# Lightweight Fault Detection in Parallelized Programs

Li Tan
CSE Department, UC Riverside
ltan003@cs.ucr.edu

Min Feng
NEC Laboratories America
mfeng@nec-labs.com

Rajiv Gupta
CSE Department, UC Riverside
gupta@cs.ucr.edu

## Abstract

A popular approach for producing parallel software is to develop a sequential version of an application and then incrementally introduce parallel constructs to parallelize different parts of the application. During the parallelization process, programming errors may be introduced, causing concurrency bugs. In this paper we develop a technique for runtime detection of data dependence faults (i.e., data races and atomicity violations) introduced during parallelization. By leveraging the availability of two versions of the program, the sequential one and the parallelized one, we *comparison check* dynamic data dependences exercised during executions of the two versions to identify faults.

To reduce the cost of *comparison checking* we develop three optimizations. The first optimization causes only a subset of dynamically exercised data dependences to be *comparison checked*. The second optimization shows that not all instances of a dynamically exercised data dependence need to be comparison checked. The third optimization shows that static analysis of parallelizing constructs can be exploited to eliminate the need for executing the parallelized version altogether. In addition, our solution is applicable when different program executions on the same input may follow different execution paths, it is effective in situations where the fault introduced manifests itself rarely during execution, and it is also effective in pinpointing the location of the fault in the program. We implemented and evaluated our approach using ten benchmarks. The experimental results indicate an average slowdown of 3x to perform fault detection.

***Categories and Subject Descriptors*** D.2.5 [*Software Engineering*]: Testing and Debugging – Debugging aids, Testing tools, Tracing; D.3.4 [*Programming Languages*]: Processors – Debuggers

***General Terms*** Algorithms, Measurement, Reliability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*CGO '13* 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

***Keywords*** program parallelization, debugging, dependence violation, data races, atomicity violations

## 1. Introduction

With an increasing number of cores on a single chip, exploiting parallelism is crucial to enhancing the performance of applications. A widely used approach is to develop a sequential version of an application and incrementally parallelize it using parallel constructs provided by high level parallel programming models such as OpenMP [9], SpiceC [11], and TBB [28]. While these models provide easy-to-use constructs for parallelization, however, parallelization can be an error-prone process. Incorrect parallelization can lead to introduction of concurrency bugs such as data races and atomicity violations. Providing tools for detecting and locating faults in parallelized programs is important to reduce debugging efforts and increase programmer productivity.

In this paper we present a technique for detecting and locating data dependence faults introduced during program parallelization. A data dependence fault is introduced when the parallelization violates a data dependence enforced by the sequential execution – data races and atomicity violations are two forms of data dependence faults. The technique we propose is based upon *comparison checking* [14, 15] which leverages the availability of two versions of the program, the sequential one and the parallelized one, to detect faults. In particular, the two versions are executed on the same input and the dynamic data dependences exercised by the two versions are compared to identify differences that result from introduction of data dependence faults.

While comparison checking is an effective approach, there are several challenges for utilizing it in practice. First, the cost of using it to solve our problem is very high. Not only both sequential and parallelized program versions must be executed, but the overhead of dynamically tracking data dependences and checking corresponding dependences is very high both in terms of execution time and memory space [22]. Second, programs may contain features (use of random numbers, time dependent code etc.) that may alter the control and data flow arbitrarily in different runs [13]. The variation of program flow further increases the difficulty of exposing faults. Third, the potential of non-deterministic events such as random interleaving involved during execu-

tions of parallel programs makes it difficult to reproduce and hence locate faults [26]. Lastly, many techniques are designed for certain types of concurrency bugs, e.g., data races [12, 19–21, 23, 24, 27, 29, 33, 34] and atomicity violations [6, 16, 26]. However, developers rarely know the bug type present in advance. Therefore, a technique that handles multiple kinds of bugs is desirable.

We propose an efficient and precise approach to locating data dependence faults in parallelized programs. Our approach is based on *Region Graphs* (RGs), which are coarse-grained data dependence graphs. Each node in an RG corresponds to a code region, which is a code block annotated by parallel constructs used in parallelization. By comparing RGs of two versions of a program, the sequential one and the parallelized one we identify faults introduced during parallelization. We also show our approach is general – it can locate data dependence faults like data races and atomicity violations and it can be employed for many parallel programming models like OpenMP, SpiceC, and TBB. In summary, the contributions of this paper are as follows:

- We show that two types of concurrency bugs cause the dynamic data dependences in the parallel run to differ from those in the sequential run;
- We propose an efficient and precise approach to locating data dependence faults in parallelized programs, which only requires dynamically profiling the execution of the sequential version of the program and statically analyzing the parallel constructs inserted for parallelization;
- We prove that our approach is equivalent to directly comparing the fine-grained dynamic data dependences in the sequential and parallel runs; and
- Our fully optimized approach achieves on average 3x slowdown compared to original executions.

The remainder of the paper is organized as follows. In section 2 we introduce comparison checking of dynamic data dependence graphs and analyze its costs. In section 3 we present optimizations to reduce the cost of comparison checking without sacrificing precision. In section 4 we present a detailed evaluation of our technique. We discuss related work in section 5 and conclude in section 6.

## 2. Comparison Checking

Comparison checking is a technique for locating faults in the scenario where two versions of a program are available – one version that is considered to be the correct version and another version that has been derived from the correct version via transformations. It was initially introduced for locating inconsistencies between unoptimized (correct) code and optimized (potentially incorrect) code to detect bugs in the optimizer by Jaramillo *et al.* [14, 15]. The values computed by the two versions of the program were *comparison checked* to locate faults.

In this section we apply the idea of comparison checking to find inconsistencies between the sequential version of a program and its parallelized version. These inconsistencies arise due to faulty parallelization performed by the programmer using a high level parallel programming model such as OpenMP [9], TBB [28], or SpiceC [11]. We observe that incorrect parallelization can lead to violation of data dependences by the parallelized version, i.e., dependences enforced by the sequential version are not preserved by the parallelized version. In particular, data dependence faults introduced during parallelization can result in data races and atomicity violations. Therefore to locate data dependence faults we can *comparison check* the data dependences exercised by the executions of the sequential and parallelized program versions on a given input. We assume that the two versions of the program are to perform the same computation and thus their dynamic dependence graphs are expected to be the same.

The executions of the *s*equential and *p*arallelized versions are characterized using their respective dynamic *D*ata *D*ependence *G*raphs, sDDG and pDDG. In dynamic data dependence graphs, nodes represent execution instances of statements (or instructions) and edges show the data dependences between nodes. By tracking reads and writes to memory locations and registers during program executions, we can construct the DDGs. The data dependences include, Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR) dependences. The formal definitions of DDGs are given below:

DEFINITION 1. *A* ***Dynamic Dependence Graph*** (DDG) *for a program execution is a directed graph* $G = (V, E)$*, where each node* $\mathtt{s_i} \in V$ *denotes the* $\mathtt{i^{th}}$ *execution instance of a statement* `s` *and each directed edge* $\mathtt{e} \in E$ *denotes a runtime data dependence from execution instance* $\mathtt{sa_i}$ *of statement* `sa` *to execution instance* $\mathtt{sb_j}$ *of statement* `sb`.

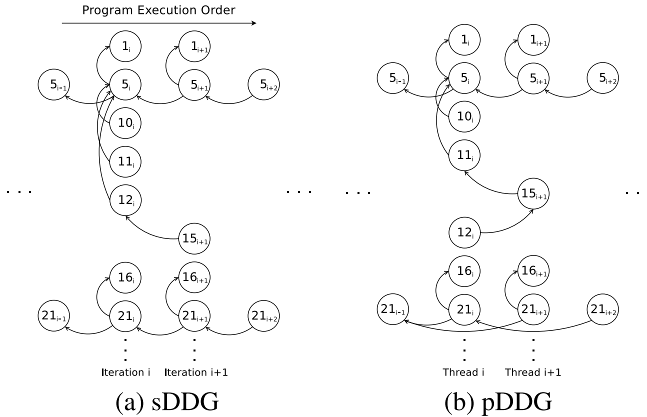
DEFINITION 2. *A* ***sequential Dynamic Dependence Graph*** (sDDG) *is a* DDG *for a sequential program run; A* ***parallel Dynamic Dependence Graph*** (pDDG) *is a* DDG *for a parallel program run.*

DEFINITION 3. *A* ***faulty*** *parallelization of a sequential program is determined if there exists a pair of* sDDG *and* pDDG *constructed using the same input that are different from each other.*

***An Example.*** Next we show how data races and atomicity violations are reflected in the form of sDDG and pDDG inconsistency. For illustration purposes we consider the `197.parser` program which is a syntactic parser for English based on link grammar. In this program, a linked dynamic data structure is employed to store dictionary information and temporary parsing results. The function `batch_process()`, called directly by the `main()` function, performs most of the detailed task of parsing and relevant

```
1: char s[MAX_LINE] = "";
2: #pragma SpiceC doacross {
3:    for(;;) {
4:       #pragma SpiceC region R1 after(ITER-1, R1) {
5:          if (fgets (s, MAX_LINE, stdin) == NULL) break;
6:          /* Other relevant code here */
7:          #pragma SpiceC commit
8:       }
9:       #pragma SpiceC region R2 {
10:         if (s[0] == '!') {
11:            special_command1 (s);
12:            special_command2 (s);
13:            continue;
14:         }
15:         if (!separate_sentence (s)) continue;
16:         print_sentence(stdout, id);
17:         /* Other relevant code here */
18:         #pragma SpiceC commit
19:      }
20:      #pragma SpiceC region R3 {
21:         fflush (stdout);
22:         /* Other relevant code here */
23:         #pragma SpiceC commit
24:      }
25:   }
26: }
```

**Figure 1.** A Motivating Example in SpiceC.




(a) sDDG (b) pDDG

**Figure 2.** DDGs for a Sequential Run and a Parallel Run.

function calls. The execution time of `batch_process()` takes up over 90% of the total execution time.

**Data races** occur when two or more threads simultaneously access the same memory location and at least one of these accesses is a memory write operation, with no locks to synchronize the accesses. Figure 1 shows an example of data races in the kernel of `197.parser` parallelized using SpiceC. We simplify this kernel for illustration purposes. Due to the presence of cross-iteration dependences on shared variables, the programmer needs to use DOACROSS parallelism and partition the loop body into three regions to parallelize this loop. Region `R1` and `R3` must be executed in sequential order due to use of shared file pointers in input and output functions `fgets()` and `fflush()` at statements 5 and 21 respectively, while region `R2` can be executed in parallel. Specifically, the programmer needs to place appropriate `after` clauses in the declaration directives of region `R1` and `R3` to allow the sequential execution order for the cross-iteration dependences on shared file pointers. Otherwise, data races will occur due to potential simultaneous writing to the same file pointer. The dependence violation can be observed by comparing sDDG and pDDG of program executions. As shown in Figure 2, the sDDG and pDDG do not match on the dependences between instances of statement 21. Instead of being data dependent upon the previous instance, each instance of statement 21 depends on instances at random distances, which indicates that a subsequent iteration may overwrite the results of previous one. Therefore, the original program semantics is not preserved by the parallelization of this loop.

**Atomicity violations** occur when the execution of a code block in one thread is interleaved by the execution of statements from other threads. The results obtained when atomicity violations occur may differ from those obtained when code blocks in different threads are executed sequentially with no interleaving. In SpiceC, atomicity violations occur when atomicity checking directives are mistakenly not introduced by the programmer. An example of atomicity violations is illustrated in Figure 1 where region `R2` is executed in parallel. Statements 11 and 12 read a shared dictionary structure, array `s[]`, while statement 15 possibly writes `s[]` when given conditions are satisfied in different threads. The reads of `s[]` need to be performed atomically to ensure statements 11 and 12 read consistent data when statement 15 writes `s[]`. If the programmer forgets to specify atomicity check in region `R2`, statements 11 and 12 may get inconsistent values of `s[]` from statement 15 in the subsequent iteration due to atomicity violations. Likewise, such a concurrency bug will be observed when comparing the sDDG and pDDG given in Figure 2. The dependence violation is found in the inconsistency of dependences among instances of statements 11, 12, and 15. In the sDDG, all instances of statements 11 and 12 depend on those of statement 5 where the value of `s[]` is initialized, while some instances of statement 12 in the pDDG depend on those of statement 15 in the subsequent iteration. The dependence violation indicates that in a parallel run one thread may read a shared variable while another thread is updating it. The inconsistent data may lead to wrong outputs or a program crash.

***Limitations of DDG based Comparison Checking.*** Next we discuss some limitations of the above comparison checking algorithm. These limitations motivate our optimized comparison checking algorithm that will be presented in the next section.

**(i) DDG Construction Overhead.** Figures 3 and 4 show the time and memory cost of DDG construction for the benchmarks in Table 1. Since DDGs record dependences between all instances of instructions, it takes considerable time and space to build them. On average, sDDG construction slows

**Table 1.** Benchmark details. From left to right: benchmark name, test case used, number of instructions executed, parallelized function, lines of code in the function, execution time percentage of the function relative to total, parallelism and parallelization system employed, and concurrency bugs introduced intentionally.

| Benchmark | Description & Test Case | Inst. Num. (in Billion) | Function | LOC | Runtime (in %) | Parallelism & Parallelized by | Faulty Versions |
|---|---|---|---|---|---|---|---|
| bodytrack | Track a 3D pose of a human with 4 cameras and 4 frames. | 15.558 | mainOMP | 48 | 97.8% | DOALL OpenMP | Data Races |
| freqmine | Perform frequent itemset mining in 990000 click streams. | 36.978 | FP_growth_first | 140 | 98.1% | DOALL OpenMP | Data Races |
| 256.bzip2 | Compress a 207 KB file. | 0.139 | compressStream | 130 | 98.9% | DOACROSS SpiceC | Data Races Atomicity Violations |
| CRC32 | Calculate 32-bit CRC for 10 files of total size 3.065 MB. | 0.236 | main | 13 | 100% | DOACROSS SpiceC | Data Races Atomicity Violations |
| Barnes-Hut | Simulate gravitational forces acting on 10000 bodies. | 4.606 | main | 92 | 100% | DOALL SpiceC | Data Races |
| 197.parser | Analyze structure and grammar of a 35.6 KB file. | 7.324 | batch_process | 284 | 91.9% | DOACROSS SpiceC | Data Races Atomicity Violations |
| ferret | Perform content-based similarity search in 34973 images. | 28.984 | do_query | 86 | 99.1% | Pipeline SpiceC | Data Races |
| DelaunayRefinement | Refine an initial Delaunay mesh of 100770 triangles. | 165.527 | read | 59 | 10.3% | DOACROSS SpiceC | Data Races Atomicity Violations |
| swaptions | Price a portfolio of 64 swaptions with 20000 simulations. | 12.609 | main | 202 | 100% | DOALL TBB | Data Races |
| streamcluster | Perform an online clustering algorithm on 16384 points. | 19.036 | pkmedian | 143 | 99.8% | DOALL TBB | Data Races |

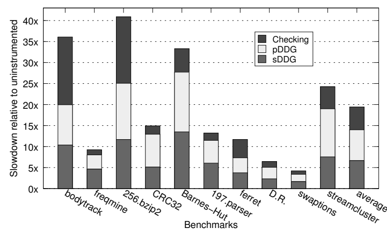



**Figure 3.** DDG: Time Overhead.

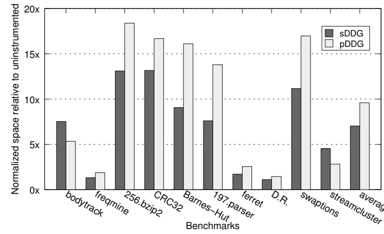



**Figure 4.** DDG: Memory Overhead.

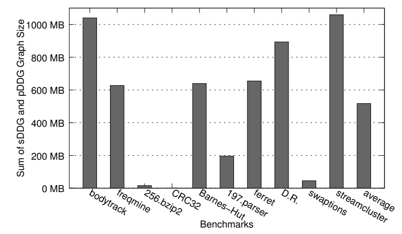



**Figure 5.** DDG: Graph Size.

down the execution of the sequential version by a factor of 6.67x and the memory consumption goes up by a factor of 7.03x. For constructing pDDGs, on average, the time slowdown is 7.32x relative to the execution of the uninstrumented parallelized version, since tracking of data accesses must be performed atomically. The memory consumption goes up by a factor of 9.59x in comparison to the parallel execution without instrumentation.

**(ii) Graph Size and Checking Time.** The graph sizes of sDDG and pDDG grow with the length of program runs; thus, they can become very large. As a consequence, comparison checking sDDG with pDDG can also take a significant amount of time. From Figure 5 we can see that the graph size reaches 1 GB for two programs. Figure 3 shows that the average checking time is 5.44x of the execution time of the uninstrumented version.

**(iii) Reproducibility Rate.** Because of the non-deterministic interleaving in parallel program runs, it may require considerable amount of runs to expose a concurrency bug (an atomicity violation in particular). For example, it has been reported that the reproducibility rate of Apache Qpid client hanging bug while joining threads is only around 10% [2]. It took 22 hours to manifest an atomicity violation from running the Apache HTTPd Server [26]. The small probability for particular interleaving of exposing concurrency bugs further increases the cost of fault localization.

**(iv) Validity of Comparing Two Runs.** Some programs depend on random numbers that may alter the control flow arbitrarily in different runs regardless of the program input [13]. For such programs the inconsistency between sDDG and pDDG may be wrongly viewed as a bug.

## 3. Optimizing Comparison Checking

In this section we present a series of three optimizations that one by one will overcome the limitations of the DDG based approach described above. Table 2 shows the benefits of the optimizations in reducing the cost of comparison checking. In addition, the final optimization (OPT-3) also eliminates the last two limitations of the DDG approach, i.e., *reproducibility rate* and *validity of comparing two runs*.

**Table 2.** Reduction Achieved by Three Optimizations.

| Optimization | Execution Time | | Memory Space | |
|---|---|---|---|---|
| | Tracking | Checking | Tracking | Graph Size |
| OPT-1 | √ | √ | √ | √ |
| OPT-2 | √ | √ | | √ |
| OPT-3 | √ | | √ | |

### 3.1 OPT-1: Region Graphs

The first optimization we present in this section performs two tasks. First, it eliminates the collection and representation of *irrelevant dependences* that have nothing to do with the parallelized portions of the program. Second, a compact dependence graph is constructed in which nodes correspond

to code blocks or *regions* instead of individual statements (or instructions). This is possible because the parallelization is typically performed at the granularity of code regions and thus it is sufficient to capture dependences between regions.

Next we show how the above approach is applied in the context of DOALL, DOACROSS, and Pipeline loops. Figure 6 shows that the control flow graph of a sequential loop as well as the graph after parallelization have been applied for all three types of parallelism. By comparing the sequential and parallelized control flow graphs, we derive the sets of dependences between code regions that *must be comparison checked* for the parallelization to be correct.

***DOALL.*** Given a loop parallelized using the DOALL construct as shown in Figure 6, the comparison checked dependences from the sequential and parallelized versions must include RAW, WAW, and WAR dependences between different instances of the same region $R_k$, i.e. ($R_k^i$, $R_k^j$), where $i$ and $j$ are distinct region instance numbers ($1 \leq (i, j) \leq n$) and $n$ is the number of iterations in the loop.

***DOACROSS.*** A body of a loop parallelized using the DOACROSS construct contains three regions ($R_1, R_2, R_3$) where the parallelism is captured by the middle region (see Figure 6). The comparison checked dependences from the sequential and parallelized versions must include RAW, WAW, and WAR dependences between following region instances: ($R_{2/3}^i$, $R_{1/2}^j$), where $i < j$ and $R_{1/2}^j$ depends on $R_{2/3}^i$ in sequential version of the program, i.e., $R_{1/2}^j \rightarrow R_{2/3}^i$.

***Pipelining.*** A loop parallelized using Pipeline construct as shown in Figure 6. The comparison checked dependences from the sequential and parallelized versions must include RAW, WAW, and WAR dependences between instances of following *different* regions and *different* loop iterations: ($R_j^k$, $R_i^{k+1..n}$), where $i < j$ and $R_i^{k+1..n}$ depend upon $R_j^k$ in the sequential version of the program, i.e., $R_i^{k+1..n} \rightarrow R_j^k$.

We can see that according to the above rules, not only are relevant subset of all dynamic data dependences comparison checked, they are also represented at the granularity of regions. Instead of building and comparing DDGs at instruction instance level, we propose to identify DDG inconsistency by constructing and checking two coarse-grained graphs at region instance level. In a *Region Graph* (RG), each node stands for a code region that denotes a code block annotated by parallel constructs and each edge represents a dependence between region instances. A *Dynamic Region Graph* (DRG) is constructed by profiling executions of the sequential or parallelized version of a program to capture dynamically arising dependences between region instances. By comparing DRGs of running both versions, we can locate dependence violation at region instance level that reflects faulty parallelization. This representation is clearly more compact. Below we define region graphs formally.

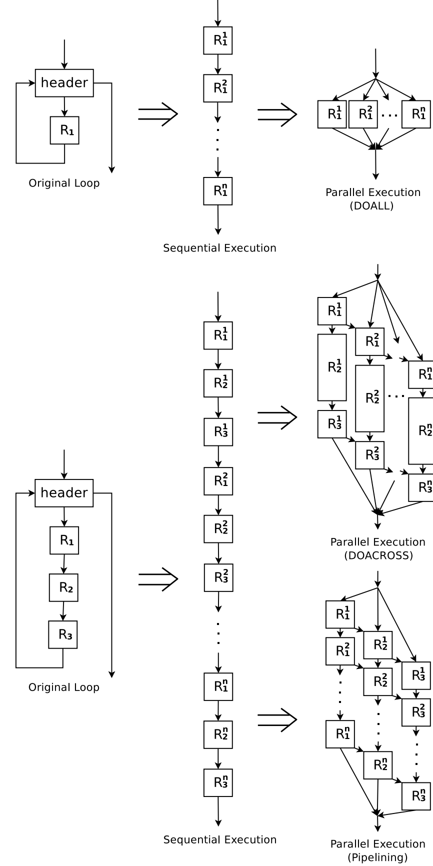



**Figure 6.** Dependence Tracking and Checking for DOALL, DOACROSS, and Pipeline Parallelism.

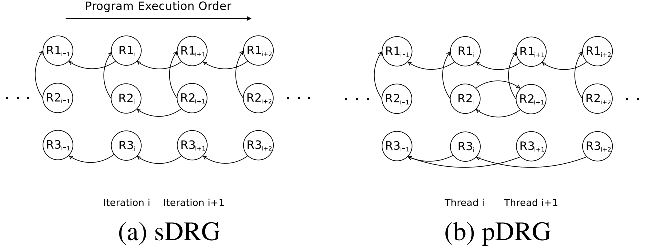
**DEFINITION 4.** ***Intra-iteration dependences*** *are* RAW, WAW, *and* WAR *dependences from an instruction instance in a region instance* $R_j$ *to an instruction instance in a region instance* $R_i$*, where* $R_j$ *follows* $R_i$ *in program execution order* ($j \geq i$)*, within the same loop iteration.*

**DEFINITION 5.** ***Cross-iteration dependences*** *are* RAW, WAW, *and* WAR *dependences from an instruction instance in a region instance* $R_j$ *of the* $t^{th}$ *loop iteration to an instruction instance in a region instance* $R_i$ *of the* $s^{th}$ *loop iteration, where* $t > s$.

**DEFINITION 6.** *A* ***Region Graph*** (RG) *is a directed graph* $G = (V, E)$*, where*: ***V*** *is a set of* ***vertices****, each of which identifies one region instance in a loop of a program parallelized using parallel constructs; and* ***E*** *is a set of* ***edges****, where each edge identifies a data dependence between region instances.*

**DEFINITION 7.** *A* ***Dynamic Region Graph*** (DRG) *is an* RG *where each edge represents a dynamic data dependence that arises in a program execution. A* ***sequential Dynamic Region Graph*** (sDRG) *is a* DRG *for a sequential program run; A* ***parallel Dynamic Region Graph*** (pDRG) *is a* DRG *for a parallel program run.*

The data races and atomicity violations can be located by comparing only data dependences at region instance level.




(a) sDRG (b) pDRG

**Figure 7.** DRGs for Two Runs of Figure 1.

Figure 7 shows the sDRG and pDRG pair for the motivating example given in Figures 1 and 2. We observe that in the sDRG an instance of `R3` depends on the region instance executed immediately before it, while in the pDRG some instances of `R3` depend on earlier region instances at an arbitrary dependence distance. This is because there are no parallel constructs specifying the sequential execution order of `R3`. The faulty parallelization causes data races in parallel runs and is reflected as an inconsistency in the sDRG and pDRG pair. Further, the sDRG is not equivalent to the pDRG due to an additional $\mathtt{R2_i} \rightarrow \mathtt{R2_{i+1}}$ edge in the pDRG. This edge reveals that in the sequential runs, cross-iteration dependences only occur from subsequent region instances to preceding ones, while in the parallel runs, cross-iteration dependences can occur in both directions. Such additional edges result when a write to a shared variable interleaves the execution of two consecutive reads of the same variable without atomicity checking directive in `R2`. Thus, the second read obtains inconsistent data revealing atomicity violation.

**THEOREM 1 (Correctness I).** *Given a parallelized program $p$ and an input $x$,* sDRG($p$, $x$) $\equiv$ pDRG($p$, $x$) $\Longleftrightarrow$ sDDG($p$, $x$) $\equiv$ pDDG($p$, $x$).

*Proof.* We prove the sufficiency and necessity separately.

***Sufficiency.*** The edges in a DDG can be divided into two categories: inter-region and intra-region edges. Inter-region edges represents data dependences between region instances while intra-region edges represents those within region instances. Given the premise that the sDRG($p$, $x$) is equivalent to the pDRG($p$, $x$), we can infer that all dynamic data dependences between region instances during the sequential execution of $p$ on the input $x$ matches their counterparts during the parallel execution of $p$ on the input $x$. That is, at region instance level, the original sequential program semantics is preserved. Therefore, the inter-region edges in the sDDG($p$, $x$) will definitely appear in the pDDG($p$, $x$). As for the intra-region edges, they will not change since the data dependences coming from outside the region instance are the same. Thus the intra-region edges in the sDDG($p$, $x$) will definitely appear in the pDDG($p$, $x$) as well. Finally, all edges in the sDDG($p$, $x$) appear in the pDDG($p$, $x$), which concludes sDDG($p$, $x$) $\subseteq$ pDDG($p$, $x$).

On the other hand, since all dynamic data dependences in the sequential run are preserved, all the branches in the sequential run will produce the same outcomes as those in the parallel run. Therefore, the control flows are the same in both runs. As both runs go through the same execution path, the total number of data dependences must be the same in both runs, i.e., $|$sDDG($p$, $x$)$| \equiv |$pDDG($p$, $x$)$|$.

With current inferences sDDG($p$, $x$) $\subseteq$ pDDG($p$, $x$) and $|$sDDG($p$, $x$)$| \equiv |$pDDG($p$, $x$)$|$, we can conclude that sDDG($p$, $x$) $\equiv$ pDDG($p$, $x$).

***Necessity.*** Given the premise that sDDG($p$, $x$) $\equiv$ pDDG($p$, $x$), we can infer that all dynamic data dependences between region instances during the parallel execution of $p$ are always the same as those arising during the sequential execution of $p$. This indicates that the original sequential program semantics at region instance level is not altered in the parallel run since data dependences between region instances are a subset of all data dependences, i.e., sDRG($p$, $x$) $\equiv$ pDRG($p$, $x$). Otherwise, there must exist a parallel run that has different dynamic data dependences between region instances from the corresponding sequential run. □

**LEMMA 1 (Basic Determination of Faulty Parallelization).** *Given a parallelized program p, when $\exists$ an input $x$,* sDRG($p$, $x$) $\neq$ pDRG($p$, $x$)*, the parallelization of p is faulty.*

*Proof.* From THEOREM 1, we can easily infer that given a parallelized program $p$ and an input $x$, sDRG($p$, $x$) $\equiv$ pDRG($p$, $x$) $\Longleftrightarrow$ sDDG($p$, $x$) $\equiv$ pDDG($p$, $x$). The inverse negative proposition of the theorem is also true, i.e., $\exists$ $p$, $x$ such that sDDG($p$, $x$) $\neq$ pDDG($p$, $x$) $\Longleftrightarrow$ sDRG($p$, $x$) $\neq$ SRG($p$). According to DEFINITION 3, when $\exists$ $p$, $x$ such that sDDG($p$, $x$) $\neq$ pDDG($p$, $x$), the parallelization of $p$ is faulty. This concludes the proof of the lemma. □

Therefore, we can employ the coarse-grained region graph approach to locate concurrency bugs instead of using the fine-grained DDG approach without loss of accuracy.

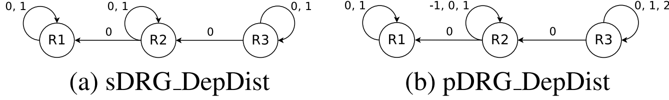
### 3.2 OPT-2: Summarizing Region Instances

While DRGs are more compact than DDGs, still the size of a DRG grows with the length of a program run as each execution instance of a code region must be represented by a distinct node in the DRG. In this section, we show that we can summarize all the dynamic data dependences involving a region such that we do not need to explicitly distinguish between its execution instances, i.e., a single node in a DRG represents all execution instances of a region. This summarization however requires that all different dependences encountered during the execution of a region are remembered. This is achieved by annotating each edge by a dynamic *dependence distances*. The value of a dependence distance of 0 indicates an intra-iteration dependence and a non-zero value indicates a cross-iteration dependence.

**DEFINITION 8.** *Given a dependence* `d` *in a region graph from one region instance in the* $\mathtt{i^{th}}$ *iteration of a loop to the*

*other in the* $\mathtt{j}^{\mathtt{th}}$ *iteration, a* ***dependence distance*** `dist` *is an integer that equals to* $\mathtt{i} - \mathtt{j}$.

A simplified version of Figure 7 according to this optimization is shown in Figure 8. It should be noted that the two simplified DRGs are not equivalent due to inconsistency related to R2 and R3. Specifically, R2 of the pDRG has an additional edge $\mathtt{R2} \xrightarrow{-1} \mathtt{R2}$ that reveals a cross-iteration dependence with a dependence distance of -1. Besides, R3 of the pDRG has additional $\mathtt{R3} \xrightarrow{2} \mathtt{R3}$ edges that reveals cross-iteration dependences with a dependence distance of 2.




(a) sDRG_DepDist (b) pDRG_DepDist

**Figure 8.** DRGs with Dependence Distances for Two Runs of Figure 1.

Next we show that by using dependence distances, the correctness of the region graph approach is preserved, since the use of dependence distances is sufficient to capture the faulty parallelization caused by dependence faults.

**THEOREM 2 (Correctness II).** *Given a parallelized program* $p$ *and an input* $x$, $\text{sDRG}'(p, x) \equiv \text{pDRG}'(p, x) \Longleftrightarrow \text{sDRG}(p, x) \equiv \text{pDRG}(p, x)$, *where* sDRG′ *and* pDRG′ *denote* sDRG *and* pDRG *with dependence distances respectively.*

*Proof.* We prove the sufficiency and necessity separately.

***Sufficiency.*** We prove it by contradiction. Let us assume that for a program $p$ and an input $x$, $\text{sDRG}'(p, x) \equiv \text{pDRG}'(p, x)$ but there exists a parallel execution that $\text{sDRG}(p, x) \neq \text{pDRG}(p, x)$. Thus there must exist at least one pair of dependences between $\text{sDRG}(p, x)$ and $\text{pDRG}(p, x)$ that do not match. Assume that a load in region instance $\mathtt{R}_{\mathtt{x}}^{\mathtt{i}}$ reads a value from region instance $\mathtt{R}_{\mathtt{y}}^{\mathtt{j}}$ (i.e., causing a dependence $\mathtt{R}_{\mathtt{y}}^{\mathtt{j}} \rightarrow \mathtt{R}_{\mathtt{x}}^{\mathtt{i}}$) in the sequential run but reading from another region instance $\mathtt{R}_{\mathtt{z}}^{\mathtt{k}}$ (i.e., $\mathtt{R}_{\mathtt{z}}^{\mathtt{k}} \rightarrow \mathtt{R}_{\mathtt{x}}^{\mathtt{i}}$) in a parallel run. Since $\mathtt{R}_{\mathtt{y}} \xrightarrow{\mathtt{j-i}} \mathtt{R}_{\mathtt{x}}$ exists in $\text{sDRG}'(p, x)$ and $\text{sDRG}'(p, x) \equiv \text{pDRG}'(p, x)$, then $\mathtt{R}_{\mathtt{y}} \xrightarrow{\mathtt{j-i}} \mathtt{R}_{\mathtt{x}}$ must also exist in $\text{pDRG}'(p, x)$. Therefore, there must exist another parallel run where $\mathtt{R}_{\mathtt{x}}^{\mathtt{i}}$ reads from $\mathtt{R}_{\mathtt{y}}^{\mathtt{j}}$. Since $\mathtt{R}_{\mathtt{x}}^{\mathtt{i}}$ could depend on different region instances (e.g., $\mathtt{R}_{\mathtt{y}}^{\mathtt{j}}$ or $\mathtt{R}_{\mathtt{z}}^{\mathtt{k}}$) in different runs, this means that the interleaving of region instances of $\mathtt{R}_{\mathtt{x}}$, $\mathtt{R}_{\mathtt{y}}$, and $\mathtt{R}_{\mathtt{z}}$ is non-deterministic. Due to the non-deterministic nature of parallel executions, in a parallel run region instance $\mathtt{R}_{\mathtt{x}}^{\mathtt{i}}$ could actually be dependent on any instance of $\mathtt{R}_{\mathtt{y}}$ and $\mathtt{R}_{\mathtt{z}}$. After summarizing the dependences, $\text{pDRG}'(p, x)$ can have any edge $e \in \{\mathtt{R}_{\mathtt{z}} \xrightarrow{\mathtt{l-i}} \mathtt{R}_{\mathtt{x}}\}$, where `l` could be any iteration number. Since $\text{sDRG}'(p, x)$ only has a limited number of edges from $\mathtt{R}_{\mathtt{z}}$ to $\mathtt{R}_{\mathtt{x}}$, there must be a parallel execution producing an edge $e \in \{\mathtt{R}_{\mathtt{z}} \xrightarrow{\mathtt{l-i}} \mathtt{R}_{\mathtt{x}}\}$ that does not exist in $\text{sDRG}'(p, x)$. Therefore, $\text{sDRG}'(p, x) \neq \text{pDRG}'(p, x)$ for certain parallel execution, which leads to a contradiction. This concludes the proof of the sufficiency.

**Table 3.** Dependences Allowed by Parallel Constructs in OpenMP, SpiceC, and TBB.

| Construct | Allowed Dependences |
|---|---|
| **OpenMP** | |
| `parallel [for|do]` | Intra-iteration dependences |
| `section` | Intra-iteration dependences |
| `critical` | Intra-iteration/cross-iteration dependences |
| `ordered` | Intra-iteration/cross-iteration dependences |
| **SpiceC** | |
| `doall` | Intra-iteration dependences |
| `doacross` | Intra-iteration dependences |
| `pipelining` | Intra-iteration/cross-iteration dependences |
| `after(ITER-x,` $\mathtt{R}_{\mathtt{y}}$`)` | Intra-iteration/cross-iteration dependences from region $\mathtt{R}_{\mathtt{y}}$ to current region with a distance `x` |
| `atomicity_check` | Intra-iteration/cross-iteration dependences |
| **TBB** | |
| `parallel_for` | Intra-iteration dependences |
| `parallel_reduce` | Intra-iteration dependences and cross-iteration dependences of `join` |
| `parallel_scan` | Intra-iteration dependences and cross-iteration dependences of `reverse_join` |
| `parallel_pipeline` | Intra-iteration dependences and cross-iteration dependences of `filter` |

***Necessity.*** Given $\text{sDRG}(p, x) \equiv \text{pDRG}(p, x)$, we can infer that all dependences at region instance level match. It is obvious that the corresponding dependence distances are the same. Thus sDRG′ and pDRG′ are equivalent. □

### 3.3 OPT-3: Static Region Graph

Previous comparison checking techniques, as well as what we have described so far, require that both versions of the program be instrumented and executed [14, 15]. In this section, we show that comparison checking can be accurately performed by only executing the sequential version. The execution of the parallelized version is not required because a precise region graph can be constructed for the parallelized version by statically analyzing the parallel constructs inserted. This optimization not only reduces the time and memory cost of our technique, it also eliminates additional limitations of the DDG based method.

Since the program semantics is reflected by the sequential run and the parallelization semantics is specified by the parallel constructs, the execution of the parallelized version is not necessary. Our approach only executes the sequential version of the program and reports real bugs that are revealed by the given sequential execution if it exercises data dependences that are prohibited by the parallel constructs used in the parallelization. Previous concurrent debugging works [21] have shown that data races can be exposed by enumerating different interleaving at accesses to synchronization variables. Our approach analyzes the parallel constructs to find if any possible interleaving at the region boundaries (i.e., accesses to synchronization variables) will violate the data dependences in the sequential run. Table 3 lists the allowed data dependences for core parallel constructs from OpenMP, SpiceC, and TBB. All these constructs are designed to parallelize loops. Each of these constructs enforces certain execution ordering, which allows some data dependences from the sequential run to be retained in the parallel run. We can easily construct a *Static Region Graph*

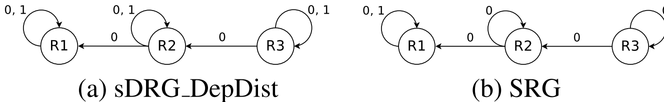
for a parallelized program conforming to the constraints in the table.

**DEFINITION 9 (Transitivity).** *If both dependences* $\mathtt{R_i} \xrightarrow{\mathtt{s}} \mathtt{R_k}$ *and* $\mathtt{R_k} \xrightarrow{\mathtt{t}} \mathtt{R_j}$ *are allowed to occur in program executions, then the dependence* $\mathtt{R_i} \xrightarrow{\mathtt{s+t}} \mathtt{R_j}$ *is also allowed.*

**DEFINITION 10.** *A* ***Static Region Graph*** (SRG) *is an* RG *where each edge represents a data dependence allowed to occur in program executions by either a parallel construct inserted, as given in* Table 3*, or* DEFINITION 9.

**LEMMA 2 (Optimized Determination of Faulty Parallelization).** *Given a parallelized program* $p$*, when* $\exists$ *an input* $x$*,* sDRG($p$, $x$) $\nsubseteq$ SRG($p$)*, the parallelization of* $p$ *is faulty.*

*Proof.* Given sDRG($p$, $x$) $\nsubseteq$ SRG($p$), we can infer there exists at least one edge $e$ in sDRG($p$, $x$) but not in SRG($p$). Now we need to prove there exists at least one parallel execution where the edge $e$ is not in pDRG($p$, $x$). We know an SRG consists of all allowed data dependence at runtime expressed by a parallel construct. If $e$ is not in SRG($p$), we know there is no such a parallel construct in the parallelized program $p$ to enforce (i.e., allow) the dependence $d$ represented by $e$. Therefore, due to the non-deterministic interleaving in parallel runs, $d$ may not occur in parallel executions of $p$. That is, there exists at least one parallel run where $e$ is not included in pDRG($p$, $x$). Thus there exists at least one edge $e$ in sDRG($p$, $x$) but not in pDRG($p$, $x$), i.e., sDRG($p$, $x$) $\neq$ pDRG($p$, $x$). From LEMMA 1, we conclude the parallelization of $p$ must be faulty. □




(a) sDRG_DepDist (b) SRG

**Figure 9.** sDRG with Dependence Distances and SRG.

Figure 9 illustrates the same example as in Figure 8 with pDRG replaced by SRG. The sDRG is not equivalent to the SRG since R2 and R3 of the sDRG have additional $\mathtt{R2} \xrightarrow{1} \mathtt{R2}$ and $\mathtt{R3} \xrightarrow{1} \mathtt{R3}$ edges respectively that reveals cross-iteration dependences, while there are no corresponding edges in the SRG due to no parallel constructs performing atomicity checking in R2 and specifying the sequential execution order of R3.

Now let us briefly discuss why the above approach eliminates two additional limitations of the DDG based approach:

- **Reproducibility Rate.** Because the parallelized version of the program is no longer executed, this problem does not arise; and
- **Validity of Comparing Two Runs.** Some programs depend on random numbers that may alter the control flow arbitrarily in different runs regardless of the program input [13]. However, since we do not run the parallelized version, this problem is also eliminated.

# 4. Evaluation

We have implemented our approach for comparison checking as summarized in Figure 10. We used the Intel's dynamic binary instrumentation tool Pin [18] to track memory and register read and write instructions during runs of the sequential program. We construct an sDRG by profiling memory and register accesses and source code locations during a sequential run. Data dependences are maintained at region level. The construction of SRGs is based upon static analysis of parallel constructs in the parallelized version. When a corresponding sDRG and SRG pair is ready, inconsistency checking is carried out by comparing the corresponding vertices, edges, and labels between two graphs.

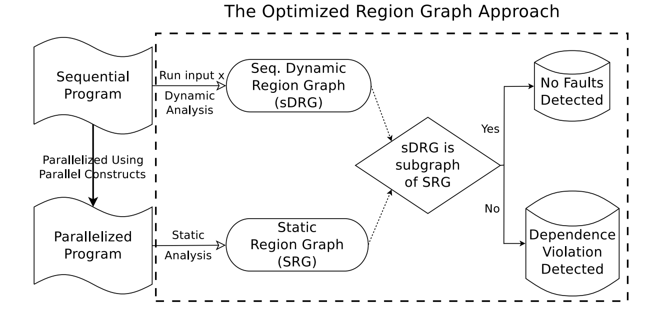



**Figure 10.** Overview of Our Implementation.

As expected, our approach successfully identifies the injected bugs that result in data dependence faults. Therefore, in the remainder of this section, we focus on evaluating our approach in terms of its efficiency. We applied our technique to ten benchmarks parallelized by OpenMP, SpiceC, and TBB to assess the costs of our technique and the benefits of using the optimizations introduced. All experiments were performed on a 2-core 2.66 GHz Intel Core Duo DELL Dimension 9200 machine with 4 GB RAM running Linux kernel 2.6.32. Time was measured using C++ time functions and GNU tool gprof. All results were normalized with respect to the time and space consumed by the uninstrumented version of each benchmark. Benchmarks used were selected from MiBench, SPEC CPU2000, Lonestar, and PARSEC suites. Table 1 shows the details of these benchmarks.

## 4.1 Overhead of the Region Graph Approach

We measured the overhead of our approach in three respects: *time overhead* in terms of the slowdown experienced by the sequential program run due to execution of instrumentation code and performing comparison checking; *space overhead* in terms of increase in memory used by the program at runtime because of tracking dynamic dependences; and *graph size* in terms of memory needed to hold the sDRG constructed by running the sequential program. These results are presented in Figures 11, 12, and 13 respectively. From Figure 11 we can see that, on average, the execution of sequential program slows down by a factor of 3x. From Figure 12 we can see that, on average, the memory used during the sequential program run increases by a factor of 6.55x.

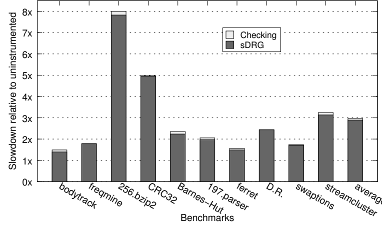



**Figure 11.** DRG: Time Overhead.

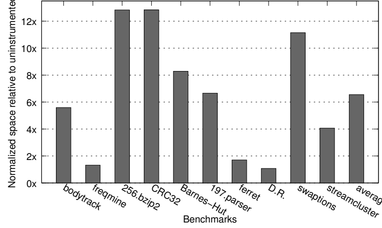



**Figure 12.** DRG: Memory Overhead.

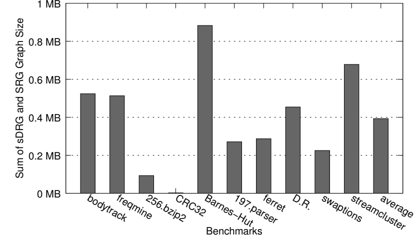



**Figure 13.** DRG: Graph Size.

The above time and memory costs are mainly the costs of tracking dependences using Pin and are therefore quite reasonable. Finally, from Figure 13 we can see that the sizes of sDRGs do not exceed 1 MB for these benchmarks. This graph is quite small and can be easily kept in memory. Moreover, it explains why the execution time cost of comparison checking the sDRG and SRG is very small in comparison to the cost of building the sDRG as can be seen from Figure 11.

## 4.2 DRG vs. DDG

While the cost of our fully optimized region graph approach is quite small, recall that the cost of the original DDG based approach is very high. This is because it requires the execution of instrumented versions of both sequential and parallelized versions of the program. Moreover, since the DDGs constructed are much larger than DRGs, the memory consumption for holding DDGs as well as the execution time spent on comparison checking them are also very high. Table 4 presents the final cost of our sDRG+SRG based approach as a percentage of using the original sDDG+pDDG based approach. As we can see, both time and memory costs of comparison checking are greatly reduced by our approach.

**Table 4.** Cost of Using DRG as a Percentage of DDG.

| Benchmark | Execution Time | | Memory Space | |
|---|---|---|---|---|
| | Tracking | Checking | Tracking | Graph Size |
| bodytrack | 4.487% | 0.371% | 45.765% | 0.049% |
| freqmine | 14.416% | 0.586% | 51.985% | 0.082% |
| 256.bzip2 | 17.594% | 0.623% | 46.155% | 0.584% |
| CRC32 | 22.854% | 0.694% | 47.240% | 1.923% |
| Barnes-Hut | 5.819% | 1.519% | 32.463% | 0.139% |
| 197.parser | 9.474% | 2.632% | 34.419% | 0.138% |
| ferret | 12.713% | 1.067% | 33.196% | 0.044% |
| DelaunayRefinement | 33.483% | 0.371% | 71.006% | 0.051% |
| swaptions | 32.437% | 1.976% | 36.379% | 0.494% |
| streamcluster | 9.884% | 1.282% | 39.001% | 0.064% |
| **GeoMean** | **13.463%** | **0.907%** | **42.534%** | **0.151%** |

The substantial savings of our approach shown above can be broken down to further observe the effectiveness of the three optimizations we presented. This breakdown is shown in Figure 14 – the four bars for each benchmark correspond to the Tracking Time (TT), the Checking Time (CT), the Tracking Memory (TM), and the Graph Size (GS). Next we analyze the impact of the optimizations on overhead costs as demonstrated by Figure 14.

**Tracking Time.** OPT-1 and OPT-3 are responsible for most of the reductions in TT. This is because OPT-1 eliminates the tracking of irrelevant dependences corresponding to those parts of the program that are executed sequentially. On the other hand, OPT-3 eliminates the need to run the parallelized version of the program and hence eliminates the corresponding TT. Finally, the reductions in TT due to OPT-2 result because the runtime overhead of maintaining the dynamically observed dependences at region instance level is simpler than maintaining them at statement instance level.

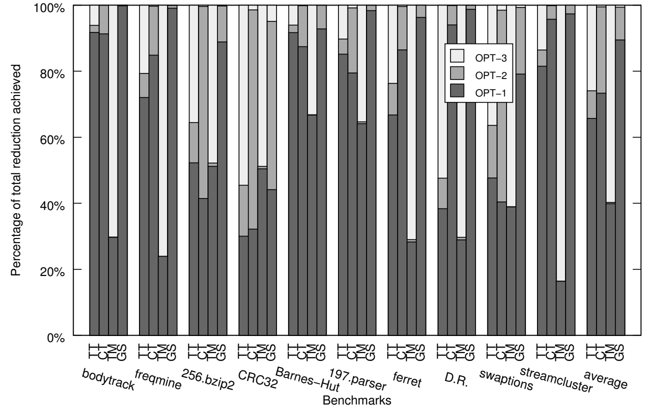



**Figure 14.** Breakdown of Total Reduction Achieved.

**Checking Time.** OPT-1 reduces the checking time as it eliminates collection, and hence checking of a significant fraction of dynamic data dependences. OPT-2 reduces checking time further as it summarizes dynamic data dependences to region level and hence greatly reduces the number of checks to be performed. Finally, OPT-3 does not reduce checking time as it does the number of checks performed is unaffected. OPT-3 does not alter the sDRG with dependence distance produced by OPT-2; thus, checking each edge in sDRG with dependence distance takes the same time.

**Tracking Memory.** For detecting all instances and forms of dynamic data dependences (i.e., RAW, WAW, and WAR) we need to buffer multiple reads to each memory address. In particular, multiple reads to a memory address must be buffered till a write to the address is encountered so that all WAR dynamic data dependences can be detected. The buffering of above information takes up significant amount of memory. The use of this memory is reduced significantly by OPT-1 because all reads and writes need not be tracked. OPT-3 further reduces this overhead because OPT-3 eliminates this overhead by eliminating the need to dynamically track data dependences in the parallel execution. OPT-2 has the potential of reducing the number of buffered reads and

writes by buffering at most one read or write to a memory address corresponding to specific execution instance of a region. However, we observed very small reductions because typically a region instance does not perform many reads and writes for the same address.

**Graph Size.** OPT-1 reduces the graph size because it only requires capture and hence representation of only a subset of dynamic data dependences. OPT-2 further reduces the graph size because it summarizes multiple dynamic dependences by representing them at region level and using dynamic dependence distances. In programs where sequential part of the computation is substantial, OPT-1 is more useful. On the other hand, in programs where most of execution has been parallelized, OPT-2 is more useful. Finally, OPT-3 does not have any impact on the graph size.

In summary, we observe that the savings in TT and TM primarily result from OPT-1 and OPT-3 while CT and reductions in GS primarily benefit from OPT-1 and OPT-2.

## 5. Related Work

**Comparison Checking.** Comparison checking [14, 15] was introduced to see if two versions of a program behave differently. It was used to compare the runtime values computed by the unoptimized and optimized versions of a program to determine if an error existed in the unoptimized code or was introduced by the optimizer. More recently it has been observed that this approach can be helpful for parallel programs [25]. In our work, we compare executions of the sequential and parallelized versions of the code. A major difference is that we require only one (sequential) version of the program to be executed while traditional comparison checking requires execution and profiling of both versions of the program. Chen *et al.* [7] extracted program intentions of message passing to check whether the intentions are fulfilled correctly by the underlying MPI libraries for detecting bugs in MPI libraries. In a similar approach proposed in [8], firstly memory accesses in the MPI applications and corresponding message transition operations in the MPI library were tracked, and then checking was performed between the MPI application and the MPI library to determine whether the correct execution order was guaranteed.

**Locating Faults in Parallelized Programs.** Several approaches have been proposed to locate faults in programs parallelized using OpenMP [9] and TBB [28]. Due to wide support on most processor architectures and operating systems, OpenMP has been widely used to parallelize sequential programs on shared memory systems. Thus there are a lot of commercial debuggers supporting debugging OpenMP programs, such as TotalView [3], VivaMP static analyzer [4], and Intel Parallel Inspector [1]. They are similar to traditional debuggers and provide the ability to control program execution and track values of variables. These debuggers can also detect some data races through static analysis. Some other static methods has been proposed for fault localization for OpenMP via use of formal verification. Basupalli *et al.* proposed a static verifier based on the polyhedral model that can only detect data races [5]. Süß *et al.* [30] have discussed common mistakes in OpenMP programs but no effective approaches and tools have been reported. In comparison to these debugging techniques, our approach is more general. We can detect all kinds of data dependence related bugs, including data races and atomicity violations. Our approach also helps developers fix bugs since it identifies the dependences that violate the user inserted parallel constructs. Our technique applies to OpenMP, SpiceC and TBB.

Recently there has been significant research on *speculative parallelization* of programs to exploit dynamic parallelism present in many applications that cannot be detected via static analysis [10, 11, 17, 31, 32]. Since parallelization is performed assuming absence of certain infrequently arising data dependences, misspeculation detection and recovery mechanisms are provided to handle the situation when these dependences are encountered during execution. Therefore the misspeculation detection is very much like data dependence violation detection presented in this paper. However, unlike speculative parallelization we do not require support to enable recovery as our approach is meant for debugging. Moreover, our solution is based upon running the sequential version of the program which overcomes the bug reproducibility problem.

**Locating Faults in Parallel Programs.** There are a large number of techniques and specialized tools released for locating concurrency bugs like data races, atomicity violations, order violations, and deadlocks for parallel programs. Some approaches are based on static analysis [21, 23, 27]. Runtime techniques have also been proposed for detecting concurrency bugs, such as data races from Eraser [29], LiteRace [19], and FastTrack [12], and atomicity violations from AFix [16] and CTrigger [26]. Some hybrid approaches have also been proposed. For example, data race detection in [24, 33, 34] combined lockset and happens-before techniques, and HAVE [6] integrated static and dynamic analysis. In addition, DRFx [20] proposed to use hardware for efficiently detecting data races. Our approach is different from the above work as it leverages the availability of two program versions and only executes the sequential version.

## 6. Conclusions

This paper proposes an efficient and precise approach for locating data races and atomicity violations in programs parallelized by introducing parallel constructs. This approach uses region level data dependence graphs and avoids the execution of the parallelized program to achieve efficiency. Comparison checking is achieved by comparing the Static Region Graph of the parallelized version (SRG) and the Dynamic Region Graph of the sequential version (sDRG). The inconsistency between SRG and sDRG in terms of dependence violation reflects faulty parallelization. The evaluation

shows that on average our approach slows down the original executions by a factor of $3\times$. This work is novel in two respects. First, comparison checking is performed without running both versions of the program. This enables us to deal with situations where the two versions of the program may behave differently due to use of features such as random number generators. Second, we debug the parallelized version of the program by executing the sequential version. This enables us to deal with the problem of reproducing bugs.

## Acknowledgments

We thank Prof. Todd Austin and the reviewers for their helpful comments and advice on improving this paper. This work is supported by NSF Grant CCF-0963996 to University of California Riverside.

## References

[1] http://software.intel.com/en-us/articles/intel-parallel-inspector/.

[2] https://bugzilla.redhat.com/show_bug.cgi?id=598948.

[3] http://www.roguewave.com/products/totalview-family/totalview.aspx.

[4] http://www.viva64.com/en/vivamp-tool/.

[5] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompVerify: Polyhedral analysis for the OpenMP programmer. In *IWOMP* 2011.

[6] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Detecting atomicity ciolations via integrated dynamic and static analysis. In *ETAPS*, pages 425–439, 2009.

[7] Z. Chen, Q. Gao, W. Zhang, and F. Qin. FlowChecker: Detecting bugs in MPI libraries via message flow checking. In *SC*, pages 1–11, 2010.

[8] Z. Chen, X. Li, J. Chen, H. Zhong, and F. Qin. SyncChecker: Detecting synchronization errors between MPI applications and libraries. In *IPDPS*, pages 342–353, 2012.

[9] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January 1998.

[10] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI*, pages 223–234, 2007.

[11] M. Feng, R. Gupta, and Y. Hu. SpiceC: Scalable parallelism via implicit copying and explicit commit. In *PPoPP* 2011.

[12] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[13] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA*, pages 73–83, 2007.

[14] C. Jaramillo, R. Gupta, and M. L. Soffa. Capturing the effects of code improving transformations. In *PACT*, 1998.

[15] C. Jaramillo, R. Gupta, and M. L. Soffa. Comparison checking: An approach to avoid debugging of optimized code. In *ESEC/SIGSOFT FSE*, pages 268–284, 1999.

[16] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, pages 389–400, 2011.

[17] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[19] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *PLDI*, pages 134–143, 2009.

[20] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A simple and efficient memory model for concurrent programming languages. In *PLDI*, pages 351–362, 2010.

[21] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[22] V. Nagarajan and R. Gupta. Architectural support for shadow memory in multiprocessors. In *VEE*, pages 1–10, 2009.

[23] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL*, pages 327–338, 2007.

[24] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *PPoPP*, pages 167–178, 2003.

[25] J. Ouyang, K. Veeraraghavan, D. Lee, P. Chen, J. Flinn, and S. Narayanasamy. Epoch parallelism: One execution is not enough. Presentation In *Research Vision Session, OSDI*, 2010.

[26] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[27] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *PLDI*, pages 320–331, 2006.

[28] J. Reinders. *Intel threading building blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.

[29] 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, November 1997.

[30] M. Süß and C. Leopold. Common mistakes in OpenMP and how to avoid them: A collection of best practices. In *IWOMP 2006 and OpenMP Shared Memory Parallel Programming, LNCS Volume 4315*, pages 312–323, 2008.

[31] C. Tian, M. Feng, and R. Gupta. Enhanced speculative parallelization via incremental recovery. In *PPoPP*, 2011.

[32] N. Vachharajani, R. Rangan, E. Raman, M.J. Bridges, G. Ottoni, and D.I. August. Speculative decoupled software pipelining. In *PACT*, pages 49–59, 2007.

[33] X. Xie and J. Xue. ACCULOCK: Accurate and efficient detection of data races. In *CGO*, pages 201–212, 2011.

[34] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.