Dancing with Uncertainty

Sasa Misailovic Stelios Sidiroglou Martin C. Rinard
MIT CSAIL
{misailo, stelios, rinard}@csail.mit.edu

Abstract

We present Dubstep, a novel system that uses the *find-transform-navigate* approach to automatically explore new parallelization opportunities in already parallelized (fully-synchronized) programs by opportunistically relaxing synchronization primitives. This set of transformations generates a space of alternative, possibly non-deterministic, parallel programs with varying performance and accuracy characteristics. The freedom to generate parallel programs whose output may differ (within statistical accuracy bounds) from the output of the original program enables a significantly larger optimization space. Dubstep 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, fully synchronized parallel program would have produced.

Initial results from our benchmarked application show that Dubstep can generate acceptably accurate and efficient versions of a parallel program that occupy different positions in performance/accuracy trade off space.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming; D.3.4 [*Programming Languages*]: Processors – Optimization

Keywords Parallelization, Accuracy, Tradeoff, Statistical Test

1. Introduction

Parallel programming is often considered difficult – programmers struggle with the design of efficient, concurrent algorithms and have difficulty reasoning about potential dependences which may limit parallelism (and thus overall efficiency). The dominant approach for reasoning about the behavior of software systems, which revolves around hard binary correctness, further exacerbates the problem. Coupled with the inherent difficulty of testing and debugging parallel code, programmers resort to conservative approaches (e.g. conservative over synchronization) that limit parallelism.

Relaxing the notion of binary correctness frees the programmer from the constraint of preserving precise semantics. This freedom may open up a much larger optimization space with additional opportunities for reduced engineering effort, reduced resource consumption, or increased functionality. But fully exploiting these new opportunities may require additional reasoning about trade-offs between accuracy, performance, and power consumption.

In recent work, our research group has advocated for the use of *accuracy-aware transformations* that exploit the freedom to change the semantics of the transformed program to trade accuracy for improved performance and/or energy consumption. We have proposed empirical, statistical, and probabilistic techniques for understanding the effects of these transformations [9, 14, 16, 17, 19–21, 24, 30].

Several of these techniques use a *find-transform-navigate* design approach to discover optimization opportunities. In this paper we present a version of this approach that uses synchronization reduction transformations to find new opportunities for improving the performance of already parallelized programs. First, we analyze the original program to *find* locations in the program that are potentially promising optimization candidates. Then, we *transform* each of these locations, one at a time, and analyze the effect of the individual transformation on the program's performance, accuracy, and safety. The transformed program has the freedom to produce an acceptably accurate result that is different from the result of the original program. Finally, understanding the effects of individual transformations lets us efficiently *navigate* the trade-off space of multiple transformations applied at the same time as we search for a transformed program that maximizes performance subject to an acceptability bound on the result that it produces.

We present Dubstep, a novel system that implements the find-transform-navigate approach to automatically explore new parallelization opportunities in already parallelized (fully-synchronized) programs by relaxing synchronization primitives. This set of transformations generates

RACES'12, October 21, 2012, Tucson, Arizona, USA.

a space of alternative, possibly non-deterministic, parallel programs with varying performance and accuracy characteristics. Dubstep explores this space to find more efficient parallel programs that, with high likelihood, produce acceptably accurate results. Dubstep builds upon our previous work on Quickstep, an interactive parallelizing compiler that finds parallelization opportunities in sequential programs and produces (potentially non-deterministic) parallel programs [13–15].

2. Approach

We next summarize how Dubstep's implements the find-transform-navigate approach to identifying and exploiting approximate optimization opportunities.

2.1 Find

Representative Inputs and Accuracy Model: A user provides Dubstep with a source code of the program and a set of inputs that represent typical program's workloads. Dubstep then uses these inputs to evaluate the accuracy and performance effects of candidate transformations.

A user also specifies an *output abstraction* function that is used to select relevant parts of the output or calculate a measure of output's quality. This is a function that takes as input the program's output and calculates a single numerical value or a vector of numerical values that summarizes the relevant properties of the output. Common output abstractions extract individual output components produced directly by the program, or compute quality metrics of the output, such as the signal to noise ratio in signal processing applications.

Analyze Original Program: Dubstep analyzes the original, fully-synchronized parallel program to help identify new optimization opportunities:

- **Baseline execution:** Dubstep runs the original parallel program on the representative inputs. It records the running times and uses the user-provided output abstraction function to compute the relevant output metrics. The analyses of the transformed programs use these baseline results to evaluate the accuracy and performance of the transformed versions.
- **Performance Profiling:** Next, Dubstep applies a performance profiler on the original parallel program to find the program locations where the programs spends most of its time. Specifically, the profiler focuses on the percentage of time that computations spend in existing parallel sections. Profiling allows Dubstep to focus on program components that are likely to yield high performance benefits.
- **Memory profiling:** Dubstep next generates an instrumented version of the program that generates a trace of memory access patterns (sequences of reads and writes). It executes this instrumented version of the program to identify program regions with high and low memory contention. Relaxing synchronization primitives around low-contention memory regions is more likely to produce highly accurate transformed programs.

2.2 Transform

Transformations: The transformations are designed to improve the performance of the parallel program while maintaining acceptable accuracy. Dubstep deploys two types of transformations to generate its search space of parallel programs:

- **Opportunistic Synchronization:** This transformation replaces synchronized operations with unsynchronized operations (by removing locks). The goal is to decrease synchronization overhead and lock waiting times by allowing data races that do not alter, or only slightly alter, the result that the program produces.
- **Opportunistic Barriers:** This transformation replaces traditional synchronization barriers (which force threads to wait until all threads have reached the barrier) with barriers that wait only for a specified fraction of the threads to reach the barrier. The barrier then instructs the remaining (non-waiting) threads to stop their work and all threads proceed past the barrier. The goal is to reduce waiting time at barriers (this waiting time is typically caused by thread scheduling anomalies and differences in the amount of work assigned to each thread).

Both of these transformations introduce non-determinism and may cause the application to produce varying results when executed on the same input.

Analyze Transformations: To evaluate, understand and predict the behavior of the resulting non-deterministic programs, Dubstep runs analyses that (1) help the developer understand the tradeoff between accuracy and performance and (2) ensure that the executions of the transformed programs are safe.

All analyses are *relational* in the sense that they use executions of the original program to reason about executions of the transformed program.

- **Performance Analysis:** The goal of this analysis is to identify transformations that improve end-to-end program performance. The performance measure is a speedup T_0/T', where T_0 is the average execution time of the original program and T' is the average execution time of the transformed program. To obtain T', the analysis executes the transformed program on the same inputs it used for the original program.
- **Accuracy Analysis:** The goal of this analysis is to identify transformations that preserve the accuracy of the program within acceptable bounds. By applying an output abstraction function on outputs of the original and transformed program, Dubstep obtains two abstracted output vectors $\mathbf{o}$ (for the original program) and $\hat{\mathbf{o}}$ (for the transformed program). Dubstep then relates these two outputs using an *accuracy metric*, which calculates the distance

between the two vectors. In this paper we will use average absolute relative error as the accuracy metric:

$$d(\mathbf{o}, \hat{\mathbf{o}}) = \frac{1}{m} \sum_{i=1}^{m} \left| \frac{o_i - \hat{o}_i}{o_i} \right| \quad (1)$$

The user also defines an *acceptable accuracy loss bound* b. If the value d of the accuracy measure is less than the accuracy loss bound b that the user defined, then the execution is considered acceptably accurate. If, in contrast, the value of d is greater than b, then the execution accuracy loss is unacceptably high.

Since the transformed programs are non-deterministic, multiple executions of a program on the same input can result in different results of the accuracy metric d. To represent accuracy in this scenario, we define an *execution reliability* as a probability that the accuracy loss d of the result that the non-deterministic transformed program produces on a single input is smaller than the bound b.

We accept the transformed program if its reliability on all tested inputs is higher than an *acceptance reliability* r that the user defines. To estimate the reliability of each parallel program, Dubstep uses *statistical testing*. We review to these techniques in Section 4.

- **Safety Analysis:** We use *criticality testing* to discover program crashing errors, which may be introduced as a consequence of an applied transformation. Criticality testing instruments the original and transformed programs using dynamic analysis to detect critical errors introduced by the applied transformations (e.g. reading uninitialized memory) [17, 24]. The goal of criticality testing is to eliminate the introduction of new memory and parallelization errors that may occur as a result of the applied transformations. Note that data races introduced by the applied transformations that do not lead to abrupt program termination (or unacceptable accuracy degradation) are not considered critical errors. Criticality testing only checks for errors introduced by transformations. In other words, if a certain error exists in the original program, then it is acceptable for the transformed program to exhibit the same error. Conversely, if an error does not exist in the original program then it should not exist in the transformed program.

2.3 Navigate

Given an initial program, the opportunistic transformations generate a new, larger, space of corresponding alternative programs with relaxed synchronization. Each alternative program may have one or more locations transformed.

Exhaustive Search: If the number of alternative programs is small, Dubstep can exhaustively explore all of them, applying performance, accuracy, and safety analysis on each explored transformed program.

Pruning: Dubstep can also use a two-stage exploration algorithm to prune parts of the tradeoff space that it explores. In the first stage, Dubstep applies a transformation to a single location in the program, and runs performance, accuracy and safety tests. If a candidate transformed program does not pass one of the tests, the Dubstep will not use this transformation in the second stage.

In the second stage, Dubstep explores the tradeoff space induced by combinations of successful transformations from the first stage (those that passed all three tests) to find additional transformed programs that profitably trade accuracy for performance. Specifically, Dubstep uses Quickstep's exploration algorithm [14, Sec. 5], which prioritizes individual transformations that demonstrated better tradeoffs between accuracy and performance.

2.4 Results

Dubstep presents its findings to developers in a detailed report. This report contains the list of applied transformations and the effects that these transformations had on accuracy and performance of programs, ordered by the tradeoff between performance increase and accuracy loss that they cause. A developer can use the report to evaluate the acceptability of the synchronization transformations and obtain insight into how the application responds to different synchronization reduction strategies.

Confidence. Dubstep implements its accuracy analysis using a statistical test. Note that statistical tests are not sound decision procedures — since they use a finite number of observations to make their decision, statistical tests may (ideally with low likelihood) make an incorrect decision. A developer can use a confidence c to control the likelihood of making an incorrect decision. The confidence is the rate at which the accuracy analysis returns a correct answer (i.e., accepts a parallelization that should be accepted and rejects a parallelization that should be rejected).

Execution Time. The confidence of the result, together with the desired acceptance reliability r, influence the number of trials that the analysis needs to execute, and thus increases the execution time of the analysis. Greater reliability and confidence require more trials for a procedure to compute its answer. We present more details about this relationship in Section 4.

3. Example

In this section we present a preliminary evaluation of Dubstep on Water, a C++ version of the Perfect Club benchmark MDG [3]. Water evaluates forces and potentials in a system of water molecules in the liquid state. Figure 1 presents the main computation that Water performs.

Computation: Water computes pairwise interactions between simulated water molecules (they are stored in wrapper `ScratchPad` objects). The parallel computation `interf`

```
void ensemble::interf(){
   parallel_for(interf_internal, 0, NumMol-1);
}

void ensemble::interf_internal(int i){
   double Res1[3][3], Res2[3][3];
   ScratchPad *p1, *p2;

   for(j = i+1; j < numMol; j++){
      p1 = getPad(j); p2 = getPad(i);

      double incr = cshift(p1,p2,Res1,Res2);
      p1->updateForces(Res1);
      p2->updateForces(Res2);
      VIR.addvalRepl(incr);
   }
}

void scratchPad::updateForces(double Res[3][3]) {
   parallel_mutex_lock(this->lock)
   this->H1force.vecAdd(Res[0]);
   this->Oforce.vecAdd(Res[1]);
   this->H2force.vecAdd(Res[2]);
   parallel_mutex_unlock(this->lock);
}
```

```
void ensemble::poteng(){
   parallel_for(poteng_internal, 0, numMol-1);
}

void ensemble::poteng_internal(){
   double Res [3][3]

   for(j = i+1; j < numMol; j++){
      computePoteng(Res, getPad(i), getPad(j));
      potenergy->vecAddRepl(Res);
  }
}

void ensemble::main(){
  // ...
  for (int t = 0; t < T; t++) {
     computation_1();
     interf();
     computation_2();
     poteng();
     computation_3();
  }
  // ...
  // output system status
}
```

Figure 1. Example Water Computation

calculates the interactions between every two molecules. It calls `cshift` to compute the results of each interaction and store the results in two 3 by 3 arrays (`Res1` and `Res2`). Then, `updateForces` takes the two arrays and updates the vectors that store the forces acting on each molecule. The `addval` updates the `VIR` accumulator object, which stores the sum of the virtual energies of all interactions. The `poteng` function takes as input the state of the system of molecules, computes the potential energy of the system and stores it in the `potenergy` vector variable. Finally, the main computation evolves the system over time — at each time step it calculates the interactions between molecules, their positions, and energies. The intermediate computations inside the main `for` loop (`computation_i`) are short sequential computations. The application outputs the final state of the system of molecules, including the total kinetic and potential energy of the system.

Parallelization Primitives: Both `interf` and `poteng` use the `parallel_for` function call. This function executes the function passed as the first parameter in parallel threads. The second and the third parameter of `parallel_for` specify the starting and ending iteration point respectively. Each thread increments the index, and passes it to the function it executes as a parameter, until it reaches the final point. In this experiment we assume the cyclic scheduling strategy (each thread increments its local index by a constant equal to the number of threads).

Internally, the function `parallel_for` uses two barriers, one at the beginning to ensure that the computation enters the loop body only after all threads have completed previous work, and the one at the end to ensure that the computation continues only after all threads have finished their work.

The `interf` computation has two synchronization primitives. The accumulator function `addvalRepl` implements a replicated update to the accumulator. The `updateForces` subcomputation for each molecule is synchronized with a lock that belongs to each `ScratchPad` object to avoid simultaneous updates from multiple threads. Each scratchpad has its own unique lock. The `poteng` computation has one synchronization primitive. The vector addition function `vecAddRepl` implements a replicated update to the vector.

Dubstep Input: Dubstep can automatically explore the application to find new parallelization opportunities when allowed the freedom to produce output that may differ from the original manually parallelized program. To use Dubstep, an application developer provides the program's original source code, a set of representative inputs, and an output abstraction function.

For this experiment, we provide a simulation of 1000 molecules for 30 time steps. The initial positions of the molecules are also part of the input. The output abstraction vector for water, $\mathbf{o}$, consists of measures of the kinetic and potential energies of the system. These are values that the original program produces; the output abstraction function extracts these values from the program's printed output. In addition, a developer provides a specified accuracy loss bound b and reliability r.

Transformation	Speedup (max 8)	Relative Speedup	Accuracy Loss
Original	6.21	1.00	0.000 ±0.000
BarrierInterf	6.34	1.02	0.027 ±0.082
BarrierPoteng	6.48	1.04	0.035 ±0.032
LockForces	6.34	1.02	0.004 ±0.001

Table 1. Empirical Results for Individual Transformations

Transformation	Speedup (max 8)	Relative Speedup	Accuracy Loss
Original	6.21	1.00	0.000 ±0.000
BarrierInterf + LockForces	6.44	1.03	0.027 ±0.044
BarrierPoteng + LockForces	6.79	1.09	0.042 ±0.033
BarierInterf + BarrierPoteng	7.10	1.14	0.053 ±0.063
All Three	7.44	1.20	0.051 ±0.070

Table 2. Empirical Results for Combinations of Transformations

Experimental Environment: We performed all experiments on a dual quad-core 2.27 GHz Intel Xeon E5520 CPU with 16 GB of RAM. The operating system is Ubuntu Linux 10.10 running kernel version 2.6.32-41. All programs are compiled with the LLVM C++ compiler version 2.7, with the optimization level `-O3`.

Roadmap: We first present the computations that Dubstep identifies as good optimization candidates in the *find* phase (Section 3.1). We next present Dubstep's exhaustive exploration of the tradeoff space induced by these optimization candidates (Section 3.2). We then present how Dubstep can navigate the tradeoff space and select the transformed program with highest performance for a set of accuracy bounds b guided by a statistical test (Section 3.3).

3.1 Find

Dubstep applies performance- and memory-profiling to find optimization opportunities in Water.

Performance Profiling: The execution time of the sequential program is 16.5 seconds. The original parallel program running with 8 parallel threads achieves speedup of 6.21 times over the sequential program. The loop profiling information (collected using Valgrind's Callgrind tool) indicates that 99.7% of the execution time is spent in the parallel sections of the `interf` and `poteng` functions. Dubstep thus narrows its optimization focus to these parallel sections. Since these computations contain synchronization barriers, they are a good target for the opportunistic barrier transformation. We transform the barriers at the end of each `parallel_for` section. We call these transformation opportunities **BarrierInterf** (for the `interf` loop) and **BarrierPoteng** (for the `poteng` loop).

Memory profiling: The memory profiling information highlights two high-contention memory regions in the parallel loops. These memory regions represent the addresses of the accumulator `VIR` accessed from the `addvalRepl` operation (inside the `interf` computation), and the vector accumulator `potenergy` accessed from the `vecAddRepl` operation (inside the `poteng` computation). These memory regions are accessed within every iteration of the parallel loops and are, therefore, not good candidates for our transformations.

On the other hand, updating vectors within `ScratchPad` objects inside the `updateForces` function results in a much smaller density of races. This is due to the fact that the number of scratch-pads is equal to the number of molecules and during the execution of multiple threads there is a much smaller chance for data races within individual scratch-pads. The lock in this function is identified by Dubstep as a promising candidate for the opportunistic synchronization transformation. In the rest of the section we call this transformation opportunity **LockForces**.

3.2 Exhaustive Tradeoff Space Exploration

In this section we present the result of the tradeoff space exploration for the Water benchmark. Since the number of candidate transformed locations is small, we use an exhaustive exploration of the tradeoff space. First, Dubstep applies one transformation at a time and evaluates the effects of that transformation on the program's performance, accuracy, and safety (through criticality testing). Then, Dubstep combines the transformations together and evaluates their joint effects.

Individual Transformations: Table 1 presents the results of executing the transformed Water benchmark with eight parallel threads. Column 1 (Transformation) presents the transformation reference name. Column 2 (Speedup) presents the speedup of the execution of the parallel program relative to the execution of the sequential program. Column 2 (Relative Speedup) presents the speedup relative to the fully-synchronized parallel program. Column 4 (Accuracy Loss) presents the average accuracy loss and the standard deviation of the accuracy loss. We used Equation 1 to calculate the accuracy loss. The results are based on 100 executions of each program on the representative input.

- **Opportunistic Barriers in `interf` and `poteng`:** Both transformations introduce non-determinism in the results as a consequence of early termination (i.e. not waiting for all threads to finish their work). For this particular experiment, we used an aggressive version of the opportunistic barrier transformation that waits for a half of the threads to arrive at the barrier before terminating the computation of the remaining threads. When half of the threads have reached the barrier, Dubstep's runtime system directs the remaining threads to terminate further computation after the current loop iteration.

  This transformation produces a faster version of the program — the program has less waiting time at the barrier and performs less work. However, due to the skipped work, the produced result is less accurate. We note that with cyclic scheduling each thread executes roughly the same amount of work. Reports produced by Dubstep indicate that the terminated threads complete more than 95% of their total iterations.

- **Remove Synchronization in `updateForces`:** This transformation introduces non-determinism due to data races that may lead to overwriting some of the intermediate sums that the `vecAdd` function computes.

  Due to a relatively large number of molecules in the input, the execution with data races (introduced by Dubstep) has minimal accuracy loss (with low variance) but it also does not drastically improve performance.

All transformed programs continued execution and terminated normally after producing output. Dynamic program analysis with Valgrind did not reveal memory errors caused by these transformations.

Combination of Transformations: It is a tractable task for Dubstep to exhaustively explore all configurations of the program. Table 2 presents results for all four combinations of multiple transformations. Note that as we combine multiple transformations performance increases, but the average accuracy loss also increases. We observe the maximum performance for the case when all three transformations are applied at the same time. This program executes 20% faster than the original, fully synchronized program. Its average accuracy loss is 0.051.

3.3 Tradeoff Space Exploration with Accuracy Tests

In this section we present how Dubstep can leverage statistical accuracy tests to determine, with high confidence, that the executions of the transformed program are reliable.

Dubstep uses the SPR (sequential probability ratio) hypothesis testing framework to determine if a transformed program is reliable. We now briefly summarize this test here and provide a more detailed description in Section 4.2. The SPR test takes as input the accuracy bound b and the acceptance reliability r, provided by the developer. The test executes the candidate transformed program on the representative input multiple times and observes after each run whether the accuracy loss d is smaller than the accuracy bound b. At the end the test returns TRUE if the actual reliability of the transformed program was greater than r (and, therefore, Dubstep accepts the transformed program) and FALSE if the actual reliability was smaller than r (and, therefore, Dubstep rejects the transformed program).

Accuracy Bound b	Best Transformation	Trials (Average)
0.01	LockForces	106
0.05	LockForces	100
0.10	All Three	161
0.15	All Three	100
0.20	All Three	100
0.25	All Three	100

Table 3. The Best Transformed Programs for Accuracy Loss Bounds

The number of the total observations that the test takes is not fixed. It depends on the previous observations of the program executions. In this experiment, we set the reliability $r = 0.9$. Our target confidence that the test returns the correct answer is also $c = 0.9$ (thus, the probability that the accuracy test fails is 0.1). This test also requires a user to set a borderline tolerance ε (this is a narrow interval around the acceptable reliability r in which the test does not make a decision). In this experiment $\varepsilon = 0.02$.

Throughout the experiment, we use different accuracy bounds b. Dubstep then selects transformed programs that can meet these bounds by successively running the SPR test for each bound. Table 3 presents the best transformations for each choice of the bound b. Column 1 (Accuracy Bound) presents the accuracy bound for which we run the test. Column 2 (Transformation) presents the transformed program with the highest performance, subject to the accuracy bound. Column 3 (Trials) presents the number of trials that the statistical test performed to provide the accuracy guarantee. This number of trial is an average over ten runs of the statistical accuracy test.

For bounds 0.01 and 0.05, the fastest r-reliable program is **LockForces**. For bounds between 0.1 and 0.25, the fastest r-reliable program is **All Three**. We now interpret the results of the statistical test. For instance, for row 2, the test states that, the transformed program **LockForces** executed on the representative input produces a result such that the difference d from the original result is smaller than the bound $b = 0.05$ with probability at least $r = 0.9$. The test also guarantees that this previous statement is correct with probability equal to confidence $c = 0.9$.

We repeated this experiment ten times. For all ten runs the statistical test produced the same answer for the best transformed program, but the number of trials that the test performs depends on the observations made during the test. The minimum number of trials that the test needs to perform

is 100. More trials indicate that some program executions produced a result whose difference d is greater than b. The average number of trials for the transformed programs that the test accepted was between 100 and 161 (the maximum number of trials for a single test was 279). To reject other transformations that do not satisfy accuracy bound the accuracy test needed between 30 and 70 trials.

4. Accuracy Analysis

In this section, we describe statistical accuracy analyses designed to help Dubstep characterize the effects that the synchronization relaxation transformations have on program's accuracy. In particular, these analyses attempt to answer the following question: how often does the execution of a non-deterministic program produce small deviations from the original result?

Program Model. Let there exist an application $\mathbb{A}$, an input $\mathbb{I}$, and an environment $\mathbb{H}$ (e.g., hardware description, operating system version – including information about its scheduler – and the compiler version – including optimization parameters – used to create application's executable). An *execution context* is a triple $\mathbb{C} = (\mathbb{A}, \mathbb{I}, \mathbb{H})$. *A sample* from the execution context $\mathbb{C}$ is a single execution of the application $\mathbb{A}$ on the input $\mathbb{I}$ in the environment $\mathbb{H}$. Each sample is randomized (due to non-deterministic program transformations).

Our analysis treats each sample from the context $\mathbb{C}$ as one of two events: the execution either produced a result whose accuracy loss d (which is calculated using the accuracy metric) is either smaller or equal to the accuracy bound b, or it is greater than b. Therefore, if we take several samples, we can represent the i-th sample (that has the accuracy loss d_i) as a Bernoulli random variable:

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

For each of the n samples (executions) from $\mathbb{C}$, we can calculate the accuracy loss d_i ($i \in \{1, \ldots, n\}$) and determine which event ($d_i \leq b$ or $d_i > b$) occurred. We call such (known) outcomes *observations* and denote them with lowercase letters $x_1, \ldots, x_n$.

Reliability. The *reliability* of an execution context $\mathbb{C}$ is the probability

$$p = \Pr[X = 1] \ .$$

In other words, the reliability is a probability that an execution of the application $\mathbb{A}$ on the input $\mathbb{I}$ in the environment $\mathbb{E}$ produces accuracy loss d smaller than the acceptable loss bound b.

The true value of the reliability p is typically unknown. A direct modeling of p is a hard task — it depends on the complex interactions between the elements of the environment (including hardware, operating system, and the compiler) and the input of the program.

Instead, we use a notion of an *acceptance reliability*, r, which represents a lower bound on the true reliability p. The acceptance reliability is provided by a user. We consider an execution context to be *acceptable* if and only if its reliability p is greater than r. Then the *acceptability test* is a procedure that determines the acceptability of an execution context.

We will review two acceptability tests. Each test returns TRUE if it determines that p is greater than r (in this case Dubstep accepts a candidate transformed program), or otherwise return FALSE (in this case Dubstep rejects a candidate transformed program). Both of these tests take observations $x_1, x_2, \ldots$ on which they base their decisions. The decisions of the tests are not sound, but a user can set the probability with which these decisions are correct. We have previously used these tests in the context of evaluating automatic approximate program parallelization [13, 14].

4.1 Hoeffding's Inequality Test

Let an estimated reliability $\hat{p}$ be an average of n observations $x_1, \ldots, x_n$ of the random variable X:

$$\hat{p} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Note that merely knowing $\hat{p}$ is not enough to reason about the acceptability of the context $\mathbb{C}$. We cannot directly compare $\hat{p}$ with the acceptance reliability r, as we do not know the relation between p and $\hat{p}$. For a *sound* decision of acceptability we would need to establish that $\hat{p} \leq p$. However, to make such a decision an additional information about parameter p is necessary. For an *probabilistically accurate* decision of acceptability, it is enough to establish that the undesired case $\hat{p} > p$ occurs only with a small probability.

Hoeffding's inequality [8] provides a bound on the probability that the estimate $\hat{p}$ is significantly different from p:

$$\Pr[|\hat{p} - p| > \varepsilon] \leq 2\exp\left(-2\varepsilon^2 n\right)$$

The inequality includes a user-defined tolerance ε that gauges the precision of the resulting confidence bound estimate. Also, note that the number of samples n affects the probability of a large distance between $\hat{p}$ and p – intuitively, the more samples the test takes, the closer the estimate will be to the true value p. The value $\delta = 2\exp\left(-2\varepsilon^2 n\right)$ is the failure rate — it is a probability that the test returns an incorrect result.

Application to Acceptability Testing: The result of Hoeffding's inequality gives us a desired connection between p, $\hat{p}$, and r. It states that $|\hat{p} - p| \leq \varepsilon$ with probability at least $1 - \delta$. Then, one can show that $\hat{p} \geq r + \varepsilon$ implies $p \geq r$ (with probability at least $1 - \delta$). A similar derivation follows in the case when $p < r$.

The test based on Hoeffding's inequality makes these decisions after computing $\hat{p}$:

- If $\hat{p} \geq r + \varepsilon$, then $p \geq r$ with probability at least $1 - \delta$. As a result, the test returns TRUE and Dubstep accepts the candidate transformed program.
- If $\hat{p} < r - \varepsilon$ than $p < r$ with probability at least $1 - \delta$. As a result, the test returns FALSE and Dubstep rejects the candidate transformed program.
- If $r - \varepsilon \leq \hat{p} \leq r + \varepsilon$, then the test returns UNKNOWN. Dubstep then acts conservatively and rejects the candidate transformed program. As an alternative, the test can be repeated with smaller ε such that the new test execution returns either TRUE or FALSE.

The confidence that the test makes a correct decision is $c = 1 - \delta$.

4.2 Sequential Probability Ratio Test

An alternative technique is to find an interval $[p', 1]$ that almost surely contains the probability p. In this case, a lower bound p' is a conservative approximation of p – if p' is greater than the acceptance reliability r, then p is also greater than r.

The SPR test [29] is a sequential hypothesis testing framework. This test performs multiple iterations to determine which of the two hypotheses, H_0 or H_1, to accept. At each iteration i the test collects a single observation x_i and evaluates which of the two hypotheses is more likely based on x_i and the previous observations $x_1, \ldots, x_{i-1}$. The test continues taking observations until its confidence becomes high enough to accept one of the two hypotheses.

The test takes as input two hypotheses of the following form:

$$H_0 : \ p \leq p'$$
$$H_1 : \ p \geq p' + \varepsilon$$

This test cannot make a sharp decision between the hypotheses involving only p and p' and therefore must use a tolerance ε, which determines a narrow interval $(p', p' + \varepsilon)$ in which the test does not make a decision. If p is outside of this interval, the test determines with high probability the correct hypothesis.

It is possible that the SPR test makes an incorrect decision, but a user can control the rate of incorrect decisions. A user of the test can set *a false positive rate* P_F (the probability that the test incorrectly accepts a bad parallelization), and *a false negative rate* $1 - P_G$ (the probability that the test incorrectly rejects a good parallelization).

Application to Acceptability Testing: The value p' must be specified before the test begins. Since a candidate transformed program should be rejected if $p < r$, and accepted otherwise, then $p' = r$. Therefore, the SPR test accepts the hypothesis H_0 (and thus rejects the candidate transformed program) if the reliability p is less than r. Conversely, it accepts the hypothesis H_1 (and accepts the candidate transformed program) if p is greater than $r + \varepsilon$.

The SPR test makes the following decisions:

- H_0 **accepted:** The decision that $p < r$ is correct with probability at least P_G. As a result, the test returns FALSE and Dubstep rejects the candidate transformed program.
- H_1 **accepted:** The decision $p > r + \varepsilon$ is correct with probability at least $1 - P_F$. As a result, the test returns TRUE and Dubstep accepts the candidate transformed program.
- **Test timeout:** The test has not made a decision after taking a maximum number of observations. As a result, Dubstep stops the test and rejects the candidate transformed program.

When $P_G = 1 - P_F$, then the confidence that the test makes a correct decision is $c = P_G$.

Comparison of the Acceptability Tests. Both tests can decide whether to accept or reject a candidate parallelization after taking enough samples. The difference is that the SPR test is an on-line test (it uses observations as the test executes to update the number of samples it needs to take), while the Hoeffding's inequality test is an off-line test (it determines the number of samples to take before it starts executing).

Most of the time the SPR test is the preferred choice because of the number of observations it needs to take. Since Hoeffding's inequality test is off-line, it must provision for the worst-case scenario, when the actual probability $p \approx r + \varepsilon$. Because of this inflexibility, this test will typically require significantly more observations than the SPR test even if its confidence p is much greater (or smaller) than this borderline case. However, the execution time of the Hoeffding's test is more predictable. In addition, it can be used to determine reliability in cases when the finite number of samples have been previously collected.

5. Related Work

Accuracy/Performance Tradeoffs. Researchers have studied various program transformations [1, 2, 6, 9, 14, 17, 19, 20, 24, 25, 30] and approximate data structures [10, 21, 23, 28] that give a promise of a faster program execution or better energy utilization at the expense of some accuracy of the result that these computations produce. One of the proposed techniques uses synchronization-free approximate data structures with data races [21]. Researchers have also proposed probabilistic techniques for reasoning about accuracy and performance effects of some of these transformations [6, 16, 30].

Quickstep. We have previously presented Quickstep [13–15], a parallelizing compiler that uses the find-transform-navigate approach to produce parallel programs with potential data races. Quickstep takes as input a sequential program, representative inputs, acceptable accuracy loss bound, and a set of *parallelization introduction* transformations and *accuracy introduction* transformations. It then explores the

tradeoff space induced by these transformations to find a (potentially non-deterministic) parallel program that executes in a maximum amount of time, subject to an accuracy bound.

Dubstep operates as a "reverse Quickstep" – it takes as input an already parallelized (deterministic) program and applies *synchronization relaxation* transformations that remove locks or terminate late tasks to improve the performance of computations, subject to accuracy bound. Dubstep, like Quickstep, uses transformed program executions and statistical tests to assess the reliability of non-deterministic parallel programs and to guide the search of the parallel programs.

Parallelization with Data Races. Meng et al. [11, 12] and Renganarayana et al. [18] propose developer-guided techniques to produce parallel programs with data races that slightly affect the accuracy of the results. These techniques rely on the developer to manually find profitable computations, parallelize programs, and provide computations that compute the quality of intermediate results (i.e., the outputs of each transformed subcomputation).

Dubstep, in contrast, automates a part of the process of finding parallel programs with relaxed synchronization. A developer provides only the representative inputs and the quality measure of the result of the whole program. Dubstep explores the tradeoff space that the synchronization relaxation transformations induce, uses statistical tests to assess the reliability of its observations, and provides the results of the exploration to the developer.

Integrity of Computations. As a related problem, researchers have also explored techniques that identify and enforce separation between regions of a program that identify or enforce separation between critical regions of a program (that must always produce correct result) and approximate regions (that may produce results with varying level of accuracy) [4, 5, 23]. With the focus on non-critical program regions, probabilistic or statistical accuracy analyses (such as those used by Dubstep) can be used to reason about the effects of transformations on the program's accuracy.

Profile Driven Parallelization Compilers. Profile-driven parallelization approaches run the program on representative inputs and use memory profiling (in some cases supported by additional static analysis) to suggest potential parallelizations that do not cause data races in the optimized program [7, 22, 26]. These approaches result in semantically equivalent optimized programs that produce the same result as the original programs. Alter [27] provides a deterministic execution model that supports parallelizations that read stale data or otherwise violate the data dependences of the sequential program. Alter is designed to provide deterministic execution and freedom from data races.

In contrast, Dubstep explores a larger optimization space that includes non-deterministic execution and data races. The transformations that Dubstep applies purposefully introduce non-determinism in transformed programs to improve performance at a cost of accuracy of the result that the transformed programs produce. Supporting a larger optimization space comes at the cost of requiring multiple executions of the same input to determine if a candidate parallelization respects the accuracy bounds with acceptable probability.

6. Conclusion

Writing parallel programs with a hard notion of correctness constraints the development of such programs to conservative practices that favor excessive use of synchronization primitives. Our results show that Dubstep's approach, which revolves around the *find-transform-navigate* approach, helps developers navigate a richer trade-off space to obtain more efficient acceptably accurate parallel programs.

Acknowledgements

This research was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0905244, CCF-1036241, and IIS-0835652), DARPA (Grants FA8650-11-C-7192 and FA8750-12-2-0110), and the United States Department of Energy (Grant DE-SC0005288).

References

[1] J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. PLDI, 2009.

[2] W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. PLDI, 2010.

[3] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of High Performance Computing Applications*, 3(3):5–40, 1989.

[4] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.

[5] M. Carbin and M. Rinard. Automatically Identifying Critical Input Regions and Code in Applications. ISSTA, 2010.

[6] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. FSE, 2011.

[7] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. PLDI, 2007.

[8] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301), 1963.

[9] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. ASPLOS, 2011.

[10] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. PODC, 2011.

[11] J. Meng, S. Chakradhar, and A. Raghunathan. Best-Effort Parallel Execution Framework for Recognition and Mining Applications. IPDPS, 2009.

[12] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna. Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution. IPDPS, 2010.

[13] S. Misailovic, D. Kim, and M. Rinard. Automatic parallelization with statistical accuracy bounds. Technical Report MIT-CSAIL-TR-2010-007, MIT, 2010.

[14] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, 2010.

[15] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing, Special Issue on Probabilistic Embedded Computing (to appear)*, 2013.

[16] S. Misailovic, D. Roy, and M. Rinard. Probabilistically Accurate Program Transformations. SAS, 2011.

[17] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. ICSE, 2010.

[18] L. Renganarayana, V. Srinivasan, R. Nair, D. Prener, and C. Blundell. Relaxing synchronization for performance and insight. Technical Report RC25256, IBM, 2011.

[19] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. ICS, 2006.

[20] M. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. OOPSLA, 2007.

[21] M. Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm. Technical Report MIT-CSAIL-TR-2012-005, MIT, 2012.

[22] S. Rul, H. Vandierendonck, and K. De Bosschere. A dynamic analysis tool for finding coarse-grain parallelism. HiPEAC Industrial Workshop, 2008.

[23] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. PLDI, 2011.

[24] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. FSE '11.

[25] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. SenSys, 2007.

[26] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. PLDI, 2009.

[27] A. Udupa, K. Rajan, and W. Thies. Alter: Leveraging breakable dependences for parallelization. PLDI, 2011.

[28] D. Ungar, D. Kimelman, and S. Adams. Inconsistency robustness for scalability in interactive concurrent-update in-memory MOLAP cubes. Technical report, IBM TJ Watson, 2011.

[29] A. Wald. *Sequential analysis*. John Wiley and Sons, 1947.

[30] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. POPL, 2012.