Accuracy Bugs: A New Class of Concurrency Bugs to Exploit Algorithmic Noise Tolerance

ISMAIL AKTURK, University of Minnesota, Twin Cities
RIAD AKRAM, MOHAMMAD MAJHARUL ISLAM, and ABDULLAH MUZAHID, University of Texas at San Antonio
ULYA R. KARPUZCU, University of Minnesota, Twin Cities

Parallel programming introduces notoriously difficult bugs, usually referred to as concurrency bugs. This article investigates the potential for deviating from the conventional wisdom of writing concurrency bug–free, parallel programs. It explores the benefit of accepting buggy but approximately correct parallel programs by leveraging the inherent tolerance of emerging parallel applications to inaccuracy in computations. Under algorithmic noise tolerance, a new class of concurrency bugs, accuracy bugs, degrade the accuracy of computation (often at acceptable levels) rather than causing catastrophic termination. This study demonstrates how embracing accuracy bugs affects the application output quality and performance and analyzes the impact on execution semantics.

CCS Concepts: • **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Error handling and recovery**; • **Computing methodologies** → *Shared memory algorithms;*

Additional Key Words and Phrases: Concurrency bugs, algorithmic noise tolerance

ACM Reference Format:
Ismail Akturk, Riad Akram, Mohammad Majharul Islam, Abdullah Muzahid, and Ulya R. Karpuzcu. 2016. Accuracy bugs: A new class of concurrency bugs to exploit algorithmic noise tolerance. ACM Trans. Archit. Code Optim. 13, 4, Article 48 (December 2016), 24 pages.
DOI: http://dx.doi.org/10.1145/3017991

1. INTRODUCTION

Historically, programmers have been trained to write programs correctly. Therefore, they obsess to eliminate as many software bugs as possible. A recent study [Britton et al. 2013] has reported that programmers spend 50% of their effort in finding and fixing bugs. This costs hundreds of billions of dollars a year throughout the world. By introducing notorious bugs such as data races, atomicity violations, ordering violations, deadlocks, or sequential consistency violations, parallel programming makes the already challenging task of writing correct programs even more daunting. These bugs are usually referred to as concurrency bugs.

There has been significant research to detect [Savage et al. 1997], avoid [Lucia and Ceze 2009], or expose [Park et al. 2009] concurrency bugs. Despite all of these efforts

This work is a new article, not an extention of a conference paper. The work was supported by the NSF under grant XPS:CCA:1438286 and CCF:SHF 1319983.
Authors' addresses: I. Akturk and U. R. Karpuzcu, 200 Union Street SE, 4-174 Keller Hall, Department of Electrical and Computer Engineering, University of Minnesota, Twin Cities, Minneapolis, MN 55455; emails: {aktur002, ukarpuzc}@umn.edu; R. Akram, M. M. Islam, and A. Muzahid, NPB 3.108, Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249, USA; emails: {riad.akram, mohammadmajharul.islam, abdullah.muzahid}@utsa.edu.

DOI: http://dx.doi.org/10.1145/3017991

to trade system efficiency for correctness, programmers continue to struggle with concurrency bugs. Accordingly, this article investigates the idea of deviating from the conventional wisdom of writing concurrency bug–free, hence correct, parallel programs. It explores the possibility of accepting buggy but approximately correct parallel programs by leveraging the inherent tolerance of emerging parallel applications to inaccuracy in computations. We introduce the concept of a new class of concurrency bugs: accuracy bugs. These are the concurrency bugs that do not lead to program failures but manifest themselves as inaccuracy in outputs. Our goal is to understand how embracing accuracy bugs can potentially improve the overall system efficiency by compromising accuracy, but not correctness.

Emerging recognition, mining, and synthesis (RMS) applications [Chen et al. 2008] exhibit an inherent ability to tolerate inaccuracy in computations [Rinard 2012, 2013]. This is because, RMS applications process noisy and highly redundant data, they are based on probabilistic, often iterative algorithms, and they generate a range of valid outputs as opposed to single golden output. These applications' output is much less sensitive to faults in data-centric program phases, as opposed to control [Li and Yeung 2007]. *Accuracy bugs comprise the subset of concurrency bugs that mainly affect the dataflow*. Therefore, embracing accuracy bugs is more likely to hurt accuracy than correctness of computation. By focusing on accuracy and integrity rather than correctness, not only can programmers be relieved from the burden of finding and fixing all concurrency bugs but also the overall system efficiency (i.e., performance, complexity, and energy efficiency can improve).

To understand the impact of embracing accuracy bugs, we carefully analyze many buggy execution scenarios. To identify accuracy bugs, and to mimic buggy execution semantics, we inject different concurrency bugs into correct programs by relaxing synchronization points. A cautious reader might wonder why we do not use existing bugs from bug databases. We do not rely on those bugs, because the bugs confirmed by the developers are clear-cut harmful or critical ones (they cause crash, deadlock, etc.) and hence cannot be considered as accuracy bugs, which is the focus of our study. The bugs that we consider as accuracy ones are not reported in bug databases and therefore are not available publicly.

Concurrency bugs may prevent program termination. Even if the program terminates successfully, the application output may be invalid (i.e., incorrect). Accuracy bugs, on the other hand, can only lead to valid (i.e., correct) application output, by definition. Accuracy bugs can degrade the accuracy of application outputs at varying levels, and the resulting output quality degradation may or may not be acceptable. Based on acceptability, we can distinguish between two classes of accuracy bugs: acceptable accuracy bugs and unacceptable accuracy bugs. *Acceptability* tightly depends on the level of accuracy bug–induced degradation in the output quality. At the same time, acceptability is a strong function of the context in which an application is used. The very same level of output degradation due to accuracy bugs, for the very same application, may be acceptable in one context and totally unacceptable in another. As our study is context oblivious, we will report ranges for expected quality degradation under different (accuracy) buggy execution scenarios.

On the other hand, accuracy bugs may also affect convergence criteria. A buggy execution may feature an increased number of iterations until convergence and thereby degrade performance, particularly for a critical class of RMS applications based on iterative refinement [Recht et al. 2011].

To understand accuracy bugs, we seek to find answers to the following questions:

—What types of concurrency bugs occur when different synchronization operations are relaxed?

—How many of these bugs are accuracy ones and why?
—What is the impact of accuracy bugs on different variables?
—How sensitive is the output quality on these variables?
—What is the overall quality degradation under different accuracy bugs?
—How does the execution time change under quality degradation? What contributes to that change?
—How can we exploit our findings to improve efficiency?

This article is the first (to the best of our knowledge) comprehensive study to provide answers to these questions. We examine all applications from PARSEC [Bienia et al. 2008] and one application from the SPLASH2 [Woo et al. 1995] benchmark suites. For each application, we conduct extensive experiments to find out answers to these questions. We experiment on both simulated and real machines. Our experiments reveal many interesting findings, such as the following: (i) 84/134 synchronizations affect the dataflow, and their relaxation introduces less than 10% inaccuracy (75/84); (ii) 404/533 injected bugs and affected variables are confined to dataflow; and (iii) applications' performance gain due to embracing accuracy bugs is not significant at a lower thread count, but the potential gain increases at a higher thread count (as we sweep the thread count from 2 to 64). Finally, we point out implications of these findings in future research for efficient system design and algorithm/software development.

We believe that the applications we examine represent many important classes of RMS applications; however, we do not intend to draw any general conclusion for all RMS or all parallel applications. In particular, it should be noted that all characteristics and findings are associated with the examined applications (and others like them). Moreover, the applications might have some latent bugs of which we are not aware. In that case, comparison with unmodified applications to quantify accuracy and correctness may lead to misleading results. Therefore, the findings of this work should be considered with the specific set of applications and the evaluation methodology (Section 3) in mind.

2. MOTIVATION

An important emerging class of parallel applications, RMS [Chen et al. 2008] exhibits, by construction, an inherent ability to tolerate faults due to massive yet noisy and redundant input data, the reliance on iterative and often probabilistic algorithms, and the existence of a range of valid outputs. Therefore, the application output is much less sensitive to faults in data-centric program phases, as opposed to control [Li and Yeung 2007]. Provided that accuracy bugs only affect the dataflow, embracing this class of concurrency bugs is more likely to hurt accuracy than correctness of computation. In other words, we can embrace concurrency bugs only if (i) they are accuracy bugs (i.e., confined to data-centric program phases and hence manifest as a degradation in accuracy) and (ii) the bug-induced degradation in accuracy remains within acceptable boundaries. Accordingly, we need to explore to what extent we can meet these two conditions.

Impact on execution semantics. Correct parallel execution demands careful orchestration of how parallel tasks access shared data. This entails preventing both (i) simultaneous accesses to shared data, particularly if at least one of the accesses is to modify the data, and (ii) any execution order that breaks producer-consumer dependencies among parallel tasks. Any violation of parallel access semantics can give rise to concurrency bugs.

Concurrency bugs may result in deadlocks. Since deadlocks prevent program termination (i.e., affect the control flow), RMS applications cannot mask them. Prevalent classes of concurrency bugs such as ordering violations, atomicity violations, and data

races, however, often affect the dataflow and hence can be classified as accuracy bugs most of the time. A recent study reports that 97% of nondeadlock concurrency bugs stem from ordering or atomicity violations and data races [Lu et al. 2008]. If multiple parallel tasks access shared data, where at least one of the accesses is a write, the program output may change with order of accesses, due to data races. Ordering violations manifest if parallel tasks execute out of the required order to guarantee correctness. Atomicity violations, on the other hand, emerge if the execution of a (supposedly) atomic code segment is interleaved with accesses from another concurrent task.

The common pathology of improper synchronization between parallel tasks of execution can easily lead to concurrency bugs—ordering or atomicity violations and data races. Injecting concurrency bugs by relaxing synchronization can change not only the interleaving but also the concurrency of threads, which in turn affect the data values protected by synchronization primitives. Depending on how such changes in data values propagate to the program outputs, we can define two distinct, broad classes for bugs. In the first class, *critical*, the bugs prevent proper termination of the program due to hangs, deadlocks, segmentation faults, and so forth. Such bugs affect the control flow of the program. The bugs of the second class, *accuracy*, on the other hand, result in output quality degradation but at varying quantities. Sometimes these bugs may render an unacceptably high degradation in application output quality. We will refer to this subclass as *unacceptable accuracy* bugs to differentiate them from *acceptable accuracy* bugs. At the same time, nondeterminism in the execution may cause the same concurrency bug to appear as a different class. To be able to exploit algorithmic fault tolerance, we need to carefully assess the conditions leading to such divergent behavior.

Implications on system design. We can analyze the impact of embracing accuracy bugs from two perspectives: implications on performance and complexity. Can accuracy bugs, if embraced, lead to any performance benefit? If so, what is the extent of such benefit? If we can harvest the performance benefit, energy efficiency and application scalability benefits are likely to follow. As for complexity, can accuracy bugs, when embraced, lead to simplicity in system design? In principle, one would expect most of the hardware overhead incurred by concurrency bug detection, avoidance, or recovery to be minimized, if not eliminated. Cache coherence and memory consistency protocols can be made flexible by relaxing constraints for accuracy oblivious data as well. However, the need for safety nets persists to sustain accuracy (at a desired level) and to guarantee program integrity. For example, checkpoint recovery support may be provided, likely of much less complexity, since embracing accuracy bugs would likely demand less frequent checkpointing of (possibly less) architectural state. At the same time, we should not overlook new sources of complexity introduced by embracing accuracy bugs, mainly due to the addition of one more degree of freedom. A system capable of embracing accuracy bugs should be at least equipped with some control logic to decide and orchestrate which concurrency bugs to embrace, at which point of execution. In this article, we would like to shed some light on these questions.

3. EVALUATION SETUP

Simulation infrastructure. We use the cycle-accurate microarchitectural simulator Sniper-6.0 [Carlson et al. 2011]—along with a real system that consists of four sockets with eight-core Intel Xeon E5-4620 v2 processors—for benchmark profiling in terms of application output quality, execution semantics, and performance. The configurations of the simulated processor and the real machine are given in Table I.

Benchmarks. We examined all benchmarks from the PARSEC-3.0 suite [Bienia 2011] and one benchmark from the SPLASH2 suite [Woo et al. 1995]. The PARSEC-3.0 suite

Table I. Technology and Architecture Parameters

Simulated	Measured (E5-4620 v2)
System & Technology Parameters	
Tech. node: 22nm	Tech. node: 22nm
Frequency = 2.60GHz (64 cores)	Frequency = 2.60GHz (8 cores/socket with 4 sockets)
Architectural Parameters	
L1-I Cache: 32KB L1-D Cache: 32KB	L1-I Cache: 32KB L1-D Cache: 32KB
L2 Cache (private): 256KB	L2 Cache (private): 256KB
L3 Cache (shared): 20MB	L3 Cache (shared): 20MB

Table II. Concurrency Bug Injection Techniques

Injection Method	Description
Lock elimination (LE)	Removal of a lock and unlock operation
Barrier elimination (BE)	Removal of a barrier
Condition elimination (CE)	Removal of a conditional wait and surrounding lock-unlock
Lock splitting (LS)	Splitting of the critical section of a lock
Atomicity elimination (AE)	Replacing an atomic operation with a conventional one

covers a representative collection of RMS applications by construction [Chen et al. 2008]. Among the PARSEC benchmarks, Blackscholes and Swaptions do not use any classic synchronization primitive (which complicates concurrency bug injection according to Table II), and Freqmine does not employ pthreads for parallelization. Accordingly, we exclude these benchmarks. The rest of the PARSEC benchmarks, Canneal, Dedup, Ferret, Streamcluster, Fluidanimate, Bodytrack, Vips, Raytrace, and X264, along with Barnes from SPLASH2, still cover a representative sample of RMS applications. For performance analysis, we profile the (prespecified) region of interest (ROI) for PARSEC, and the parallel section for SPLASH2, where the actual computation takes place.

Concurrency bug injection. Ideally, we would like to start by exploring various concurrency bugs that either arose during development time or are reported by users. Unfortunately, bugs of the first category are not available to anyone other than the developers. Bugs of the second category, as mentioned in Section 1, are either clear-cut harmful ones (because they cause crash, hang, deadlock, etc.) or not classified as bugs at all (as decided by developers). Hence, we resort to injecting different bugs artificially for this study. Table II lists the techniques that we applied for concurrency bug injection.[1] Table III provides the points of injection (i.e., synchronization points subject to the methods in Table II) for each application. The second column specifies the range (which we use as labels to identify the synchronization points throughout evaluation) of the synchronization points within the source file given in the third column. Synchronization points are numbered according to the order of appearance in the source file. The last column tabulates the respective line numbers in the source files. An interested reader can examine the source files to determine the type of each synchronization point.

We use Thread Sanitizer [Serebryany and Iskhodzhanov 2009] and Helgrind [Nethercote 2004] to detect data races. Since there is no publicly available tool to detect atomicity and ordering violations, we manually inspect the reported data races and bug injection sites to determine these violations. If a bug can be categorized as

[1]In principle, these injection methods may lead to bugs other than concurrency bugs, but we did not observe such cases in our evaluation.

Table III. Points of Injection (i.e., Synchronization Points Subject to the Techniques from Table II)

Benchmark	Synchronization Points	Source File	Line Number
barnes	sync 1-7	code.c	285, 410, 414, 658, 721, 735, 807
	sync 8-12	load.c	45, 51, 53, 217, 233
canneal	sync 1-3	annealer_thread.cpp	90, 120, 121
	sync 4	main.cpp	119
	sync 5-6	netlist.cpp	58, 84
	sync 7-10	netlist_elem.cpp	57, 63, 81, 90
	sync 11	rng.h	46
dedup	sync 1-3	encoder.c	267, 317, 486
	sync 4-6	mbuffer.c	197, 230, 280
	sync 7-9	queue.c	39, 53, 87
fluidanimate	sync 1-16	pthreads.cpp	603, 732, 741, 834, 843, 1126, 1129, 1131, 1133, 1135, 1137, 1139, 1141, 1143, 1149, 1262
streamcluster	sync 1-27	streamcluster.cpp	706, 738, 744, 759, 770, 785, 789, 794, 803, 812, 825, 956, 991, 1008, 1021, 1036, 1073, 1102, 1115, 1150, 1212, 1231, 1239, 1503, 1514, 1569, 1608
ferret	sync 1-3	queue.c	27, 34, 56
bodytrack	sync 1-4	AsyncIO.cpp	66, 73, 88, 93
	sync 5-12	ParticleFilterPthread.h	110, 112, 114, 120, 125, 127, 129, 138
	sync 13-24	TrackingModelPthread.cpp	142, 144, 149, 154, 167, 169, 174, 180, 188, 190, 195, 200
	sync 25-31	WorkerGroup.cpp	71, 78, 99, 112, 114, 118, 129
vips	sync 1-2	window.c	139, 355
	sync 3	im_close.c	311
	sync 4-5	semaphore.c	87, 124
	sync 6	init.c	188
	sync 7	debug.c	477
	sync 8	threadpool.c	499
	sync 9-14	region.c	203, 230, 253, 287, 335, 387
	sync 15	im_XYZ2Lab.c	80
raytrace	sync 1-8	RTThread.cxx	24, 186, 218, 239, 263, 287, 342, 361
x264	sync 1-2	frame.c	882, 890

both a data race and an atomicity (ordering) violation, we categorize it as an atomicity (ordering) violation.

Metrics to capture application output quality. Canneal implements a simulated annealing (SA)-based optimization algorithm and generates a numeric output corresponding to the minimum (routing cost). To quantify how the outcome of buggy and bug-free runs differ, we use the buggy routing cost normalized to the bug-free one as a relative quality metric. For Dedup, a compression algorithm, we deploy the relative compression rate when compared to the bug-free execution as the quality metric, provided that the output generated by the buggy execution can be decompressed.[2] For Ferret, a content-based similarity search algorithm, our quality metric is based on the

[2]For Dedup, we explicitly check whether the output can be decompressed, as the output cannot be valid if decompression is not possible.

Table IV. Benchmarks Deployed

Benchmark	Domain	Quality Metric
Barnes (ba)	N-body simulation	Difference in body positions
Canneal (ca)	Optimization	(Relative) Routing cost
Dedup (de)	Compression	(Relative) Compression rate
Fluidanimate (fa)	N-body simulation	Difference in particle positions
Streamcluster (sc)	Clustering	Number of common clusters
Ferret (fe)	Similarity search	Number of common images
Bodytrack (bt)	Computer vision	SSD of output vectors
Vips (vp)	Image processing	Peak-signal-to-noise ratio (PSNR)
Raytrace (rt)	Real-time animation	PSNR
X264 (x2)	Video encoding	Structural similarity (SSIM)

number of common images that buggy and bug-free versions of the code produce for any given query image. A similar metric is used for Streamcluster, a clustering application, to capture the intersection of buggy and bug-free outputs. For Vips and Raytrace, an image processing and a real-time animation benchmark, respectively, we use peak signal-to-noise ratio (PSNR) to determine the output quality. Specifically, we normalize the PSNR of the output image of the buggy version to the PSNR of the bug-free version. For X264, a video encoding benchmark, we use structural similarity (SSIM) [Wang et al. 2004] as the quality metric. Specifically, we normalize the SSIM of the encoded video under buggy execution to the SSIM of the bug-free version. The rest of the benchmarks output vectors, so we rely on the average relative error per vector element as the quality metric.[3] Vector elements correspond to three-dimensional (3D) particle positions for Fluidanimate and 3D body positions for Barnes. For Bodytrack, the output is a vector of tracked configurations, and we rely on the sum of squared distance (SSD) as the quality metric. Table IV summarizes the quality metrics. To calculate the quality degradation under buggy execution, we repeat each experiment 100 times and report the mean along with statistical significance.

4. EVALUATION RESULTS

To characterize how embracing accuracy bugs can enhance system efficiency, we inject concurrency bugs following Table II and provide an in-depth analysis of application output quality, execution semantics, and performance.

4.1. Impact on Application Output Quality

To demonstrate the impact of concurrency bugs on application output quality, we profile representative RMS applications from the PARSEC suite [Bienia et al. 2008] with the input dataset simsmall[4] and thread count (16) fixed, along with Barnes from the SPLASH2 suite [Woo et al. 1995]. We run the experiments 100 times on the real system from Table I and report the statistical significance. For each application, we use a numeric metric to quantify the relative quality degradation with respect to the bug-free (i.e., fully synchronized) execution, as shown in Table IV. To inject concurrency bugs, we deploy the techniques from Table II. We inject one concurrency bug at a time.

Classification. Some synchronization points, if relaxed according to Table II to inject a concurrency bug, prevent proper program termination. We call them *critical*

[3]We also provide maximum relative error; both average and maximum relative error are within reasonable range of each other.

[4]Except for X264, where we use simlarge, as this benchmark requires a larger problem size for runs involving more than eight threads.

Table V. Impact of Injected Concurrency Bugs on Application Output

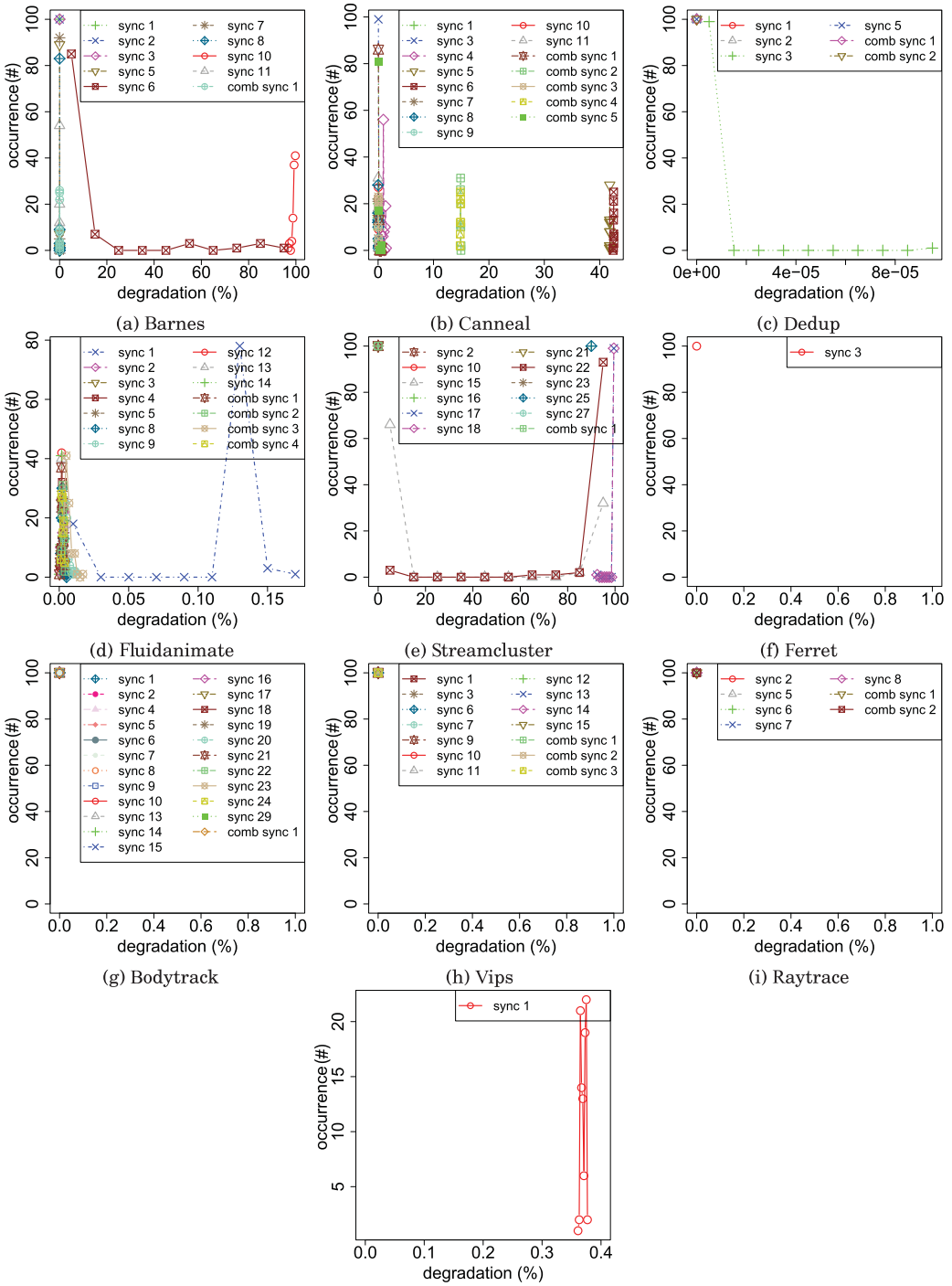
Benchmarks	Quality Degradation Bin					
	= 0%	< 1%	< 50%	< 100%	No Termination	Invalid
	Share of Synchronization Points in Quality Degradation Bin (%)					
ba	25	33.3	8.3	8.3	25	0
ca	0	63.6	27.3	0	0	9.1
de	33.3	11.1	0	0	11.1	44.4
fa	56.25	6.25	0	0	25	12.5
sc	18.5	0	3.7	18.5	59.3	0
fe	33.3	0	0	0	66.6	0
bt	71	0	0	0	29	0
vp	73.3	0	0	0	26.7	0
rt	62.5	0	0	0	37.5	0
x2	0	50	0	0	50	0

synchronizations. No output can be generated due to hangs or segmentation faults. Accordingly, concurrency bugs induced by the relaxation of critical synchronization points cannot be masked by algorithmic fault tolerance. Critical synchronization points are essential for the proper functioning of the applications. The rest of the synchronization points, which are referred to as *noncritical*, do not prevent proper program termination if relaxed according to Table II to inject a concurrency bug. Concurrency bugs due to the relaxation of the noncritical synchronization points (i.e., accuracy bugs) render varying degrees of degradation in the application output quality and hence can potentially be masked by algorithmic fault tolerance.

Table V characterizes concurrency bug–induced degradation in application output quality. Each column corresponds to a specific bin for the percentage of quality degradation. Two separate columns capture the bins for *invalid* outputs and *no termination*. For each benchmark, we report the percentage share of synchronization points falling in a particular bin (as captured by the columns). We observe that out of 134 synchronization points, 43 lead to nonterminating execution and hence are critical. Similarly, 7 synchronization points lead to termination; however, invalid outputs are generated, so these are also considered critical. The rest (i.e., 84 synchronization points) are noncritical. We also observe that (across all benchmarks) out of 84 noncritical synchronization points, only 9, when relaxed, cause more than 10% degradation in output quality. For the majority of the noncritical synchronizations (i.e., 71), the maximum output quality degradation remains below 0.1%.

We relaxed one synchronization point at a time and repeated each relaxation experiment 100 times to estimate the quality bin for each synchronization point independently. The presence of a synchronization point p in a quality bin (i.e., column) q in Table V indicates that the relaxation of p caused a degradation corresponding to q in more than 90 of the 100 experiments. Using normal approximation for binomial confidence intervals [Leon-Garcia 1994], this translates into less than 5.88% binning error with 95% confidence on a per synchronization point basis.

Noncritical synchronization points. We next analyze how the relaxation of each noncritical synchronization point changes the application output quality. Figure 1 provides histograms for the percentage of quality degradation. The x-axis captures the percentage of degradation in output quality, and the y-axis shows how many times we observed the corresponding output quality degradation (out of 100 trials) when synchronizations are relaxed either individually or in combination with other (noncritical) synchronizations. For example, *combined sync 1* in Figure 1(a) characterizes the combined relaxation of synchronization points 3, 5, 7, and 11 for Barnes. For Canneal,



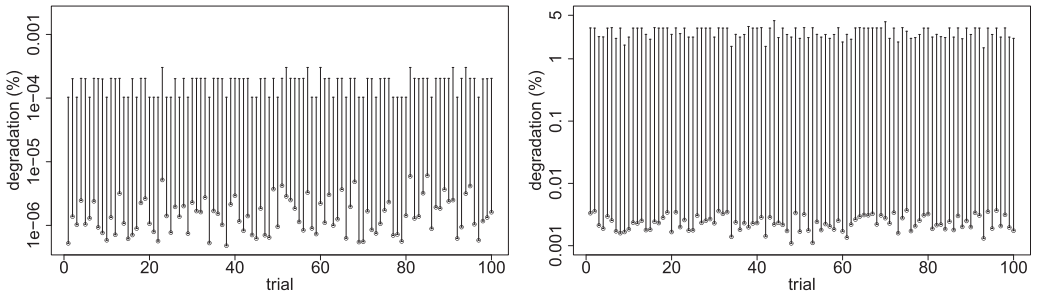
(a) Barnes (b) Canneal (c) Dedup

(d) Fluidanimate (e) Streamcluster (f) Ferret

(g) Bodytrack (h) Vips (i) Raytrace

(j) x264

Fig. 1. Percentage of quality degradation due to relaxed synchronization.



(a) Barnes (b) Fluidanimate

Fig. 2. Maximum relative error observed per vector element for vector-based outputs due to relaxed synchronization.

combined sync 1, 2, 3, 4, and 5 from Figure 1(b) characterize the combined relaxation of synchronization points (1, 3), (1, 5, 6), (1, 7, 10), (1, 5, 6, 9), and (1, 3, 7, 9), respectively. For Dedup, *combined sync 1 and 2* in Figure 1(c) characterizes the combined relaxation of synchronization points (1, 5) and (2, 5), respectively. For Fluidanimate, *combined sync 1, 2, 3, and 4* in Figure 1(d) characterizes the combined relaxation of synchronization points (2, 13), (3, 5, 14), (3, 4, 9, 14), and (2, 3, 13, 14), respectively. For Streamcluster, *combined sync 1* in Figure 1(e) characterizes the combined relaxation of synchronization points (21, 27). For Bodytrack, *combined sync 1* in Figure 1(g) characterizes the combined relaxation of synchronization points 7, 8, and 10. For Vips, *combined sync 1, 2, and 3* in Figure 1(h) characterize the combined relaxation of synchronization points (11, 13, 14), (9, 10, 11) and (7, 11, 13), respectively. For Raytrace, *combined sync 1 and 2* in Figure 1(i) characterize the combined relaxation of synchronization points (2, 6) and (6, 8), respectively.

No combined relaxation applies for Ferret and X264, as these benchmarks have only one noncritical synchronization point (see Figures 1(f) and (j)).

For combined cases, we only consider noncritical synchronization points. We could not experiment with all possible combinations of these due to the excessively large number of possibilities. Instead, in Figure 1, we report how the application output quality changes as we combine several noncritical synchronization points that caused relatively low quality degradation in isolation as a best-case study variant. Overall, we observe that combining such noncritical synchronization points is unlikely to result in excessive quality degradation.

For vector-based outputs, the average of (relative) degradation across all vector elements can hide notable deviations and hence be misleading. Figure 2 shows the maximum relative degradation observed per vector element, along with the average, across all experiments (we do not show the minimum, as it is always zero). Figure 2(a) characterizes the combined relaxation of synchronization points (3, 5, 7, and 11) for Barnes, and Figure 2(b) characterizes the combined relaxation of synchronization points (2, 3, 13, and 14) for Fluidanimate. The x-axis shows the experiment number (recall that we run each experiment 100 times for statistical significance), whereas the y-axis (in log scale) captures the percentage of output quality degradation. The error bars show the maximum relative degradation observed per vector element, and the data points show the average quality degradation of all vector elements, on a per-experiment basis. We observe that the maximum relative degradation per vector element remains less than 4% for Fluidanimate and less than 1% for Barnes. Bodytrack is the only remaining benchmark with vector-based output, where the experiments did not result in any sizable difference between the maximum relative degradation and the average relative

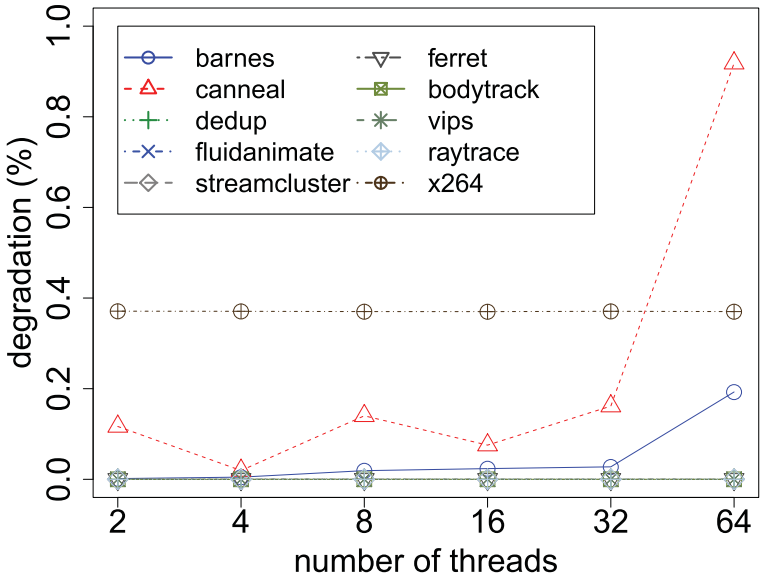


Fig. 3. Percentage of quality degradation versus thread count.

degradation (recall that we found the average relative degradation for Bodytrack to be practically negligible).

Injecting concurrency bugs by relaxing synchronization can change not only the interleaving but also the concurrency of threads, which in turn affects the data values protected by the synchronizations and hence the application output quality. At the same time, both the interleaving and the concurrency strongly depend on the number of parallel threads of execution. Figure 3 shows how the output quality degrades as a function of the number of threads. For this analysis, as representative combinations, we relax synchronization points 3, 5, 7, and 11 for Barnes; 1, 3, 7, and 9 for Canneal; 2 and 5 for Dedup; 2, 3, 13, and 14 for Fluidanimate; 21 and 27 for Streamcluster; 7, 8, and 10 for Bodytrack; 7, 11, and 13 for Vips; and 2 and 6 for Raytrace. We relax synchronization point 3 for Ferret and synchronization point 1 for X264. These combinations resulted in the lowest-quality degradation for our 16-threaded default runs. We observe that Barnes and Canneal are the most sensitive to the changes in thread count, featuring a slight increase in percentage of degradation in output quality as the thread count increases. However, the degradation across all benchmarks still remains less than 1%. The output quality of the remaining benchmarks (Dedup, Fluidanimate, Streamcluster, Ferret, Bodytrack, Vips, Raytrace, and X264) is practically unaffected. For the benchmark applications considered, we observe that the execution outcome is practically insensitive to the number of parallel threads of execution. This finding appears to be counterintuitive, as a higher thread count usually implies higher contention critical sections, which increases the likelihood of simultaneous write accesses to shared data under relaxation (hence of more corruption in the value of respective shared data). Even under this scenario, however, the quality degradation in the application output may remain negligible, depending on how sensitive the application output is to the changes in the values of such corrupted shared variables. On the other hand, as we will discuss in the following case study, it may also be the case that injected concurrency bugs are never activated (i.e., the injected execution follows the very same trajectory as the bug-free execution) depending on thread interleavings, even at a relatively high thread count.

The values of data protected by synchronizations and their impact on output quality also depend on the input of the application. To quantify the impact of input size on output quality under buggy executions, we repeated the analysis for simmedium and simlarge inputs of PARSEC (and equivalents for SPLASH2). To be consistent with previous evaluations, we relax synchronization points 3, 5, 7, and 11 for Barnes; 1, 3, 7, and 9 for Canneal; 2 and 5 for Dedup; 2, 3, 13, and 14 for Fluidanimate; 21 and 27 for Streamcluster; 3 for Ferret; 7, 8, and 10 for Bodytrack; 7, 11, and 13 for Vips; and 2 and 6 for Raytrace. We fixed the thread count to 16. We observe that the output quality degradation is not sensitive to the input size for Fluidanimate, Streamcluster, Bodytrack, Vips, and Raytrace. For Ferret, output quality degradation increases by 1.2% for simlarge over simsmall. The output quality remains below 1% for both Barnes and Canneal across all input sizes—simlarge leads to even smaller quality degradation when compared to simmedium and simsmall. For X264, we relax synchronization point 1 and fix the thread count to eight for simsmall and simmedium. X264 does not generate any output when more than eight threads are used for simsmall and simmedium. The quality degradation under simsmall remains around 0.14%; under simmedium and simlarge, it remains around 0.37%. This implies that our findings are highly oblivious to the input size for the benchmark applications considered.

Observation: On average, 62% of all synchronizations (84/134) are found to be noncritical. When noncritical synchronization points are relaxed, output quality degradation remains less than 10% for the majority of cases. In addition, the degradation is found to be highly insensitive to different thread counts and inputs for the benchmark applications deployed.

The reported numbers strongly depend on the classification of synchronization points, which incurs less than 5.88% error with 95% confidence.

Case study. As we relax synchronization points to inject concurrency bugs, each time we execute the given portion of the code we may encounter a different interleaving of threads. Moreover, the execution outcome may change as a function of these interleavings. For example, we can have the buggy and bug-free executions resulting in the very same thread interleaving, which translates into the injected concurrency bug not being activated. If the bug is activated, depending on thread interleavings, a variety of execution outcomes are possible, ranging from nonterminating cases (e.g., due to segmentation faults) to varying degrees of output quality degradation. The thread interleavings in the presence of injected concurrency bugs may affect the performance of the application in positive or negative ways as well.

```
1  // streamcluster.cpp: barrier @ 991 in pgain()
2  ...
3  if(pid == 0){
4    work_mem = (double *) malloc (...);
5    ...
6  }
7  pthread_barrier_wait(barrier);
8  for(i = k1; i < k2 ; i++){
9    if(is_center[i]){
10     center_table[i] = count++;
11   }
12 }
13 work_mem[pid*stride] = count;
14 ...
```

Listing 1. Elimination of barrier @ 991 of Streamcluster.

Next we review a case study to demonstrate the impact of interleavings in the presence of injected concurrency bugs, specifically how interleavings can hide or expose the injected concurrency bugs, which in turn can affect the output quality and performance in different ways. In Listing 1, we show a code snippet taken from Streamcluster. We inject a concurrency bug by eliminating the barrier at line 7. Now consider an interleaving where the master thread (whose pid (i.e., thread id) is 0) is right at line 3. At the same time, a worker thread may reach line 13. Since the master thread has not allocated memory for *work_mem* yet (the allocation takes place at line 4), the worker thread will try to access an unallocated memory location at line 13. This will cause a segmentation fault. On the other hand, all other possible interleavings will lead to successful termination without any output degradation if the master thread allocates memory for *work_mem* at line 4, before any worker thread gets to line 13. This example shows how interleavings can hide or expose the injected concurrency bugs.

```
1  // streamcluster.cpp: barrier @ 1231 in pFL()
2  ...
3  while( change/cost > 1.0*e){
4    change = 0;
5    numberOfPoints = points->num;
6    if(pid == 0){
7      intshuffle(feasible, numfeasible);
8    }
9    pthread_barrier_wait(barrier);
10   for(i = 0; i < iter; i++){
11     x = i % numfeasible;
12     change += pgain(feasible[x], ...);
13   }
14   cost -= change;
15   pthread_barrier_wait(barrier);
16 }
17 return cost;
```

Listing 2. Elimination of barrier @ 1231 of Streamcluster.

Next, let us demonstrate how thread interleavings in the presence of injected concurrency bugs can change output quality as well as performance. In Listing 2, the concurrency bug is injected by eliminating the barrier at line 9. An array called *feasible* stores integer values. The integer values stored in this array are shuffled at every iteration by the master thread at line 7. This very same array is accessed by the rest of the threads, and an element from this array is passed as a value to the *pgain()* function at line 12. The value passed to *pgain()* is used as an index (to select a point in calculating the cost). Now consider the case where a worker thread passes a value from the *feasible* array to the *pgain()* function at line 12, before the *feasible* array is shuffled by the master thread at line 7. The *pgain()* function receives a value from the unshuffled array, so it may generate a wrong return value to be assigned to the *change* variable at line 12, which in turn determines *cost*. Eventually, the value of the *cost* variable can be different than anticipated as it is updated at line 14.

Notice that both *cost* and *change* variables are used in evaluating the condition of the while loop at line 3. Depending on the corruption in these variables, the number of iterations executed by the while loop may be less or more than anticipated. This can change the performance of the application. We observed that under certain thread interleavings, the execution time increases considerably. Similarly, for the interleavings where threads pass values from the unshuffled *feasible* array to the *pgain()* function, we observed degradation on the output quality at varying degrees.

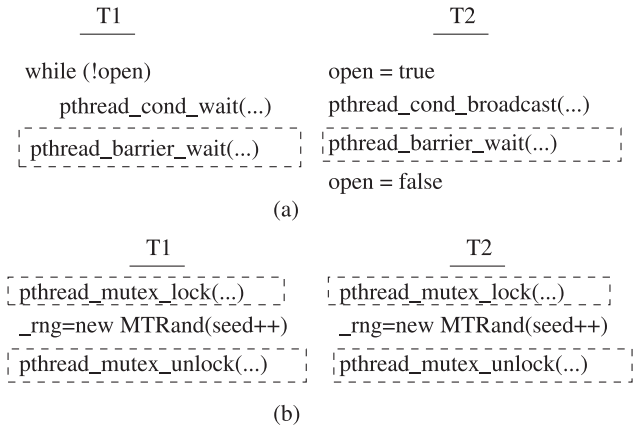


Fig. 4. A critical(a) and an accuracy bug (b). Dotted boxes represent relaxed synchronization operations.

Similar to Streamcluster, many RMS applications rely on iterative refinement. As the case study reveals, and as observed in previous studies [Recht et al. 2011; Dean et al. 2012], controlling the number of iterations until convergence may be a more effective knob for these applications in trading performance for accuracy than embracing accuracy bugs by relaxing synchronization. This is mainly because accuracy bugs may render both premature or late convergence (i.e., the number of iterations until convergence (hence, execution time) may decrease or increase), which challenges deterministic control in the performance-accuracy trade-off space.

4.2. Impact on Execution Semantics

In this section, we inspect how the concurrency bugs induced by the injection techniques from Table II change the execution semantics. In Section 4.2.1, we start with the categorization of different types of injected bugs. We then provide, in Sections 4.2.2 and 4.2.4, an in-depth analysis of the characteristics of various affected variables and their impact on the accuracy of end results, along with a function-level analysis in Section 4.2.3.

4.2.1. Bug Categorization. We apply our injection techniques to one synchronization point at a time. We experiment with simsmall input (except X264, where we use simlarge as in Section 4.1) and 16 threads on the real machine (Table I). We run each experiment 100 times. We use program termination status, thread sanitizer report, and manual code inspection to analyze the bugs. More specifically, we determine whether a particular bug can cause deadlock, hang, crash, memory leak, inaccurate computation, and so forth. We categorize the bugs that lead to accuracy loss in program end results as *accuracy* bugs, as opposed to *critical* bugs, which result in no termination or improper termination. Notice that accuracy bugs that do not alter the final results at all are referred to as *benign* bugs by prior literature [Narayanasamy et al. 2007]. Thus, benign bugs represent a subset of accuracy bugs. Both critical and accuracy bugs may correspond to data races, atomicity violations, ordering violations, and others.

Figure 4(a) shows an example of a critical bug in Streamcluster. Here, the worker thread T1 is waiting on a condition variable after it finds *open* to be false. The master thread T2 sets *open* to be true and broadcasts a signal on the condition variable. As a result, T1 wakes up. However, if we relax the barrier, T2 immediately goes on and sets *open* to be false. As a result, T1 again finds *open* to be false and keeps on waiting on the condition variable. This leads to a nonterminating state. Therefore, this is a

Table VI. Categorization of Bugs

Injection	Data Race		Atomicity Violation		Ordering Violation		Others	
	Critical	Accuracy	Critical	Accuracy	Critical	Accuracy	Critical	Accuracy
LE	12	93	6	41	0	0	2	23
BE	30	52	4	19	0	0	3	15
CE	0	0	0	0	7	7	4	2
LS	0	0	60	144	0	0	0	1
AE	0	0	0	0	0	0	1	7

Table VII. Categorization of Synchronizations

Benchmark	Lock		Barrier		Condition Variable		Atomic Operation	
	Critical	Noncritical	Critical	Noncritical	Critical	Noncritical	Critical	Noncritical
ba	1	4	2	5	0	0	0	0
ca	0	1	0	2	0	0	1	7
de	3	3	0	0	2	1	0	0
fa	0	5	6	5	0	0	0	0
sc	2	0	13	11	1	0	0	0
fe	1	0	0	0	1	1	0	0
bt	4	16	3	5	2	1	0	0
vp	3	11	0	0	1	0	0	0
rt	1	4	0	0	2	1	0	0
x2	0	1	0	0	1	0	0	0
Total	15	45	24	28	10	4	1	7

critical bug. Figure 4(b) shows an accuracy bug on the variable *seed* that causes a random number generator to be initialized with a different value than usual. However, it does not affect the generator's ability to produce random numbers, and hence the final output is not affected. This is an accuracy bug taken from Canneal.

Table VI shows the bugs injected by different techniques. Lock elimination (LE) and barrier elimination (BE) inject mostly data races and atomicity violations, 80% of which are accuracy related. Condition elimination (CE) mostly introduces ordering violations. Half of them are accuracy related. Lock splitting (LS) injects atomicity violation bugs. In summary, we introduce a total of 533 bugs. Of them, a total of 404 bugs are accuracy related.

Based on these results, it might be tempting to believe that we can eliminate most of the synchronizations and still get a workable program. However, this is not true. The very same injection technique applied to the very same synchronization point may give rise to accuracy as well as critical bugs, mainly as a function of interleavings. To clarify this issue, we categorize a synchronization point as a critical one if its elimination leads to at least one critical bug. Otherwise, it is categorized as a noncritical one. Critical synchronization points cannot be eliminated without the risk of no or improper termination. On the other hand, noncritical synchronizations can be eliminated if we can accept some inaccuracy. Table VII shows the categorization for different applications. If the synchronization operation is a lock or atomic operation, it is more likely to be a noncritical one. Most of the conditional synchronization operations are found to be critical. As for barrier operations, 46% are critical and the rest are noncritical. In summary, 62% of the synchronization operations are found to be noncritical.

4.2.2. Variable Categorization. We next categorize shared variables into critical or accuracy ones. For this purpose, we consider only the variables affected by the injected bugs—that is, the shared variables protected by synchronization primitives that are

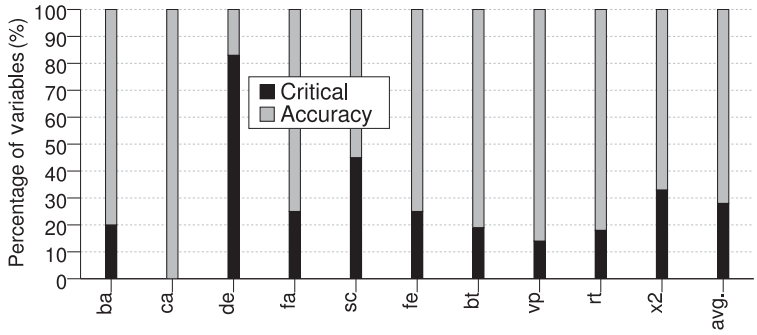


Fig. 5. Categorization of shared variables.

relaxed to inject concurrency bugs. We manually inspect the code to determine how different shared variables are used. If a variable is used during the process of computing the (intermediate or final) results, then we categorize it as an accuracy one. Other shared variables are categorized as critical. These variables are the ones usually used during pointer manipulation, data structure maintenance, array index calculation, and so forth. We cross validate our categorization by analyzing the related synchronizations. An accuracy variable might be affected by both critical and noncritical synchronizations, whereas a critical variable can be affected only by critical synchronizations. Figure 5 shows the classification of variables. On average, 72% of the bug-affected variables are accuracy related and the rest are critical for the benchmark applications considered.

4.2.3. Functional Analysis. Finally, we investigate the functions encompassing the noncritical or critical synchronization points. More specifically, we would like to know if there is any correlation between the type of synchronizations and what the function does. We are going to present three case studies: two cases for noncritical synchronizations and one case for critical synchronizations.

Fluidanimate keeps track of the current and last positions of fluid particles for the purpose of simulating an incompressible fluid. The *compute densities* phase of the application estimates the fluid density at the position of each particle by analyzing how closely other particles are packed in its neighborhood. The *ComputeDensitiesMT* function calculates the particle's and its neighboring particles' density, which are protected by two locks. If we relax both of these locks individually or jointly, we affect density calculation that can impact the final positions of the particles. *ComputeDensitiesMT* is an accuracy-related function because its density calculation affects the accuracy of the program, not the termination. Its locks are noncritical as well.

Canneal uses cache-aware SA to minimize the routing cost of a chip design. The annealing algorithm is implemented in the *Run* function. In this function, per iteration, two netlist elements are attempted to be swapped to calculate the impact on the routing cost. This loop continues until the chip design stabilizes or a preset maximum number of steps is reached. A barrier is placed at the end of the loop body for each thread to complete the calculation. If we relax the barrier, some threads may start the next iteration before the rest of the threads complete the current iteration. This might affect the dereferencing and interchanging of pointers to the netlist elements. Canneal uses atomic operations to recover from these data races, but this might still increase the routing cost. Thus, relaxing the barrier is more likely to cause accuracy loss, which makes *Run* an accuracy-related function.

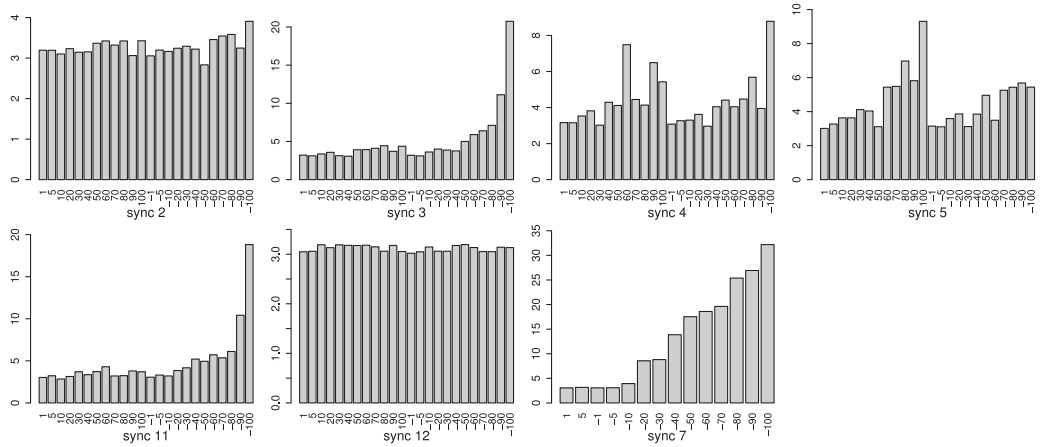


Fig. 6. Percentage of output quality degradation versus percentage of value corruption in shared variables protected by synchronization primitives (Fluidanimate).

Dedup uses a compression mechanism called *deduplication*. The *mbuffer_split* function splits a memory buffer into two buffers using metainformation, including a reference counter. This metainformation serves the orchestration of malloc or free operations. In this function, the updates to the reference counter are protected by a lock. If we relax the lock, the reference counter may not get updated correctly, which may corrupt malloc and result in a program crash. This means that the *mbuffer_split* function is critical in nature as it updates control-centric (i.e., critical) data (pointers).

4.2.4. Sensitivity Analysis. We next explore how sensitive the accuracy loss in the end result of applications is with respect to corruption in shared variables protected by synchronization primitives subject to relaxation. We analyze Fluidanimate as a case study. We corrupt the value of various representative shared variables, each protected by a synchronization primitive, one at a time, for 1% of the total execution time, by a quantity varying from −100% to +100% of the bug-free, *correct* value. We do not relax the synchronization points during this sensitivity study. Rather, the goal is to measure the sensitivity of application output to corruption of these particular variables in an attempt to better understand our findings from Sections 4.1 and 4.2. Figure 6 captures the percentage of degradation in output quality as a function of the percentage of data corruption, separately for each variable. We identify the variables by the synchronization primitives protecting them. In each graph, the x-axis captures the percentage of corruption in the corresponding variable. We analyze seven variables, protected by seven synchronization points. We observe that the variables protected by synchronizations 3, 7, and 11 have the highest impact on the output quality—in these cases, the output accuracy loss can exceed 20%, as captured by the y-axis. Synchronization 3 degrades output quality by more than 5% only if the associated variable's value is corrupted by more than −50%. For the variable protected by synchronization point 7, the execution fails if we corrupt the value by more than +5%. Synchronization 11's impact on output quality is similar to synchronization 3. These results are in line with our findings from Sections 4.1 and 4.2, which identified synchronization points 7 and 11 as critical. For synchronizations 2 and 12, output accuracy loss remains virtually the same across different values of the corruption in the associated variables. For synchronization points 4 and 5, on the other hand, fluctuations in the output accuracy loss become more visible. Synchronization points 2, 4, 5, and 12 all were deemed noncritical by our analysis from Sections 4.1 and 4.2.

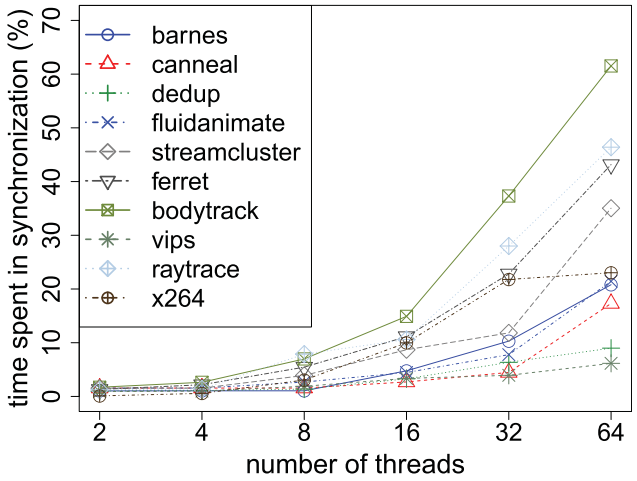


Fig. 7. Percentage of time overhead of synchronization.

Observation: On average, 76% of all injected bugs are accuracy related (404/533). CE introduces approximately the same number of accuracy and critical bugs. Other techniques tend to introduce more accuracy bugs. Locks and atomic operations tend to be noncritical, whereas conditional synchronizations tend to be critical. A barrier can be critical or noncritical with similar probabilities (i.e., 46.2% and 53.8%, respectively). On average, we observe that 72% of the bug-affected variables are accuracy related. Most accuracy bugs cause insignificant accuracy loss due to the fact that the values of the affected variables do not get corrupted significantly for the benchmark applications deployed.

4.3. Impact on Performance

In this section, we assess how embracing accuracy bugs can impact the execution time and scalability of parallel programs. As in the previous sections, we inject accuracy bugs by eliminating synchronization points 3, 5, 7 and 11 for Barnes; 1, 3, 7 and 9 for Canneal; 2 and 5 for Dedup; 2, 3, 13 and 14 for Fluidanimate; 21 and 27 for Streamcluster; 3 for Ferret; 7, 8 and 10 for Bodytrack; 7, 11, 13 for Vips; 2 and 6 for Raytrace; and 1 for X264, as representative combinations.

4.3.1. Time Overhead of Synchronization. Figure 7 shows how the time overhead of synchronization changes as a function of the number of threads, considering the original, bug-free execution. The y-axis depicts the percentage of total execution time spent in synchronization events. We observe that for all applications, the synchronization overhead increases with the number of threads. This is because (as we keep the problem size, i.e., the input dataset constant) per thread work, the overall execution time of the program reduces as the number of threads increases, whereas the number of sharers for a given chunk of data tends to grow. When we inject concurrency bugs by eliminating synchronizations, time spent in synchronization represents a loose upper bound for the potential speedup in execution. With 64 threads, the time spent on synchronizations ranges from 6.16% to 61.52% across the benchmarks.

4.3.2. Impact on Concurrency. When we inject accuracy bugs by eliminating synchronization points, synchronization-induced stalls are minimized—if not eliminated. Thus, we expect to have more threads to be active concurrently. Figure 8 shows how the

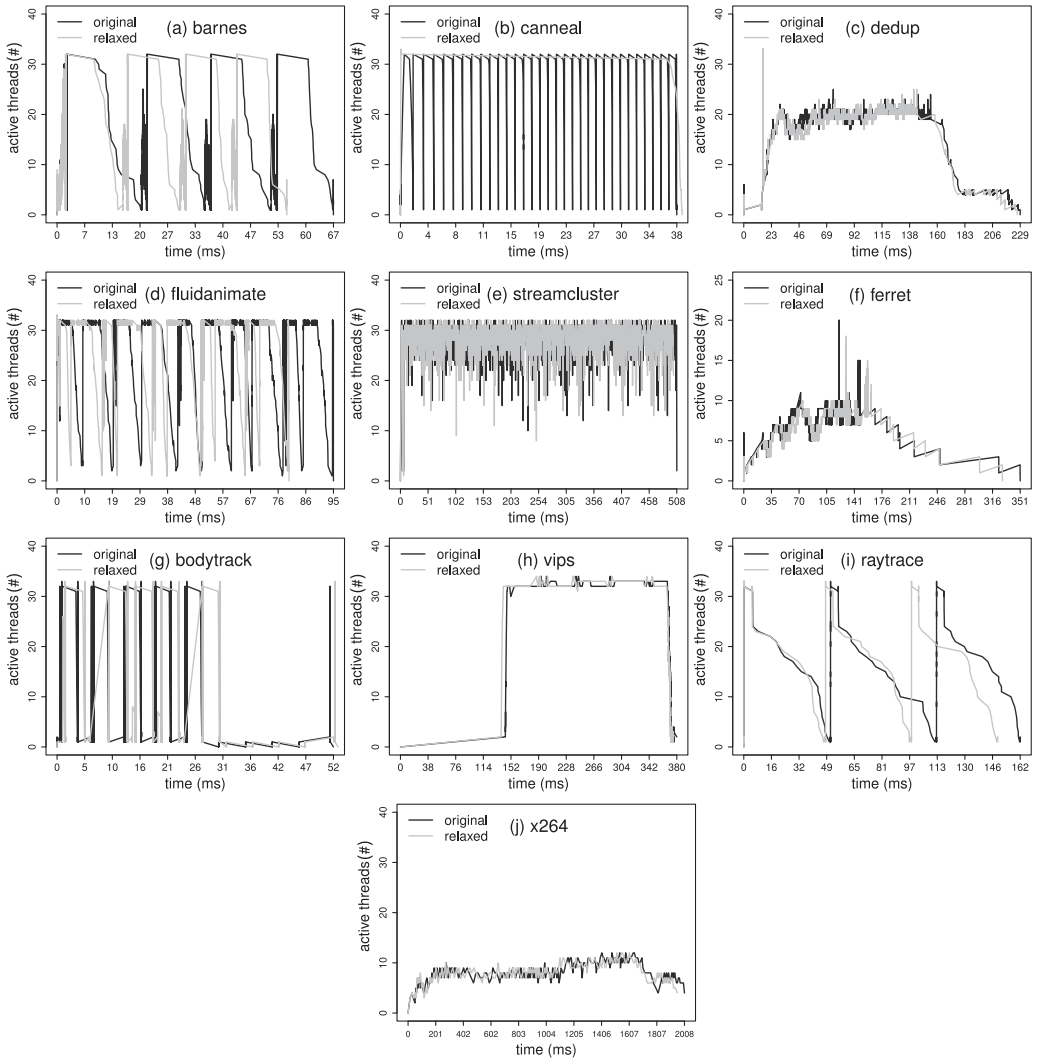


Fig. 8. Impact of accuracy bugs on concurrency.

concurrency, as measured by the number of active threads (y-axis), changes over the course of the execution (x-axis), until the application terminates, for 32-threaded runs. We obtained these data from the simulator in Table I. We monitored changes in the state of each thread over the course of execution by keeping track of sleep and wake-up times on a per-thread basis. In each graph, the profile in black corresponds to the *bug-free* (i.e., default) execution, whereas the *buggy* outcome is depicted in gray.

In Figure 8, we observe that the original and relaxed profiles mostly overlap for Dedup, Streamcluster, and Vips, which is contrary to our expectations. For these cases, we can still identify a slight increase in the level of concurrency each time a synchronization point is relaxed; however, this sporadic boost in performance does not render a notably reduced execution time for the buggy run: only 1.06%, 1.2%, and 0.19% reduction for Dedup, Streamcluster, and Vips, respectively. Yet we see 16.9%, 16.07%, 6.42%, 8.14%, and 2.55% reduction in execution time due to increased concurrency for Barnes,

Table VIII. Impact of Concurrency Bugs on Performance

Benchmark	Bug-Free Exec. Time (ms)	Buggy Exec. Time (ms)	Change (%)
barnes	66.50	55.27	16.9
cannel	38.11	38.83	–1.8
dedup	229.17	226.73	1.06
fluidanimate	95.06	79.78	16.07
streamcluster	509.97	503.84	1.2
ferret	351.35	328.79	6.42
bodytrack	52.05	52.93	–1.6
vips	379.7	378.97	0.19
raytrace	161.88	148.89	8.14
x264	2,038.86	1,986.75	2.55

Fluidanimate, Ferret, Raytrace, and X264, respectively. On the contrary, buggy runs are slowed down by 1.8% and 1.6% for Canneal and Bodytrack. This slowdown is due to deteriorated cache performance, as an increased number of active threads causes more cache contention—this is a known potential artifact of concurrency bugs [Liu et al. 2014]. Dedup, Ferret, and X264 are pipelined parallel programs. Ferret and X264 have only one synchronization point subject to relaxation, which is a lock confined in a pipeline stage (Table III). Accordingly, the slight divergence in concurrency between bug-free and buggy executions is confined to the time window in which this particular pipeline stage is executed. Despite this, we still observe 6.42% and 2.55% reduction in the overall execution time for Ferret and X264, respectively. The impact of accuracy bugs on performance is summarized in Table VIII.

In Figure 8, for Canneal, Fluidanimate, and Bodytrack, the active number of threads fluctuates in the bug-free run between 32 and 1 due to frequent synchronization. For Canneal, relaxation eliminates these fluctuations, rendering a practically constant number of active threads (i.e., 32) within the course of execution. However, the buggy run takes slightly longer due to performance-limiting artifacts of increased concurrency such as increased cache contention and off-chip bandwidth pressure. For Fluidanimate, on the other hand, the buggy run does not eliminate the fluctuations, as was the case for Canneal, but reduces the number of fluctuations, leading to a reduced execution time: the bug-free run finishes at 95.06ms and the buggy run at 79.78ms. For Bodytrack, the buggy run takes longer due to increased cache contention and off-chip bandwidth pressure, as in the case of Canneal. For Bodytrack, the bug-free run finishes at 52.05ms and the buggy run at 52.93ms.

We should note that relaxing synchronization to inject concurrency bugs can affect convergence conditions of parallel applications (e.g., Streamcluster) and can result in an even larger execution time than the bug free, although our experiments did not generate a severely slowed down buggy run when compared to the bug free. If this were the case, convergence criteria needed to be adjusted accordingly.

Observation: Although we did not observe a significant performance boost for the majority of benchmarks with which we experimented, per Amdahl's law the trend of quickly increasing time overhead of synchronization from Figure 7 suggests to expect notable performance benefits at much higher degrees of parallelism than 32 threads, even if we are not able to relax all of the synchronization points.

5. DISCUSSION

In this section, we discuss key implications of our findings.

Implication 1. As mentioned in Section 4.1, even when we eliminate synchronization points, critical concurrency bugs (i.e., nonterminating cases) occur only under few special thread interleavings. Other interleavings either do not expose the bugs or introduce accuracy bugs. If we restrict the program execution so that it avoids the critical bug-triggering interleavings, we can get away with concurrency bugs being present in the program. To this end, we need to find an efficient and lightweight way to manage interleavings. In other words, relaxing synchronizations and thereby embracing concurrency bugs become more predictable and controllable in interleaving-restrictive environments such as deterministic systems [Devietti et al. 2009; Bocchino et al. 2009]. More specifically, we can envision a deterministic system that enforces determinism only for critical synchronizations while ignoring the noncritical ones.

Implication 2. Section 4.1 shows that in an inadequately synchronized program, certain thread interleavings might cause longer execution time, whereas others accelerate execution. If we find a way to determine and enforce those interleavings (e.g., as facilitated by a deterministic system), we might embrace accuracy bugs to improve performance.

Implication 3. Section 4.2.1 analyzes critical and noncritical synchronizations as well as variables. These can be utilized to design a bug filtering and ranking tool. We can think of an automated tool that, given a buggy execution, can determine the bug-related variables using program slicing techniques [Weiser 1981]. By determining the type of variables and bugs automatically, it can produce a ranking of reported bugs so that critical bugs are placed higher in the rank than accuracy ones. The programmer can then focus her effort in fixing the critical ones and leaving behind the accuracy ones. This is likely to save programmer's debugging effort.

Implication 4. We validated that RMS programs feature a sizable fraction of accuracy variables that can tolerate inaccuracy (e.g., due to errors). We do not need to strictly protect accuracy variables, and therefore we can simplify the complexity of checkpoint and recovery schemes. Presumably, we would need to check point only critical variables. During rollback, we omit recovery of accuracy ones and leave them in an inaccurate state without significantly affecting the final result. The knowledge of accuracy variables can be utilized during software development and testing process as well. Programmers can be less rigorous in synchronizing and testing accuracy variables. This knowledge can also be utilized in other dimensions of architectural design for simplification.

6. RELATED WORK

There has been significant research over the years to detect concurrency bugs. Among the various types of concurrency bugs, data races are the ones most commonly studied [Savage et al. 1997]. Research on data races leads to commercial tools like Intel Inspector [Petersen 2011]. ToleRace [Ratanaworabhan et al. 2012] provides runtime support for detecting and masking asymmetric data races. Proposals like AVIO [Lu et al. 2006] and MUVI [Lu et al. 2007] detect atomicity violation bugs where a group of memory accesses that is supposed to be executed atomically by a single thread gets interleaved by accesses from other threads. Most of these proposals share a common shortcoming—they target only one type of bug. To remedy this, researchers have proposed schemes like PSet [Yu and Narayanasamy 2009] and Bugaboo [Lucia and Ceze 2009] that do not rely on the symptoms of any particular bug. Instead, these proposals focus on identifying correct data communications among threads and provide a general solution to handle any type of concurrency bug. In addition to these, there are proposals to expose and fix concurrency bugs. CHESS [Musuvathi et al. 2008],

CTrigger [Park et al. 2009], and RaceFuzzer [Sen 2008] expose concurrency bugs mainly by changing thread interleavings. Some of them are based on model checking, whereas others use a delay mechanism. Liu et al. [2014] propose to fix concurrency bugs by using locks. Grail uses a context-aware analysis to find a minimal set of locks that do not introduce any deadlock. There is not much prior work that investigates the effect of concurrency bugs on output accuracy. The closest one is from Renganarayana et al. [2012], who perform a study on the effect of synchronization relaxation in the context of approximate computing. This study shows performance improvement of the programs without providing any detailed characterization. Narayanasamy et al. [2007] propose a way to filter out benign data races from the harmful ones. They define a data race to be "benign" if it does not change the end result at all. Our goal is to study not only benign data races but also any other concurrency bugs that affect end results within (or beyond) an acceptable limit (i.e., the accuracy bugs).

There has been a growing interest in the field of approximate computing. Early work focuses on trading off accuracy for performance [Lin et al. 1987]. Proposals to exploit algorithmic fault tolerance for energy efficiency follow [Hegde and Shanbhag 1999; Shanbhag 2002]. The impact of soft errors on the output quality is analyzed by fault injection in the context of multimedia [Li and Yeung 2007], artificial intelligence [Li and Yeung 2007], and probabilistic inference algorithms [Wong and Horowitz 2006]. For emerging RMS applications, the output quality is shown to be insensitive to errors in the dataflow as opposed to control [Wong and Horowitz 2006; Li and Yeung 2007; Cho et al. 2012]. This characteristic is leveraged by dividing multicore compute power into two groups: unreliable cores in charge of data and reliable cores in charge of control [Cho et al. 2012]. In de Kruijf et al. [2010], a software recovery approach is explored based on the observation that emerging applications can tolerate even discarding of computations. Recently, approximate data structures [Rinard 2013], along with the required hardware support [Esmaeilzadeh et al. 2012], have been explored. Efforts have been made to design language constructs and verify acceptability properties of approximate programs [Carbin et al. 2012]. To the best of our knowledge, this and a similar body of work on approximate computing mainly exploit algorithmic fault tolerance to mask lower level faults, mainly stemming from hardware, as opposed to concurrency bugs.

7. CONCLUSION

This work investigated the potential for deviating from the conventional wisdom of writing concurrency bug–free, and hence correct, parallel programs. It explored the benefits and pitfalls of accepting buggy but approximately correct parallel programs by leveraging the inherent tolerance of 10 representative parallel applications to inaccuracy in computations. This work is the first (to the best of our knowledge) to do a comprehensive study on this issue. For each application, we conducted extensive experiments on both simulated and real machines. Our experiments revealed many interesting findings, some of which are the following: (i) most of the synchronizations are noncritical, and their relaxation introduces less than 10% inaccuracy; (ii) most of the injected bugs and affected variables are accuracy related; and (iii) the applications' performance gain due to embracing accuracy bugs is not significant at a lower thread count, but the potential gain increases significantly at a higher thread count. The article also provided possible implications of such findings, namely how to use a deterministic environment as an enabler to embrace accuracy bugs, how to reduce debugging effort, and how to take advantage of accuracy variables.

REFERENCES

Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University, Princeton, NJ.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*.

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*.

Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible Debugging Software. Available at https://www.jbs.cam.ac.uk/media/2013/.

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. 2012. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Proceedings of the 33rd SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*.

Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*.

Yen-Kuang Chen, Jatin Chhugani, Pradeep Dubey, Christopher J. Hughes, Daehyun Kim, Sanjeev Kumar, Victor W. Lee, Anthony D. Nguyen, and Mikhail Smelyanskiy. 2008. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE* 96, 5, 790–807.

Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. 2012. ERSA: Error resilient system architecture for probabilistic applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 4, 546–558.

Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA'10)*.

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Proceedings of the 26th Annual Conference on Neural Information Processing Systems (NIPS'12)*.

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*.

Rajamohana Hegde and Naresh R. Shanbhag. 1999. Energy-efficient signal processing via algorithmic noise-tolerance. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'99)*.

Alberto Leon-Garcia. 1994. *Probability and Random Processes for Electrical Engineering* (2nd ed.). Addison Wesley.

Xuanhua Li and Donald Yeung. 2007. Application-level correctness and its impact on fault tolerance. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA'07)*.

Kwei-Jaiy Lin, Swaminathan Natarajan, and Jane W. S. Liu. 1987. Imprecise results: Utilizing partial computations in real-time systems. In *Proceedings of the IEEE 8th Real-Time Systems Symposium (RTSS'87)*.

Peng Liu, Omer Tripp, and Charles Zhang. 2014. Grail: Context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*.

Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*.

Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.

Brandon Lucia and Luis Ceze. 2009. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*.

Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*.

Nicholas Nethercote. 2004. *Dynamic Binary Analysis and Instrumentation*. Ph.D. Dissertation. Computer Laboratory, University of Cambridge, UK.

Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*.

Paul Petersen. 2011. Intel parallel inspector. In *Encyclopedia of Parallel Computing*, D. Padua (Ed.). Springer.

Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. 2012. Efficient runtime detection and toleration of asymmetric races. *IEEE Transactions on Computers* 61, 4, 548–562.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*. Curran Associates.

Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with relaxed synchronization. In *Proceedings of the ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES'12)*.

Martin Rinard. 2012. Unsynchronized techniques for approximate parallel computing. In *Proceedings of the ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES'12)*.

Martin Rinard. 2013. Parallel synchronization-free approximate data structure construction. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism*.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* 15, 4, 391–411.

Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*.

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA'09)*.

Naresh Shanbhag. 2002. Reliable and energy-efficient digital signal processing. In *Proceedings of the 39th Design Automation Conference (DAC'02)*.

Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. 2004. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4, 600–612.

Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*.

Vicky Wong and Mark Horowitz. 2006. Soft error resilience of probabilistic inference applications. In *Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE'06)*.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'05)*.

Jie Yu and Satish Narayanasamy. 2009. A case for an interleaving constrained shared-memory multiprocessor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*.

Received April 2016; revised October 2016; accepted November 2016