A

Section Based Program Analysis to Reduce Overhead of Detecting Unsynchronized Thread Communication

MADAN DAS, GABRIEL SOUTHERN, JOSE RENAU, UC Santa Cruz

Most systems that test and verify parallel programs, such as deterministic execution engines, data race detectors and software transactional memory systems, require instrumenting loads and stores in an application. This can cause a very significant runtime and memory overhead compared to executing uninstrumented code. Multithreaded programming typically allows any thread to perform loads and stores to any location in the process's address space independently, and such tools monitor all these memory accesses. However, many of the addresses in these unsynchronized memory accesses are only used by a single thread, and do not affect other executing threads.

We propose Section-Based Program Analysis (SBPA), a novel way to decompose the program into disjoint code sections to identify and eliminate instrumenting such loads and stores during program compilation, so that the program runtime overhead is significantly reduced. Our analysis includes improvements to pointer analysis, and uses a few user directives to increase the effectiveness of SBPA further. We implemented SBPA for a deterministic execution runtime environment, and were able to eliminate 51% of dynamic memory access instrumentations. When combined with directives, such reduction increased to 63%. We also integrated SBPA with ThreadSanitizer, a state of the art dynamic race detector, and achieved a speed-up of 2.43 (2.74 with directives) on a geometric mean basis.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel Programming

General Terms: Program Analysis, Race Detection, Deterministic Execution, Software Transactional Memory

ACM Reference Format:
Madan Das, Gabriel Southern and Jose Renau, 2015. Section Based Program Analysis to Reduce Overhead of Detecting Unsynchronized Thread Communication. *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Typical multithreaded applications assume implicit data sharing among threads. Dynamic data race detection [Savage et al. 1997], Software Transactional Memory (STM) [Shavit and Touitou 1995], and deterministic execution [Bergan et al. 2010] have been proposed as ways to reduce errors associated with multithreaded programming. These techniques detect unsynchronized communications among threads and either report them as possible bugs (for data-race detection), or change the behavior of running threads (for STM and deterministic execution).

However, the use of these techniques is limited in practice by significant instrumentation overhead associated with software implementations [Cascaval et al. 2008]. These techniques perform runtime checks on loads and stores that cause significant slowdown in program execution. One way to improve the performance of such systems

This work is supported in part by NSF grants 1337278 and 1318943. Authors' address: M. Das, G. Southern and J. Renau, Computer Engineering Department, UC Santa Cruz, 1156 High St, Santa Cruz, CA, USA

DOI: http://dx.doi.org/10.1145/0000000.0000000

is to decrease the cost of each dynamic check operation. For instance, FastTrack [Flanagan and Freund 2009] improved on the performance of Djit+ [Pozniansky and Schuster 2003] by reducing the cost of most runtime checks needed for data race detection.

An alternative and complementary way to improve performance is to eliminate instrumentation of memory accesses that do not need to be monitored. Choi et al. [2002] proposed using static race detection in conjunction with dynamic race detection to reduce instrumentation overhead. In this paper we propose a new method of identifying disjoint program sections in a multi-threaded program and use this to reduce instrumentation overhead for dynamic data race detection, deterministic execution runtime systems, and STMs. We share the same goal of reducing instrumentation overhead by combining static analysis with dynamic runtime checks, but have developed a novel technique to decompose applications into sections defined by thread creation, barrier, and join operations.

The majority of memory accesses by a thread do not affect the behavior or state of other threads. For example, Figure 1 shows a thread $T1$ performing a write to a shared memory location W_x in a code section $CS1$, and another thread $T2$ performing a read of the same location after a synchronized barrier. The barrier creates a *happens before* relationship between the two sections, so the two code sections cannot execute simultaneously. Consequently, the read is race free and does not need to be instrumented.

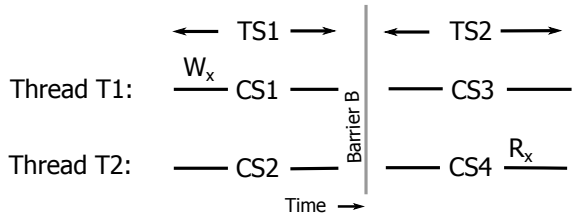

Fig. 1: Two threads T1 and T2 execute four different code sections (CS1, CS2, CS3, CS4) separated by a barrier B. The barrier implicitly creates two thread sections (TS1, TS2) that cannot execute simultaneously. W_X and R_X represent write and read of some memory location X. Since X is not modified anywhere in TS2, R_X does not need to be instrumented. W_X is instrumented, as CS1 could potentially be executed by multiple threads in TS1, resulting in conflicting writes.

Previous work on race detection [Mellor-Crummey 1993; Raman et al. 2010; 2012] has also avoided instrumenting loads and stores in serial sections of code. However, prior work mainly targeted structured programming environments such as Cilk [Blumofe et al. 1995] or Fortran using nested parallel loops. Structured programming explicitly restricts communication among threads, which makes it easier to determine when an application is executing a serial section of code. In contrast, our work focuses on statically partitioning an application written using unstructured parallelism with thread spawns and barriers. Our algorithms segment the whole program into sections, and use this information to eliminate unnecessary instrumentation. We also improve the precision of alias analysis for memory locations accessed in these different sections, by extending flow insensitive Steensgaard [1996] points-to analysis to use dynamically adjusted non-unified points-to sets. As a final addition we evaluate adding user directive to further reduce instrumentation for some applications.

We implemented and evaluated our techniques with a deterministic execution system similar to CoreDet [Bergan et al. 2010] and measured the reduction in instrumentation. Our implementation and evaluation works with C and C++ applications that use Pthreads; however, the SBPA concepts could be applied to other languages and

multithreading techniques. We also integrated our pass with ThreadSanitizer [Serebryany et al. 2012], a widely used dynamic race detector, and measured runtime improvements using our optimizations.

In this paper we propose Section-Based Program Analysis (SBPA) with the following contributions:

— Propose a novel method to statically decompose multithreaded applications into code sections, and analyze their communication pattern.
— Propose a set of user directives to improve static analysis and programmer reasoning about memory accesses.
— Evaluate SBPA in a deterministic runtime system similar to CoreDet and show 51% average reduction in dynamic instrumentation for a variety of benchmarks. When user directives are added, the total instrumentation reduction increased to 63%. Reduction by current state-of-the-art compiler techniques, as used in CoreDet, is 16%.
— Integrated SBPA with ThreadSanitizer, improving execution time of instrumented programs by an average of 2.74 times.

Section 2 of this paper defines the terms used. Section 3 describes how we create a reduced inter-procedural CFG of the whole program needed by SBPA. Section 4 describes the partitioning of an application into sections and how this is used to reduce instrumentation based on such partitioning. Section 5 describes our experimental setup, Section 6 presents results, Section 7 surveys related work, and Section 8 concludes.

2. DEFINITIONS OF TERMS

We define a *code section* as a collection of basic blocks that have a common dominator and post-dominator, and a *thread section* as a collection of code sections that *might* be executed concurrently. We define two types of thread sections as follows:

Single-threaded thread section *Single-TS* (Section 4.1) is a collection of code sections that can only execute when a single thread is active.

Disjoint thread section *Disjoint-TS* (Section 4.2) is a collection of code sections that *might* be executed by multiple threads simultaneously. Note that the same code section can appear in multiple Disjoint-TS. Two different Disjoint-TS sections cannot execute concurrently; only one Disjoint-TS can execute at any given time. A union of all the Disjoint-TS sections constitutes a conservative closure of multi-threaded code regions of the program.

Von Praun and Gross [2003] use the term *conflicting* for potentially unsynchronized accesses to shared objects. We define *conflicting* read-write (or equivalently load-store) as a pair of read and write of the same memory location by different threads that are executing concurrently, without a happens-before edge between them. For data-race detection, conflicting loads and stores must be synchronized with barriers and/or mutual exclusion checks. For deterministic execution, the requirement is stricter, since the lock acquire-release orders can affect the program output. So, even the lock protected non-racy accesses that do not have a happens-before relationship are treated as conflicting for our analysis. Thus, a conflicting access can be racy or non-racy. We define the other accesses separated by happens before edges as *non-conflicting*. A non-conflicting memory access by a thread does not affect the behavior of other threads executing concurrently.

In this paper, a *memory location* implies an abstract set of memory addresses that are seen as equivalent by pointer analysis used in SBPA. We use read/load and write/-store interchangeably for a memory read/write access.

3. CONSTRUCTING THE RICFG

SBPA requires an inter-procedural control flow graph (ICFG) of the whole program to understand the multi-threaded structure of the program. Since it is used only to detect

the multi-threaded regions of a program, and is an auxiliary graph that points back to original CFGs of functions, some optimizations can be performed to reduce the size of the ICFG. We call this reduced ICFG (*RICFG*) in subsequent discussions.

Algorithm 1 is a high level description of creation of the RICFG. The RICFG of the top level function `main` is constructed recursively. In the RICFG, if a function f or any function called within f has any thread related directive, we replace the call to function f at all its call sites with the RICFG of f. To do this, first the RICFG of f is created in a recursive manner. In RICFG of f, a common return node $R(f)$ is created that has incoming edges (R_i, $R(f)$) for each of the original return nodes R_i in RICFG of f. Then, for each call site of f, we create two nodes $B1$ and $B2$ for the basic block containing the call instruction. The call to f is replaced with an edge from $B1$ to the entry node in RICFG of f, and another edge from $R(f)$ to $B2$. The nodes in RICFG of f point back to original CFGs of functions, which remain unmodified.

Algorithm 1 Constructing RICFG for a function F.

```
1.start = First instruction of F
2.Copy CFG of F to its RICFG, with RICFG nodes pointing to original basic blocks.
3 foreach basic block B in F:
 .1 foreach instruction I in B:
  .1 if I is a call to function f with spawn, join or sync,
       or if I is a call to spawn a thread:
    .1 ricfg(f) = Create or find RICFG for callee function f at I.
    .2 Create node B1 for instructions in B before I.
    .3 Create node B2 for instructions in B after I.
    .4 Add edge from B1 to entry of ricfg(f)
    .5 Add edge from return node R(f) of ricfg(f) to B2.
  .2 else if I is a global barrier call:
    .1 n = find or create global sync node for used ID
    .2 Create B1 and B2 as above, partitioning B at I
    .3 Add edges B1 to n, and n to B2.
```

We do not insert the whole RICFG of f at its call sites, but only add two edges from and to its RICFG. Thus, there is no scalability issue with this approach. For each call site of f, 2 edges are added to the RICFG, and 2 nodes are created for the containing basic block. For our purpose of identifying threaded sections, we are interested in analyzing the control flow of functions that contain thread spawn, join, barrier or other thread directives. If a function does not have these directives, we do not need the inter-procedural flow of control through that function. This eliminates the need to include a vast majority of the basic functions in the RICFG. The interesting functions are tagged in a global hash table, so each function is inspected only once to check if it has a thread directive. Similarly, when the RICFG of a function is created, its entry node and the return node are recorded in a hash table indexed by this function. So, subsequent queries for the function simply returns information for the already constructed RICFG of the function in $O(1)$ time. Adding the incoming and outgoing edges to and from the RICFG of f at each call site is also $O(1)$ in complexity. For these reasons, the RICFG construction is runtime and memory efficient for large programs.

4. FINDING THREAD SECTIONS

We observe that in multithreaded applications, phase often exist naturally due to the kind of problems they are solving, and it is possible to identify those phases using flow analysis techniques. Figure 1 shows an example execution, with two thread sections TS1 and TS2, and four different code sections (CS1, CS2, CS3, CS4). TS1 consists

of CS1 and CS2, whereas TS2 consists of CS3 and CS4. Although not shown in the example, the same code section can appear in more than one thread section when multiple thread sections execute the same code.

Intuitively, a multithreaded application has thread sections because the programmer is responsible for ensuring that inter-thread communication is properly synchronized. The synchronization directives that the programmer uses divide the program naturally into thread sections, and the communication pattern among the thread sections can be detected at compile-time.

Once the program is segmented into these sections, we perform an improved points-to analysis to extract modified-referenced information for memory locations accessed in overlapping thread sections (described in Section 4.3). These techniques, combined, help identify the real conflicting loads and stores in the program. Loads from a memory location are conflict free if they happen in a thread section, which does not overlap with any other thread section that writes to the same location. These loads do not need to be instrumented. Similarly, if a store is conflict free (only writes to thread local memory), it does not need to be instrumented in tools that check data races or provide determinism. However, STMs still need to log or version the data in such stores to support restarts that may be triggered by conflicts with other memory accesses in a transaction. So, even conflict free stores in Disjoint-TS might need to be instrumented for STMs. Consequently, SBPA classifies the stores as conflicting, non-conflicting, and log-only.

4.1. Single-Threaded Thread Sections (Single-TS)

Programming Patterns: As defined earlier, Single-TS code can only be executed in single-threaded mode. Single-TS exist because some code is safe to be executed only when a single thread is active. All the concurrent threads are either not created or not executing in this mode. Application initialization and finalization in many programs fall in this category. In many applications, a parent thread waits for all the spawned threads to complete before running some finalization code. Also in the fork/join programming model it is common to alternate between parallel and serial regions. In all these cases, the serial regions could be detected as Single-TS, thereby eliminating the need to detect unsynchronized communication or races in these code sections.

If a memory location is determined to be modified only in Single-TS, all reads to the same memory location in any thread section are conflict free. Then, identifying Single-TS in a program can not only help eliminate instrumentation in the Single-TS regions of application, but also in the regions that are not Single-TS.

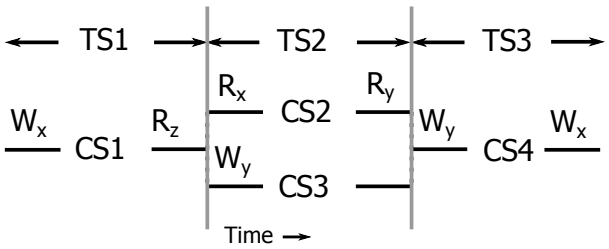

Fig. 2: A program comprised of 4 code sections in 3 thread sections

Detailed Example: Figure 2 shows a program divided into 4 code sections appearing in 3 thread sections TS1, TS2 and TS3. CS1 and CS4 are in single-TS TS1 and TS3 respectively, while CS2 and CS3 can execute concurrently within TS2. CS1 and CS4 are single threaded, so they don't need instrumentation. Memory location X is only

written in CS1 and CS4, and is accessed only for reads in TS2. So no instrumentation is required for X in any of the thread sections. Only Y needs instrumentation in TS2, as it is both modified and read in the same Disjoint-TS.

Algorithmic Solution: Algorithm 2 is a high level description of our method to detect code sections executed in single-threaded mode only. The algorithm marks code sections as either single-threaded or multithreaded. If a code section can be run in both single-threaded and multithreaded sections, it conservatively marks that as multithreaded code section (MTCS). While it is possible to replicate functions that are called in both Single-TS and MTCS, and assign one copy to Single-TS to avoid instrumentation, we did not see opportunities for this in the benchmarks we studied.

Algorithm 2 Finding multi-threaded code sections (MTCS).

```
1 create RICFG for top level function main
2 remove exit branches from global RICFG
3 foreach spawn call c in the RICFG:
 .1 if a join j is control equivalent with c:
  .1 code between c and j is multithreaded
 .2 else if c is in a loop:
  .1 foreach control equivalent loop with same iterations:
   .1 if the loop join calls always match c loop:
    .1 code between c and j is multithreaded
 .3 else:
  .1 code between c and exit is multithreaded
```

The algorithm finds multithreaded code sections (MTCS). Single-TS code is any code that is not included in any MTCS. It first creates the RICFG as described in Section 3. Thereafter, it splices away exit edges such as asserts and exits for error conditions. These are irrelevant for our analysis of which code can execute in parallel. This simplifies the RICFG for the purposes of post-dominance relationship. These two steps are shown as steps 1 and 2 in Algorithm 2.

The next step is to iterate over each spawn call (step 3). If the spawn call is followed by a single join call, and both are control equivalent [1] (step 3.1), all the code sections dominated by c and post-dominated by j are marked as MTCS (step 3.1.1). Since we use the RICFG for this, it includes all the function calls embedded in this section as well as the functions invoked by the thread spawns.

More commonly, a loop spawning a number of threads is followed by a similar loop that joins or waits until all the threads have finished execution. This is a typical fork/join program pattern in parallel applications. Both spawn calls and join calls should be in control equivalent loops that have the same number of iterations (step 3.2.1). In addition, we check that the join and spawn calls inside the loop are not control dependent, which guarantees that both loops call exactly the same number of spawn and join calls (step 3.2.2). When these conditions are met, we mark the loops and the code sections dominated by the spawn loop and post-dominated by the join loop as MTCS.

When we cannot find the control equivalent join of the loop, we conservatively assume that the rest of the program is multithreaded (step 3.3.1). This ensures that the single-TS identification is conservatively correct.

[1] Two nodes (x and y) in a control flow graph are control equivalent if x dominates y and y post-dominates x.

4.2. Disjoint Thread Sections (Disjoint-TS)

As defined earlier, code appearing in any Disjoint-TS can potentially run in multi-threaded mode. So, loads and stores in these sections need to be instrumented unless proven unnecessary. Portions of Disjoint-TS can overlap with each other. While some code sections of a Disjoint-TS are executed only by one Disjoint-TS, others might be executed by many Disjoint-TS.

Programming Patterns: Disjoint-TS occur due to non-overlapping parallel phases separated by Single-TS code sections, or due to global synchronization barriers within the same parallel phase. A common example of Disjoint-TS occurs when there is a global barrier in a program. The barrier divides the code in a parallel phase into two disjoint segments which cannot execute concurrently. This finer-grained partitioning helps identify non-conflicting reads and writes.

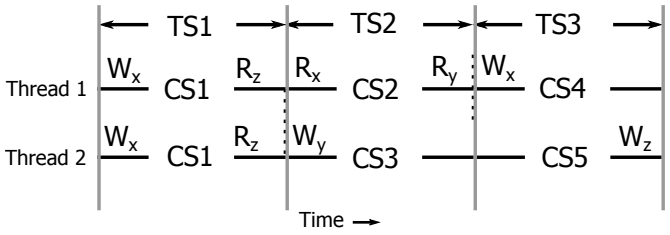

Fig. 3: Two threads executing 5 code sections in 3 disjoint thread sections.

Detailed Example: Figure 3 shows a program with 5 different code sections, CS1 to CS5, which read and write memory locations X, Y, and Z. Since all memory accesses are performed in multithreaded thread sections, without Disjoint-TS, all the accesses are assumed conflicting. However, the barriers guarantee that some accesses cannot overlap. Reads and writes to Z in code sections CS1 and CS5 belong to non-overlapping thread sections TS1 and TS3, so R_Z in TS1 is non-conflicting. Similarly, R_X in TS2 is data-race free as there are no writes to X in TS2. Accesses to Y in CS2 and CS3 may be conflicting, and must be instrumented. Stores are instrumented, as we can't determine how many threads are executing the same code section within a thread section. If the memory locations are thread local, then for data race detection, the stores don't need instrumentation. For STMs, stores to local memory are race free. But SBPA cannot prove that there is no restart due to some other conflict. So, those stores for STMs are instrumented as log-only stores.

Algorithmic Solution: To identify disjoint-TS in parallel code sections, the `pthread_barrier` function calls from the POSIX library are replaced by calls to `sync_barrier` with the same functionality, but that additionally provides a barrier identification ID. This method is required, as `pthread_barrier` is not a global synchronization directive. This step must be performed by the programmer, to annotate the global synchronization points in the program. The ID allows the compiler to determine the barriers where all threads synchronize. In the benchmarks we evaluated barriers were typically used in a global barrier style thus the `sync_barrier` annotation did not change the programmer's intent (although we did not integrate SPBA with OpenMP, we note that the barrier directive in OpenMP also acts as a global barrier).

We note that SBPA does not distinguish whether a code section in a Disjoint-TS is actually executed by a single thread or multiple threads at runtime. It pessimistically assumes that each code section that appears in multithreaded regions of a program can be executed by any number of threads. For the example in Figure 3, there could be one or more threads executing code sections CS1 through CS5. SBPA only relies on the

strict ordering of code sections, based on global synchronization, to determine if two code sections can overlap in any execution.

Algorithm 3 Finding disjoint thread sections in MTCS

```
foreach thread section (ts) found by Single-TS:
  1. if ts is single threaded, skip.
  2. let b = beginning of ts
  3. let e = end of ts
  4. while b != e:
    .1 C = reachable barrier nodes starting from b.
    .2 if C has a single node:
       .1 Code from b to C is a new Disjoint-TS
       .2 b = C
    .3 else exit search.
```

Due to factors such as code duplication by compilers for other optimizations, barriers across multiple functions, or barriers within loops, identifying the barriers where all the threads synchronize is complex, despite the use of `sync_barrier` annotations to identify global barriers. Hence we use a flow-based algorithm, Algorithm 3, that incrementally identifies these barriers starting from the beginning of a multithreaded section. The algorithm uses the RICFG created by Algorithm 2. In the RICFG, a single, global sync node is created for each `sync_barrier` call with a unique ID. The Disjoint-TS algorithm proceeds by iterating over each multithreaded thread section found by the Single-TS algorithm. It finds the reachable barrier nodes from start node through a depth first search (DFS) traversal of the RICFG, stopping at the sync nodes. If all the paths from the beginning of a thread section reach a single ID sync node (steps 4.1 and 4.2), then we know that all possible paths that start from the beginning of the multithreaded code section have to go through the same barrier, and so this section can safely be demarcated as a new Disjoint-TS (4.2.1). The new common sync node is then used as a starting point for another search (4.2.2), to find the next Disjoint-TS. If the reachable set of barriers has multiple IDs or is an empty set, then it conservatively abandons finding any new Disjoint-TS (4.3), and marks the end of the multithreaded section as the end of current Disjoint-TS. At the end of this process, each of the multi-threaded thread sections is partitioned into one or more Disjoint-TS separated by global barriers.

4.3. SBPA Pointer Analysis Framework

SBPA uses the DSA pointer analysis framework [Lattner and Adve 2005] which is a unification based [Steensgaard 1996] pointer analysis framework. This is also used in CoreDet [Bergan et al. 2010] from which we derived our baseline set of optimizations (henceforth referred to as *Base*) used in our evaluation. In addition, DSA works with the LLVM compiler framework which is used by SBPA. DSA is context sensitive in the sense that it can be applied to a local context of the program. However, both SBPA and *Base* need to identify the potential points-to set for the whole program to detect where a pointer used in an instruction may point, and whether it conflicts with any other access in the whole program. Therefore, for SBPA the analysis is performed in the global context. The pointer analysis first creates a local points-to graph for each function in a bottom-up manner. For a function call, the points-to sets of formal function arguments are merged with those of the actual arguments. We modified DSA as described below for improved results. Note that some of these techniques are covered in literature in various forms [Zhang et al. 1996] [Choi et al. 2002] [Landi et al. 1993]. We include

these here to provide a complete explanation of the differences with $Base$ method, The use of these techniques is categorized as *Alias* in our results section.

4.3.1. Perform modref analysis per section. We performed modified and referenced analysis for the points-to sets (memory regions) for each Disjoint-TS. For each points-to set, two separate bit arrays R and W are maintained for read and write accesses respectively. If $R[i]$ is true for a points-to set S, it implies some instruction in Disjoint-TS i reads some memory location in S. Similarly, $W[i]$ indicates a write to any location in S in Disjoint-TS i. A depth-first traversal for each section i sets $R[i]$ and $W[i]$ of the accessed points-to sets.

4.3.2. Adaptive non-unification for points-to sets of function arguments. Generally, in unification methods, two points-to nodes (abstract memory locations) are merged when any pointer can point to both of those nodes. In DSA [Lattner and Adve 2005], the calls to a function f are resolved by merging the points-to nodes of actuals with those of the formals of f at each callsite. This ultimately leads to the pointers corresponding to arguments of f pointing to an unified node that includes all the actuals at different call sites. So, if $f(X)$ and $f(Y)$ are two such calls, and $P(x)$ represents the points-to node of x, then this leads to a merger of $P(X)$ and $P(Y)$. However, if X and Y are accessed in a section where $P(Y)$ has a write and $P(X)$ only has a read, this false aliasing caused by merging $P(X)$ and $P(Y)$ results in instrumenting read-only accesses of X in that section.

To circumvent this issue, we modified DSA to avoid these mergers of actuals for some functions. To compute the points-to node for a pointer within f in the program context, we must then recursively traverse the function call graph in a bottom-up manner until the actual is not a function argument passed from above. This can lead to deep recursions, and an exponentially growing number of points-to nodes for the pointer. So, we used three thresholds to constrain the runtime and memory requirements of this scheme. First, this was only used for functions with at most $T1$ callsites. Second, if the pointer argument of f was passed many levels through the function call graph, then f was resolved by merging its arguments with actuals. This can be detected during the upward traversal in the call graph starting from each callsite of f. If the depth of such argument passing exceeded $T2$, then calls to f were also resolved by the original method. Third, if during the process, the number of elements in a points-to set of any argument of f exceeded another threshold $T3$, then the most commonly occurring elements within the set were merged to reduce the size of points-to set to $T3$. For its implementation, these non-unified points-to sets were maintained as a `map`, with each points-to node mapping to its occurrence count within the set. Whenever the size of the set exceeded $T3$, the two nodes with the most occurrences counts were merged into a single points-to node, making them indistinguishable. The calls to functions in strongly connected components in the call graph were resolved by the original method of unification.

For our experiments, we used $T1$ = 4, $T2$ = 4 and $T3$ = 10. To resolve the points-to set for any formal argument of a function, it can take at most $T1^{T2}$ recursions. Since both $T1$ and $T2$ are small constants, the worst case runtime of such computation is $O(1)$. $T1$ and $T2$ should not be large, as that leads to a high constant multiplier. The third threshold guarantees that the size of non-unified sets are within $T3$. Once the above process is completed in a top-down manner, the non-unified points-to sets of all function arguments that were not merged are available for use by SBPA analysis, and recomputation is not required. The benefit derived is the disambiguation of points-to sets that would otherwise have been merged if they appeared as actual arguments in multiple calls of the same function. Very small functions are inlined by the compiler, so their formals aren't merged by unification in most cases, thus making the pointer

resolution within these inlined functions context sensitive. So, our above heuristic only affects function call resolutions for relatively large functions.

The result is never worse than the original approach, since the different actuals would always have been merged otherwise. The pointer analysis remains context insensitive; only the mod-ref analysis becomes context sensitive for the points-to nodes not merged by this approach. We found this hybrid, adaptive approach to be less memory and runtime intensive while providing significant disambiguation benefits by making the mod-ref analysis more precise. As described above, we performed this non-unification strategy only for function calls. At other places, as in function bodies, the points-to nodes are unified as usual.

4.3.3. Field sensitivity for array elements. DSA is field sensitive only for small sizes of pointed memory objects. For large arrays, it loses field sensitivity as it collapses the points-to node to a size of 1. We modified it to reduce it to the size of the elements in the array. Thus, the array indices alias for all accesses, but the offsets within the array element still remain distinct. We added a check for strided accesses to unalias the store to one field from load from another field. To minimize performance overheads, we used a simple scheme based on offsets from the beginning of a structure's memory. Assume a read r and a write w in the same Disjoint-TS accessed data through the same points-to set that has elements of type *struct ss*. Also assume that the access sizes are s_r and s_w respectively, at offsets of o_r and o_w respectively within the struct. In such a case, if $o_r + s_r < o_w$, or $o_w + s_w < o_r$, there is no overlap, and so r and w can be classified as non-conflicting. This helped significantly for arrays of structures, where in some phases, only a certain field is written while other fields are only read.

4.4. Programmer Annotations (Directives)

In this work, we focus on automatically reducing instrumentation without any program modification. However, in practice, due to various other overlapping compiler optimizations that can make the CFG more complex to analyze, and the pessimism of points-to analysis, sometimes static analysis fails to derive the most precise sets of conflicting loads and stores. In such cases, it may be useful for the programmer to insert directives that help compilers understand the program flow more easily. The directives act as hints; inexpensive runtime checks are performed to verify that they hold true at runtime. The runtime checks are executed very infrequently compared to the frequency of instrumented memory accesses. We propose two types of directives: a directive to indicate multi-threaded read-only memory, and a pair of directives to mark the multi-threaded sections.

4.4.1. Specifying Multi-Thread Read Only Memory. We define Multithreaded Read-Only Memory (MTROM) as a region of memory that is read-only, *only when* an application has multiple active threads. For data races to happen across multiple threads, at least one thread accessing a shared memory location must perform a write to that location while some other thread reads or writes to the same location in the same thread section.

Programming Patterns: Frequently, parallel sections of code only read a vast amount of data that is created in a non-parallel mode. A common example is the parallel dense matrix multiplication. During the parallel phase of the program, the matrices being multiplied are not written to; only the product matrix is modified. Hence, the input matrices can be allocated from MTROM memory space. Another such example of this pattern is operations on large graphs, where additions and deletions happen only in single-threaded mode, but the search operations happen in multi-threaded mode. In the benchmarks we tried, many exhibited this nature of parallel execution. The MTROM model also helps the programmer understand the communication patterns in a multithreaded application easily.

All MTROM memory is allocated from a special heap, called *MTROM-heap*. This heap is a reserved virtual address segment in the program's heap memory. At beginning of the multi-threaded phases, these heaps are marked as read-only through page fault protection available on most systems.

Theorem 1.: If all memory locations that a pointer can point to at a load instruction are MTROM memory locations, then that load instruction does not need to be instrumented.

Proof: Since it is guaranteed by the compiler and additionally checked by the runtime system that MTROM memory is never written in multi-threaded mode, all the possible memory locations pointed to by this pointer are invariant in all disjoint-TS sections. Hence, there is no race or non-determinism through this access. So, excluding this load access from instrumentation is conservatively correct. This access can only be a load, as a store will be an immediate violation.

Ideally, alias analysis as described in Section 4.3 would identify all such accesses. But due to limitations in static alias analysis, it is not always possible to do so. In the MTROM approach, the programmer allocates such memory from MTROM heap by replacing allocation routines with corresponding MTROM routines. The programmer can also mark some globals as MTROM, if desired. In practice, we observed that most large chunks of such memory are heap allocated or mmaped, as their sizes are not known upfront. We provide MTROM versions of functions (such as `malloc` and `mmap`), and require the programmer to use them when allocating MTROM memory. Pointer analysis marks the memory objects created by these methods as *MTROM-heap*, and similarly for global memories, it marks them as *MTROM-global*. During access instrumentation in thread sections, if a read memory access is traced to a points-to set that only contains MTROM-heap or MTROM-global regions, then that memory read access is not instrumented. On the other hand, if there is a write access to a points-to set containing only MTROM locations in some parallel section of the program, then the compiler pass can flag that as an error or warning. If the points-to set is a mixture of both MTROM and non-MTROM memory regions, then the read and write operations are instrumented as usual.

To detect incorrectly marked MTROM memory regions, a test can be implemented either in all the executions or during debug and test. To avoid adding overhead with additional checks in the write instrumentation functions, we protect the pages of MTROM heap by marking them as read-only. The read-only protection happens during multi-threaded execution only, and is marked as read-write whenever the program goes to a single thread mode. These checks are to detect incorrect MTROM annotation by the programmer, and also to alert inadvertent writes to such memory locations in parallel mode.

In the benchmarks we analyzed, using MTROM required changes in only a few lines of code to replace `malloc`, `mmap`, `free` and `munmap` with their MTROM equivalents, and an annotation at one place within a function to indicate that the variable is MTROM. This method is not automatic, but it is robust and general, since the writes to such memory are protected in parallel phases. It provides a strict guarantee to the programmer that there are no races through the MTROM memory regions.

4.4.2. Marking Single Threaded Code Sections. Sometimes, thread spawn and join calls have conditional calls that make it difficult to automatically detect the fork/join pattern. For these cases, we propose a pair of directives, `single_mode_begin` and `single_mode_end`. These directives do not change the behavior of the program; they just notify the compiler that the code section between the `single_mode_begin` and the first immediate post-dominator `single_mode_end` is executed by a single thread. To verify that the programmer does not introduce bugs, the runtime system verifies that when these are invoked, only a single thread is active. Otherwise, a runtime error is gen-

erated. These same directives can be used when the programmer knows that a single thread is performing work while the others are waiting. These were used only in 2 benchmarks in our evaluation (ocean_ncp and kmeans).

4.5. Overall Instrumentation Flow

The overall SBPA compilation flow proceeds as follows: We first perform an interprocedural pointer analysis. Then, we build the RICFG. Then Single-TS and Disjoint-TS algorithms are applied on the RICFG consecutively. Once we have detected the correct sets of memory locations accessed in different sections, we leverage per section modified/reference information to detect if the loads and stores in various sections can potentially share data with other threads. If a load/store is in single threaded code sections only, it is guaranteed to not have sharing issues. As the evaluation shows, we correctly detect nearly 80% of the single-threaded loads and stores.

For multithreaded code sections, we evaluate each load instruction as follows: If any store accesses the same points-to set as accessed in the load instruction in any of the Disjoint-TS where this load instruction can be executed, it implies that the access could potentially conflict with the store. So the load must be instrumented. Otherwise, the memory accessed by this load is only read in all the Disjoint-TS in which it is executed, and the load is not instrumented. All thread sections in which the load instruction can appear are considered before the load can be determined as conflict free. Since single threaded sections do not overlap with multi-threaded sections, the writes in single threaded mode are automatically excluded from consideration.

5. EXPERIMENT SETUP

We implemented SBPA as an LLVM compiler pass for a software runtime system which enforces deterministic execution of multithreaded applications. The software runtime system is similar to CoreDet [Bergan et al. 2010] and must instrument loads and stores to detect conflicts, unless proven to be non-conflicting. We used CoreDet's optimizations as our baseline for a system that does not perform task logging. These optimizations include escape analysis to avoid instrumenting thread-local data, and elimination of redundant instrumentations for same memory address on successive accesses.

Our compiler pass is implemented for LLVM 3.6. We included the poolalloc [Lattner and Adve 2005] module to incorporate Data Structure Analysis (DSA) as our starting pointer analysis engine. DSA is a context sensitive, unification based [Steensgaard 1996] alias analysis. We enhanced it as described in Section 4.3 to work with SBPA. We compiled each benchmark with Clang's -O4 optimization level to include link time optimization (LTO). Thereafter, we applied SBPA analysis and instrumentation pass on the intermediate representation (IR) of the whole program.

To evaluate SBPA we used benchmarks from the PARSEC [Bienia 2011], SPLASH [Woo et al. 1995], and Phoenix [Ranger et al. 2007] benchmark suites which are shown in Table I. We included results from all of the benchmarks that we are currently able to compile and run with our deterministic execution runtime system, which is under development. At runtime, we recorded the dynamic counts of the number of loads and stores instrumented.

For measuring dynamic instrumentation counts, all benchmarks were run with 2 threads, as the number of threads does not influence these counts. For comparing runtime improvement with ThreadSanitizer, as explained in Section 6.3, we used 2,4 and 8 threads. Unless otherwise stated, each benchmark is unmodified from its original state. Any improvement due to annotations in program are shown as part of *Directives* in our results.

Some benchmarks were excluded because of compatibility problems with the deterministic execution runtime system that we used for testing. The reasons for excluding certain PARSEC benchmarks were: *bodytrack*, *facesim*, and *ferret* had C++ exceptions

Suite	Benchmarks
PARSEC	blackscholes (bl), dedup (de), fluidanimate (fl), swaptions (sw), canneal (ca), streamcluster (st)
SPLASH	fft (ft), lu (lu), ocean_cp (oc), ocean_ncp (on), raytrace (rt), radix (rd)
Phoenix	histogram (hi), kmeans (km), linear regression (lr), matrix multiply (mm), pca (pc), reverse index (ri), string match (sm)

Table I: Benchmarks used, abbreviation shown in parentheses.

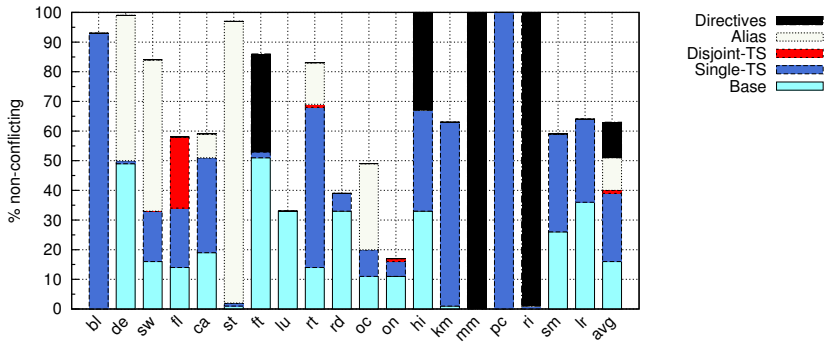

Fig. 4: Percentage reduction in instrumented accesses at runtime achieved by SBPA with different techniques.

which are not supported by our runtime system. *Freqmine* uses OpenMP instead of pthreads. *VIPS* and *x264* had several dependencies that we were unable to get working with our toolchain.

For SPLASH our runtime system did not work with the ad hoc synchronization used in *barnes* and *water*. We were unable to compile *radiosity* and *volrend* in 64-bit mode, while *cholesky* and *fmm* had segfaults with our runtime system.

Other deterministic execution runtime systems such as CoreDet [Bergan et al. 2010] and DThreads [Liu et al. 2011] are also unable to run many of these benchmarks. However, we do not think the SBPA analysis technique is incompatible with these benchmarks, but rather that our benchmark selection was limited by our runtime system. Furthermore, the 19 benchmarks that we used in our evaluation provide a solid foundation for assessing the usefulness of SBPA.

6. EVALUATION

The goal of SBPA is to identify as many non-conflicting loads and stores as possible. A non-conflicting memory access is guaranteed to be data-race free, and it also does not require checking for conflicts in Software Transactional Memory (STM). Non-conflicting stores can be classified as log-only (LogOnly) for STMs to reduce overhead, or can be ignored for data-race detection. We compare SBPA results with the current state-of-the-art compiler technique used by CoreDet [Bergan et al. 2010] that we call *Base* in this section. On top of Base, we add four improvements: Single-TS (Section 4.1),

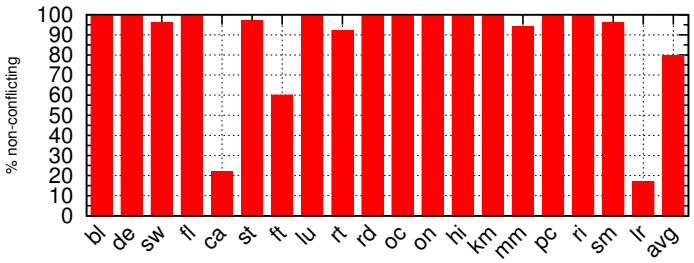

Fig. 5: Percentages of single threaded dynamic memory access instrumentation identified by SBPA as non-conflicting. Average reduction is 80%.

Disjoint-TS (Section 4.2), Improved Alias Analysis (Section 4.3) and Directives (Section 4.4).

The evaluation is divided into five sections: Overall Results (Section 6.1), Analysis of Reduction in Instrumentation (Section 6.2), Runtime with ThreadSanitizer (Section 6.3), Benchmark Insights (Section 6.4), and Compilation Overheads (Section 6.5). Overall Results provides details about the effectiveness of SBPA. Benchmark Insights explains the source of instrumentation and/or the reason SBPA failed to identify more memory accesses as non-conflicting. The evaluation finishes by showing that SBPA pass requires minimal compilation time, and that in fact, sometimes it makes the compilation process faster because it reduces the number of loads and stores that require instrumentation.

6.1. Overall Results

Figure 4 summarizes the results for SBPA. For each benchmark, it shows the percentage of non-conflicting memory operations that are identified as such at compile time.

On average, 51% of the memory operations are proved non-conflicting, while $Base$ only proved 16% of the memory operations as such. This is the dynamic count of loads and stores rather than the static count. In all our plots, we show the dynamic counts of memory operations, not the number of load or store instructions in the binary, as the latter does not represent runtime overheads. The average bar avg in Figure 4 shows that without any annotations, the reduction in instrumentation is 51%. It also shows that when we include `Directives`, the reduction increases to 63%.

The most effective technique is Single-TS. It is able to avoid checks for 22% of the memory operations. $Alias$ helped disambiguate some pointer aliasing, thereby reducing 11% of the memory access instrumentations on average. The benefit is most apparent in 5 applications (`de, sw, st, rt, oc`) as the base alias analysis was creating alias between different thread sections. Since we can guarantee that some loads and stores cannot execute simultaneously, we were able to improve the precision of alias analysis. $Directives$ significantly affected 4 applications (`ft, hi, mm, ri`). Section 6.4 provides a detailed explanation per benchmark, but the main reason is the declaration of some variables as MTROM, which helped by marking the truly invariant memory regions that were not detected as such by pointer analysis.

From these benchmarks, only `fluidanimate (fl)` has a significant improvement due to Disjoint-TS. The reason is that very few benchmarks have different barriers during multithreaded code sections, or perform a significant percentage of work in each disjoint thread section. Instead, most applications use a spawn fork/join model where threads are spawned to perform a task, and then are joined by the main thread. This kind of model is where Single-TS benefits the most.

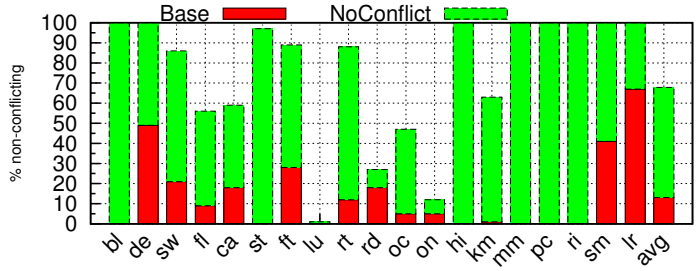

Fig. 6: Reduction in dynamic load instrumentation by SBPA. 68% of loads were identified as non-conflicting on average, with few applications reaching 100%.

6.2. Analysis of Reduction in Instrumentation

To measure the effectiveness of Single-TS pass, we ran the benchmarks with and without SBPA optimization and measured the number of memory access that were instrumented when only one thread was active in the program. Figure 5 shows the percentage reduction of one over the other. A 0% reduction implies that SBPA couldn't prove any access in single threaded code sections as non-conflicting. A 100% reduction implies that all such memory accesses were identified. On average, SBPA identified 80% of these memory operations. Most of the benchmarks have over 90% reduction except `ca, ft` and `lr`. The `lr` benchmark has very few such memory accesses due to use of `mmap` operation for the input file, and all the data is read in multithreaded mode.

To understand SBPA behavior, we also show the effectiveness for loads and stores separately. Figure 6 shows the percentage reduction in instrumentation for loads when applying SBPA optimization pass. The state-of-the-art (Base) removes only 13% of loads on average. SBPA can remove an additional 55% of loads. This means that on average 68% of the load operations can be proven non-conflicting (NoConflict). The results are better than those for the aggregate access counts because SBPA achieves a significant improvement for read-only accesses. The breakdown is also fairly disparate; for 8 benchmarks (`bl, de, hi, pc, ri, mm, sm, lr`) out of 19, it is able to remove over 99% of the loads that would need to be instrumented. This leaves little opportunity for other techniques to show any further improvement. The worst result is in `lu`, where nearly all the loads are marked as conflicting loads. The problem in `lu` is that the input matrix is modified in-place. The accesses are partitioned between threads in sub-blocks, but all threads access the same matrix. As a result, the alias analysis collapses all the memory accesses to a single points-to set. Even an ideal pointer analysis will have this problem, unless a range or partition analysis is performed.

Figure 7 shows the percentage reduction in instrumentation for all stores. In the case of stores, we divided the reductions as $Base$, $LogOnly$, and $NoConflict$. The state-of-the-art (Base) is able to mark 38% of the stores as NoConflict. SBPA improves that result, removing 61% of the stores for tools like data-race detectors. For STMs, SBPA proves over 52% of the stores as non-conflicting.

$Alias$ also plays a part in these instrumentation reductions. For the benchmarks we evaluated, it helped to reduce 12% of load accesses for baseline, and 19% of loads for SBPA. Its effect on store reductions was minimal ($< 1\%$).

6.3. A case study with ThreadSanitizer

To evaluate the effect of SBPA on runtime for dynamic race detection, we combined SBPA optimizations with ThreadSanitizer [Serebryany et al. 2012], a state of the art dynamic race detection tool. ThreadSanitizer works within the LLVM framework and uses Clang's front end to insert the instrumentation instructions. This is also one pri-

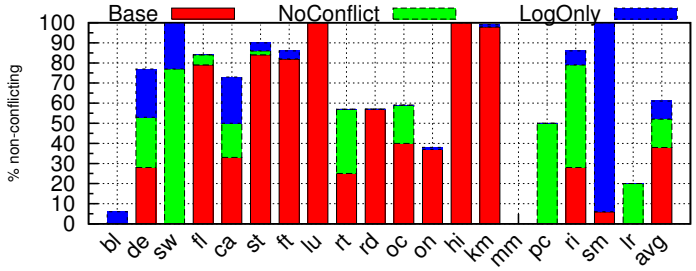

Fig. 7: Reduction in dynamic store instrumentation by SBPA. 61% of the stores do not need to be tracked by tools like data-race detectors, which is much better than 38% with Base. For STMs that may have restarts, 52% of the stores can be proven as safe.

mary reason to choose ThreadSanitizer over other such race detectors. We let ThreadSanitizer instrument the binary as it deems suitable. Thereafter, SBPA is applied on the whole program model to remove the unnecessary instrumentations inserted by ThreadSanitizer. In this method, our pass has very low interactions with the ThreadSanitizer instrumentation process. We evaluated each benchmark with the following options:

— No-Tsan: ThreadSanitizer pass not applied.
— Tsan-Zero: ThreadSanitizer pass is applied; then all instrumentations are removed.
— Tsan: ThreadSanitizer instrumented; none of the Base or SBPA optimizations applied
— Tsan-Base: Apply Base optimization on Tsan
— Tsan-SBPA: Tsan with SBPA, that includes Base, Single-TS, Disjoint-TS and Alias
— Tsan-Directives: Tsan with SBPA, and annotations where applicable

Tsan-Zero is included to measure the overhead that remains even when all instrumentations are removed. This is not a correct optimization, but it demonstrates a lower bound on the runtime that could possibly be achieved by by removing all instrumentation. Some benchmarks have a significant overhead regardless of the amount of instrumentation, while others have low fixed overheads.

Tsan-Base includes the optimizations that we described in previous sections, with the exception of certain optimizations that are not applicable to precise data race detection, and are dependent on the cache line size used in the previous section. Instead, we included an additional optimization to exclude accesses that are enclosed by the locking and unlocking of the same mutex variable. This is somewhat similar to the *Unlocked Pairs Computation* technique in [Naik et al. 2006], except that we can only remove an instrumentation when it is guaranteed that such removal will not miss any race. So, instead of may aliasing, our checking is based on must aliasing of the lock variables. If a memory location is accessed only in lock protected regions that are all protected by the same lock, then we remove instrumentation of accesses to that memory location. Although this technique should theoretically yield improved results, in reality only a few accesses are made in such critical regions due to the overhead of locking and unlocking, and because it causes serialization. Secondly, if the mutex variable is not global, then it is difficult to prove statically that the accessed memory regions are indeed protected by the same mutex. In our benchmarks, we only observed minimal benefit by using this optimization.

In Table II, we show the runtime in seconds of the instrumented executables with these optimizations. These runtimes were observed using 2 threads on a machine with dual processor Intel Xeon E5-2689 system (16 cores in total) with a clock frequency of

Test	No-Tsan	Tsan-Zero	Tsan	Tsan-Base	Tsan-SBPA	Tsan-Directives
bl	90.3	107.4	174.1	173.1	112.9	113.1
de	26.4	66.2	100.6	100.7	69.9	67.5
sw	22.0	32.5	180.8	171.1	148.9	148.9
fl	6.5	12.4	105.4	97.2	69.0	69.3
ca	28.7	119.3	153.8	151.5	119.7	118.6
st	11.3	11.8	281.9	277.8	21.3	21.3
ft	7.6	7.6	57.0	40.7	32.7	32.2
lu	1.3	1.4	71.9	60.4	53.2	53.2
rt	10.2	14.9	142.3	136.1	17.2	17.2
rd	27.5	27.7	59.5	60.0	57.7	57.5
oc	20.4	23.6	271.1	257.6	228.0	228.0
on	45.5	51.9	578.6	561.2	546.9	547.6
hi	0.9	0.9	29.1	22.4	11.9	7.9
km	26.6	28.9	440.4	440.6	245.5	245.4
mm	3.5	3.5	16.5	16.4	16.5	3.5
pc	25.5	27.1	939.6	936.5	28.0	27.9
ri	1.8	7.6	10.4	10.5	10.4	7.0
sm	5.1	155.3	213.2	211.3	196.4	196.6
lr	1.1	1.1	15.4	15.4	1.1	1.1

Table II: ThreadSanitizer dynamic race detection runtime in seconds for 2 threads.

2.60 GHz. In order to make the runtime comparisons meaningful, we selected the input for each benchmark to produce at least a few seconds of runtime in NoTsan mode. We also allocated specific CPU cores to the jobs and copied input data to local machine to exclude network delays from observed runtimes.

In Figure 8, we show the speed-up for each benchmark with SBPA, which includes all the optimizations. These are compared to the runtimes of $Tsan$, as measured with 2, 4 and 8 threads. Some benchmarks have higher speed-ups as the number of threads are increased. In some cases with many reported races, the race detection and reporting dominate the total runtime. In those cases, even removing the instrumentation didn't help the runtime significantly, regardless of the number of threads. In several of these cases, ThreadSanitizer reports races. And in all of those cases, the races continued to be reported after our pass was applied. This shows that our pass is working as expected, and that it is not removing any instrumentation that is necessary to detect races.

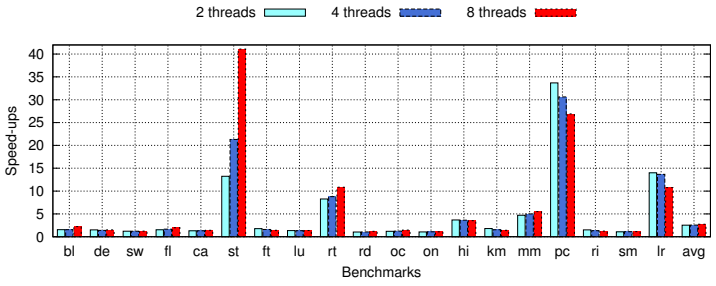

Fig. 8: Geometric mean speed-ups of ThreadSanitizer instrumented executables with various options.

We also show the overall speed-ups achieved with different optimizations in Figure 9. These are the geometric means of speed-ups of all the benchmarks for NoTsan, Tsan-Zero, Tsan-Base, Tsan-SBPA and Tsan-Directives compared to *Tsan*. (The prefix "Tsan-" is removed in the figure for brevity.) Overall, SBPA and Base achieved speed-ups of 2.425 and 1.116 respectively on geometric mean basis. When combined with directives, the speed-up increased to 2.736 times. The upper bound of speed-up is 6.5 as with *Tsan-Zero*. We note that *Tsan* is about 12 times slower than *No-Tsan* for these benchmarks.

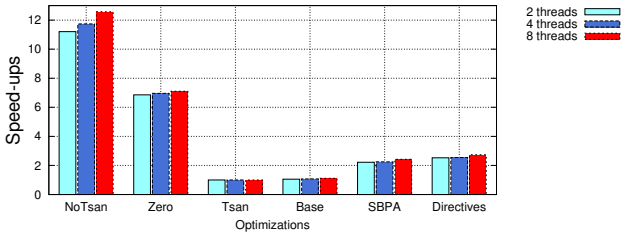

Fig. 9: Geometric mean of speed-ups of ThreadSanitizer instrumented executables with different optimizations.

In general, the speed-ups correlate fairly well with the percentage reductions in instrumentation we showed in an earlier section. But we note that the improvement in runtime is not as high as the reduction in dynamic instrumentation count in some cases. For certain benchmarks, such as `de`, runtime overhead didn't decrease as the optimizations used for our STM as part of Base was specific to cache line size used for that system. On some other benchmarks, such as `st`, `pc`, `rt`, `mm`, and `lr`, we see tremendous improvements. Yet in some others, the overhead even with zero instrumentation is quite high and dominates the runtime, so we do not observe as much speed-up as would be indicated by the reduction in instrumentation counts.

6.4. Benchmark Insights

To better understand these results, we provide some insights on the benchmarks below.
blackscholes (bl): bl iterates over 4 global read vectors. SBPA detects most of these accesses as read-only, and reduces load access instrumentation by 100%. For store accesses, the same array is written by multiple threads and so it cannot eliminate the store instrumentations. SBPA still determines 93% of memory accesses as non-communicating.
dedup (de): This is a pipelined parallel processing benchmark. With base optimization, the read accesses are already optimized by almost 50%, as there is significant local heap memory. With Single-TS and $Alias$, instrumentation is reduced significantly more, by 98%.
swaptions (sw): In swaptions, aliasing of locally allocated memory with global memory caused many instrumentations of local memory as well. $Alias$ with Single-TS unaliases some of these to reduce access tracking in the critical functions.
fluidanimate (fl): This uses fork-join model parallelism with many global barriers. It reads and modifies different fields of the structures in a large array of structures, and so field-sensitive analysis of $Alias$ combined with Disjoint-TS made significant improvement in this case.
canneal (ca): This is a simulated annealing algorithm, where only coordinates of objects are modified during the parallel phase. Single-TS shows a significant decrease, as

the graph creation process did not require any instrumentation. Field sensitive pointer analysis helped, since only the coordinates of each node in the graph are modified, and not their connectivity matrix.

streamcluster (st): This is an RMS kernel that clusters a large number of points in different clusters based on a pre-determined number of medians. We observed several local memory allocations and also noticed that the medians are computed in a separate array. So, most of the fields of the points are not modified and field-sensitive modref analysis significantly reduced the amount of instrumentation needed.

FFT (ft): ft has a global array of input time domain points that is not modified. SBPA detects accesses to this as non-communicating, and reduces load counts by 30%. $Alias$ reduced this further by disambiguating memory regions pointed by various function arguments. FFT is one of the few benchmarks that benefit from Directives, in which we used a per thread array that effectively privatized the data before each FFT iteration.

LU (lu): In LU, threads modify the input matrix in-place with complex strides. Our methods do not show significant improvement except for reducing some access tracking that happens in single threaded mode. The benefit comes from detecting all the memory initialization operations as non-communicating, which account for over 80% of the stores.

raytrace (rt): Single-TS reduced 61% of total read and 49% of total write instrumentations. With $Alias$, reductions were 98% for read and 57% for write instrumentation. Many major functions are invoked from multiple locations with different arguments, and $Alias$ used these differences in improving the results. Disjoint-TS identified 2 sections within the multithreaded program segment, but this only helped marginally.

radix (rd): rd performs in-place sorting of the keys. We see some benefit from avoiding instrumentation during initialization, but very little benefit from applying SBPA specific analysis on the multithreaded sections. Radix is one of the worst benchmarks. Nevertheless, SBPA still proved 40% of the memory accesses as non-communicating.

ocean_cp (oc): Loads only benefited from $Alias$. Three main functions perform most of the memory accesses. $Alias$ unaliased some of the points-to sets due to calls to some major functions with different arguments. Some memory regions become read-only, thus increasing the number of non-communicating loads. Stores mainly benefited from Single-TS analysis.

ocean_ncp (on): In this case, the threads access global memory in such a manner that our methods do not find significant optimization opportunities. Our methods still remove approximately 8% more read accesses than the baseline. The additional reduction in store instrumentation over baseline was negligible.

histogram (hi): The threads read values from global data, and increment counters each time they read a value. Histogram uses a large memory malloc for a single array and passes a section to each thread, which creates aliasing among the memory operations. We modified the memory allocation to use per-thread malloced memory. This improved the aliasing, resulting in nearly 100% non-communicating memory accesses.

kmeans (km): km has a very high load-to-store ratio, stores being only a small fraction of total accesses. It also uses a global array of points, so SBPA can determine several accesses as read-only. Due to other interfering optimizations, such as loop trip optimization that replicates loops with branches to handle trip counts of 0, SBPA algorithm is not able to identify the most precise points where the multi-threaded regions begin or end. Still, it is able to remove a significant portion of instrumentation calls.

matrix_multiply (mm): This is a classic case of multiple threads accessing large portions of read-only memory, where the input matrices are not modified at all during the multiplication process. The problem in this benchmark is that a single large array is created for all the threads. When the input matrices were marked read-only (MTROM), SBPA reduced 100% of the load instrumentations, but it still instrumented all the multithreaded store operations.

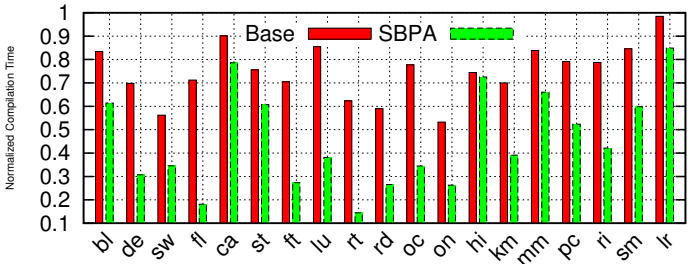

Fig. 10: Compilation times of Base and SBPA normalized to compile times when no optimization is applied. On average, Base was 75% and SBPA was 46% of unoptimized compile times.

pca (pc): This program has two separate threaded regions. In the mean computation region, it is reading matrix data and writing only the means of each row. In the subsequent threaded region to compute covariance, mean is only read. Hence, in the second threaded section, both the matrix data and mean data are treated as read-only, and do not need instrumentation.
reverse_index (ri): In this case, large input data set is modified in-place, to NULL-terminate the end of http links embedded in it. This rendered its memory read-write in multi-threaded mode, although in principle it is only used as input. When the link end point information was annotated separately using less than 10 lines of code change, this large input data became read-only in each Disjoint-TS. Then, SBPA was able to identify nearly all the memory accesses as non-communicating.
string match (sm): In this case, most of the stores were log-only. So, for data-race detection, nearly 100% of the memory accesses do not need to be checked. For STM-like systems, 60% of the memory accesses were still identified as non-communicating (Figure 4).
linear regression (lr): In this case, there are successive loads of the fields of the same structure. As a result, the state-of-the-art (Base) optimizations could reduce load tracking significantly. SBPA is able to detect all the reads as read-only, marking 100% of the loads as non-communicating, with an overall 62% reduction in aggregate access instrumentation. There is a significant amount of stores in this case, which results in smaller reduction in total access tracking compared to load tracking reduction.

6.5. Compilation Overhead

In this section we show that SBPA, incorporated as optimization passes in LLVM compiler, is not computationally expensive, which makes it quite practical. SBPA uses the Clang front end to create LLVM byte codes. These byte codes are optimized and linked to create one final byte code, which is converted to machine code executable. SBPA pass is run as part of *opt* after all the input files have been combined into a single byte-code file. The instrumentation functions are inlined for faster execution. These steps are very similar to those used by CoreDet [Bergan et al. 2010].

Figure 10 shows the compile times of Base and SBPA normalized against compile times without SBPA or Base. SBPA compilation for STM systems is between 15% and 90% faster than the same STM compiler pass without any optimization. In the benchmark with the highest overhead (`on`), the SBPA optimization pass accounts for less than 1% of the compilation time. For most of the benchmarks, it is less than 0.3% overhead. So, SBPA pass is runtime efficient, as even with a small overhead of its own, it reduces overall compilation times for STMs and other deterministic engines, primarily because the instrumentation overhead for compiler is reduced significantly.

Base optimization also reduces compilation time over unoptimized version. The extra computation time required for Base and SBPA is significantly less than the extra cost incurred in instrumenting the additional loads and stores. The second bar shows that SBPA reduces compile times significantly more than Base for every case.

For ThreadSanitizer experiments, we used a slightly different flow as described in Section 6.3. In this case, SBPA removes the unnecessary instrumentations already inserted by ThreadSanitizer. ThreadSanitizer instrumentation calls are external library functions and not inlined, so SBPA adds some compile time overhead in this flow. In our experiments, we saw a maximum slowdown of 42% in compile time compared to that of ThreadSanitizer pass. Still, for many benchmarks we observed shorter compile times due to reduced linking times. The geometric mean slowdown was 1%. We believe that a small slowdown is acceptable, considering how much runtime can be saved, particularly when many tests are run with the same executable.

7. RELATED WORK

The most closely related work is Choi et al. [2002], which uses static analysis to identify potentially racy accesses and instruments these accesses for use with a dynamic race detector. An earlier paper, Choi et al. [1999], describes a reachability based program analysis technique to identify objects that can be stack allocated or that are accessed by a single thread only, where unnecessary synchronization can be removed. In both cases, the techniques presented targeted Java applications and, in contrast to our work, do not consider barrier or join operations during the static analysis phase.

In the rest of this section we compare our SBPA technique with related work in the fields of static analysis (Section 7.1), dynamic race detection (Section 7.2), deterministic execution (Section 7.3), and STM (Section 7.4).

7.1. Static Analysis

The methods used in our paper have a solid foundation in static program analysis, a field that has matured with decades of research. It covers multiple techniques such as control and data flow analysis, constraint based analysis, concurrency analysis, type and effect systems, and many others. These techniques can have a variety of objectives, such as code optimization, static race detection, and correction checks to name a few.

Naik et al. [2006] developed *Chord* which uses escape and alias analysis techniques similar to the ones we used, but that focus on lock-protected regions. Naik and Aiken [2007] extended this work by using *conditional must not aliasing* to improve the alias analysis of pairs of objects in static race detection. We implemented and evaluated a similar algorithm (see Section 6.3) but saw negligible performance improvement in the benchmarks we evaluated, because there were relatively few memory accesses in lock protected regions. We included this in our baseline, and SBPA extends these techniques by considering program sections.

Von Praun and Gross [2003] proposed *object use graphs* to detect potential sharing between threads. Their technique has many similarities to SBPA, and in fact we adopt the terminology of *conflicting* and *non-conflicting* memory accesses from them. However, their technique uses a symbolic execution phase to classify memory accesses. SBPA does not use symbolic execution, and uses a different algorithm to associate memory accesses with program sections.

RacerX [Engler and Ashcraft 2003] is another static race detector that uses lockset analysis together with flow sensitive interprocedural analysis. Its goal is to effectively analyze code that cannot easily be run with a dynamic race detector (such as OS kernel and driver code), and it focuses on lock protected regions. While there are some similarities to SBPA, our goal is to reduce instrumentation for dynamic runtime systems, and our algorithms focus on identifying code section differentiated by thread create/join directives and barrier.

Aldrich et al. [1999] present a static analysis technique to reduce synchronization overhead in Java programs by identifying cases where a monitor is nested, entered multiple times by a single thread, or accesses only by one thread. The purpose of the analysis is similar to SBPA, but the implementation is very different. They track locksets for objects while SBPA divides the application into thread sections for tracking of memory accesses.

Locksmith [Pratikakis et al. 2011] is another static race detector that analyzes locks, uses fork-join and mod-ref of memory locations to avoid instrumenting in single threaded code sections, and also includes some field sensitivity. However, SBPA further refines the contexts to global barriers, and our algorithm for detecting sections differs from the one used in Locksmith.

Fang et al. [2003] and Kuperstein et al. [] describe memory fence insertion and optimization algorithms to guarantee sequential consistency. In many ways their goal is the converse of ours, because they analyze programs in order to determine when to divide it into sections using fences, while with SBPA we remove unnecessary instrumentation based on how the program is already divided into sections.

Agarwal et al. [2007] introduce *may happen in parallel* analysis for X10, which is applicable to languages that adopt concepts of places, async, finish, and atomic sections from X10. Although there are some similarities, their analysis is path sensitive while SBPA is path insensitive. SBPA also targets a different programming model.

7.2. Dynamic Race Detection

There is also a vast body of literature and tools covering dynamic race detection. One of the seminal papers in this field, Lamport [1978], described a simple model to represent the causal relationships and developed the notion of *happens before* among pairs of events. Most dynamic race detectors are based on this notion, and check that conflicting memory accesses are separated by a synchronization event.

Savage et al. [1997] proposed Eraser, a tool that detects data races dynamically in lock-based programs. FastTrack [Flanagan and Freund 2009] is an optimized version of a precise dynamic race detector using vector clocks to track the *happens before* relationships. Ponzniansky and Schuster [2003] developed a dynamic data race detector for C++ programs called Djit+. In theory, SBPA techniques could be integrated with any of these dynamic race detectors to reduce instrumentation overhead.

ThreadSanitizer [Serebryany et al. 2012] is a commercially deployed dynamic race detector, developed at Google, and integrated with Clang and LLVM. We integrated SBPA with ThreadSanitizer and presented our results in Section 6.3.

Raman et al. [2010; 2012] developed a dynamic data race detector for use with structured parallelism in languages such as X10. They implement some of the same optimization as SBPA, such as eliminating checks in sequential regions of code and checks for read-only data in parallel code sections. However, the algorithms differ from those used in SBPA because their work is intended for structured parallelism.

Effinger-Dean et al. [2012] proposed using *interference-free regions* to reduce the overhead of dynamic race detection. This work shares SBPA's goal of reducing instrumentation by segmenting an application, but takes a different approach to achieving this. Mellor-Crummey [1993] developed one of the first dynamic race detection tools that also used static analysis to reduce instrumentation overhead. In that sense, it is similar to SBPA; but their tool was intended for Fortran, and the design and implementation differ from those of SBPA. RedCard [Flanagan and Freund 2013] describes eliminating redundant access tracking in dynamic race detection using static analysis. SBPA does not focus on redundant accesses; instead it eliminates conflict free accesses. Their method is orthogonal to what is proposed in this paper.

7.3. Deterministic Execution

Recent work in the field of deterministic execution has seen the development of a number of software runtime systems which enforce deterministic execution of shared memory multithreaded programs. SBPA was originally developed to improve the performance of a deterministic execution system we worked on, and it can be applied to other systems. Kendo [Olszewski et al. 2009] is a system that achieves weak determinism through deterministic locking among multiple threads. CoreDet [Bergan et al. 2010] is a compiler and runtime environment that together enforces determinism. We use the static analysis techniques included in CoreDet as our baseline and extend them with SBPA to reduce instrumentation overhead. DThreads [Liu et al. 2011] is a deterministic execution system, where threads are implemented as processes and OS memory protection enforces isolation between threads; as a result, it does not use compiler instrumentation. Bond et al. [2013] developed OCTET, a framework for controlling cross-thread dependencies that allows for optimization of dynamic analyses. However, OCTET is largely dynamic while SBPA is largely static.

7.4. Software Transactional Memory

There is also prior work to reduce STM overhead, mainly for strongly typed languages. Beckman et al. [2009] describes object ownership in Java to reduce logging operations in STM. Blanchet [1999] describes escape analysis for object oriented languages like Java. Li and Verbugge [2005] describe similar work on *May Happen In Parallel*, for Java, but it performs flow sensitive analysis of locks and synchronizations, which is runtime intensive. Our analysis does not rely on locks, and uses a different approach of program segmentation orthogonal to theirs. SBPA can complement synchronization detection mechanisms such as Xiong et al. [2010] and Tian et al. [2008], but currently does not use such methods and relies on global barriers only. Yehuda [2011] describes static analysis techniques to reduce STM overhead that are similar to ones implemented in CoreDet and used in our baseline system.

8. CONCLUSION

We conclude that a section-based static analysis of programs can significantly reduce the runtime overhead for race detection, STMs, and other systems that detect unsynchronized memory accesses in multithreaded programs, at very low compile time overheads. The most effective technique was Single-TS, which identified single-threaded sections of code in a multithreaded application. Overall, SBPA eliminated a much higher percentage of load and store instrumentations compared to the baseline. Our ThreadSanitizer results show that SBPA can be used to improve performance of tools such as data-race detectors, STM, and deterministic execution systems.

In some cases, the fact that Single-TS removed nearly all of the instrumentation meant that there was no opportunity to demonstrate the potential of Disjoint-TS. Additionally, limitations in the pointer analysis prevented the compiler from proving that some instrumentations could be removed. We also made improvements to alias analysis that increased the effectiveness of the SBPA pass. However there is more which can be done, such as applying range analysis to partitions points-to sets for which the range of addresses accessed by different threads is disjoint. This is a topic of future work for us. From our experiments, we believe the solution to concurrency related issues lies in a multifaceted approach of static and dynamic analyses, memory modeling, and more appropriate programming models.

Acknowledgments

We thank the reviewers for their feedback. This work was supported in part by NSF grants 1337278 and 1318943. Opinions, findings and conclusions expressed herein are those of the authors and do not reflect the views of the NSF.

REFERENCES

Yehuda Afek, Guy Korland, and Arie Zilberstein. 2011. Lowering STM overhead with static analysis. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing (LCPC'10)*. 31–45.

Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K Shyamasundar. 2007. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. 183–193.

Jonathan Aldrich, Craig Chambers, EminGun Sirer, and Susan Eggers. 1999. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis*. Lecture Notes in Computer Science, Vol. 1694. 19–38.

Nels E. Beckman, Yoon Phil Kim, Sven Stork, and Jonathan Aldrich. 2009. Reducing STM overhead with access permissions. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO '09)*. 2:1–2:10.

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. 53–64.

Christian Bienia. 2011. *Benchmarking modern multiprocessors*. Ph.D. Dissertation. Princeton University.

Bruno Blanchet. 1999. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the 14th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. 20–34.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*. 207–216.

Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. 2013. OCTET: Capturing and controlling cross-thread dependences efficiently. In *Proceedings of the 2013 International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '13)*. 693–712.

Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software transactional memory: Why is it only a research toy? *Commun. ACM* 51, 11 (Nov. 2008), 40–46.

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *Proceedings of the 14th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. 1–19.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. 258–269.

Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: Interference-free regions for dynamic data-race detection. In *Proceedings of the 2012 International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '12)*. 467–484.

Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP '03)*. 237–252.

Xing Fang, Jaejin Lee, and Samuel P. Midkiff. 2003. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. 285–294.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI '09)*. 121–133.

Cormac Flanagan and Stephen N. Freund. 2013. RedCard: Redundant check elimination for dynamic race detectors. In *Proceedings of the 27th European Conference on Object-Oriented Programming*. 255–280.

Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (FMCAD '10)*. 111–120.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565.

William Landi, Barbara G. Ryder, and Sean Zhang. 1993. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. 56–67.

Chris Lattner and Vikram Adve. 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI '05)*. 129–142.

Lin Li and Clark Verbrugge. 2005. A practical MHP information analysis for concurrent Java programs. In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing (LCPC'04)*. 194–208.

Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient deterministic multithreading. In *Proceedings of the 23rd Symposium on Operating Systems Principles (SOSP '11)*. 327–336.

John Mellor-Crummey. 1993. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proceedings of the 1993 Workshop on Parallel and Distributed Debugging (PADD '93)*. 129–139.

Mayur Naik and Alex Aiken. 2007. Conditional must not aliasing for static race detection. In *Proceedings of the 34th Annual Symposium on Principles of Programming Languages (POPL '07)*. 327–338.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 2006 Conference on Programming Language Design and Implementation (PLDI '06)*. 308–319.

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. 97–108.

Eli Pozniansky and Assaf Schuster. 2003. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the 9th Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. 179–190.

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1 (Jan. 2011), 3:1–3:55.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient data race detection for async-finish parallelism. In *Proceedings of the 1st International Conference on Runtime Verification (RV'10)*. 368–383.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation (PLDI '12)*. 531–542.

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. 13–24.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP '97)*. 27–37.

Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2012. Dynamic race detection with LLVM compiler. In *Proceedings of the 2nd International Conference on Runtime Verification (RV'11)*. 110–114.

Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *Proceedings of the 14th Annual Symposium on Principles of Distributed Computing (PODC '95)*. 204–213.

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL '96)*. 32–41.

Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. 2008. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. 143–154.

Christoph von Praun and Thomas R. Gross. 2003. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. 115–128.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*. 24–36.

Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad hoc synchronization considered harmful. In *Proceedings of the 9th Conference on Operating Systems Design and Implementation (OSDI'10)*.

Sean Zhang, Barbara G. Ryder, and William Landi. 1996. Program decomposition for pointer aliasing: A step toward practical analyses. In *Proceedings of the 4th Symposium on Foundations of Software Engineering (SIGSOFT '96)*. 81–92.