457–467, 2014.

[15] Y. Jiang, D. Li, C. Xu, X. Ma, and J. Lu. Optimistic shared memory dependence tracing. In *Proceedings of the 30th International Conference on Automated Software Engineering*, ASE, pages 524–534, 2015.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[17] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, pages 463–474, 2012.

[18] P. Liu, X. Zhang, O. Tripp, and Y. Zheng. Light: Replay via tightly bounded recording. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*, pages 55–64, 2015.

[19] T. Liu and E. D. Berger. Sheriff: Precise detection and automatic mitigation of false sharing. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 3–18, 2011.

[20] A. W. Madison and A. P. Batson. Characteristics of program localities. *Commun. ACM*, 19(5):285–294, 1976.

[21] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR workshop on Parallel and distributed debugging*, PADD, pages 1–11, 1993.

[22] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 25–36, 2009.

[23] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pages 11–21, 2008.

[24] J. Silva, J. Simao, and L. Veiga. Ditto - deterministic execution Replayability-as-a-Service for Java VM on multiprocessors. In *Middleware*, volume 8275 of *Lecture Notes in Computer Science*, pages 405–424. Springer Berlin Heidelberg, 2013.

[25] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.

[26] J. Torrellas, M. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *Computers, IEEE Transactions on*, 43(6):651–663, Jun 1994.

[27] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE, pages 205–214, 2007.

[28] J. Wilcox, P. Finch, C. Flanagan, and S. Freund. Array shadow state compression for precise dynamic race detection. In *Proceedings of the 30th International Conference on Automated Software Engineering*, ASE, pages 155–165, 2015.

[29] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[30] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings. of the 30th Annual International Symposium on Computer Architecture*, pages 122–133, June 2003.

[31] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 49–60, 2006.

[32] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. ORDER: Object centric deterministic replay for java. In *Proceedings of the USENIX conference on USENIX annual technical conference*, ATEC, pages 30–43, 2011.

[33] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP, pages 221–234, 2005.

[34] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE, pages 27–38, 2011.

[35] J. Zhou, X. Xiao, and C. Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 892–902, 2012.