Automatic Atomic Region Identification in Shared Memory SPMD Programs *

Gautam Upadhyaya, Samuel P. Midkiff and Vijay S. Pai
Purdue University
{gupadhya, smidkiff, vpai}@purdue.edu

Abstract

This paper presents *TransFinder*, a compile-time tool that automatically determines which statements of an unsynchronized multithreaded program must be enclosed in atomic regions to enforce *conflict-serializability*. Unlike previous tools, TransFinder requires no programmer input (beyond the program) and is more efficient in both time and space.

Our implementation shows that the generated atomic regions range from being identical to, or smaller than, the programmer-specified transactions in the three Java Grande benchmarks considered, and in five of the eight STAMP benchmarks considered, while still providing identical synchronization semantics and results. The generated atomic regions are between 5 and 38 lines larger in the three remaining STAMP benchmarks. In the most conservative case, TransFinder can, based on the program structure, successfully identify and suggest an alternative that conforms exactly to the programmer-specified atomic regions. By generating small, highly-targeted, conflict-serializable atomic regions, TransFinder allows the programmer to focus further tuning efforts on only a small portion of the code (when further tuning is needed).

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

General Terms Algorithms, Performance, Design

Keywords Automatic transactional region identification, conflict-serializability, parallel programming

1. Introduction

A major difficulty when programming shared memory parallel machines is identifying shared variables (whose storage is accessed by many threads), and controlling the access to those variables across the threads. When writing a parallel program, the programmer must both determine the shared variables and analyze the interactions among them to determine when sequences of shared memory accesses must execute as a single atomic region, when they may execute as a sequence of atomic regions, or when some (or all) of them can execute outside of any atomic region. Once the programmer identifies the code's atomic regions, they may be enforced using either transactions [18–20, 26, 33, 34], user-generated locks, or locks automatically generated from user-identified atomic regions or transactions [7, 14, 17, 21, 25, 35, 41].

All of these techniques require the programmer to understand the interactions between different threads and the flow of data across threads. Errors in generating atomic sections and locks can lead to non-determinate bugs that are extremely difficult to correct. This paper describes the TransFinder tool that attacks these fundamental problems head-on. TransFinder assumes that programmers desire operations on shared data within a thread to be performed without interference from other threads; that is, without the values of the shared data being read or written by other threads while the operations are performed. Using compiler analyses, some of which are inspired by concepts from database theory, TransFinder automatically identifies regions in Java programs that need to be executed as atomic regions to enforce the semantics implied by the above constraint. TransFinder produces as output a C++ program with the atomic regions marked. Unlike the only other known attempt at automatically identifying atomic regions in programs [37, 38], TransFinder does not require user annotations of the code. This makes TransFinder easy to use. The close match of atomic regions identified by TransFinder, and hand-inserted atomic regions in our benchmark programs indicates that the semantics TransFinder ascribes to multithreaded programs are valid over the benchmarks examined, and useful.

The primary goal of TransFinder is to suggest atomic regions to a programmer, and allow the programmer to determine the final scope of the atomic regions. It does this by analyzing cycles of conflicting operations and encapsulating those cycles in atomic regions. In most cases studied, however, the atomic regions specified by TransFinder are close enough to hand generated atomic regions to be used unchanged. For certain program control flow structures — loops in particular — TransFinder can suggest alternative atomic regions in cases where the first choice suggested by TransFinder is too conservative. In one additional bench-

*This work is supported in part by the National Science Foundation under Grant Nos. CCF-0532448, CNS-072212,and CNS-0751153.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

mark, this alternative was close enough to hand-generated code to be used unchanged. Moreover, it is always the case that the atomic regions suggested by TransFinder allow the programmer to focus on a small subset of the program (less than 7% of the code in the benchmarks studied) to evaluate the atomic regions. TransFinder is conservative in that the atomic regions specified include all variable references that *might* need to be in some atomic region.

We use programs from two benchmarks suites: we use the three multithreaded full-scale applications from the Java Grande Benchmarks [15] and the STAMP benchmarks [6]. All of these benchmarks are characterized by having a single thread type, of which multiple instances simultaneously execute. Following the example of [22] we refer to these as *SPMD (Single Program, Multiple Data)* programs. Our analysis can be applied to programs with multiple thread types.

Finally, TransFinder does not check for code within transactions that might lead to deadlocks or similar behaviors as discussed in [39, 42].

This paper's contributions are as follows:

- It describes the TransFinder system that automatically detects atomic regions in shared memory SPMD programs;
- It describes the correctness criteria used by TransFinder, and its relationship to conflict-serializability;
- It describes the analyses used by TransFinder to enforce these correctness criteria;
- It provides experimental data showing how close TransFinder comes to hand-generated atomic regions in the three full-scale applications among the multithreaded Java Grande benchmarks [15] and in the STAMP benchmarks [6]. It does this by both comparing lines of code contained within atomic regions, and by measuring the performance of programs using hand-coded and automatically-generated atomic regions. Experimental results are provided for executions using versions of the programs where the atomic regions are enforced using software transactional memory.
- It discusses extensions of the TransFinder technique to non-SPMD shared memory programs.

The rest of the paper is organized as follows. Section 2 provides an overview of our approach. Section 3 describes the theoretical underpinnings of our approach and the analyses we use. Section 4 describes the implementation of these principles in our system. Section 5 describes our evaluation using the STAMP and Java Grande benchmark suites. Sections 6 and 7 describe related work and conclusions. Appendix A contains a detailed example.

2. Overview

In this section we introduce, at a fairly high level, our approach to identifying atomic regions in shared memory programs. We introduce the concepts and some useful definitions, leaving the more formal analysis for Section 3.

Our analysis is built on the notion of *serializability*. Serializability is a universally recognized correctness criterion that has its roots in database theory. To motivate the rest of this paper, we briefly explain the necessary definitions here. A fuller exploration of some of these topics can be found in [13].

A trace of a program involving multiple transactions is called a *schedule*. A schedule is said to be *serial* if statements in an individual transaction execute one after the other, with no interference from any statements in other transactions. A schedule is *serializable* if it is equivalent to a serial schedule (the resultant database state is the same as that of some serial schedule). Operations in different transactions are said to *conflict* if they access the same memory location(s), and at least one of the operations is a write.

Broadly speaking, there are two widely-used definitions of serializability: *view-serializability* and *conflict-serializability*. View-serializability matches the general definition of serializability above. Conflict-serializability is a conservative approximation to serializability that focuses on the respective order of conflicting operations. In particular, a schedule is conflict-serializable if the respective order of mutually conflicting operations is the same as that of some serial schedule (non-conflicting operations may be reordered). We note that conflict-serializability is more restrictive than serializability, in that any conflict-serializable schedule is also serializable, but the converse does not necessarily hold. In the rest of the paper, we will use the terms serializable and conflict-serializable interchangeably.

For a schedule involving multiple transactions to be conflict-serializable, therefore, only non-conflicting operations may be interleaved (here we consider an operation in its entirety, and do not break it down into its constituent reads and writes). This leads to the following insight:

> If two transactions have multiple operations on a given (shared) memory location, and if these operations mutually conflict, then all of the operations within a given transaction which access that location must be allowed to execute without the possibility of an interleaving from statements in other transactions.

Intuitively, if such an interleaving were to be allowed to exist (which ordered conflicting operations in a certain manner), then it would also be possible for an alternate interleaving to exist which ordered these operations in a different manner. The end result would, by definition, not be conflict-serializable. Adding synchronization constructs eliminates the possibility of such an interleaving, and forces a conflict-serializable schedule.

We note that conflict-serializability generally conforms to a user's definition of correctness: a sequence of operations on shared storage within a thread generally should appear to complete as if the sequence executed without interference from other threads. Thus, shared variable state after such a sequence should be the same as if there was no interference from other threads.

When analyzing multithreaded shared memory programs, the TransFinder compile-time tool initially considers each possible runtime thread to correspond to a single, large transaction. In other words, the TransFinder tool assumes the transaction is the entire thread. With these semantics, and given our discussion so far, any interleavings of conflicting operations can lead to non-conflict-serializable execution runs. Clearly such interleavings may be avoided by forc-

Thread 1	Thread 2
x = 1	x = 2
y = x	y = x

(a) Program Fragment

Thread 1	Thread 2
x = 1	
	x = 2
y = x	
	y = x

(b) Invalid Interleaving

$write\ x_{T_1} \succ write\ x_{T_2} \succ (read\ x_{T_1}, write\ y_{T_1}) \succ (read\ x_{T_2}, write\ y_{T_2})$

(c) Order of conflicting operations ($S_i \succ S_j$ denotes S_i executes before S_j)

Thread 1	Thread 2
begin_atomic()	**begin_atomic()**
x = 1	x = 2
y = x	y = x
end_atomic()	**end_atomic()**

(d) Synchronized Code

Thread 1	Thread 2
x = 1	
y = x	
	x = 2
	y = x

(e) Serializable Schedule

$write\ x_{T_1} \succ (read\ x_{T_1}, write\ y_{T_1}) \succ write\ x_{T_2} \succ (read\ x_{T_2}, write\ y_{T_2})$

(f) Order of conflicting operations ($S_i \succ S_j$ denotes S_i executes before S_j)

Figure 1. Valid and invalid interleavings: x and y are variables shared by both threads.

ing the threads to execute sequentially. However, such an arrangement would defeat the purpose of a multithreaded program. In this work, our purpose is to protect against only those interleavings which would lead to non-conflict-serializable schedules. To do so, we first identify conflicting operations, and then determine which interleavings of those operations are disallowed. We then prevent the possibility of such interleavings by synchronizing only the offending blocks of code via atomic regions, and thus enforce conflict-serializability.

Consider the example of Figure 1, where two threads attempt to first write to a (shared) variable x, and then use the value of x to update a different shared variable y. The input to TransFinder consists of program source code with no synchronization constructs. Executing such code could lead to any and all interleavings, including those that may lead to non-conflict-serializable schedules. Figure 1(b) shows one such invalid interleaving of the statements in the program fragment from Figure 1(a). The order of conflicting operations in the schedule is shown in Figure 1(c). Note that this order is inconsistent with that of the serial schedule from Figure 1(e), (shown in Figure 1(f)). In fact, the ordering shown in Figure 1(c) is inconsistent with that of *any* serial schedule. The invalid interleaving of Figure 1(b) can be prohibited by enclosing the conflicting operations within an atomic block, as shown in Figure 1(d). Doing so leads to conflict-serializable schedules.

Intuitively, when the threads have finished executing the program fragments shown, it might reasonably be expected that the values assigned to x and y have been assigned in a single thread. Enclosing the relevant operations in atomic regions ensures this.

We now formalize the criteria we use for enclosing conflicting references within an atomic region.

DEFINITION 1. *Two threads, T_i, T_j have a non-serializable execution run if there exists some interleaving of the individual memory operations of the two threads which is not conflict-serializable.*

Again, Figure 1(a) gives an example code fragment and Figure 1(b) shows one possible invalid interleaving. The interleaving is possible because of a lack of synchronization on the two globally-scoped variables, x and y. Inserting synchronization (Figure 1(d)) leads to a serializable schedule (Figure 1(e)).

Conflict-serializability can be tested using a *precedence graph*. A precedence graph of a schedule is a directed graph with a node for each (completed) atomic region in the schedule. An edge exists between two transactions, X_i and X_j if an action of X_i precedes and conflicts with an action of X_j. The following theorem illustrates the use of precedence graphs to test the conflict-serializability of a schedule:

THEOREM 1. *A schedule is conflict-serializable if, and only if, its precedence graph is acyclic.*

This theorem is a fundamental and well-known result [13].

Figure 2 shows the precedence graph for the schedule shown in Figure 1(b). An edge exists from the node for Thread 1 to the node for Thread 2 because the update of the shared variable x occurs in Thread 1 before it occurs in Thread 2. In addition, an edge exists from Thread 2 to Thread 1 because the update of x by Thread 2 precedes and conflicts with the read of x during the assignment to the shared variable y by Thread 1. Therefore, a cycle exists, and the schedule shown in Figure 1(b) is not conflict-serializable.

Thus, detection of non-serializable schedules is a matter of cycle detection in the precedence graphs of shared memory parallel programs. Notice that the cycles in the graph of Figure 2 could have been avoided by forcing one thread to execute the relevant conflicting instructions in toto, without any possibility of interference from the other thread. Doing so forces a total ordering of the instructions in the program fragment, and therefore disallows cycle formation. Figure 1(d) shows correctly synchronized code (through the use of an atomic region), and Figure 1(e) shows one possible resultant schedule.

To summarize, we have the following observations:

- Cycles in a precedence graph of a schedule indicate a lack of conflict-serializability.

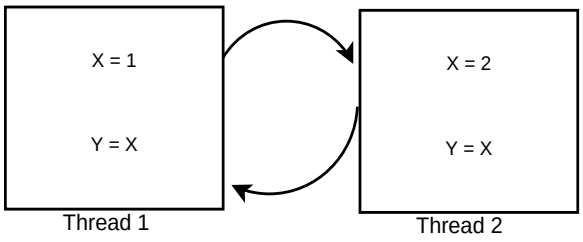


Figure 2. The precedence graph of the schedule shown in Figure 1(b). The cycle indicates a non-conflict-serializable schedule.

- These cycles arise because conflicting operations in different threads may be interleaved with each other.
- Breaking these cycles (and therefore, ensuring conflict-serializable execution runs) involves synchronizing code to ensure that no such interleaving is possible.

In Section 3, we will demonstrate how to detect such *conflict cycles* in multithreaded shared memory programs. In addition, we will show how the cycle-detection process may be optimized for a certain class of shared memory programs, namely SPMD programs.

3. Serializability Analysis

This section defines the notion of *conflict interleavings* and demonstrates their equivalence to cycles in the precedence graph of a program. This section also shows how such conflict interleavings can be avoided through the proper application of synchronization constructs, providing conflict-serializability.

Our analysis makes the following assumptions: (a) that the semantics of the program allow us to treat every thread as a single transaction (as described in Section 2), and (b) that every access to a shared memory location is atomic (we enforce this assumption in our implementation). Section 2 showed how conflict cycles indicate a lack of conflict-serializability. Notice that enforcing the atomicity of the individual memory references does not protect against such cycles, because such instruction-level atomicity enforcement does not necessarily prevent interleavings of conflicting operations among different threads. It is only when multiple operations are synchronized that the edges in the precedence graph are unidirectional. Accordingly, this work aims to detect and protect against such illegal interleavings by enlarging the scope of atomic regions.

3.1 Conflict Interleavings and Serializability

We start with several key definitions:

DEFINITION 2. *Two statements* conflict *if they access the same (shared) memory location and at least one of the accesses is a write. The conflict relation* S_x conflicts with S_y *is written:* $S_x \rightleftharpoons S_y$.

Note that the conflict relation is symmetric, i.e if $S_x \rightleftharpoons S_y$ then $S_y \rightleftharpoons S_x$.

DEFINITION 3. *A statement* S_x may precede *another statement* S_y *if it is possible for* S_x *to execute before* S_y *in some execution of the program. The precedence relationship* S_x may precede S_y *is written:* $S_x \preceq S_y$. *If the statements may execute in parallel, then* $S_x \preceq S_y$ *and* $S_y \preceq S_x$ *and is written* $S_x \prec\succ S_y$.

DEFINITION 4. *Two threads,* T_i, T_j *have an* interleaving *if there exist statements* $S_{i,x}, S_{i,z} \in T_i$ *and* $S_{j,y} \in T_j$ *such that* $S_{i,x} \preceq S_{i,z}$, $S_{i,x} \preceq S_{j,y}$ *and* $S_{j,y} \preceq S_{i,z}$. *The thread interleaving relation is written:* $T_i \prec\succ T_j$.

Given these definitions, we define a *Conflict Interleaving (CI)* as follows:

DEFINITION 5. *Threads* $T_1, T_2, \ldots, T_N$ *have a Conflict Interleaving (CI) if the following condition holds:*

- *Let* $S_i \equiv (S_{i,1}, S_{i,2}, \ldots, S_{i,m_i})$ *be the set of statements of each thread* T_i, $1 \leq i \leq N$, *and let* $\{S_{i,j} \preceq S_{i,(j+1)}\}_{j=1}^{j=(m_i-1)} \forall S_i$. *Then*

$$\Big\{(S_{1,a} \preceq S_{2,b}), (S_{1,a} \rightleftharpoons S_{2,b}),$$
$$(S_{2,c} \preceq S_{3,d}), (S_{2,c} \rightleftharpoons S_{3,d}) | c \geq b,$$
$$\ldots\ldots$$
$$(S_{N,y} \preceq S_{1,z}), (S_{N,y} \rightleftharpoons S_{1,z}) | z > a\Big\}$$

The Conflict Interleaving relationship is written as: $T_1 \prec\rightleftharpoons\succ T_2 \ldots \prec\rightleftharpoons\succ T_N$

Intuitively, threads have a conflict interleaving if there exists some interleaving of the threads involving mutually conflicting operations.

We next explore the role of conflict interleavings in detecting non-conflict-serializable schedules. We then discuss how to detect conflict interleavings in shared memory programs, with a special emphasis on SPMD programs — that is, programs where all threads are of the same type. Finally, we show how to insert atomic regions into SPMD programs by eliminating conflict interleavings.

The following theorems illustrate how conflict interleavings may be used to detect non-conflict-serializable executions.

THEOREM 2. *In a shared memory program, any conflict interleaving will result in a non-conflict-serializable execution.*

Proof: Let the CI be between threads $T_1, T_2 \ldots T_N$, and let $S_i \equiv (S_{i,1}, S_{i,2}, \ldots, S_{i,m_i})$ be the set of statements of each thread T_i, where $\{S_{i,j} \preceq S_{i,(j+1)}\}_{j=1}^{j=(m_i-1)} \forall S_i$. Then, from Definition 5, we have: $(S_{1,a} \preceq S_{2,b})$, $(S_{1,a} \rightleftharpoons S_{2,b})$ for some $S_{1,a} \in S_1$, $S_{2,b} \in S_2$. In the precedence graph of the program, therefore, an edge exists from the vertex corresponding to thread T_1, to the vertex corresponding to thread T_2. Similarly, $(S_{2,c} \preceq S_{3,d})$, $(S_{2,c} \rightleftharpoons S_{3,d})$ for some $S_{2,c} \in S_2$, $S_{3,d} \in S_3$, leading to an edge from T_2 to T_3, and so on for all N threads. Finally, we have: $(S_{N,y} \preceq S_{1,z})$, $(S_{N,y} \rightleftharpoons S_{1,z})$, leading to an edge from T_N to T_1, thus completing the cycle. From Theorem 1, therefore, the schedule is non-serializable.

THEOREM 3. *If an execution run is non-serializable then there exists at least one conflict interleaving.*

Proof: Let threads $T_1, T_2, \ldots, T_N$ have a non-serializable run, and let $S_i \equiv (S_{i,1}, S_{i,2}, \ldots, S_{i,m_i})$ be the set of statements of each thread T_i, where $\{S_{i,j} \preceq S_{i,(j+1)}\}_{j=1}^{j=(m_i-1)} \forall S_i$. Because the execution is non-serializable, there must exist a cycle in the precedence graph of the program. Without loss of generality, assume the path traced by the cycle is $[T_1, T_2, \ldots, T_N, T_1]$ i.e., an edge exists from the vertex corresponding to any thread T_i, to the vertex corresponding to thread T_{i+1}. Furthermore, let there be statements $(S_{1,a}, S_{1,z}) \in S_1, (S_{2,b}, S_{2,c}) \in S_2, \ldots, S_{N,y} \in S_N$. Because an edge exists from T_1 to T_2, we have $(S_{1,a} \preceq S_{2,b})$, and because the only edges allowed in the precedence graph are conflict edges, we have $(S_{1,a} \rightleftharpoons S_{2,b})$. Similarly, the edge from T_2 to T_3 gives us: $(S_{2,c} \preceq S_{3,d})$, $(S_{2,c} \rightleftharpoons S_{3,d})$, and so on for the other threads. And finally, the edge from T_N to T_1 gives us: $(S_{N,y} \preceq S_{1,z})$, $(S_{N,y} \rightleftharpoons S_{1,z})$ (here we assume, without loss of generality, that $S_{1,a} \preceq S_{1,z}$). From Definition 5, therefore, we have a conflict interleaving.

THEOREM 4. *Threads, $T_1, T_2, \ldots, T_N$ have a non-serializable execution run if, and only if, there is a conflict interleaving between them.*

Proof: Follows from Theorems 2 and 3.

COROLLARY 1. *Eliminating conflict interleavings in a shared memory program will result in a conflict-serializable execution.*

Proof: Follows from Theorem 4.

3.2 Detecting Conflict Interleavings in general shared memory programs

The previous section established the relationship between cycles in the precedence graph of a program, and conflict interleavings. The precedence graph is a runtime construct: it depicts precedence orders between *committed* transactions. As such, it is not directly usable in a static context because it is impossible, in general to determine a priori in what order the individual statements in a multithreaded program will execute. TransFinder, on the other hand, is a tool that, at compile-time, determines the synchronization constructs necessary to achieve conflict-serializability. In order to allow TransFinder to achieve its goal, we require a construct to statically represent the various possible (runtime) interleavings in the program. This section introduces the *Conflict Graph*, which is just such a construct.

We begin by stating some useful definitions:

DEFINITION 6. *A Concurrent Shared Basic Block (CSBB) is a basic block with the added constraint that there is at most one occurrence, within the block, of a variable that references shared memory locations.*

We note that the shared variable within a CSBB may, over time, reference different memory locations.

We define conflicting CSBBs as follows:

DEFINITION 7. *Two Concurrent Shared Basic Blocks* conflict *if they contain statements that conflict with one another. More precisely, let there be two CSBBs, B_i, B_j. Then $B_i \rightleftharpoons B_j$ if $\{\exists S_i \in B_i, S_j \in B_j | S_i \rightleftharpoons S_j\}$. Moreover, a concurrent shared basic block B_i* may precede *another, B_j, if any statement in B_i may precede any statement in B_j, i.e. $B_i \preceq B_j$ if $\{\exists S_i \in B_i, S_j \in B_j | S_i \preceq S_j\}$.*

```
class Thread
    int []data
    procedure Thread(int []d)
        // Thread constructor
        data = d
    end procedure
    procedure void run()
        // run function for the thread
        int x = d[0]
        d[0] = 3
        int y = d[1]
        ...
    end procedure
end class
...
procedure main()
    int []a = new int[10]
    Thread T1 = new Thread(a), T2 = new Thread(a)
    T1.run(); T2.run()
end procedure
```

Figure 3. A Typical SPMD program.

Note that we conservatively assume that if two CSBBs *may* conflict then they *do* conflict.

We rely on the conflict graph (CG) to identify CIs in multithreaded programs. The CG is closely related to the Concurrent Control Flow Graph (CCFG) of Lee et al. [23]. A CG is defined as follows:

DEFINITION 8. *A Conflict Graph (CG) is a directed graph $G = (V, E, Entry, Exit)$, where:*

1. *V is the set of vertices in G. Each vertex is a CSBB, denoted B.*
2. *$E = E_F \cup E_C$. E_F is the set of control flow edges, including transitive edges. E_C is the set of conflict edges. An undirected conflict edge exists between any two CSBBs that conflict. Only conflict edges may exist between CSBBs in different threads.*
3. Entry *is the CSBB through which all control flow enters the graph.*
4. Exit *is the CSBB through which all control flow leaves the graph.*

It is sometimes desirable to refer to the smallest subgraph of a CG that contains a specific thread. We denote the subgraph of some CG that contains the nodes of thread T_i as CG_i.

Figure 3 gives an example of a typical SPMD program, and Figure 4 shows the CG for the program of Figure 3. The *Thread Start* and *Thread Stop* nodes indicate thread initializations (one per outgoing edge) and destructions, respectively. *Thread run* and *Thread join* are special nodes which indicate where the thread starts and stops executing, respectively. In what follows, we will assume the existence of the *Entry* and *Exit* nodes and will generally omit mentioning them. Figure 11 in the appendix contains a more detailed example of a CG.

The CG can be analyzed to determine CIs by detecting cycles involving control flow edges and inter-thread conflict edges. In general, a conflict occurs because of multiple accesses to shared memory locations. These accesses are either

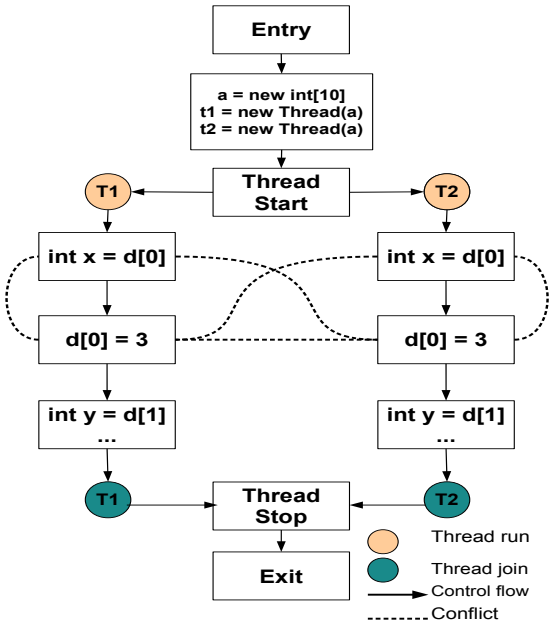


Figure 4. The Conflict Graph of the SPMD program of Figure 3.

read accesses or write accesses. We note that while individual CSBBs may contain a single access to any shared memory location, these accesses do not necessarily conflict with the accesses from other CSBBs. To detect conflicts between CSBBs, therefore, we must first detect the set of accesses to shared memory locations. The *read set* of a CSBB B_i in the conflict graph CG_i for thread T_i is the set of shared memory locations being read by the statements in B_i, and is denoted R_i. Similarly, the *write set*, W_i, is the set of shared memory locations being written by the statements in B_i. Two CSBBs, B_i and B_j, *conflict* if any of the following conditions hold: (a) $R_i \cap W_j \neq \emptyset$, (b) $W_i \cap R_j \neq \emptyset$ or (c) $W_i \cap W_j \neq \emptyset$. The *read-write set* of CSBB B_i is the set of shared memory locations being read from or written to by the statements in B_i, and is denoted RW_i where $RW_i = R_i \cup W_i$.

We are now ready to define a *conflict cycle*:

DEFINITION 9. *Let* $B_i \equiv (B_{i,1}, B_{i,2}, \ldots, B_{i,m_i}) \in CG_i$ *be the CSBBs of some thread* T_i*, and let* $\{(B_{i,j}, B_{i,(j+1)}) \in E_F\}_{j=1}^{j=(m_i-1)} \forall B_i$. *Then threads* $[T_1, T_2, \ldots, T_N]$ *have a conflict cycle if the following condition holds: there exists a path* $[B_{1,a}, B_{2,b}, B_{2,c}, B_{3,d}, \ldots, B_{N,y}, B_{1,z}]$ *such that:*

$$\Big\{(B_{1,a}, B_{2,b}) \in E_C, B_{1,a} \preceq\succeq B_{2,b}$$
$$(B_{2,c}, B_{3,d}) \in E_C | c \geq b, B_{2,c} \preceq\succeq B_{3,d},$$
$$\ldots\ldots$$
$$(B_{N,y}, B_{1,z}) \in E_C | z \geq a, B_{N,y} \preceq\succeq B_{1,z}\Big\}$$

Intuitively, a conflict cycle is a cycle in the CG involving both control flow and conflict edges.

THEOREM 5. *An inter-thread conflict cycle in a shared memory program indicates a CI.*

Proof: Let the inter-thread conflict cycle occur between threads $[T_1, T_2, \ldots, T_N]$. Let $B_i \equiv (B_{i,1}, B_{i,2}, \ldots, B_{i,m_i}) \in CG_i$ be the CSBBs of thread T_i, and let $\{(B_{i,j}, B_{i,(j+1)}) \in E_F\}_{j=1}^{j=(m_i-1)} \forall B_i$. From Definition 9, we know that $(B_{1,a}, B_{2,b}) \in E_C, B_{1_a} \preceq\succeq B_{2,b}$. But $(B_{1,a}, B_{2,b}) \in E_C$ if $B_{1,a} \rightleftharpoons B_{2,b}$ and, from Definition 3, $B_{1_a} \preceq\succeq B_{2,b}$ if $B_{1,a} \preceq B_{2,b}$ and $B_{1,a} \succeq B_{2,b}$. Similarly, $(B_{2,c}, B_{3,d}) \in E_C$ if $B_{2,c} \rightleftharpoons B_{3,d}$, and $B_{2,c} \preceq\succeq B_{3,d}$ if $B_{2,c} \preceq B_{3,d}$ and $B_{2,c} \succeq B_{3,d}$. Extending this to all N threads gives us:

$$(B_{1,a} \preceq B_{2,b}), (B_{1,a} \rightleftharpoons B_{2,b}),$$
$$(B_{2,c} \preceq B_{3,d}), (B_{2,c} \rightleftharpoons B_{3,d}) | c \geq b,$$
$$\ldots\ldots$$
$$(B_{N,y} \preceq B_{1,z}), (B_{N,y} \rightleftharpoons B_{1,z}) | z > a$$

which, from Definitions 5 and 7, is a conflict interleaving.

To detect conflict interleavings in multithreaded programs, then, we detect inter-thread conflict cycles in the conflict graph of the program.

3.3 Detecting Conflict Interleavings in SPMD programs

In Section 3.2, we noted the equivalence of inter-thread conflict cycles with conflict interleavings. Detecting and removing conflict interleavings is therefore a matter of detecting conflict cycles in the CG of the program. However, such cycle detection can be resource-intensive (common techniques for cycle detection are derived from Depth First Search (DFS) algorithms, which have a worst case performance of $O(|V|+|E|)$; for a large program with many conflicts, $E \rightarrow V^2$). It is possible to optimize this cycle detection process for SPMD programs. This is done by *projecting* the effects of every thread onto a single thread, and then identifying the Strongly Connected Components of the resultant conflict graph (the resulting time complexity is still $O(|V| + |E|)$, but the number of edges, $|E|$, is much smaller). To that end, we first define a Projection Conflict Graph (PCG) and then show how the PCG of an SPMD program enables conflict interleaving detection.

DEFINITION 10. *Two CSBBs,* $B_i, B_j \in CG$ *are* equivalent *(written* $B_i \equiv B_j$*) if they are isomorphic (contain the same set of edges) and contain the same set of statements.*

DEFINITION 11. *Let* $B_x \in CG_i$ *and* $B_y \in CG_j$*. If* $B_x \equiv B_y$ *then the* projection *of* B_y *onto* B_x *is the vertex* $B_{xy} \in CG_i, B_{xy} \equiv B_x$*, with* $RW_{xy} = RW_x \cup RW_y$*. More precisely,*

$$RW_{xy} = \begin{cases} RW_x \cup RW_y & \text{if } B_x \equiv B_y \\ RW_x & \text{otherwise} \end{cases}$$

Intuitively, the *projection* of $B_y \in CG_j$ onto $B_x \in CG_i$ merges the read-write sets of the two nodes, but leaves CG_i structurally unchanged (i.e. $B_{xy} \equiv B_x$).

The projection of thread T_j onto thread T_i is the projection of the individual CSBBs of CG_j onto CG_i.

3.3.1 The Projection Conflict Graph (PCG)

DEFINITION 12. *The Projection Conflict Graph, or PCG, of an SPMD program with threads $T_1, T_2 \ldots, T_N$ is the conflict graph of any one thread, say T_1, on which the conflict graphs of the other N-1 threads have been projected.*

Note that there can be only one PCG for any SPMD program. To see why this is so, consider an SPMD program with threads $T_1, T_2 \ldots, T_N$. Let the PCG for this program be constructed by projecting every thread onto the CG of thread T_1. Call this graph PCG'. Note that, by definition, $PCG' \equiv CG_1$. Construct another PCG by projecting the conflict graphs of threads $T_1, T_3 \ldots T_N$ onto the conflict graph of thread T_2 and call it PCG''. Then $PCG'' \equiv CG_2$. But, because this is an SPMD program, $CG_1 \equiv CG_2$ and therefore $PCG' \equiv PCG''$. Thus, the resultant PCGs are structurally equivalent. The equivalence of the read-write sets may be demonstrated in a similar fashion.

We can use the PCG to detect conflict interleavings in the program. The following theorems illustrate how inter-thread conflict cycles in the CG have equivalent cycles in the PCG:

THEOREM 6. *Every inter-thread conflict edge in the CG of an SPMD program has an equivalent conflict edge in the PCG for the program.*

Proof: Let the inter-thread conflict edge occur between nodes $B_i \in CG_i$ and $B_j \in CG_j$ (that is, $(B_i, B_j) \in E_C$). In other words, one or more of the following is true: (a) $R_i \cap W_j \neq \emptyset$, (b) $W_i \cap R_j \neq \emptyset$ or (c) $W_i \cap W_j \neq \emptyset$. Without loss of generality, assume condition (a) above is true. Let B_i', B_j' be the projections of B_i, B_j onto the PCG. Then, by definition, $R_i \subseteq R_i'$ and $W_j \subseteq W_j'$. Thus, if $R_i \cap W_j \neq \emptyset$ then $R_i' \cap W_j' \neq \emptyset$, and therefore $(B_i', B_j') \in E_C$.

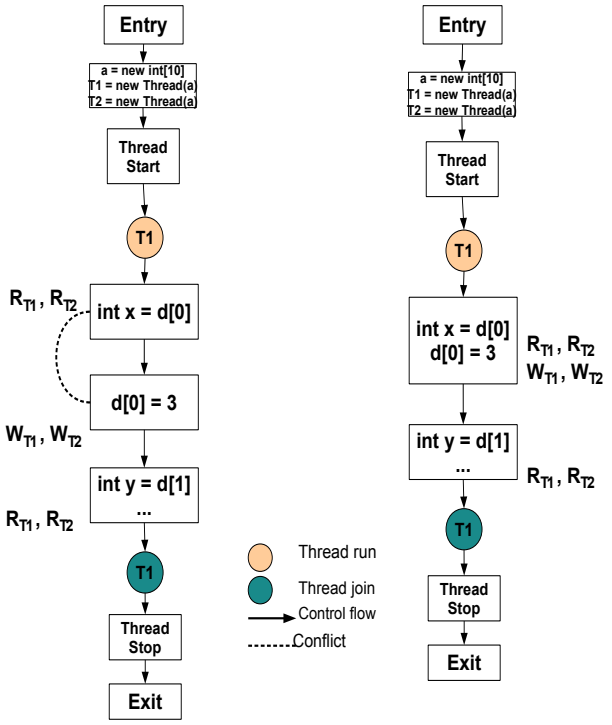
COROLLARY 2. *Every inter-thread conflict cycle in the CG of an SPMD program has an equivalent conflict cycle in the PCG for the program.*

Proof: An inter-thread conflict cycle consists of control flow edges and inter-thread conflict edges. Control flow edges are always present in the PCG (because the PCG is structurally equivalent to the CGs of the threads). From Theorem 6, every inter-thread conflict edge in the CG has an equivalent conflict edge in the PCG. Therefore, every edge in the inter-thread conflict cycle has an equivalent edge in the PCG, and the cycle itself is in the PCG.

Figure 5(a) shows the projection conflict graph of the conflict graph depicted in Figure 4. Here, thread `T2` has been projected onto thread `T1`. The read-write sets have also been shown: an R_{T_i} indicates a read by thread T_i, while a W_{T_i} indicates a write by thread T_i.

3.4 Inserting atomic regions into SPMD programs

So far we have focused on detecting conflict interleavings (CIs). Once CIs are detected, the next step is to insert atomic regions that will ensure that the CIs cannot lead to non-conflict serializable executions. In what follows, we assume that the Projection Conflict Graph (PCG) has been constructed.



(a) Projection Conflict Graph: Thread T_2 projected onto T_1 (b) SCC contraction of PCG

Figure 5. The Projection Conflict Graph and SCC Contraction of the SPMD program of Figure 3.

```
class Thread
  ...
  procedure void run()
    // run function for the thread
    atomic_region_begin()
    {
      // Generated transactional region
      int x = d[0]
      d[0] = 3
    }
    atomic_region_end()
    int y = d[1]
    ...
  end procedure
end class
```

Figure 6. Program with Atomic Regions Added.

We will first consider programs without loops, i.e. without back-edges. Later we discuss how to insert atomic regions into SPMD programs with loops.

Corollary 1 states that eliminating conflict interleavings will lead to a conflict-serializable execution run, while Theorem 5 shows that an inter-thread conflict cycle indicates a conflict interleaving. Corollary 2 states that every inter-thread conflict cycle in the CG has an equivalent conflict cycle in the PCG. Consider one such conflict cycle in the PCG. The cycle consists of control flow edges and conflict edges. Intuitively, the conflict cycle exists because two nodes that are transitively reachable via control flow edges are also

connected via a (by definition, bidirectional) conflict edge. Eliminating these conflict edges will then eliminate the conflict cycle, and will lead to a serializable execution run.

Note that because conflict edges are bidirectional, any node in the conflict cycle is reachable from every other node in the cycle, thus creating a Strongly Connected Component (SCC). Therefore, to eliminate the conflict cycle, we contract the SCC into a single node. The contraction is accomplished by merging the individual nodes in the SCC, with the merged node containing every statement contained in the individual nodes. The merged node also inherits the read-write sets (and therefore, the conflict edges to nodes not contained in the SCC) of the constituent nodes. Figure 5(b) shows the SCC contraction of the PCG from Figure 5(a).

The contracted nodes now constitute the final atomic regions in the program. Protecting against mutual access across these regions (e.g., through the use of lock sets or transactional memory implementations) will now prevent all interleavings which could result in a non-serializable schedule. Figure 6 shows the the input program from Figure 3 annotated with the (automatically generated) atomic region boundary markers.

3.5 Using semantic information in our analysis

Commutative Operations By definition any interleaving involving commutative operations leaves the program in the same state (and is therefore serializable), regardless of whether the memory accesses performed by the operations conflict. Note that this is true only if we treat the operation in toto and do not decompose the operation into its component reads and writes (an increment of a shared variable may first read the value of that variable and then write the incremented value back into the variable). TransFinder supports commutative operations in three ways (i) it detects commutative operations on shared memory locations, (ii) it treats such operations in their entirety (by merging the CSBBs which contain the component reads and writes comprising the operation) and (iii) it ensures that no conflict cycles can form as a result of conflict edges between any two commutative operations by deleting such edges. We currently only detect a small class of commutative operations (namely, arithmetic increments and decrements). This is sufficient for our purposes but support for detecting a larger class of such operations (using, e.g., the commutativity analysis of Rinard and Diniz [29]) could be added to TransFinder's analysis, if necessary.

Loops Loops have control flow back-edges. Thus, each loop is a strongly connected component. To avoid detecting and contracting these components, therefore, we avoid traversing control flow back-edges. In addition, the read-write sets of the nodes within a loop may contain accesses to shared arrays whose subscript functions do not cause loop-carried dependences, and other arrays and scalars that lead to loop-carried dependences. These references lead to two kinds of conflict cycles: (a) cross-iteration (or loop-carried) cycles, where the conflict arises from accesses to shared scalars or array references that have loop-carried dependences, and (b) intra-iteration cycles on array references involved in no loop-carried dependences. SCC detection and contraction resolves intra-iteration conflict cycles. Cross-iteration conflict cycles remain after SCC contraction; in such cases, the generated atomic region boundary markers are hoisted out of the loop so that the entire loop is in an atomic region. This is because the conflicts are loop-carried and TransFinder conservatively assumes every iteration is a part of the conflict cycle.

Conditional Branches Conflict cycles occur because nodes that are transitively reachable via control flow edges are also connected by a conflict edge. The case of conditional branches, such as *if* or *switch* statements, must be handled carefully. If the conditional branch is not contained within a loop, or the branch condition is invariant in all enclosing loops, then when a thread executes statements in one clause of the conditional branch it will never execute statements in the other clauses of that instance of the conditional branch since if one branch of the instance is taken then the other is not. Thus, in the compile-time representation of the program (the PCG), no strongly connected component is allowed to form across multiple clauses of such a conditional branch statement, and an atomic region will either span the entire branch (because a statement external to the branch conflicts with a statement contained within the branch) or will be confined to the statements within the individual clauses. When the conditional branch is within one or more loops and the branch condition is not loop invariant, a thread may execute different clauses of the conditional branch in different iterations of the loop. In this case statements in different clauses of a conditional branch are transitively reachable from each other along control flow edges. To handle this case, TransFinder allows the traversal of control flow back-edges of the loop when detecting SCCs whenever the compiler cannot guarantee the thread will execute only one clause of the branch within the loop. Traversing the back-edge ensures that every statement in the loop is transitively reachable from every other statement, and allows statements in the different clauses of the conditional branch to be reachable from one another.

4. Implementation

The ideas presented in Section 3 have been implemented in TransFinder and evaluated. TransFinder is a source-to-source compiler whose input is an SPMD shared memory Java program and whose output is C++ code that has been annotated with atomic regions inserted using the analyses and optimizations of Section 3. We now describe the implementation of TransFinder.

We first describe, in Section 4.1, the escape analysis that TransFinder uses to conservatively determine what variable references in the program are to storage that is accessed in more than one thread, and how the results of this analysis are combined with the results of Section 3 to build a PCG. We then describe situations that lead to TransFinder inserting atomic regions that are different than the hand-coded regions, and how TransFinder can help the programmer generate better atomic regions in these cases.

4.1 Escape analysis and detecting atomic regions

The TransFinder compiler uses the points-to analysis and code of [35], which builds upon the SPARK analysis [24]

```
procedure GenCG(CFG)
    input CFG: Control Flow Graph for the program
    returns cg: Conflict Graph
    cg = new CG( )
    curr = new CSBB( ), prev = NULL
    for each Node N_i ∈ CFG do
        if N_i does not contain a shared memory access then
            add all statements in N_i to curr
        else
            add curr to cg
            if prev ≠ NULL then
                add program edge (prev, curr) to cg
            end if
            prev = curr, curr = new CSBB( )
        end if
    end for
    return cg
end procedure

procedure ConstructPCG(CG, L)
    input CG: Conflict Graph for the program
    input L: List of Threads
    returns pcg: Projection Conflict Graph of the program
    CG_1 = CG of thread T_1
    for each Thread T_i ∈ L, 1 < i ≤ N do
        CG_i = CG of thread T_i
        for each CSBB N_i ∈ CG_i do
            Project N_i onto CG_1
        end for
    end for
    return CG_1
end procedure

procedure ConstructCG(CFG, L, T)
    input CFG: Control Flow Graph for the program
    input L: List of allocation sites
    input T: List of Threads
    returns cg: Conflict Graph
    cg = GenCG(CFG)
    for each a_i ∈ L do
        TW = list of CSBBs in CG writing to a_i
        TR = list of CSBBs in CG reading from a_i
        for each w_i ∈ TW do
            for each r_i ∈ TR do
                add conflict edge (w_i, r_i) to cg
            end for
        end for
    end for
    return cg
end procedure

procedure doSCC(PCG)
    input PCG: Projection Conflict Graph of program
    // Perform SCC detection and contraction
    stack = new Stack( )
    root = root of PCG
    SCC = tarjan(stack, root, 0)
    for each List L ∈ SCC do
        // Merge all nodes into a single node
        // This node is now the SCC contraction
    end for
end procedure

procedure TransFind(CFG, L, T)
    input CFG: Control Flow Graph of the program
    input L: List of allocation sites
    input T: List of threads
    cg = ConstructCG(CFG, L, T)
    pcg = ConstructPCG(cg, L)
    doSCC(pcg)
end procedure
```

Figure 7. Algorithm to identify atomic regions in code.

found in Soot [36]. The points-to analysis annotates allocation sites with the various contexts where they are read or written. The TransFinder compiler uses these read-write sets to perform an escape analysis as follows: if multiple threads have entries in the read-write sets of an allocation site then that site is considered to have thread-escaped. After the compiler has determined thread-escaping locations, the Conflict Graph (CG) is constructed from the Control Flow Graph (CFG) by using the thread-escape and read-write information obtained from the earlier phases to construct conflict edges between the various Concurrent Shared Basic Blocks (CSBBs). The Projection Conflict Graph (PCG) is then constructed and SCC detection and contraction proceed as described in Section 3.4. We use a modification of Tarjan's algorithm to detect SCCs. We modify the algorithm by considering only those previously visited successors of a given node which are connected to the node via a conflict edge. Figure 7 gives the complete algorithm for identifying an atomic region (the modified Tarjan's algorithm is not shown). In the interest of brevity, we omit the special cases mentioned in Section 3.5 from the algorithms.

4.2 Program tuning and optimization

The goal of TransFinder is to facilitate the insertion of optimized atomic regions or locks by identifying sequences of statements that must be protected against concurrent access to ensure conflict-serializable execution runs. In this section we discuss TransFinder's utility when the identified regions are different from those that a skilled programmer might identify.

TransFinder's atomic regions are smaller than manually identified atomic regions In some programs TransFinder identifies atomic regions that are too fine grained, leading to too-large overheads from starting and committing the transactions, or acquiring and releasing the locks used to enforce the atomic region. In these situations, TransFinder assists the programmer by focusing attention on the small percentage of program statements that likely should be protected by atomic regions, allowing the programmer to incrementally adjust and merge the suggested regions to create a correct and efficient program.

TransFinder's atomic regions are larger than manually identified atomic regions TransFinder's atomic regions are sometimes larger than the hand-coded atomic regions, particularly when memory accesses with inter-thread conflicts create loop carried dependences. In this case, TransFinder will hoist the atomic region out of the loop and enclose the entire loop in the atomic region, as described in Section 3.5. The atomic regions can be overly conservative for three reasons. First, the alias, escape and dependence analysis used by TransFinder are, like all such analyses, conservative, and thus the loop-carried dependence may not actually exist. Second, the semantics of the computation realized by the program may allow (or require) the accesses of

```
transaction_begin( )
{
   for i = 1 to numOps do
      op = opType[i]
      if op == A then
         // Perform op "A"
      else
         // Perform op "B"
      end if
   end for
}
transaction_end( )
```

(a) Atomic regions considering back-edges

```
for i = 1 to numOps do
   op = opType[i]
   if op == A then
      transaction_begin( )
      {
         // Perform op "A"
      }
      transaction_end( )
   else
      transaction_begin( )
      {
         // Perform op "B"
      }
      transaction_end( )
   end if
end for
```

(b) Atomic regions ignoring back-edges

Figure 8. Benchmark tuning example taken from the Vacation benchmark.

different iterations to be in different atomic regions. Third, the computations performed by different iterations may be commutative, but this commutativity may not be recognized by TransFinder.

As a heuristic to give the programmer insights into the behavior of the program, TransFinder first inserts atomic regions into the entire program and presents the results to the user. Next, TransFinder can optionally ignore the effects of the loop back-edges, starting with the outermost loop, then the outermost two loops, and so forth until all loops are ignored. The programmer is then presented with the atomic regions that result from each of these cases. By examining the identified atomic regions the programmer can see the effect that the different loops have on the atomic regions. From these results, the programmer can either pick the set of atomic regions that properly reflects the program semantics, or can incrementally adjust the atomic regions using domain specific semantic knowledge about the computation.

The Vacation benchmark in the STAMP benchmark suite [6] exhibits this behavior. A succession of operations on a database are simulated by a loop that picks, at random, different operations on different data. Because the data induces a loop-carried dependence, TransFinder initially generates an atomic section that encloses the entire loop, as shown in Figure 8(a). By ignoring the back-edge for the loop, the atomic regions in Figure 8(b) are produced, and are exactly those present in the original, hand-transformed code.

5. Experimental Evaluation

This section presents the experimental evaluation of TransFinder, including the experimental methodology, quantitative performance results, and detailed analysis.

5.1 Methodology

The strategies discussed in this paper are evaluated in two ways: by comparing the similarity of the generated atomic regions to hand-generated versions and by comparing the performance of the programs using automatically-generated atomic regions with respect to programs with hand-coded atomic regions. We use the Stanford Transactional Applications for Multi-Processing (STAMP) benchmark suite and the multithreaded version of the Java Grande benchmark suite for our experiments [6, 15].

The STAMP suite is a set of benchmarks originally designed to evaluate the efficacy of various transactional memory implementations. However, its use of atomic regions provides an objective way to evaluate the ability of TransFinder to identify atomic regions. The original STAMP benchmarks are `C` programs synchronized with explicit atomic regions. TransFinder, on the other hand, accepts as input un-annotated Java code. Accordingly, we modify the original STAMP benchmark programs by stripping out the *begin* and *end* statements of the various atomic regions and manually converting the code to Java, before feeding it to TransFinder.

The STAMP benchmark suite consists of: (1) *Genome*, a gene sequencing program; (2) *Intruder*, a network intrusion detection program; (3) *Kmeans*, a K-means clustering program[1]; (4) *Labyrinth*, a maze routing program; (5) *Vacation*, a travel reservation program; (6) *Bayes*, a bayesian network learning algorithm implementation[2] (7) *Yada*, an implementation of a Delauney mesh refinement algorithm and (8) *Ssca2*, a program which implements Kernel 1 of the Scalable Synthetic Compact Applications 2 (SSCA2) graph-based benchmark [2].

The Java Grande benchmark suite is a collection of low-level kernels and applications for scientific and numerical computing. We use the suite's three large-scale applications to test the applicability and performance of our system on non-transactional programs. The three Java Grande programs considered were: (1) *Moldyn*, a molecule dynamics simulation; (2) *Raytracer*, a ray-tracing simulation and (3) *Montecarlo*, a Monte Carlo simulation. The *Moldyn* and *Montecarlo* codes do not use synchronized blocks, relying on a careful thread-local partitioning of data to ensure conflict-serializability — we use these codes to determine if TransFinder generates unnecessary atomic regions for large codes. For *Raytracer*, we manually remove all synchronized blocks and regions before inputting the code into TransFinder, just as with the STAMP benchmarks.

Similarity of the TransFinder generated programs to the original "Native" programs is measured in three ways. First, we give the number of atomic regions that are present in both forms of the program. Second, we give the total number of lines contained within atomic regions in each program. Third, we give the percentage of code, i.e. the percentage of the lines of program code, within atomic regions. Only non-comment lines of code are used in our measurements, and we used the `sclc` line-counting tool [4] for measuring the non-comment, source code size of the generated atomic regions with respect to the original program. For the TransFinder versions of the STAMP benchmarks, line number measurements were conducted on the generated C++ code. For the Java Grande benchmarks, we first analyzed the programs

[1] We converted the TL2 version of Kmeans to use double precision arithmetic because of a bug in TL2 which caused spurious writes to successive `float` array elements within tight loops.

[2] We use the `CM_DELAY` contention manager for the TinySTM versions of Bayes; the default contention manager caused an inordinate number of conflicts in the TransFinder version of the benchmark.

Applications	Program Version	# static atomic regions	total lines in atomic regions	% of code in atomic regions
Bayes	Native	15	52	1.70
	TransFinder	5	80	2.62
Genome	Native	5	50	3.91
	TransFinder	5	44	3.44
Intruder	Native	3	8	0.63
	TransFinder	2	13	1.02
Kmeans	Native	3	14	2.26
	TransFinder	4	12	1.94
Labyrinth	Native	3	18	1.75
	TransFinder	3	18	1.75
Ssca2	Native	3	21	0.61
	TransFinder	5	18	0.53
Vacation-untuned	Native	3	83	4.71
	TransFinder	1	121	6.87
Vacation-tuned	Native	3	83	4.71
	TransFinder	3	83	4.71
Yada	Native	6	11	0.69
	TransFinder	6	9	0.57
Moldyn	Native	0	0	0
	TransFinder	0	0	0
Monte carlo	Native	0	0	0
	TransFinder	0	0	0
Raytracer	Native	1	2	0.29
	TransFinder	1	2	0.29

Table 1. Comparison of atomic regions generated by TransFinder to atomic regions in the original benchmarks.

with TransFinder and then manually mapped the indicated atomic regions (if any) onto the original (Java) programs to conduct line number measurements.

The performance tests in this paper use software transactional memory (STM) as the concurrency control mechanism. We consider three such STM systems: the TL2 system [11], the TinySTM system [16] and the SwissTM system [12]. Each application that includes atomic regions is evaluated with two different versions: a version that uses the default atomic regions in the benchmarks (called "Native") and a version that uses the atomic regions generated by TransFinder (called "TransFinder").

All experiments are performed on a Dell Poweredge 2950 server with two 1.8 GHz quad-core Intel Xeon E5320 processors based on the Core 2 microarchitecture (8 cores total). This system has 16 GB of system RAM and 4 MB L2 caches shared between pairs of processors. This system runs Linux kernel 2.6.18 and GNU C library 2.3.6 with the Native POSIX Threads Library in the Debian AMD64 distribution. All C/C++ programs were compiled using the GNU `gcc` compiler at optimization level: `-O3`. We used Java Grande Thread Version 1.0 and STAMP version 0.9.10 for the benchmarks. For the STM backends we used version 0.9.6 of the x86 port of TL2, version 0.9.9 of TinySTM and version 2009-09-10 of SwissTM.

5.2 Analysis Running Times

Table 2 details the analysis times for the TransFinder tool for the eight STAMP benchmarks considered. The TransFinder-specific phase of the analysis took a negligible amount of time for the three Java Grande programs and they are not presented here. The first column gives the name of the application. The second column depicts the average time taken by the alias analyser to perform the points-to analysis of [35], while the last column shows the average time taken to perform the phases of the analysis specific to TransFinder (including the construction of the PCG and the SCC detection and contraction phases). As can be seen, the TransFinder-specific phase of the analysis incurs only a minimal overhead, ranging from a minimum of 0.37 seconds (Kmeans) to a maximum of 7.16 seconds (Vacation Tuned). In addition, the loop elision technique described in Section 4.2 adds an average overhead of less than 1 second.

Applications	Alias Analysis Time (secs)	TransFinder-specific Time (secs)
Bayes	23.2777	5.772
Genome	12.21067	1.36233
Intruder	11.676033	0.488667
Kmeans	10.421033	0.370667
Labyrinth	16.48837	2.86633
Ssca2	13.8843	8.734
Vacation-untuned	22.171	6.36667
Vacation-tuned	15.89603	7.15667
Yada	15.20433	1.85067

Table 2. Analysis times for the TransFinder tool

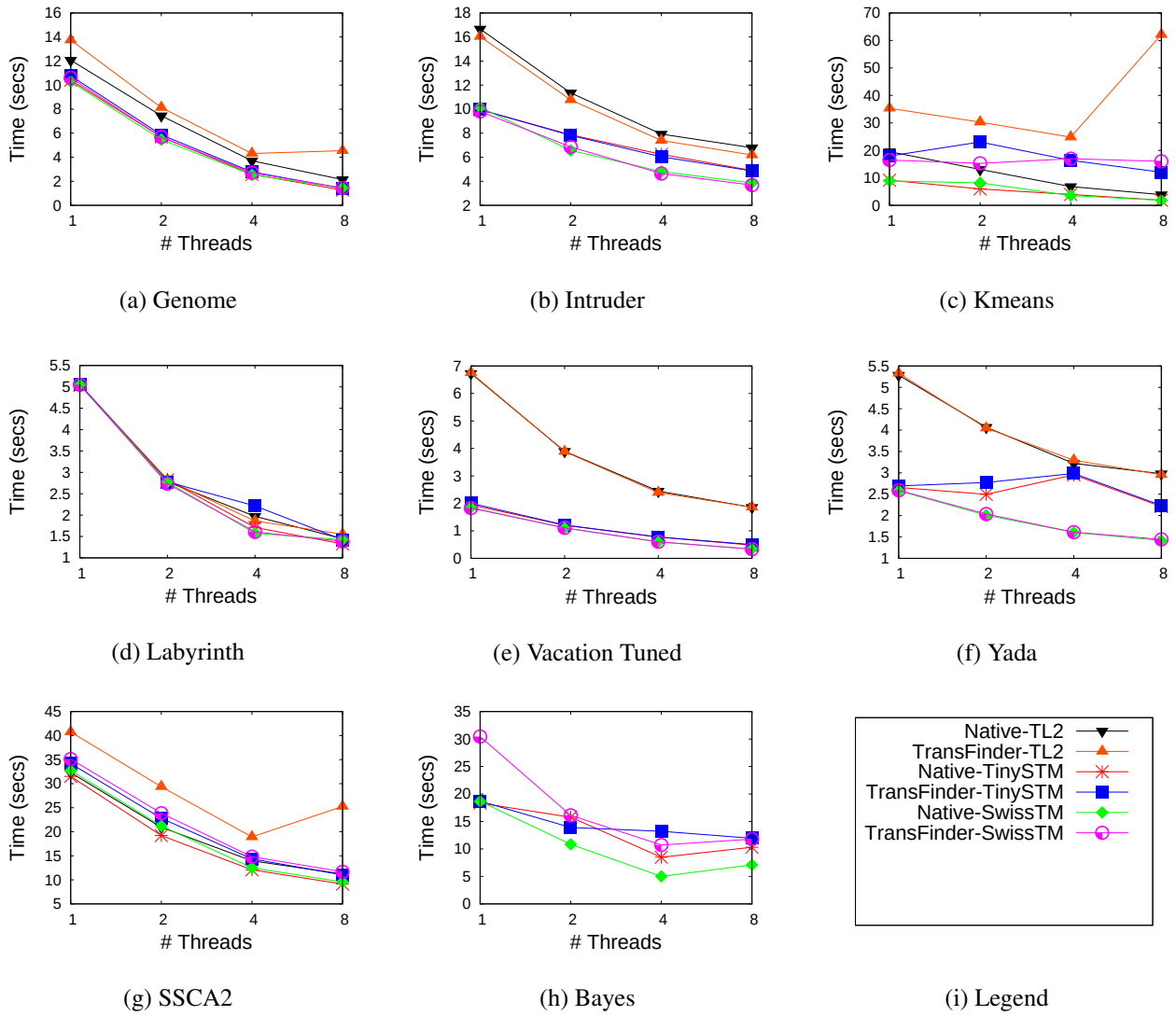


Figure 9. STAMP benchmark performance and scalability results of the various STM backends.

5.3 Code Quality Results

Table 1 gives a measure of the effectiveness of TransFinder in determining atomic regions. The first two columns give the name and version of the program for which metrics are provided in the other columns. The "Native" version is the original benchmark, and the "TransFinder" version is the benchmark transformed by TransFinder. The "# static atomic regions" column presents the number of atomic regions in the Native and TransFinder versions of the programs. The "total lines in atomic regions" column presents the total number of non-comment source lines of code in the atomic regions for each of the versions of the benchmark programs, while the "% of code in atomic regions" column expresses this number as a percentage of the total (non-library) lines of code in the respective programs.

The three applications at the bottom present results for the three Java Grande benchmarks considered. All three are in 100% agreement with the original atomic regions. Among the STAMP benchmarks, Labyrinth shows the best result, with the TransFinder version of the benchmark having the same number of atomic regions as the Native version. In addition, each of these regions has exactly the same code in both versions. The TransFinder version of Intruder, on the other hand, generates a smaller number of larger atomic blocks than are present in the Native version of the benchmark. A closer inspection reveals that conservatism in the thread-escape analysis leads to an over-estimation of the size of the generated atomic regions. Genome, Kmeans, Yada and Ssca2 are interesting cases: the generated atomic regions are, on average, actually *smaller* than those in the original STAMP benchmarks. In the case of Genome, an atomic region which should properly contain only a single hashmap insertion within a loop has been hoisted out of the loop by the benchmark designers, presumably to minimize transaction startup and shutdown times. This pattern recurs in Ssca2, where three separate atomic blocks containing code

accessing disjoint memory locations have been agglomerated into a single atomic block, and in Yada, where atomic blocks containing accesses to two disjoint memory locations have been agglomerated into a single block. In Kmeans, the original benchmark encloses an entire inner loop within an atomic region, even though the shared memory accesses in different loop iterations are independent of each other. TransFinder, on the other hand, recognizes the iterations as independent and thus generates atomic regions only large enough to enclose the shared memory accesses within a single iteration. As in the other benchmarks, the atomic regions in the original benchmark version seem to have been coded to reduce the transaction startup and shutdown time.

In the two remaining STAMP benchmarks, Vacation and Bayes, the number of atomic regions indicated by TransFinder was less than the number of such regions in the original programs. Of these, the least similar was Vacation, where the TransFinder version of the benchmark agglomerated three separate atomic regions into a single block (although even in this case, the absolute difference in the size of the atomic regions between the two versions of the benchmark is only 38 lines). Vacation, however, is amenable to the tuning techniques detailed in Section 4.2. Accordingly, we present two results for Vacation: a version which uses atomic regions generated after the outermost loop has been elided using the methodology described in Section 4.2 (called Vacation Tuned) and a version which uses the atomic regions generated without the benchmark tuning (called Vacation Untuned). As we can see, TransFinder can properly identify benchmark artifacts that preclude correct identification of independent atomic regions, and can suggest alternatives with a high degree of accuracy: the tuned version of the benchmark is 100% conformant with the STAMP version. The number of atomic blocks in the TransFinder version of the Bayes benchmark is a third of the number present in the original benchmark. A closer inspection of the code reveals the cause: two switch statements (with common case expressions) cause multiple conflicting statements to be transitively reachable from each other. This causes the SCC contraction phase to agglomerate 11 different atomic regions into a single, large atomic block, and serves to illustrate a shortcoming in this, and indeed, any static analysis, namely, an inability to incorporate complex semantic information in the decision making process. The user-synchronized code, on the other hand, takes advantage of application-specific information to break up the large block of code into multiple, (semantically) independent chunks.

The differences in the Native and TransFinder versions are also shown by providing the percentage of each version of the program that is contained in atomic blocks. The percentages and the absolute number of lines in atomic blocks show that even when TransFinder's code differs from the hand-tuned code, both the absolute and relative amount of code that needs to be examined by the programmer when doing performance tuning of atomic regions is small.

5.4 Performance Results

Figure 9 presents experimental results showing the performance and scalability of the eight STAMP benchmarks under each of the implementations described above. We do not show the Java Grande benchmarks because, as discussed in Section 5.3, the TransFinder code and original code are both identical.

In each case, the X axis represents the number of threads while the Y axis shows the execution time, in seconds, on the given platform. We note that numbers for "Vacation Untuned" are not presented: as discussed in Section 4.2, the presence of loop-carried dependences results in a single, serialized block. Accordingly, the results for Vacation use the loop elision technique of Section 4.2, as described above. In six of the benchmarks (Genome, Intruder, Labyrinth, Vacation, Yada, and Ssca2), the performance of the TransFinder versions that use TinySTM and SwissTM for concurrency control closely track that of the respective Native versions. This is to be expected, since the generated atomic regions have a high degree of conformity to the default atomic regions in the original STAMP benchmarks (the least accurate, Intruder, generates an atomic region which is only 5 lines larger than the original). In addition, the TransFinder-TL2 version of the benchmark performs similarly to the Native version for Intruder, Labyrinth, Vacation and Yada. In the case of Genome and Ssca2, however, the TransFinder-TL2 version does not scale past 4 threads. A detailed analysis of Genome revealed high overheads for a transaction containing a single hashmap insertion call within a loop: in the original version of the benchmark the transaction is hoisted out of the loop, thus minimizing transaction startup and shutdown times. Similarly, an analysis of Ssca2 revealed higher overheads due to the added number of transaction startups and shutdowns in the TransFinder generated code as discussed in Section 5.3. The TransFinder versions of the Kmeans benchmark did not scale well because of the added overheads of the extra transaction begin and end function calls (the TransFinder versions execute 32 pairs of these calls for each pair executed by the Native versions).

We do not present TL2 results for Bayes here: both the Native-TL2, and the TransFinder-TL2 versions of the benchmark terminated with a segmentation fault. Figure 9(h) shows the results of the other versions of the benchmarks. In general, the TransFinder versions of the benchmark run slower than the corresponding native versions. This is likely due to the agglomeration of multiple atomic regions into a single, large atomic block, as described in Section 5.3: an analysis revealed a large number of conflicts within this region. It should be noted, however, that the TransFinder versions follow the same trend as the Native versions of the benchmark.

Where there are differences in either code quality or performance between the results of TransFinder and the original benchmarks, these differences stem primarily from situations where benchmarks have been tuned for performance by using atomic regions that are not the minimal set of statements that must be protected against concurrent access. In these situations, TransFinder enables the programmers' tuning efforts to be focused on only the small part of the program that must be covered by atomic regions. The programmer can then use additional semantic knowledge about the program to determine if those generated atomic regions can be altered without compromising correctness. These ideas form the basis of the TransFinder approach to generating

atomic regions: first automatically producing atomic regions with conflict-serializability, and then allowing the programmer to focus code tuning efforts only on those regions that appear to be bottlenecks.

6. Related Work

Vechev et al. have also attempted to analyze shared memory programs with a view to enforcing whole-program correctness criteria [37, 38]. In [37] the authors parameterize (finite-state) programs through the use of an n-tuple characterized by the values of the various shared variables, and the program counters of the various threads participating in the program. The authors then enumerate all possible program states and attempt to successively remove states which would cause an incorrect program condition, for some (user-defined) correctness criterion. The authors extend this approach in [38] by allowing infinite-state programs (programs which may iterate indefinitely). This work may be viewed as a program verification tool which verifies programs under a (user-supplied) correctness criterion and, if necessary, modifies programs (by introducing atomicity constraints) to conform to the correctness conditions. The approach is similar to their previous work in that programs are represented as a set of states, and invalid transitions between these states (more properly, transitions which may lead to invalid interleavings) are eliminated. However, there are some significant differences: the program states are parameterized through the use of various abstractions (more precisely, they use abstract representations to parameterize the shared variable values in the state tuples). This allows them to verify infinite-state programs. Moreover, at every stage, when faced with a possibly invalid program state, the algorithm chooses to either (a) remove the underlying interleaving by adding atomicity constraints to an (evolving) atomicity formula or (b) re-verify the program under a new, more refined abstract representation.

Our approach differs from these works in the following, significant ways: (1) they use a user-supplied correctness condition, whereas we rely on conflict-serializability as our (fixed) consistency criterion and (2) they use an enumeration of program states, each characterized via the use of an abstract representation, whereas we rely on cycles in a *Conflict Graph* to determine violations of serializability. Providing a user-supplied whole-program correctness condition has the advantage of generality: their approach works equally well whether analyzing the program with a view towards enforcing serializability, or some other correctness condition. However, the specific benefits to this approach are, in our opinion, somewhat muted by the fact that the onus of responsibility for providing the (sometimes convoluted) correctness criteria now falls on the end-users. Indeed, [38] cites cases where these conditions were too "long and tedious" to reproduce. Also, using a set of program states allows them to exercise greater control over the granularity of the various correctness criteria, but leads to a larger search space of invalid interleavings (possibly exponential in the number of shared variables and threads), whereas our *Projection Conflict Graph* approach takes up no more space than is required to express the control flow of a single thread. In addition, these works target numerical programs, whereas our approach applies to general shared memory parallel programs.

The problem of serializability-preserving transaction optimization has received much attention in the database community [1, 3, 5, 31, 40]. Of these, the work most relevant to this paper is the transaction chopping approach of Shasha et al. [31], in which the authors attempt to reduce (or "chop") the individual transactions in a database program while guaranteeing serializable executions. They do so by identifying a set of primitive database operations, by determining the resultant connected components and then by enforcing a precedence order to determine the final chopping. However, their work, unlike ours, targets database transactions and not general shared memory programs. Also, their tool operates on programs with known transactional regions, unlike ours. Finally, their work is predicated on the assumption that the user of the tool can characterize all of the transactions that may run in some time interval, whereas we make no such assumption (indeed, it can be argued that the problem of a priori transactional region identification in shared memory parallel programs is hard precisely *because* of the lack of such semantic information).

There exists a rich body of work exploring the problem of concurrency control in shared memory programs. Given some user-identified atomic scopes, researchers have proposed solutions to efficiently guard against concurrent modifications in those sections, either by providing mutual exclusion (compiler-directed lock generation [7, 14, 17, 21, 25, 35, 41]) or by using optimistic techniques (transactional memory [18–20, 26, 33, 34]). We view these approaches as being complementary to ours: the techniques discussed in this paper may be used to identify the atomic regions that must be guarded by the various concurrency control mechanisms.

Shasha and Snir [32] provide a qualitative graph-based analysis of the requirements for enforcing sequential consistency in shared memory programs under a given set of atomicity constraints. Given a shared memory program, their work attempts to define the set of delays and locks which must be used to enforce sequential consistency. Krishnamurthy and Yelick [22] present a set of optimizations for SPMD programs that extend the delay set analysis of Shasha and Snir by incorporating synchronization analysis and optimizing the resultant cycle analysis for SPMD programs. Both of these works detect violations of sequential consistency in shared memory programs by detecting cycles in their underlying conflict graphs. In addition, Krishnamurthy and Yelick improve the accuracy of the delay set analysis by incorporating additional information from an analysis of the synchronization constructs in the program. They, like us, are optimizing based on a conflict graph of a shared memory SPMD program. We differ in two significant ways: first, they project all threads onto two, rather than one, thread, and second, their goal is to detect violations of sequential consistency, which is a weaker correctness criterion than conflict-serializability. Moreover, they rely on user-supplied transactional boundary information rather than attempting to automatically discover this information.

Data race detection tools [8, 9, 25, 27, 28, 30] analyze programs to determine data races: unsynchronized accesses

to the same memory location by more than one thread, where at least one access is a write. They do so by using static analyses, at runtime, or both. Unlike our work, none of these tools attempts to enforce any whole-program correctness criteria such as serializability.

Our use of the conflict graph is inspired by the Concurrent Control Flow Graph of Lee et al. [23]. In that work, the authors present a mechanism for extending the SSA form of Cytron et al. [10] to parallel shared memory programs. They then show how the resultant Concurrent Static Single Assignment (CSSA) form can be used for optimizations such as constant propagation in parallel programs. They do not, however, optimize the flow graph for SPMD programs. Nor do they attempt to discover atomic regions in the code.

7. Conclusions

This paper presented an effective method for identifying atomic regions in multithreaded SPMD programs. The paper systematically mapped the conflict-serializability problem onto the space of multithreaded programs by qualitatively demonstrating the equivalence of conflict graphs and precedence graphs. It then presented optimizations of conflict graphs for SPMD programs and demonstrated how, in this special case, it is possible to project the effects of multiple threads onto a single thread, thus reducing the space and time requirements for the atomic region identification process. Implementing these ideas allowed us to reproduce the atomic regions with 100% accuracy in the three Java Grande benchmarks considered. The generated atomic regions were also either equal to or smaller than the default atomic regions in a majority of the STAMP benchmarks, and were never more than 38 lines larger in the remaining STAMP benchmarks. In addition, this paper also presented a methodology to aid in benchmark tuning via the use of iterative loop elision and demonstrated how this technique is particularly helpful in situations where benchmark design dictates the placement of multiple independent atomic regions within large loops.

In general, atomic region identification will always involve some programmer effort due to the semantic nature of the problem. However, we have demonstrated, for the first time, how it is possible to automatically identify atomic regions in multithreaded SPMD programs, using only an unannotated program as input, with a surprising degree of accuracy for a varied class of benchmarks. Much work still needs to be done in this area but we believe that, with the ideas mentioned in this paper, we have provided an important stepping stone in the process.

References

[1] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy. Relative serializability (extended abstract): an approach for relaxing the atomicity of transactions. In *PODS '94: Proceedings of the ACM Symposium on Principles of Database Systems*, pages 139–149, New York, NY, USA, 1994. ACM.

[2] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *HiPC '05: Proceedings of the High Performance Computing Conference*, pages 465–476, 2005.

[3] R. Bayer. Consistency of transactions and random batch. *ACM Transactions on Database Systems*, 11(4):397–404, 1986.

[4] Brad Appleton. Sclc and Cdiff: Perl scripts for ClearCase. At `http://www.cmcrossroads.com/broadapp/clearperl/sclc-cdiff.html`.

[5] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD '08: Proceedings of the ACM International Conference on Management of Data*, pages 729–738, New York, NY, USA, 2008. ACM.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[7] S. Cherem, T. M. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI '08: Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 304–315, 2008.

[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 258–269, New York, NY, USA, 2002. ACM.

[9] M. Christiaens and K. De Bosschere. Trade, a topological approach to on-the-fly race detection in Java programs. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the International Symposium on Distributed Computing*, pages 194–208, 2006.

[12] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 155–165, New York, NY, USA, 2009. ACM.

[13] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[14] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL '07: Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 291–296, New York, NY, USA, 2007. ACM.

[15] EPCC. The Java Grande Forum Benchmark Suite. At http://www.epcc.ed.ac.uk/research/java-grande/. Last accessed March 24, 2010.

[16] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[17] R. L. Halpert, C. J. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT '07: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[18] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[20] M. Herlihy, J. E. B. Moss, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[21] M. Hicks, J. S. Foster, and P. Prattikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[22] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38(2):130–144, 1996.

[23] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *PPoPP '99: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1999.

[24] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Proceedings of the International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[25] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 346–358, New York, NY, USA, 2006. ACM.

[26] M. Moir, K. Moore, and D. Nussbaum. The adaptive transactional memory test platform: a tool for experimenting with transactional code for rock (poster). In *SPAA '08: Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, pages 362–362, 2008.

[27] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. ACM.

[28] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 167–178, New York, NY, USA, 2003. ACM.

[29] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):1–47, 1997.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[31] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3):325–363, 1995.

[32] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[33] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[34] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 78–88, New York, NY, USA, 2007.

[35] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Using data structure knowledge for efficient lock generation and strong atomicity. In *PPoPP '10: Proceedings of the ACM Symposium on Principles and Practice Of Parallel Programming*, pages 281–292, 2010.

[36] R. Vallee-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction (CC '09)*, pages 18–34, 2000.

[37] M. T. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *TACAS '09: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 139–154, 2009.

[38] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL '10: Proceegings of the ACM Symposium on Principles of Programming Languages*, pages 327–338, 2010.

[39] H. Volos, N. Goyal, and M. Swift. Pathological interaction of locks with transactional memory. In *Proceedings of the ACM Workshop on Transactional Computing (TRANSACT) '08*, 2008. article available at http://www.unine.ch/transact08/papers/Volos-Pathological.pdf. URL last checked on Nov. 19, 2009.

[40] O. Wolfson. The virtues of locking by symbolic names. *Journal of Algorithms*, 8(4):536–556, 1987.

[41] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In *Proceedings of the 21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC '08)*, 2008.

[42] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for Java. In *ECOOP '08: Proceedings of the European Conference on Object-Oriented Programming*, pages 129–154, 2008.

A. Example

This section presents a more detailed example of an SPMD program, and its associated conflict and projection conflict graphs. Figure 10(a) shows a code fragment adapted from the `Kmeans` benchmark in the STAMP transactional memory benchmark suite [6] (shared variables in the figure have been annotated for clarity; the code analyzed by TransFinder does not contain any annotations). Figure 11 shows the resultant Conflict Graph (CG). Figure 12(a) shows the Projection Conflict Graph (PCG), with Thread `T2` projected onto Thread `T1`, and Figure 12(b) shows the result after SCC contraction. The CSBBs with a dashed outline represent atomic regions. Blue dashes indicate CSBBs which were formed as a result of an SCC contraction, while red dashes indicate CSBBs which were carried over unchanged from the PCG. Figure 10(b) shows the code fragment from Figure 10(a) with the appropriate `atomic` synchronization constructs added (only the `run` procedure is shown).

```
class Thread
  // Constructor and members elided
  procedure void run()
    // Local variable declarations elided
    while start < npoints do
      int stop = ...
      for i = start to stop do
        index = foo(...)
        if shared_mem[i] ≠ index then
          delta ++
        end if
        shared_mem[i] = index
        shared_arr1[index][0] =
          shared_arr1[index][0] + 1
        for j = 0 to nfeatures do
          shared_arr2[index][j]          =
          shared_arr2[index][j]          +
          shared_feat[i][j]
        end for
      end for
      start = shared_i
      shared_i = start + CHUNK
    end while
    shared_delta = shared_delta + delta
  end procedure
end class
...
procedure main()
  // Initialization code elided
  Thread T1 = new Thread(...)
  Thread T2 = new Thread(...)
  T1.run(); T2.run()
end procedure
```

(a) Original Code

```
procedure void run()
  // Local variable declarations elided
  while start < npoints do
    int stop = ...
    for i = start to stop do
      index = foo(...)
      atomic {
        if shared_mem[i] ≠ index then
          delta ++
        end if
        shared_mem[i] = index
      } // end atomic
      atomic {
        shared_arr1[index][0] =
          shared_arr1[index][0] + 1
      } // end atomic
      for j = 0 to nfeatures do
        atomic {
          shared_arr2[index][j]          =
          shared_arr2[index][j]          +
          shared_feat[i][j]
        } // end atomic
      end for
    end for
    atomic {
      start = shared_i
      shared_i = start + CHUNK
    } // end atomic
  end while
  atomic {
    shared_delta = shared_delta + delta
  } // end atomic
end procedure
...
```

(b) Synchronized Code

Figure 10. Kmeans benchmark code: globally-scoped variables are annotated with `shared_`.

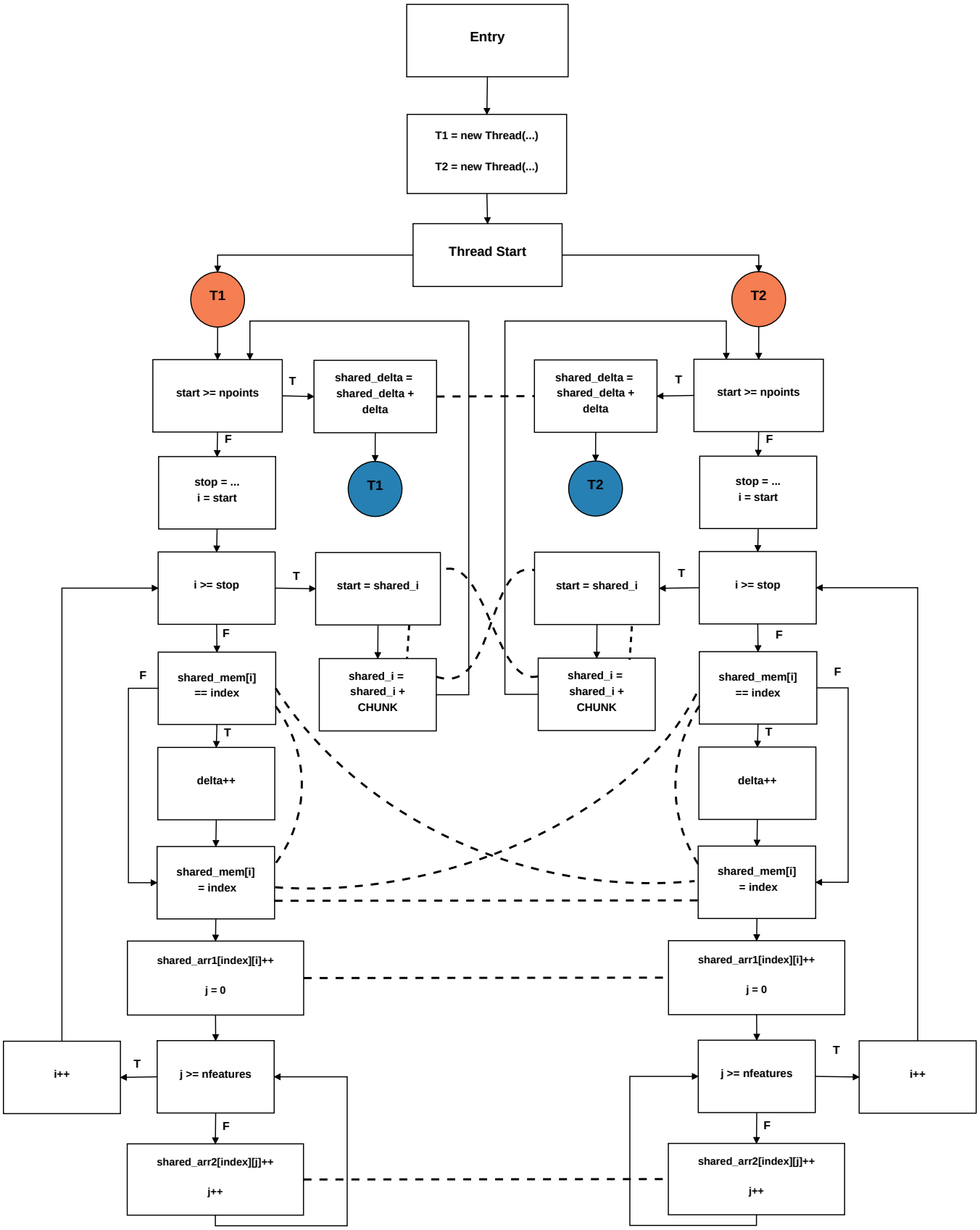
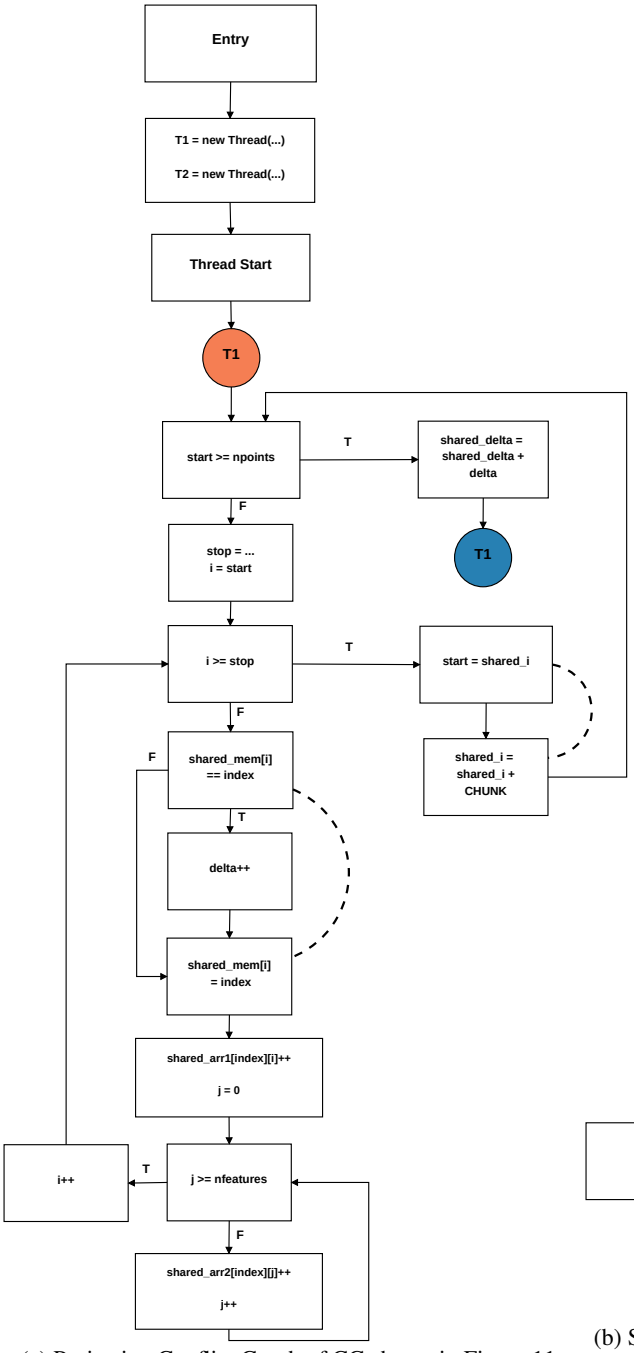
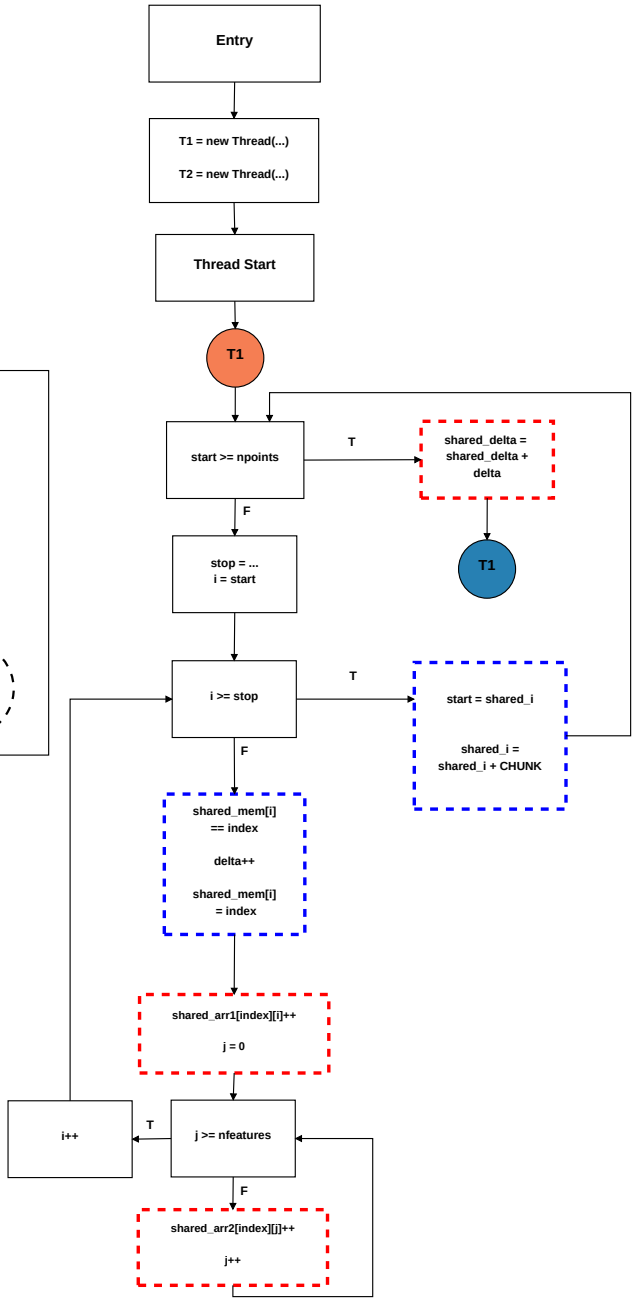


Figure 11. Conflict Graph of program fragment from Figure 10. The Thread Stop and Exit nodes are not shown.



(a) Projection Conflict Graph of CG shown in Figure 11.



(b) SCC Contraction of the PCG for the `kmeans` example

Figure 12. PCG and SCC contraction.