# Parallelizing Sequential Programs with Statistical Accuracy Tests

SASA MISAILOVIC, DEOKHWAN KIM, and MARTIN RINARD, MIT

We present QuickStep, a novel system for parallelizing sequential programs. Unlike standard parallelizing compilers (which are designed to preserve the semantics of the original sequential computation), QuickStep is instead designed to generate (potentially nondeterministic) parallel programs that produce acceptably accurate results acceptably often. The freedom to generate parallel programs whose output may differ (within statistical accuracy bounds) from the output of the sequential program enables a dramatic simplification of the compiler, a dramatic increase in the range of applications that it can parallelize, and a significant expansion in the range of parallel programs that it can legally generate.

Results from our benchmark set of applications show that QuickStep can automatically generate acceptably accurate and efficient parallel programs—the automatically generated parallel versions of five of our six benchmark applications run between 5.0 and 7.8 times faster on eight cores than the original sequential versions. These applications and parallelizations contain features (such as the use of modern object-oriented programming constructs or desirable parallelizations with infrequent but acceptable data races) that place them inherently beyond the reach of standard approaches.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Parallelization,accuracy, trade-off, interactive

**ACM Reference Format:**
Misailovic, S., Kim, D., and Rinard, M. 2013. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embed. Comput. Syst.* 12, 2s, Article 88 (May 2013), 26 pages.
DOI:http://dx.doi.org/10.1145/2465787.2465790

## 1. INTRODUCTION

The dominant paradigm for reasoning about the behavior of software systems revolves around hard binary correctness properties. A standard approach is to identify (either formally or informally) a concept of correct behavior, then engineer with the goal of producing a perfect system that delivers correct behavior in all circumstances.

One advantage of this approach is that it simplifies reasoning for modularity purposes—perfect components that always behave correctly can significantly reduce the number of scenarios that a developer must consider when using the components in a larger system. The complexity reduction is so valuable that developers typically operate under this assumption even though the components are almost never perfect—virtually all components exhibit incorrect behavior in some (ideally rare) scenarios.

A key disadvantage, however, is overengineering. When attempting to build a perfect system, developers often invest substantially more engineering effort and produce a

This research was supported by DARPA Cooperative Agreement FA8750-tion, and NSF Awards 0811397, 0835652, 0905244, and 1036241.

Authors' address: S. Misailovic (corresponding author), D. Kim, and M. Rinard, MIT CSAIL, The Stata Center, 32 Vassar St., Cambridge, MA 02139; email: misailo@csail.mit.edu.

DOI:http://dx.doi.org/10.1145/2465787.2465790

system that consumes more resources (such as time or power) than strictly required to deliver acceptable behavior [Rinard 2003].

### 1.1. An Alternate Approach

Overengineering motivates an alternate approach whose goal is to produce systems that deliver acceptably accurate results acceptably often. Instead of a guarantee that the system will, on every execution, produce correct behavior, such systems may instead come with a statistical or probabilistic guarantee.

This approach can provide both developers and automated systems with significantly more engineering freedom than standard approaches. This freedom, in turn, can translate directly into reduced engineering effort, reduced resource consumption, or increased functionality.

### 1.2. QuickStep

This article presents a system, *QuickStep*, for automatically parallelizing sequential programs. Freed from the constraint of preserving the precise semantics of the original sequential program, QuickStep instead generates a search space of parallel programs (whose performance and accuracy may vary), then searches this space to find a parallel program that will, with high likelihood, produce outputs that are acceptably close to the outputs that the original sequential program would have produced.

Because QuickStep does not aspire to preserve the semantics of the original sequential program, it has no need to analyze problematic constructs, such as pointers, object references, and dynamic method invocations. It instead simply applies a statistical test to determine if test executions are acceptably accurate. As a result, QuickStep is able to effectively parallelize programs whose use of modern programming constructs place them inherently beyond the reach of standard techniques. The fact that the field of traditional parallelizing compilers has been active for decades but still has yet to produce a compiler capable of parallelizing programs that are within QuickStep's reach demonstrates the advantages of QuickStep's statistical acceptability approach in comparison with standard semantics-preserving approaches.

Note that because the programs that QuickStep generates only need to satisfy statistical accuracy bounds, QuickStep has the freedom to generate efficient nondeterministic parallel programs (as long as these programs satisfy the desired accuracy bounds). Our current QuickStep implementation generates parallel programs with two potential sources of nondeterminism (data races and variations in the execution order of parallel loop iterations), but in general any parallel program, deterministic or nondeterministic, is acceptable as long as it satisfies the statistical accuracy guarantee. Such efficient nondeterministic parallelizations, of course, lie inherently beyond the reach of standard approaches.

### 1.3. Transformations

QuickStep deploys three kinds of transformations to generate its search space of parallel programs.

- *Parallelism-Introduction Transformations.* Transformations that introduce parallel execution. QuickStep currently implements one parallelism introduction transformation: *loop parallelization*. Note that because the iterations of the resulting parallel loop execute without synchronization, anomalies such as data races may cause the parallel program to crash or produce an unacceptably accurate result.
- *Accuracy-Enhancing Transformations.* Transformations that are designed to enhance the accuracy of the parallel program. If a parallelism-introduction

transformation produces an unacceptably accurate program, accuracy-enhancing transformations may restore acceptably accurate execution. QuickStep implements two accuracy-enhancing transformations: *synchronization introduction*, which replaces unsynchronized operations with synchronized operations (the goal is to eliminate data races), and *privatization*, which gives each thread its own copy of otherwise shared local variables (the goal is to eliminate interference between parallel loop iterations).

—*Performance-Enhancing Transformations.* Transformations that are designed to enhance the performance of the parallel program. QuickStep implements two performance-enhancing transformations: *replication introduction*, which replicates objects accessed by parallel tasks then combines the replicas for sequential access (the goal is to eliminate bottlenecks associated with frequently executed synchronized operations on shared objects) and *loop scheduling*, which applies different parallel loop scheduling algorithms (the goal is to find a scheduling policy that interacts well with the specific characteristics of the loop).

### 1.4. Searching the Parallelization Space

Given an initial sequential program, the parallelization transformations induce a space of corresponding parallel programs. QuickStep searches this space as follows. QuickStep attempts to parallelize a single loop at a time, prioritizing the attempted parallelization of loops that consume more execution time over the attempted parallelization of loops that consume less execution time. QuickStep first applies the parallelization transformation to the current loop. It then explores the parallelization space for this loop by repeatedly applying accuracy- and performance-enhancing transformations (prioritizing accuracy-enhancing transformations until it has obtained acceptably accurate execution).

If it is unable to obtain an acceptably accurate parallelization, QuickStep abandons the current loop and moves on to the next. Once it has processed the most time-consuming loops and obtained the final parallelization, QuickStep produces an interactive parallelization report that a developer can navigate to evaluate the acceptability of this parallelization and obtain insight into how the application responds to different parallelization strategies.

### 1.5. Accuracy Metric

In many cases, it may be desirable to produce a parallel program that produces the same result as the original sequential program. But in other cases, the best parallel version may, because of phenomena such as infrequent data races or reordered parallel accumulations, produce a result that differs within acceptable bounds from the result that the sequential program produces. QuickStep is therefore designed to work with an *accuracy metric* that quantifies the difference between an output from the original sequential program and a corresponding output from the automatically generated parallel program run on the same input [Rinard 2006, 2007]. The accuracy metric first uses an *output abstraction* (which typically selects relevant output components or computes a measure of the output quality) to obtain a sequence of numbers $o_1, \ldots, o_m$ from a sequential execution and a corresponding sequence $\hat{o}_1, \ldots, \hat{o}_m$ from a parallel execution on the same input. It then uses the following formula to compute the *distortion* $d$, which measures the accuracy of the parallel execution.

$$d = \frac{1}{m} \sum_{i=1}^{m} \left| \frac{o_i - \hat{o}_i}{o_i} \right| .$$

The closer the distortion $d$ is to zero, the less the parallel execution distorts the output. Given an accuracy metric, an *accuracy bound* $b$ is an upper bound on the acceptable distortion.

### 1.6. Statistical Accuracy Test

QuickStep's statistical accuracy test executes the generated parallel program multiple times, treating each execution as a Bernoulli trial which succeeds if the execution satisfies the accuracy bound $b$ (i.e., if the observed distortion $d$ is at most $b$) and fails if it does not. The test terminates when QuickStep has observed enough executions to make a statistically well-founded judgement to either accept or reject the parallelization. QuickStep uses the Wald sequential probability ratio (SPR) test (see Section 4) to make this judgement. During its search of the parallel program space, QuickStep sets the parameters of this test to accept a parallelization if the program satisfies the accuracy bound at least 90% of the time with a false positive rate of at most 10% (i.e., QuickStep accepts an unacceptably inaccurate program at most 10% of the time). During the final accuracy test, the program must satisfy the accuracy bound at least 99% of the time with a false positive rate of at most 1%. See Section 4 for more details.

It is possible for a parallelization to cause the program to fail to produce a well-formed output (typically because the program crashes). In such cases, QuickStep rejects the parallelization (although it would be possible to configure QuickStep to accept parallelizations that have a low likelihood of failing in this way).

### 1.7. Parallelization Reports

We anticipate that some developers may wish to examine the QuickStep parallelization or use it as a foundation for further development. QuickStep therefore produces an interactive parallelization report that summarizes the applied transformations and the accuracy and performance results. The developer can use this report to evaluate the overall acceptability of the parallelization or to obtain the understanding required to further modify the program.

### 1.8. Experimental Results

To evaluate the effectiveness of QuickStep's approach, we obtained a set of benchmark sequential programs, then used QuickStep to parallelize this set of programs and generate corresponding parallelization reports. Our results show that QuickStep is able to effectively parallelize five out of the six programs, with the final parallel versions running, on our test inputs, between a factor of 5.0 and 7.8 faster (on eight cores) than the corresponding sequential versions.

We used the parallelization reports to evaluate the acceptability of the final parallelizations. Our evaluation shows that, for the programs in our benchmark set, QuickStep is able to produce parallelizations that are acceptable for all inputs (and not just the representative inputs that it uses to drive the exploration of the search space). Moreover, each final parallel program contains at most a handful of parallel loops, each of which requires at most only several synchronization or replication transformations to produce an acceptably accurate program with good parallel performance. The parallelizations are therefore amenable to developer evaluation with reasonable developer effort.

### 1.9. Contributions

This article makes the following contributions.

—*Basic Approach.* It introduces the basic QuickStep approach of developing a set of parallelization transformations then searching the resulting induced space of parallel programs to find a program that maximizes performance while preserving acceptably accurate execution. It also identifies specific parallelization transformations that work together to produce efficient and accurate parallel programs.

—*Search Algorithm.* It presents an algorithm for automatically searching the induced space of parallel programs. This algorithm uses profiling information, performance measurements, and accuracy results from executions on representative inputs to guide the search.

—*Statistical Accuracy Tests.* It introduces the use of statistical accuracy tests to determine if the likelihood that a candidate parallel program will produce an acceptably accurate result is acceptable.

—*Interactive Parallelization Reports.* It introduces interactive parallelization reports that present the performance and accuracy characteristics of the candidate parallel programs, identify the applied transformations for each program, and summarize the overall search process. These reports are designed to facilitate developer evaluation of the automatically generated parallel programs and to help the developer obtain insight into how the original sequential program responds to various parallelization strategies.

—*Experimental Results.* It presents experimental results obtained by using QuickStep to parallelize a set of benchmark sequential applications. These results show that QuickStep is able to produce accurate and efficient parallel versions of five of the six applications. And our examination of the resulting parallelizations indicates that they are acceptable for a wider set of inputs than just the representative inputs used to drive the parallelization process.

A comparison with parallelizations produced by the Intel icc compiler shows that the icc compiler, in contrast, was able to parallelize only small inner loops and produced no improvement in the performance of the application.

—*Accuracy vs. Time Trade-off.* It presents additional evidence that there is a fundamental trade-off between accuracy and performance. It also presents a specific mechanism for exploiting this trade-off in the context of parallelizing compilers.

## 2. EXAMPLE

Figure 1 presents an example that we use to illustrate the operation of QuickStep. The example computes pairwise interactions between simulated water molecules (both stored temporarily in `scratchPads` for the purposes of this computation). The two loops in `interf` generate the interactions. `interact` calls `cshift` to compute the results of each interaction into two 3 × 3 arrays (`Res1` and `Res2`). `updateForces` then uses the two arrays to update the vectors that store the forces acting on each molecule, while `addval` updates the `VIR` accumulator object which stores the sum of the virtual energies of all the interactions.

### 2.1. Profiling Runs

QuickStep starts with the source code of the program, some representative inputs, an accuracy abstraction (which identifies relevant outputs), and an accuracy bound of 0.003, all specified by the developer. QuickStep next runs the sequential computation on the representative inputs and records the running times and outputs. It next generates an instrumented version of the program that counts the number of instructions executed in each loop and records the dynamic loop nesting information. It then runs this instrumented version of the program to obtain the loop profiling information (which identifies the most time-consuming loops in the program and the dynamic nesting relationships between these programs). QuickStep next generates an instrumented version of the program that generates a trace of the addresses that the program reads

```
void ensemble::interf(){
  int i, j; scratchPad *p1, *p2;
  for(i = 0; i < numMol-1; i++) {
    for(j = i+1; j < numMol; j++){
      p1 = getPad(j); p2 = getPad(i); interact(p1,p2);
    }
  }
}
void ensemble::interact
  (scratchPad *p1, scratchPad *p2) {
  double incr, Res1[3][3], Res2[3][3];
  incr = cshift(p1,p2,Res1,Res2);
  p1->updateForces(Res1); p2->updateForces(Res2);
  VIR.addval(incr);
}
void scratchPad::updateForces(double Res[3][3]) {
  this->H1force.vecAdd(Res[0]); this->Oforce.vecAdd(Res[1]);
  this->H2force.vecAdd(Res[2]);
}
```

Fig. 1. Example computation.

```
class accumulator {
 double *vals; volatile bool isCached;
 volatile double cachedVal; pthread_mutex_t mutex;
 double cache() {
  double val = 0.0;
  for (int i = 0; i < num_thread; i++) { val+= vals[i]; vals[i] = 0; }
  cachedVal += val; isCached = true; return val;
 }
public:
 double read() {
  if (isCached) return cachedVal;
  return cache();
 }
 void addval(double d) {
  if (!isCached) cache();
  cachedVal = cachedVal + d; isCached = true;
 }
 void addvalSync(double d) {
  pthread_mutex_lock(&mutex); addval(d); pthread_mutex_unlock(&mutex);
 }
 void addvalRepl(double d) {
  if (isCached) isCached = false;
  vals[this_thread] += d;
 }
};
```

Fig. 2. Example accumulator.

and writes. It then runs this version to obtain the memory profiling information (which identifies potentially interfering accesses from parallel loop iterations).

### 2.2. Search of the Parallelization Space

The loop profiling information indicates that almost all of the execution time is spent in an outer loop that iterates over the time steps in the simulation. QuickStep's attempted parallelization of this loop fails because it produces computations with unacceptable accuracy.

QuickStep next parallelizes the interf outer loop. The resulting parallel computation is close to accurate enough but, in the end, fails the statistical accuracy test. The memory profiling information indicates that there are potential data races at multiple locations in the parallel loop, with the densest races occurring within the accumulator addval operation invoked from the interact method. Figure 2 presents (relevant methods of) the accumulator class. Each accumulator contains several additional implementations of the basic addval operation—the addvalSync operation, which uses a multiple exclusion lock to make the addval execute atomically, and the addvalRepl operation, which adds the contributions into local replicas without synchronization.[1] Based on the memory profiling information, QuickStep invokes the synchronized version of the addval method, changing the call site in interact to invoke the addvalSync method instead of the addval method.

This transformation produces a parallel program with statistically acceptable accuracy but unacceptable performance. In fact, the parallel program takes longer to execute than the original sequential program! QuickStep operates on the assumption that there is a bottleneck on the synchronized addvalSync updates and replaces the call to addvalSync with a call to the replicated version (addvalRepl) of the addval method. This version has good accuracy—the remaining data races (which occur when the computation updates the vectors that store the forces acting on each molecule) occur infrequently enough so that the computation produces acceptably accurate results.

After a similar parallelization process for the remaining time-intensive loop, the outer loop in the poteng method (not shown) which accounts for the vast majority of the remaining execution time, QuickStep has found several parallelizations that exhibit both good performance and acceptably accurate output. QuickStep takes the parallelization with the best performance and runs the final, more stringent, statistical accuracy test on this parallelization, which passes the test to become the final accepted parallelization.

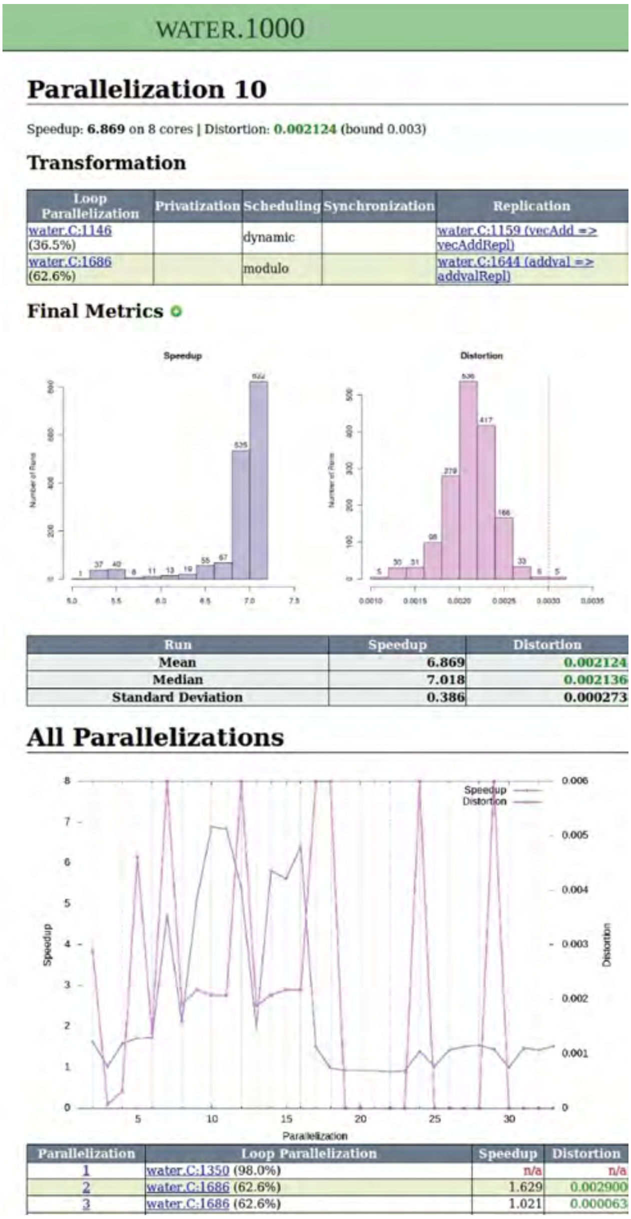
### 2.3. Interactive Report

As it explores the induced space of parallel programs, QuickStep produces a log whose entries specify the applied parallelization transformations, the performance, and the accuracy for each execution of each generated parallel program. QuickStep processes this log to produce an interactive report that the developer can navigate to evaluate the final parallelization and gain insight into how the different transformations affect the parallel behavior of the application. The interactive report takes the form of a set of (dynamically generated) linked webpages.

Figure 3 presents a screen shot of the start page of the interactive report for the Water application from our benchmark suite (the example presented in this section is a simplified version of this application) running on the 1,000 input. This page summarizes the final parallelization of this application. The header presents the revision number (in this case, Revision 10) of the final parallelization, the mean speedup (in this case, 6.869 on eight cores), and the mean distortion (in this case, 0.002).

The Transformation section of the page provides information about the applied transformations for the final version, with links to the transformed source code. In our example, the final parallel program has two parallel loops (at lines 1146 and 1686 of the file Water.C). The computation spends 36.5% of its time in the first loop and 62.6% of its time in the second. The links in the report take the developer directly to these loops in the source code. Each loop has a single applied replication transformation (the call to vecAdd at line 1159 of Water.C was replaced with a call to the replicated version vecAddRepl of this operation, and the call to addval at line 1644 of Water.C was replaced with a call to the replicated version addvalRepl of this operation), no private variables, and no applied synchronization transformations (other parallel versions have applied synchronization transformations). The interf loop uses the modulo loop scheduling policy, while the poteng loop uses the dynamic scheduling policy. The Final Metrics section of the page summarize speedups and distortions from the final statistical accuracy test. Histograms summarize speedups and distortions of

[1] The actual implementation of this accumulator uses padding to avoid potential false sharing interactions.



WATER.1000

## Parallelization 10

Speedup: **6.869** on 8 cores | Distortion: **0.002124** (bound 0.003)

### Transformation

| Loop Parallelization | Privatization | Scheduling | Synchronization | Replication |
|---|---|---|---|---|
| water.C:1146 (36.5%) | | dynamic | | water.C:1159 (vecAdd => vecAddRepl) |
| water.C:1686 (62.6%) | | modulo | | water.C:1644 (addval => addvalRepl) |

### Final Metrics ⊕

| Run | Speedup | Distortion |
|---|---|---|
| Mean | 6.869 | 0.002124 |
| Median | 7.018 | 0.002136 |
| Standard Deviation | 0.386 | 0.000273 |

## All Parallelizations

| Parallelization | Loop Parallelization | Speedup | Distortion |
|---|---|---|---|
| 1 | water.C:1350 (98.0%) | n/a | n/a |
| 2 | water.C:1686 (62.6%) | 1.629 | 0.002900 |
| 3 | water.C:1686 (62.6%) | 1.021 | 0.000063 |

Fig. 3. QuickStep interactive report screen shot.

individual runs of the final parallel programs. Note that only 5 of the 1,606 runs exceeded the accuracy bound 0.003. Clicking on the green + sign (following the Final Metrics title) opens up a table with results from the individual executions.

The All Parallelizations section summarizes the search process. A single graph plots the speedup (left Y axis) and distortion (right Y axis) as a function of the revision number of the generated parallel program (QuickStep assigns the next available revision number every time it generates a new parallel program).

The final section (below the graphs and truncated in this screen shot at Revision 3 of 33 total parallel versions) contains links to similarly formatted pages that summarize the applied parallelization transformations, performance results, and distortion results for the full set of parallel programs that QuickStep visited (starting with Revision 1) as it explored the parallel program search space.

## 3. ANALYSIS AND TRANSFORMATION

QuickStep is structured as a source-to-source translator that augments the original sequential program with OpenMP directives to obtain a parallel program. The QuickStep analysis phases use the LLVM compiler infrastructure [Lattner and Adve 2004].

### 3.1. Loop Profiler

The QuickStep loop profiler produces an instrumented version of the sequential program that, when it runs, counts the number of times each basic block executes. The instrumentation also maintains a stack of active nested loops and counts the number of (LLVM bit code) instructions executed in each loop, propagating the instruction counts up the stack of active nested loops so that outermost loops are credited with instructions executed in nested loops.

The loop profiler produces two outputs. The first is a count of the number of instructions executed in each loop during the loop profiling run. The second is a directed graph that captures the dynamic nesting relationships between different loops (note that the loops may potentially be in different procedures).

### 3.2. Memory Profiler

The QuickStep memory profiler produces an instrumented version of the sequential program that, when it executes, generates a trace of the addresses that it reads and writes. Given the memory profiling information for a given loop, the memory profiler computes the *interference density* for each store instruction $s$, that is, the sum over all occurrences of that store instruction $s$ in the trace of the number of store instructions $p$ in parallel iterations that write the same address. Conceptually, the interference density quantifies the likelihood that a store instruction $p$ in a parallel iteration will interfere with a sequence of (otherwise atomic) accesses that completes with the execution of the store instruction $s$. The interference density is used to prioritize the application of the synchronization and replication transformations, with the transformations applied first to operations that contain store instructions with the higher interference density. QuickStep is currently configured to apply synchronization introduction transformations only to loops in which all writes to a given address are preceded by a corresponding read from that address. The goal is to apply the transformation only within loops that access the object exclusively with atomic read/write updates (and not other accesses such as initializations).

### 3.3. Parallelization and Privatization

QuickStep uses OpenMP directives to specify parallel loops. The following example illustrates the use of these directives.

```
int t;
#pragma omp parallel for private(t) schedule(static)
for (i = 0; i < n; i++) { t = a[i]+b[i]; c[i] = t; }
```

This `parallel for` OpenMP directive causes the iterations of the loop to execute in parallel. When the parallel loop executes, it uses the `static` scheduler (we discuss available schedulers below in Section 3.5) to schedule its iterations onto the underlying parallel machine. Each iteration of the loop is given its own private version of the variable `t` so that its accesses to `t` do not interfere with those of other parallel iterations.

QuickStep uses a simple intraprocedural dataflow analysis to determine which variables to privatize. This analysis finds all local scalar variables that iterations of the parallel loop first write, then read. The parallelization transformation inserts all such variables into the `private` clause of the OpenMP loop parallelization directive.

### 3.4. Synchronization and Replication Transformations

QuickStep's synchronization and replication transformations operate on objects that provide multiple implementations of standard operations. Some operations are synchronized for atomic execution in parallel environments. Others operate on thread-local replicas of the object state (with the replicas coalesced when appropriate). QuickStep can work with any object that provides synchronized or replicated methods. We have implemented a QuickStep component that, given a class, automatically generates an augmented class that contains synchronized and replicated versions of the methods in the class. This component performs the following steps.

— *Synchronization.* It augments instances of the class with a mutual exclusion lock then generates synchronized versions of each method. Each synchronized method acquires the lock, invokes the standard (unsynchronized) version of the method, then releases the lock.
— *Replication.* It augments instances of the class with an array of replicas. Each array element contains a replica of the state of the object. There is one array element (and therefore one replica) for each thread in the parallel computation. It also generates the replicated version of each method (this version accesses the local replica).
— *Caching.* It modifies the original versions of the methods to check whether the object state is distributed across multiple replicas and, if so, to combine the replicas to obtain a single cached version of the state. The method then operates on this cached version. The specific combination mechanism depends on the object. In our current system, the automatically generated code simply adds up the values in the replicas. QuickStep, of course, supports arbitrary combination mechanisms.
— *Padding.* It generates additional unused memory (pads) to separate the replicas so that they fall on different cache lines. The goal is to eliminate any false sharing [Bolosky and Scott 1993] that might otherwise degrade the performance.

QuickStep can work with instances of any class that use any mechanism to obtain alternate implementations of standard operations. So a developer could, for example, provide implementations that use atomicity mechanisms, such as transactional memory or other lock-free synchronization mechanisms [Herlihy and Moss 1993].

As it searches the space of parallel programs, the QuickStep search algorithm may direct the QuickStep compiler to perform synchronization or replication introduction transformations. The QuickStep compiler implements each such transformation by modifying the corresponding call site to invoke the appropriate synchronized or replicated version of the method instead of the original version.

### 3.5. Loop Scheduling

OpenMP supports a variety of loop scheduling policies via the `schedule` clause of the `parallel for` directive [Dagum and Menon 1998]. In general, these policies may vary in the cache locality, load balancing, and scheduling overhead that they elicit from the loop, with different policies working well for different loops. Note that the policies may also have an effect on the result of the computation due to different orders of accessing and processing data.

QuickStep currently searches three loop scheduling policies: `dynamic` (when a thread goes idle, it executes the next available loop iteration), `static` (each thread executes a single contiguous chunk of iterations, with the chunks assigned at the start of the parallel loop), and `static,1` (each thread executes a discontiguous set of iterations, with the iterations assigned to threads at the start of the parallel loop in a round-robin manner). The parallelization reports refer to the `static,1` scheduling strategy as `modulo` scheduling.

## 4. STATISTICAL ACCURACY TEST

QuickStep uses the Wald sequential probability ratio (SPR) test [Wald 1947] as its statistical accuracy test. The SPR test works with a sequence of independent, identically-distributed (IID) Bernoulli variables $X_i \in \{0, 1\}$ from a distribution with unknown reliability (i.e., probability of success) $p = \mathbb{P}(X = 1)$. The test takes as parameters a desired reliability $r$, an approximate false positive bound $\alpha$, an approximate false negative bound $\beta$, a lower accuracy interval $\epsilon_0$, and an upper accuracy interval $\epsilon_1$. The SPR test accepts one of two disjoint hypotheses $H_0 : p \leq p_0$ or $H_1 : p \geq p_1$, where $p_0 = r - \epsilon_0$ and $p_1 = r + \epsilon_1$, after observing $n$ Bernoulli variables (the number $n$ is dynamically determined by the parameters and the values of the previously observed Bernoulli variables). The decision of the test will be correct with high probability, determined by the bounds $\alpha$ and $\beta$.

*Program Executions as Random Variables.* QuickStep's statistical accuracy test works with a candidate parallelized application (QuickStep automatically generates this parallelization) an accuracy bound $b$, and an accuracy metric. The test repeatedly executes the parallelized application; for each execution, the accuracy metric produces a distortion $d$. We represent the $i$th program execution as a random variable $X_i$ which has as the value a result of comparison of the distortion $d_i$ with the bound $b$.

$$X_i = \begin{cases} 1 & \text{if } d_i \leq b; \\ 0 & \text{if } d_i > b. \end{cases}$$

*Reliability Tolerances.* The probability that the program will produce an acceptable result $p = \mathbb{P}(X = 1)$ is, in general, unknown—the test uses observations to reason about the relationship between $p$ and a user-provided reliability goal $r$, but (in general) will never obtain complete information about $p$. The SPR test therefore works with a tolerance around $r$ within which it may give arbitrary answers. Specifically, the user needs to provide a lower accuracy interval $\epsilon_0$ and an upper accuracy interval $\epsilon_1$. Together, $r$, $\epsilon_0$, and $\epsilon_1$ define the lower target probability $p_0 = r - \epsilon_0$ and the upper target probability $p_1 = r + \epsilon_1$. Since we typically view incorrectly accepting a bad parallelization as less acceptable than incorrectly rejecting a good parallelization, we use a one-sided test in which $\epsilon_0 = 0$ (i.e., $p_0 = r$).

*Hypotheses.* We represent the acceptability of a parallel program as two hypotheses $H_0 : p \leq p_0$ (i.e., the probability that the application produces an acceptable result is at most $p_0$) and $H_1 : p \geq p_1$ (i.e., the probability that the application produces an acceptable result is at least $p_1$). If the test accepts $H_0$, QuickStep will reject the parallelization. If the test accepts $H_1$, QuickStep will accept the parallelization.

*True and False Positive Bounds.* Ideally, QuickStep's statistical accuracy test should select all parallelizations with a high enough probability of producing an acceptable result ("good" parallelizations), and reject all other parallelizations ("bad" parallelizations). In practice, however, the statistical accuracy test may encounter an unlikely sequence of samples that cause it to make an incorrect decision (either rejecting a good parallelization or accepting a bad parallelization). We therefore consider the following four probabilities: $P_G$ is the probability that the test will correctly accept a good parallelization, $1 - P_G$ is the probability that the test will incorrectly reject a good parallelization, $P_B$ is the probability that the test will incorrectly accept a bad parallelization, $1 - P_B$ is the probability that the test will correctly reject a bad parallelization. Together, $P_G$ and $P_B$ completely characterize the risks associated with making a wrong decision. Ideally, $P_G$ should be close to 1 and $P_B$ should be close to 0. Note that the accuracy test uses an approximate false positive bound $\alpha$ and an approximate true positive bound $\beta$. To ensure that the true desired bounds $P_B$ and $P_G$ are guaranteed by the test, we can use Inequalities (5) and (6) from Section 4.1 to calculate $\alpha$ and $\beta$.

*IID Executions.* The statistical accuracy test assumes that test executions of the parallel application are independent and identically distributed. To promote the validity of this assumption, we ensure that executions do not reuse results from previous executions and that no two executions run at the same time.

We note that in the presence of execution schedule dependent phenomena such as data races, changing aspects of the underlying execution environment (such as the computational load, thread scheduler, number of cores, or other hardware characteristics) may change the probability that an execution of the program will satisfy the accuracy bound $b$. So the statistical accuracy tests are valid only for the specific execution environment in which they were performed.

### 4.1. SPR Test Detailed Description

We now provide an overview of the mechanism that the SPR test uses to determine which hypothesis to accept. Given two hypotheses, $H_0 : p \leq p_0$ and $H_1 : p \geq p_1$, the test performs a minimal number of trials (corresponding to a parallel program execution) before accepting one of the hypotheses. The test accepts the correct hypothesis with high probability, as determined by the correctness bounds $P_G$ and $P_B$. If the true probability $p$ lies between $p_0$ and $p_1$, the test nondeterministically accepts one of the hypotheses.

The SPR test executes for multiple iterations, performing a single trial in each iteration. Let $X_i$ be a Bernoulli variable from single test iteration and $\vec{X} = (X_1, X_2, \ldots X_n)$ be an $n$-dimensional vector of variables observed in the first $n$ iterations. At each iteration $n$, the test collects the observation of the variable $X_n$ and calculates the log likelihood ratio $L(\vec{X})$ between the two interpretations of the observations seen so far.

$$L(\vec{X}) = \log \frac{\mathbb{P}(\vec{X}|H_1)}{\mathbb{P}(\vec{X}|H_0)}. \tag{1}$$

The algorithm defines two stopping conditions, $A$ and $B$ ($B < A$), which can be calculated using the probabilities $P_G$, and $P_B$. The algorithm performs trials until $L(\vec{X}) \geq A$, when it accepts the hypothesis $H_1$, or $L(\vec{X}) \leq B$, when it accepts the hypothesis $H_0$.

For a single execution, $\mathbb{P}(X_i|H_0) = \mathbb{P}(X_i|p \leq p_0) \leq p_0$ (the inequality follows immediately from $\mathbb{P}(X_i|p \leq p_0) = p$), and $\mathbb{P}(X_i|H_1) = \mathbb{P}(X_i|p \geq p_1) \geq p_1$. The function $L(X_i)$ has a minimum value when $\mathbb{P}(X_i|H_0) = p_0$ and $\mathbb{P}(X_i|H_1) = p_1$. Using these equalities will provide the most conservative decision by the test.

Since all random variable $X_i$ are independent, $\mathbb{P}(\vec{X}|H_k) = \prod_{i=0}^{n} \mathbb{P}(X_i|H_k) = (1 - p_k)^f p_k^{n-f}$ ($k \in \{0, 1\}$), where $f$ is the number of executions that failed to produce an acceptable result, and $n - f$ is the number of executions that produced an acceptable result. The formula for the log likelihood ratio becomes

$$L(\vec{X}) = (n - f)\log\frac{p_1}{p_0} + f\log\frac{1 - p_1}{1 - p_0}. \tag{2}$$

The values $A$ and $B$ used as stopping conditions for the test are related to the risk that the test may make a wrong decision. Specifically, we derive the relation between $A$ and $B$ and the probabilities of accepting a good parallelization, $P_G = \mathbb{P}[H_1|H_1]$, and accepting a bad parallelization, $P_B = \mathbb{P}[H_1|H_0]$. For the approximate bounds $\alpha$ and $\beta$, assume the following.

$$\frac{P_B}{P_G} \le \frac{\alpha}{\beta} \le e^{-A}; \tag{3}$$

$$\frac{1 - P_G}{1 - P_B} \le \frac{1 - \beta}{1 - \alpha} \le e^{B}. \tag{4}$$

To find the values for $A$ and $B$ that satisfy the previous inequalities, Wald showed that $A = \log\frac{\beta}{\alpha}$ and $B = \log\frac{1-\beta}{1-\alpha}$ are the computationally simplest solutions [Wald 1947]. Since $P_B, P_G \in (0, 1)$, then $P_B \le \frac{P_B}{P_G}$ and $1 - P_G \le \frac{1-P_G}{1-P_B}$, and so

$$P_B \le \frac{\alpha}{\beta}; \tag{5}$$

$$1 - P_G \le \frac{1 - \beta}{1 - \alpha}. \tag{6}$$

While, in general, the values of $\alpha$ and $\beta$ are not equal to $P_B$ and $P_G$ when $A$ and $B$ have the values described in the previous paragraph, it is possible to use Inequalities (5) and (6) to calculate appropriate values of parameters $\alpha$ and $\beta$ that yield the desired probabilities $P_B$ and $P_G$. For example, if $\alpha = 0.091$ and $\beta = 0.91$, then $P_G \ge 0.9$ and $P_B \le 0.1$. As a special case, when $\alpha \to 0$, and $\beta \to 1$, the bounds can be approximated to $P_B \le \alpha$ and $\beta \le P_G$. In that case, $\alpha \approx P_B$ and $\beta \approx P_G$. For example, if $\alpha = 0.01$ and $\beta = 0.99$, $P_G = 0.9898\ldots$ (which is very close to $\beta$), and $P_B = 0.0101\ldots$ (which is very close to $\alpha$).

### 4.2. Accuracy vs. Time Trade-offs

The values of $\alpha$, $\beta$, $p_0$, and $p_1$, together with the (at the start of the test) unknown number of failed executions $f$, determine the number of iterations required for the SPR test to terminate and the overall execution time of the test.

Higher accuracy requires more observations, that is, more executions of the parallel application, for the SPR test to accept or reject the parallelization. We can see from Equation (2) that the coefficient $\log\frac{1-p_1}{1-p_0}$ will have a significantly larger absolute value (but the opposite sign) compared to $\log\frac{p_1}{p_0}$ when $p_1$ and $p_0$ are close to 1. As a consequence, the number of failed executions in previous steps, $f$, will have a major impact on the total number of steps of the test. Note that if the true value of $p$ is close to $p_1$, the number of failed executions is likely to increase as the test progresses, requiring a substantially larger total number of steps.

For example, if $p_0 = 0.9$, $p_1 = 0.95$, $\alpha = 0.091$, and $\beta = 0.91$, the SPR test requires 43 executions in order to accept the parallelization if all results are acceptable. In contrast, the test requires only 4 executions to reject the parallelization if all results are unacceptable. If there is one execution that produced unacceptable result, the SPR test requires 57 executions before accepting the parallelization. For stricter bounds,

$p_0 = 0.99$, $p_1 = 0.995$, $\alpha = 0.01$, and $\beta = 0.99$, the parallelization is accepted after at least 913 executions (if all executions are acceptable) and rejected after at least 7 executions (if none of the executions is acceptable). Accepting a parallelization when a single failed execution is observed will require 1051 executions in total.

In general, it is necessary to make a trade-off between the accuracy of the statistical test and the time required to perform the test. QuickStep uses two sets of accuracy bounds to improve the speed of exploration. The bounds are relaxed during the intermediate steps in the search space exploration, resulting in a significantly smaller number of required executions, but also a larger probability of an incorrect decision. QuickStep performs the final accuracy test with stronger accuracy bounds, and therefore a smaller probability of an incorrect decision. This design promotes a more rapid exploration of the search space at the potential cost of the search producing a final parallelization that the final test rejects (in which case the final test moves on to try the next best parallelizations until it finds a parallelization that it can accept).

During the parallelization space search, we set $r = 0.90$, $\epsilon_0 = 0$, $\epsilon_1 = 0.05$, $\alpha = 0.091$, and $\beta = 0.91$. The probability that the SPR test will accept a good parallelization is $P_G \geq 0.9$ and the probability that it will accept a bad parallelization is $P_B \leq 0.1$. During the final accuracy test we set $r = 0.99$, $\epsilon_0 = 0$, $\epsilon_1 = 0.005$, $\alpha = 0.01$, and $\beta = 0.99$. The probabilities that the SPR test will accept good and bad parallelizations, are, respectively $P_G \geq 0.99$ and $P_B \leq 0.01$.

*The Hoeffding Inequality*. In previous work [Misailovic et al. 2010a], we used the Hoeffding inequality [Hoeffding 1963], to determine if a candidate parallelization satisfies a desired statistical accuracy bound. The test performs test executions to obtain an estimator $\hat{p}$ of the unknown probability $p$ that the parallelization will produce an acceptably accurate result. It works with a desired precision $\langle \delta, \epsilon \rangle$ and performs enough test executions so that it can use the Hoeffding inequality $\mathbb{P}\left(\hat{p} > p + \epsilon\right) \leq \delta$. Unlike the SPR test, the number of test executions is fixed in advance as a function of $\delta$ and $\epsilon$—the test does not use results from completed test executions to determine if it has enough information to determine if the parallelization is acceptably accurate and terminate. In general, using the Hoeffding inequality requires the test to perform substantially more executions than the SPR test to obtain comparable statistical guarantees.

## 5. PARALLELIZATION SEARCH SPACE ALGORITHM

The parallelization algorithm explores the search space induced by the possible parallelizations, looking for the parallel program that delivers the largest performance increase while keeping the distortion within the accuracy bound for the representative inputs. The algorithm consists of two main parts—the first part of the algorithm explores the parallelization space associated with a single loop; the second part of the algorithm combines single-loop parallelizations into parallelizations involving multiple loops (in the process invoking the single-loop algorithm).

### 5.1. Parallelization Search for Individual Loops

Figure 4 presents the algorithm that searches the parallelization space associated with a single loop. The algorithm takes as input a loop $\ell$ whose parallelization to explore, a set $P$ of loops that are already parallelized, and a set $\mathbb{A}$ of information about previously explored parallelizations. Specifically, $\mathbb{A}$ contains, for each such parallelization, sets $S$ and $T$ of synchronizations and replications (respectively) applied to call sites in the parallelized loops, a set $T$ of scheduling policies for the parallelized loops, and a mean speedup $s$ and distortion $d$ for the parallelization.

**INPUTS**

$\ell$ – the loop to parallelize
$P$ – the set of loops that were previously parallelized
$\mathbb{A}$ – the set of all results

**OUTPUTS**

$\mathbb{A}$ – updated set of parallelization results.

**LOCALS**

$S$ – the set of call sites for synchronization introduction
$R$ – the set of call sites for replication introduction
$T$ – the set of loop scheduling policies
$S_\ell$ – a set of synchronization introduction call sites for loop $\ell$
$R_\ell$ – a set of replication introduction call sites for loop $\ell$
$C$ – a set of call sites for synchronization or replication introduction
$p$ – indicator of whether the parallelization is acceptable

**AUXILIARY FUNCTIONS**

$\mathsf{visit}(P, S, R, T)$. – runs the program with the loops in $P$ parallelized with loop scheduling policies $T$, synchronization introduction transformations applied at $S$, and replication introduction transformations applied at $R$. Returns the status (pass/fail) according to the selected statistical accuracy bounds, mean speedup and distortion from the test runs. Also generates log entries used to produce the interactive report.

$\mathsf{candidates}(\ell)$. – returns the set of candidate synchronization and replication introduction call sites, $C_S$ and $C_R$ for loop $\ell$

$\mathsf{next}(C)$. – the highest priority synchronization or replication introduction call site in $C$ according to the memory profiling information

$\mathsf{transformations}(P, \mathbb{A})$. – returns a set of transformations for the set of loops $P$ from the results set $\mathbb{A}$

**SEARCH-LOOP**

```
$S,\ R,\ T\ =$ transformations$(P,\ \mathbb{A})$
$p,\ s,\ d\ =$ visit$(P\ \cup\ \{\ell\},\ S,\ R,\ T)$
$C_S,\ C_R\ =$ candidates$(\ell)$, $C\ =\ C_S\ \cup\ C_R, S_\ell\ =\ \emptyset\ \ B\ =\ \emptyset$
repeat
  while $b\ <\ d\ \wedge\ C\ \neq\ \emptyset$ do
    $c\ =$ next$(C)$
    $p',\ s',\ d'\ =$ visit$(P\ \cup\ \{\ell\},\ S\ \cup\ S_\ell\ \cup\ \{c\},\ R,\ T)$
    if $p'\ =\ true\ \wedge\ d'\ <\ d$
      $S_\ell\ =\ S_\ell\ \cup\ \{c\},\ d\ =\ d',\ s\ =\ s', p\ =\ true$
    end
    $C\ =\ C\ \setminus\ \{c\}$
  end
  $C\ =\ S_\ell\ \cap\ C_R, R_\ell\ =\ \emptyset$
  while $C\ \neq\ \emptyset$ do
    $c\ =$ next$(C)$
    $p'\ ,\ s',\ d'\ =$ visit$(P\ \cup\ \{\ell\},\ (S\ \cup\ S_\ell)\ \setminus\ \{c\},\ R\ \cup\ R_\ell\ \cup\ \{c\},\ T)$
    if $p'\ =\ true\ \wedge\ s\ <\ s'\ \wedge\ d'\ \leq\ b$
      $S_\ell\ =\ S_\ell\ \setminus\ \{c\}, R_\ell\ =\ R_\ell\ \cup\ \{c\}, d\ =\ d', s\ =\ s',\ p\ =\ true$
    end
    $C\ =\ C\ \setminus\ \{c\}$
  end
until $d\ \leq\ b$ or $S_\ell\ =\ \emptyset$
foreach $t$ in OpenMP loop schedule types do
  $p', s',\ d'\ =$ visit$(P\ \cup\ \{\ell\},\ S\ \cup\ S_\ell,\ R\ \cup\ R_\ell,\ \ (T\ \setminus\ \{\langle\ell,\ \_\rangle\})\ \cup\ \{\langle\ell,\ t\rangle\})$
  if $p'\ =\ true\ \wedge\ s\ \leq\ s'\ \wedge\ d'\ \leq\ b$
    $S\ =\ S\ \cup\ S_\ell, R\ =\ R\ \cup\ R_\ell$
    $T\ =\ (T\ \setminus\ \{\langle\ell,\ \_\rangle\})\ \cup\ \{\langle\ell,\ t\rangle\},\ p\ =\ true$
  end
end
if $p\ =\ true$ then return $\mathbb{A}\ \cup\ \{(P\ \cup\ \{\ell\},\ S,\ R,\ T,\ s,\ d)\}$
else return $\mathbb{A}$
```

Fig. 4. Parallelization space search algorithm for one loop.

The algorithm searches the space of parallel programs for the candidate loop $\ell$ as follows.

(1) *Initial Parallelization.* QuickStep first generates a parallelization that executes $\ell$ in parallel (with no applied synchronization or replication transformations) in the context of the previous parallelization of the loops in $P$ (this parallelization includes any applied synchronization and replication transformations for these previously parallelized loops). It uses the statistical accuracy test to determine if this parallelization is acceptably accurate.

(2) *Synchronization Transformations.* If the current parallelization is not acceptably accurate, QuickStep next applies synchronization introduction transformations to $\ell$ in an attempt to restore acceptably accurate parallel execution. It applies these transformations according to the priority order established by the memory profiling information, with operations containing higher interference-density store instructions prioritized over operations containing lower interference-density store instructions. As it applies the transformations, it builds up a set $S_\ell$ of call sites to apply synchronization introduction transformations. It only places a transformation into $S_\ell$ if it improves the accuracy of the parallelization.

(3) *Replication Transformations.* QuickStep next applies replication introduction transformations in an attempt to maximize the performance while preserving acceptably accurate execution. For each synchronization introduction call site in $S_\ell$, it replaces the synchronization introduction transformation with the corresponding replication introduction transformation. As it applies the transformations, it builds up a set $R_\ell$ of replication introduction transformations. It only places a replication transformation into $R_\ell$ (and removes it from $S_\ell$) if it improves performance while maintaining acceptable accuracy.
Note that it is possible for the performance improvements associated with replication to increase the data race density within the loop. In the worst case, this increase can cause the parallelization to exceed its accuracy bound. In this case, the algorithm returns back to apply additional synchronization transformations (potentially followed by additional replication transformations) to restore acceptably accurate execution.

(4) *Loop Scheduling.* QuickStep finally tries all of the different loop scheduling policies for the current loop $\ell$. If it encounters an acceptably accurate parallelization with better performance than the previous best alternative, it accepts this new scheduling policy. Note that since they change data access patterns of individual threads, scheduling policies may also influence accuracy and we need to ensure that the parallel program remains within the accuracy bound $b$.

### 5.2. Exploring the Search Space

Figure 5 presents the algorithm for the exploration of the entire parallelization space. While, in general, finding the optimal parallel program (the one with the highest performance increase subject to the accuracy bound) may require exploring all loops, Quickstep uses the profiling information to search only a part of the optimization space that is likely to contain parallel programs that substantially improve the performance. In particular, QuickStep prioritizes the parallelization of the most time-consuming loops by traversing the dynamic loop nesting graph $G$. It starts from the most time-consuming loops in priority order, with the priority determined by the amount of time spent in the loop (as indicated by the loop profiling information). QuickStep explores the graph in depth-first fashion, pruning the search space (1) when the resulting parallelization is unacceptably inaccurate and (2) once it has successfully parallelized an

```
INPUTS
    $L$ – the set of loops in execution time priority order
    $G$ – the dynamic loop nesting graph
OUTPUTS
    $\mathbb{A}$ – set of parallelization results for the explored loops
LOCALS
    $V$ – the set of visited loop sets
    $R$ – the set of results for all loops
    $S$ – the stack containing loop states to visit
AUXILIARY FUNCTIONS
    successor($\ell, G$). – returns loops that are immediate successors of a loop $\ell$ in graph $G$.
    filter($L, P, G$). – filters out the nested loops of loops in set $P$ from the set $L$.
    order($\mathbb{A}$). – orders the results according to speedup and returns the set of loops and belonging transformations
QUICKSTEP
$V = \emptyset, \mathbb{A} = \emptyset, S = \emptyset$
foreach $\ell_0$ in $L$
    $S \leftarrow (\ell_0, \emptyset)$
    while $S \neq \emptyset$ do
        $S \rightarrow (\ell, P')$ , $P = P' \cup \{\ell\}$
        if $P \in V$ then continue
        $V = V \cup \{P\}$
        $\mathbb{A} = \text{SEARCH} - \text{LOOP}\ (\ \ell,\ P',\ \mathbb{A}\ )$
        foreach $\ell_s$ in filter($L,\ P,\ G$) do
            $S \leftarrow (\ell_s,\ P)$
        end
    end
end
foreach $P,\ S,\ R,\ T$ in order($\mathbb{A}$) do
    $p,\ s,\ d =$ visit($P,\ S,\ R,\ T$)
    if $p = true$ then return $(P,\ S,\ R,\ T,\ s,\ d)$
end
return $\emptyset$
```

Fig. 5. QuickStep search algorithm.

outer loop, it does not attempt to parallelize any inner loops which can be executed only from the body of the outer loop.

QuickStep tries to discover the synchronization, replication, and scheduling policies for the current loop that, when applied together with previously parallelized loops, maximally increase performance while keeping the accuracy within the bound $b$. The algorithm maintains a set of all acceptable parallelizations $\mathbb{A}$. For each such parallelization $\mathbb{A}$, it records the sets $S$ and $R$ of applied synchronization and replication transformations, the set $T$ of loop scheduling policies, and the observed mean speedup $s$ and distortion $d$. It updates $\mathbb{A}$ whenever it successfully parallelizes another loop.

When QuickStep finishes exploring the parallel program search space, it orders the accepted parallelizations according to the performance then runs the final statistical accuracy test on these programs. As we noted in Section 4.2, QuickStep relaxes the accuracy bounds in the intermediate steps of the algorithm to improve the speed of the exploration. The algorithm may therefore return more parallel programs that pass the relaxed test. In the final step QuickStep performs SPR test on these candidate parallelizations with the desired, stronger accuracy bounds. Quickstep returns the parallel program with the best performance that passes the final accuracy test as the final parallelization.

### 5.3. Interactive Parallelization Report

The report generator, written in Ruby programming language, processes the log generated by the search algorithm, extracting the relevant information and placing this

Table I. Quantitative Results for Benchmark Applications

| Application | Input | Best Parallel Version | Speedup | Distortion | Search Time (min) | Check Run/Fail | Check Time (min) |
|---|---|---|---|---|---|---|---|
| Barnes-Hut | 16K bodies | 2 of 16 | 6.168 | 0.0000 | 27.4 | 913/0 | 41.3 |
| | 256K bodies | 2 of 16 | 5.760 | 0.0000 | 47.4 | 913/0 | 77.6 |
| Search | 500 particles | 16 of 16 | 7.743 | 0.0000 | 63.6 | 913/0 | 33.5 |
| | 750 particles | 16 of 16 | 7.756 | 0.0000 | 92.9 | 913/0 | 50.0 |
| String | `big` | 4 of 10 | 7.608 | 0.0005 | 8.98 | 913/0 | 18.0 |
| | `inv` | 4 of 13 | 7.582 | 0.0005 | 17.1 | 913/0 | 34.3 |
| Volume Rendering | `head` | 3 of 7 | 6.153 | 0.0000 | 23.5 | 913/0 | 52.4 |
| | `head-2` | 3 of 7 | 5.047 | 0.0000 | 6.8 | 913/0 | 15.3 |
| Water ($b$ = 0.003) | 1000 molecules | 10 of 33 | 6.869 | 0.0021 | 180.2 | 1606/5 | 64.5 |
| | 1728 molecules | 11 of 34 | 7.009 | 0.0010 | 228.9 | 913/0 | 43.9 |
| Water ($b$ = 0.0) | 1000 molecules | 12 of 35 | 6.183 | 0.0000 | 189.2 | 913/0 | 40.7 |
| | 1728 molecules | 12 of 35 | 6.123 | 0.0000 | 231.5 | 913/0 | 50.1 |

information in an SQLite database. It uses the Ruby on Rails Web development framework[2] to retrieve the appropriate information from the database and dynamically generate webpages that present this information to the developer.

## 6. EXPERIMENTAL RESULTS

We used QuickStep to parallelize five scientific computations.

- *Barnes-Hut.* A hierarchical N-body solver that uses a space-subdivision tree to organize the computation of the forces acting on the bodies [Barnes and Hut 1986]. Barnes-Hut is implemented as an object-oriented C++ computation.
- *Search.* Search uses a Monte-Carlo technique to simulate the elastic scattering of each electron from the electron beam into a solid [Browning et al. 1995]. Search is implemented in C.
- *String.* String constructs a two-dimensional discrete velocity model of the geological medium between two oil wells [Harris et al. 1990]. String was originally developed as part of the Jade project [Rinard 1994] and is implemented in C.
- *Volume Rendering.* Volume Rendering renders a three-dimensional volume dataset for graphical display [Nieh and Levoy 1992]. Volume Rendering appears in the SPLASH-2 benchmark suite [Woo et al. 1995] and is implemented in C.
- *Water.* Water evaluates forces and potentials in a system of water molecules in the liquid state. Water is an object-oriented C++ version of the Perfect Club benchmark MDG [Blume and Eigenmann 1992].

We also applied QuickStep to Panel Cholesky (a program that factors a sparse positive-definite matrix). Because the parallelism in this application is not available by executing loops in parallel, it is beyond the reach of the current set of QuickStep transformations [Misailovic et al. 2010a, 2010b].

### 6.1. Methodology

We obtained the applications in our benchmark suite along with two representative inputs for each application. We specified appropriate accuracy bounds for each application and used QuickStep to automatically parallelize the applications, using the representative inputs to perform the required profiling and parallel executions. We

[2] `http://rubyonrails.org/`

configured QuickStep to search the parallelization space for each input independently with the final accuracy test performed on both inputs. Note that it also makes sense to perform the validation on the input used during the search because the final accuracy step provides stronger statistical bounds than the exploration steps.

We performed all executions on an Intel Xeon E5520 dual quad-core machine running Ubuntu Linux, using LLVM 2.7 to compile all versions of the applications. All parallel executions use eight threads. We set the accuracy bound $b$ to be 0.003 for Water and String and 0.01 for the other benchmarks.

### 6.2. Quantitative Results

Table I presents the quantitative results for each application. All of the numbers in this table either appear directly in the corresponding interactive report or are computed from numbers that appear in this report. The table contains one row for each combination of application and input. The third column (Best Parallel Version) contains entries of the form $x$ of $y$, where $x$ is the revision number of the final parallel version of the application (i.e., the version with the best performance out of all parallel versions that satisfy the accuracy requirement) and $y$ is the number of revisions that QuickStep generated during its exploration of the search space. The fourth column (Speedup) presents the mean speedup (over all test executions) of the final version of the application when run on eight cores. The speedup is calculated as the mean execution time of the original sequential program (with no parallelization overhead whatsoever) divided by the mean execution time of the final version. The fifth column (Distortion) presents the mean measured distortion over all executions of the final parallel version; the number of executions is equal to the sum of the numbers in the column Check Run/Fail. The sixth column (Search Time) presents the total time (in minutes) required to search the space of parallel programs. The seventh column (Check Run/Fail) shows the number of executions that were required in order to pass the statistical accuracy test. Five of the 1,606 executions of Water failed to satisfy the accuracy bound $b$. For all of the other benchmarks all of the executions satisfied the accuracy bound. The eighth and final column (Check Time) presents the total time (in minutes) required to perform the test executions during the final statistical accuracy test for the final version of the application.

We note that, with the exception of String, both inputs induce identical parallelizations. As discussed further, for String, the two different parallelizations are acceptable and produce roughly equivalent performance for both inputs.

*Barnes-Hut.* The representative inputs for Barnes-Hut differ in the number of bodies they simulate and the number of time steps they perform. The accuracy metric computes the relative differences between the various aspects of the state of the final system, specifically the total kinetic and potential energy and the position and velocity of the center of mass.

The report indicates that Barnes-Hut spends almost all of its time in the outer loop that iterates through the time steps in the simulation. The attempted parallelization of this loop crashes with a segmentation violation. QuickStep proceeds on to the main force computation loop which iterates over all the bodies, using the space subdivision tree to compute the force acting on that body. This loop is nested within the time step loop; the report indicates that Barnes-Hut spends the vast majority of its time in this loop. Because there are no cross-iteration dependences in the force computation loop, QuickStep's parallelization of this loop produces a parallel program that deterministically produces the same result as the original sequential program. QuickStep proceeds on to attempt to parallelize several more loops (both alone and in combination with each other and the main force computation loop), but even after the application of replication introduction transformations in these loops it is unable to produce a parallelization that outperforms the parallelization of the main force computation loop by itself. We attribute the performance loss to unsuccessfully amortized overhead of initialization of the replicated versions of data structures.

*Search.* The representative inputs for Search differ in the number of traced particles. The accuracy metric computes the relative difference between the number of particles that scatter out of the front of the solid in the sequential and parallel computations.

The report indicates that Search spends almost all of its computation time within a single inner loop. All parallelizations that attempt to execute this loop in parallel fail because the program crashes with a segmentation fault. QuickStep also explores the parallelization of various combinations of five outer loops, each of which executes the inner loop. Together, these loops account for almost all of the computation time. QuickStep is able to parallelize all of these loops with no applied synchronization or replication transformations and no distortion. An examination of the source code reveals that all iterations of these loops are independent and the parallelization is valid for all inputs.

*String.* The representative inputs for String differ in the starting geology model and in the number of rays that they trace through the velocity model. The accuracy metric computes the mean scaled difference between corresponding components of the velocity models from the sequential and parallel computations.

The report indicates that String spends almost all of its time within a single outer loop that updates the velocity model. QuickStep is able to parallelize this loop. Our examination of the source code indicates that parallel iterations of this loop may contain (infrequent) data races—it is possible for two iterations to update the same location in the geology model at the same time. The fact that the distortion is small but nonzero (0.0005 for the representative inputs) reinforces our understanding that while some data races may actually occur in practice, they occur infrequently enough so that they do not threaten the acceptable accuracy of the computation.

For each input, QuickStep is also able to successfully parallelize an additional loop. The search for the `big` input finds an outer loop in a loop nest, while the search for the `inv` input finds an inner loop from the same loop nest. In both cases, the loop nest contributes to less than 4% of the execution time. Manual inspection shows that the iterations of these loops are independent, and therefore parallelizing the loops does not affect the accuracy.

We executed each final parallelization on the input used to derive the other parallelization, with the number of executions determined by the statistical accuracy test. All executions on all inputs satisfied the statistical accuracy bounds. The parallel version from the `inv` input produced a mean speedup of 7.55 and distortion of 0.0004 for the `big` input; the parallel version from the `big` input produced a mean speedup of 7.57 and distortion of 0.0005 for the `inv` input. These numbers suggest that both parallelizations are acceptable; inspection of the source code via the interactive report confirms this acceptability.

*Volume Rendering.* The representative inputs for Volume Rendering differ in the size of the input volume data. The accuracy metric is based on the final image that the application produces—specifically, it computes the mean scaled difference between corresponding pixels of the images from the sequential and parallel computations.

The report indicates that Volume Rendering spends almost all of its time in three nested loops: an outer loop that iterates over the views in the view sequence, a nested loop that iterates over the y-axis of the current view, and an inner loop that, given a y-axis from the enclosing loop, iterates over the x-axis to trace the rays for the corresponding x,y points in the view. QuickStep first tries to parallelize the outer loop. This parallelization fails because the application crashes with a segmentation violation.

QuickStep next attempts to parallelize the nested loop (which iterates over the y-axis). This parallelization succeeds and produces a parallel computation with good performance and no distortion. QuickStep also succeeds in parallelizing the inner loop by itself but this parallelization does not perform as well as the parallelization of the nested loop (which contains the inner loop). An examination of the source code indicates that the ray tracing computations are independent and that the parallelization is valid for all inputs.

*Water.* The representative inputs for Water differ in the number of water molecules they simulate. The accuracy metric for Water is based on several values that the simulation produces as output, specifically the energy of the system, including the kinetic energy of the system of water molecules, the intramolecular potential energy, the intermolecular potential energy, the reaction potential energy, the total energy, the temperature, and a virtual energy quantity.

The reports indicate that the vast majority of the computation time in Water is consumed by an outer loop that iterates over the time steps in the simulation. The attempted parallelization of this loop fails because the resulting distortion is much larger than the accuracy bound. As discussed in Section 2, QuickStep proceeds to successfully parallelize the outer loops in the `interf` and `poteng` computations.

While the final computation has some unsynchronized data races, these data races occur infrequently enough to keep the final distortion (0.002) within the accuracy bound (0.003 for this example). When running the statistical test on the input `1000`, five of the executions produce output whose distortion is over the accuracy bound. For that reason, the test must perform more executions to obtain the evidence that the likelihood of a parallel execution satisfying the accuracy bound is acceptable.

When we set the accuracy bound $b$ to 0.0, QuickStep is able to discover a parallel version of the computation which does not have unsynchronized data races. This parallel version is over six times faster than the sequential program. Manual code inspection reveals that this parallelization produces valid results for all inputs. However, because of the additional synchronizations in the `interf` computation, this parallel version is 11% slower than the parallel version with infrequent data races. This example highlights QuickStep's ability to produce the most appropriate parallelization for the specific accuracy bound.

### 6.3. Comparison with icc Compiler

To provide at least one comparison point with standard parallelizing compiler techniques, we attempted to use the Intel icc compiler 11[3] to parallelize our benchmark applications. In this section we summarize the comparison presented in Misailovic et al. [2010b]. We used three values of the Intel icc `par-threshold` parameter, which determines how aggressively the compiler parallelizes loops. Higher threshold levels direct icc to parallelize only loops that are more likely to improve the performance.

At the default threshold level 100, icc parallelizes no loops. At threshold level 99, icc parallelizes zero loops in String, Water, and Barnes-Hut, one loop in Search, and five loops in Volume Rendering. At threshold level 0 icc parallelizes all loops that it identifies as parallelizable. At this level, it parallelizes 12 loops in String, 13 in Search and Volume Rendering, 21 loops in Water, and 22 loops in Barnes-Hut. However, only one of these loops accounts for more than 1% of the execution time of the application.

[3] `http://software.intel.com/en-us/intel-compilers/`

The loops discovered by icc are all small inner loops whose parallelization never significantly improves the performance and in many cases significantly degrades the performance. Intel icc did not select any of the loops parallelized by QuickStep. These results highlight the difficulty of performing the static analysis required to parallelize large outer loops using standard techniques and the effectiveness of the QuickStep approach in parallelizing programs that use modern programming constructs.

## 7. RELATED WORK

We discuss related work in parallelizing compilers, interactive profile-driven parallelization, statistical accuracy models for parallel computations, unsound program transformations, and data race detection and repair.

*Parallelizing Compilers.* There is a long history of research in developing compilers that can automatically exploit parallelism available in programs that manipulate dense matrices using affine access functions. This research has produced several mature compiler systems, including Open64[4] with demonstrated success at exploiting this kind of parallelism [Blume et al. 1995; Hall et al. 1996]. Our techniques, in contrast, are designed to exploit parallelism available in loops regardless of the specific mechanisms the computation uses to access data. Because the acceptability of the parallelization is based on an analysis of the output of the parallelized program rather than an analysis of the program itself with the requirement of generating a parallel program that produces identical output to the sequential program, QuickStep is dramatically simpler and less brittle in the face of different programming constructs and access patterns. To cite just one example, our results show that it can effectively parallelize object-oriented computations written in C++ that heavily use object references and pointers. The use of any one of these programming language features is typically enough to place the program beyond the reach of standard parallelizing compilers.

Commutativity analysis [Aleen and Clark 2009; Kim and Rinard 2011; Rinard and Diniz 1997] analyzes sequential programs to find operations on objects that produce equivalent results regardless of the order in which they execute. If all of the operations in a computation commute, it is possible to execute the computation in parallel (with commuting updates synchronized to ensure atomicity). QuickStep takes a simpler approach that can successfully parallelize a broader range of programs.

Motivated by the difficulty of exploiting concurrency in sequential programs by a purely static analysis, researchers have developed approaches that use speculation. These approaches (either automatically or with the aid of a developer) identify potential sources of parallelism before the program runs, then run the corresponding pieces of the computation in parallel with mechanisms designed to detect and roll back any violations of dependences that occur as the program executes [Bridges et al. 2007; Prabhu and Olukotun 2005; Rauchwerger and Padua 1995; Tinker and Katz 1988]. These techniques typically require additional hardware support, incur dynamic overhead to detect dependence violations, and do not exploit concurrency available between parts of the program that violate the speculation policy. Our approach, in contrast, operates on stock hardware with no dynamic instrumentation. It can also exploit concurrency available between parts of the program with arbitrary dependences (including unsynchronized data races) as long as the violation of the dependences does not cause the program to produce an unacceptably inaccurate result.

Another approach to dealing with static uncertainty about the behavior of the program is to combine static analysis with runtime instrumentation that extracts additional information (available only at runtime) that may enable the parallel execution

[4] Open64: Open Research Compiler. `http://www.open64.net`.

of the program [Ding and Li 2003; Rauchwerger and Padua 1995; Rauchwerger et al. 1995; Rus et al. 2007]. Once again, the goal of these approaches is to obtain a parallel program that always produces the same result as the sequential program. Our approach, on the other hand, requires no runtime instrumentation of the parallel program and can parallelize programs even though they violate the data dependences of the sequential program (as long as violations do not unacceptably perturb the output).

*Profile-Driven Parallelization.* Profile-driven parallelization approaches run the program on representative inputs, dynamically observe the memory access patterns, then use the observed access patterns to suggest potential parallelizations that do not violate the observed data dependences [Ding et al. 2007; Rul et al. 2008; Tournavitis et al. 2009]. The dynamic analysis may be augmented with a static analysis to recognize parallel patterns, such as reductions.

QuickStep uses a fundamentally different parallelization approach—instead of attempting to preserve the data dependences, it deploys a set of parallelization strategies that enable it to explore a much broader range of potential parallelizations, including parallelizations with synchronized commuting operations, acceptable data races, and acceptable reorderings that violate the data dependences. Our experimental results show that QuickStep's broader reach is important in practice. The final parallelizations of two of the applications in our benchmark set (String and Water) contain unsynchronized data races that violate the underlying data dependences, which places these efficient parallelizations beyond the reach of any technique that attempts to preserve these dependences.

Alter provides a deterministic execution model that supports parallelizations that read stale data or otherwise violate the data dependences of the sequential program [Udupa et al. 2011]. QuickStep produces simpler parallelizations, including efficient nondeterministic computations with acceptable unsynchronized data races.

Researchers have also studied the effects of program parallelizations with relaxed data dependencies on several data mining benchmarks [Meng et al. 2009, 2010]. The researchers use profiling information to identify computational patterns that, when parallelized with potential remaining data races, execute significantly faster while incurring only small accuracy losses. However, the parallelization is manual and does not provide statistical accuracy bounds that the nondeterministic parallel program will produce an acceptably accurate result. Kirsch et al. [2011] propose an approximate concurrent FIFO queue with the relaxed constraint on the order of the elements and derive probabilistic bound on the amount of deviation of this approximate queue from a perfect FIFO queue.

*Statistical and Probabilistic Accuracy Models.* Recent research has developed statistical accuracy models for parallel programs that discard tasks, either because of failures or to purposefully reduce the execution time [Rinard 2006]. A conceptually related technique eliminates idle time at barriers at the end of parallel phases of the computation by terminating the parallel phase as soon as there is insufficient computation available to keep all processors busy [Rinard 2007]. The research presented in this article, on the other hand, uses user-defined accuracy tests in combination with the SPR test to obtain a statistical guarantee of the accuracy of an envisioned parallelization. In comparison with previous approaches, this approach requires fewer assumptions on the behavior of the parallel computation but more test executions to obtain tight statistical distortion bounds.

Loop perforation [Hoffmann et al. 2009; Misailovic et al. 2010; Sidiroglou et al. 2011] can be seen as a specialization of discarding tasks in which the discarded tasks are loop bodies. It would be possible to apply the statistical accuracy techniques presented in this article to obtain statistical accuracy bounds for perforated programs. It is also possible to use Monte-Carlo simulation to obtain statistical models of the effect of perforating loops [Rinard et al. 2010]. Static analyses can produce probabilistic models that characterize this effect [Chaudhuri et al. 2011; Misailovic et al. 2011a, 2011b].

*Unsound Program Transformations.* We note that this article presents techniques that are yet another instance of an emerging class of unsound program transformations. In contrast to traditional sound transformations (which operate under the restrictive constraint of preserving the semantics of the original program), unsound transformations have the freedom to change the behavior of the program in principled ways. Previous unsound transformations have been shown to enable applications to productively survive memory errors [Berger and Zorn 2006; Rinard et al. 2004], code injection attacks [Perkins et al. 2009; Rinard et al. 2004], data structure corruption errors [Demsky and Rinard 2005; Demsky et al. 2006], memory leaks [Nguyen and Rinard 2007], and infinite loops [Carbin et al. 2011]. The fact that all of these techniques provide programs with capabilities that were previously unobtainable without burdensome developer intervention provides even more evidence for the value of this new approach.

*Data Race Detection and Repair.* Researchers have developed tools that detect the presence of data races in parallel programs [Dinning and Schonberg 1991]. It is possible to use the synchronization introduction transformation presented in this article to automatically eliminate data races (to the best of our knowledge, this is the first such mechanism capable of automatically eliminating data races [Misailovic et al. 2010a, 2010b]). The AFix system introduces synchronization to eliminate single-variable atomicity violations [Jin et al. 2011]. This mechanism could also be used to enhance the accuracy of the parallel programs that QuickStep generates.

## 8. CONCLUSION

Parallelizing compilers have achieved demonstrated successes within specific computation domains, but many computations remain well beyond the reach of traditional approaches. The difficulty of building compilers that use these approaches and the large classes of programs that currently (and in some cases inherently) lie beyond their reach leaves room for more effective techniques that can parallelize a wider range of programs. Our results indicate that QuickStep's basic approach, which involves the combination of parallelization transformations and search of the resulting induced space of parallel programs guided by test executions on representative inputs, provides both the simplicity and broader applicability that the field requires.

## ACKNOWLEDGMENTS

We would like to thank Dan Roy for his help with the statistical accuracy test and Stelios Sidiroglou and Danny Dig for their useful comments on the earlier drafts of this work.

## REFERENCES

Aleen, F. and Clark, N. 2009. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proceedings of ASPLOS*.

Barnes, J. and Hut, P. 1986. A hierarchical O(NlogN) force calculation algorithm. *Nature 324,* 4, 446–449.

Berger, E. and Zorn, B. 2006. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of PLDI*.

Blume, W. and Eigenmann, R. 1992. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Trans. Parallel Distrib. Syst. 3,* 6.

Blume, W., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, W., Raughwerger, L., Tu, P., and Weatherford, S. 1995. Effective automatic parallelization with Polaris. *Int. J. Parallel Program*.

Bolosky, W. and Scott, M. 1993. False sharing and its effect on shared memory performance. In *Proceedings of SEDMS*.

Bridges, M., Vachharajani, N., Zhang, Y., Jablin, T., and August, D. 2007. Revisiting the sequential programming model for multi-core. In *Proceedings of MICRO*.

Browning, R., Li, T., Chui, B., Ye, J., Pease, R., Czyzewski, Z., and Joy, D. 1995. Low-energy electron/atom elastic scattering cross sections for 0.1-30keV. *Scanning 17,* 4, 250–253.

Carbin, M., Misailovic, S., Kling, M., and Rinard, M. 2011. Detecting and escaping infinite loops with Jolt. In *Proceedings of ECOOP*.

Chaudhuri, S., Gulwani, S., Lublinerman, R., and Navidpour, S. 2011. Proving programs robust. In *Proceedings of ESEC/FSE*.

Dagum, L. and Menon, R. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5,* 1, 46–55.

Demsky, B. and Rinard, M. 2005. Data structure repair using goal-directed reasoning. In *Proceedings of ICSE*.

Demsky, B., Ernst, M., Guo, P., McCamant, S., Perkins, J., and Rinard, M. 2006. Inference and enforcement of data structure consistency specifications. In *Proceedings of ISSTA*.

Ding, C., Shen, X., Kelsey, K., Tice, C., Huang, R., and Zhang, C. 2007. Software behavior oriented parallelization. In *Proceedings of PLDI*.

Ding, Y. and Li, Z. 2003. An adaptive scheme for dynamic parallelization. In *Proceedings of LCPC*, H. Dietz Ed., Lecture Notes in Computer Science, vol. 2624. Springer-Verlag, 274–289.

Dinning, A. and Schonberg, E. 1991. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*.

Hall, M., Anderson, J., Amarasinghe, S., Murphy, B., Liao, S., Bugnion, E., and Lam, M. 1996. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*.

Harris, J., Lazaratos, S., and Michelena, R. 1990. Tomographic string inversion. In *Proceedings of the 60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*.

Herlihy, M and Moss, J. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of ISCA*.

Hoeffding, W. 1963. Probability inequalities for sums of bounded random variables. *J. Am. Stat. Assoc. 58,* 301, 13–30.

Hoffmann, H., Misailovic, S., Sidiroglou, S., Agarwal, A., and Rinard, M. 2009. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Tech. rep. MIT-CSAIL-TR-2009-042, MIT, Cambridge, MA.

Jin, G., Song, L., Zhang, W., Lu, S., and Liblit, B. 2011. Automated atomicity-violation fixing. In *Proceedings of PLDI*.

Kim, D. and Rinard, M. C. 2011. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of PLDI*.

Kirsch, C., Payer, H., Röck, H., and Sokolova, A. 2011. Performance, scalability, and semantics of concurrent FIFO queues. Tech. rep. 2011-03, Department of Computer Sciences, University of Salzburg.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of CGO*.

Meng, J., Chakradhar, S., and Raghunathan, A. 2009. Best-effort parallel execution framework for recognition and mining applications. In *Proceedings of IPDPS*.

Meng, J., Raghunathan, A., Chakradhar, S., and Byna, S. 2010. Exploiting the forgiving nature of applications for scalable parallel execution. In *Proceedings of IPDPS*.

Misailovic, S., Kim, D., and Rinard, M. 2010a. Automatic parallelization with statistical accuracy bounds. Tech. rep. MIT-CSAIL-TR-2010-007, MIT, Cambridge, MA.

Misailovic, S., Kim, D., and Rinard, M. 2010b. Parallelizing sequential programs with statistical accuracy tests. Tech. rep. MIT-CSAIL-TR-2010-038, MIT, Cambridge, MA.

Misailovic, S., Roy, D., and Rinard, M. 2011a. Probabilistic and statistical analysis of perforated patterns. Tech. rep. MIT-CSAIL-TR-2011-003, MIT, Cambridge, MA.

Misailovic, S., Roy, D., and Rinard, M. 2011b. Probabilistically accurate program transformations. In *Proceedings of SAS*.

Misailovic, S., Sidiroglou, S., Hoffmann, H., and Rinard, M. 2010. Quality of service profiling. In *Proceedings of ICSE*.

Nguyen, H. and Rinard, M. 2007. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of ISMM*.

Nieh, J. and Levoy, M. 1992. Volume rendering on scalable shared-memory MIMD architectures. Tech. rep. CSL-TR-92-537, Computer Systems Laboratory, Stanford Univ., Stanford, CA.

Perkins, J., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W., Zibin, Y., Ernst, M. D., and Rinard, M. 2009. Automatically patching errors in deployed software. In *Proceedings of SOSP*.

Prabhu, M. and Olukotun, K. 2005. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of PPoPP*.

Rauchwerger, L. and Padua, D. 1995. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. In *Proceedings of PLDI*.

Rauchwerger, L., Amato, N., and Padua, D. 1995. Runtime methods for parallelizing partially parallel loops. In *Proceedings of ICS*.

Rinard, M. 1994. The design, implementation and evaluation of Jade, a portable, implicitly parallel programming language. Ph.D. dissertation, Dept. of Computer Science, Stanford Univ., Stanford, CA.

Rinard, M. 2003. Acceptability-oriented computing. In *Proceedings of OOPSLA Onwards! Session*.

Rinard, M. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of ICS*.

Rinard, M. 2007. Using early phase termination to eliminate load imbalancess at barrier synchronization points. In *Proceedings of OOPSLA*.

Rinard, M. and Diniz, P. 1997. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst. 19,* 6.

Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and William S. Beebee, J. 2004. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of OSDI*.

Rinard, M., Hoffmann, H., Misailovic, S., and Sidiroglou, S. 2010. Patterns and statistical analysis for understanding reduced resource computing. In *Proceedings of OOPSLA Onwards!*

Rul, S., Vandierendonck, H., and De Bosschere, K. 2008. A dynamic analysis tool for finding coarse-grain parallelism. In *Proceedings of HiPEAC Industrial Workshop*.

Rus, S., Pennings, M., and Rauchwerger, L. 2007. Sensitivity analysis for automatic parallelization on multicores. In *Proceedings of ICS*.

Sidiroglou, S., Misailovic, S., Hoffmann, H., and Rinard, M. 2011. Managing performance vs. accuracy tradeoffs with loop perforation. In *Proceedings of ESEC/FSE*.

Tinker, P. and Katz, M. 1988. Parallel execution of sequential Scheme with Paratran. In *Proceedings of LFP*.

Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M. 2009. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of PLDI*.

Udupa, A., Rajan, K., and Thies, W. 2011. Alter: Leveraging breakable dependences for parallelization. In *Proceedings of PLDI*.

Wald, A. 1947. *Sequential Analysis*. John Wiley and Sons.

Woo, S., Ohara, M., Torrie, E., Singh, J., and Gupta, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of ISCA*.

Received July 2011; revised September 2011; accepted November 2011