Appears in the Proceedings of the
33rd International Conference on Software Engineering (ICSE), May 2011.

LIME: A Framework for Debugging Load Imbalance in Multi-threaded Execution

Jungju Oh
Georgia Institute of Technology

Christopher J. Hughes
Intel Corporation

Guru Venkataramani
George Washington University

Milos Prvulovic
Georgia Institute of Technology

ABSTRACT

With the ubiquity of multi-core processors, software must make effective use of multiple cores to obtain good performance on modern hardware. One of the biggest roadblocks to this is *load imbalance*, or the uneven distribution of work across cores. We propose LIME, a framework for analyzing parallel programs and reporting the cause of load imbalance in application source code. This framework uses statistical techniques to pinpoint load imbalance problems stemming from both control flow issues (e.g., unequal iteration counts) and interactions between the application and hardware (e.g., unequal cache miss counts). We evaluate LIME on applications from widely used parallel benchmark suites, and show that LIME accurately reports the causes of load imbalance, their nature and origin in the code, and their relative importance.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Performance, Measurement

Keywords

Load imbalance, Performance debugging, Parallel section

1. INTRODUCTION

In recent years, the number of cores available on a processor has increased rapidly, while the performance of an individual core has increased much more slowly. As a result, achieving a large performance improvement for applications now requires programmers to leverage the increased core count. This is often a very challenging problem, and many parallel applications end up suffering from *performance bugs* caused by *scalability limiters*. These prevent performance from improving as much as it should with more cores. Since we expect core counts to continue increasing for the foreseeable future, addressing scalability limiters is important for developing software that will obtain better performance on future hardware.

Load imbalance is one of the key scalability limiters in parallel applications. Ideally, a parallel application assigns an equal amount of work to all cores, keeping all of them busy for the entire application. Load imbalance occurs when some cores run out of work and must wait for the remaining cores to finish their work. Load imbalance is relatively easy to detect—we can watch for threads waiting at the end of a parallel section (i.e., at a barrier) or at a thread-join point. However, it is much more difficult to diagnose the cause of load imbalance with sufficient precision to help programmers decide what changes to make to the application to reduce the imbalance.

The cause of load imbalance can be hard to diagnose because there are a variety of candidates. For instance, load imbalance can be caused by assigning an unfair proportion of tasks to a thread, or by assigning too many long tasks to the same thread—both of these manifest as control flow differences between threads. Diagnosing causes for imbalance becomes even harder when it occurs due to interactions between the application and the underlying hardware (e.g., threads having different numbers of cache misses). Such causes cannot be easily detected through code inspection or static analysis.

Reducing load imbalance has long been an active research topic. The most common approach to reducing load imbalance is to use dynamic task scheduling, such as that provided with OpenMP [18] and TBB [8]. Dynamic task scheduling involves partitioning the parallel work into many more tasks than threads, and using a run-time system to assign tasks to threads on-demand. Dynamic scheduling significantly reduces load imbalance, but introduces significant runtime overheads, both from executing scheduling code and from the loss of cache locality among tasks. These overheads increase with the number of cores/threads, and can dominate performance [12].

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

Because load imbalance is a major issue in performance debugging of parallel programs, and because existing solutions do not satisfactorily address this problem, there is a need for tools to help programmers efficiently find and eliminate causes of load imbalance in their code.

This paper presents LIME, a framework that uses profiling, statistical analysis, and control flow graph analysis to automatically determine the nature of load imbalance problems and pinpoint the code where the problems are introduced. Unlike prior work that addresses load imbalance, LIME does not aim to automatically exploit or reduce load imbalance. Instead, it provides highly accurate information to programmers about what is causing the imbalance and where in the code it is introduced, with the goal of minimizing trial-and-error diagnosis and the programming effort needed to alleviate the problem.

We built and evaluated our LIME framework on 15 parallel sections from SPLASH-2 [24] and PARSEC [4] benchmark suites, which are commonly used to evaluate performance of multi-processor and multi-core machines. Our results show that LIME is highly accurate in pinpointing load imbalance problems caused by cache misses and control flow differences among threads. We confirmed the accuracy of the tool's cache miss results by eliminating misses it reports and confirming that doing so dramatically reduces load imbalance. We confirmed the accuracy of the tool's control flow results by verifying that, when the tool reports control flow differences as the primary cause of load imbalance, the reported line of code is the actual location where the load imbalance is introduced and leads the programmer to the code where it can be repaired.

2. OVERVIEW OF LIME

To help explain our framework and the problem it addresses, we use the code example in Figure 1. This is an actual parallel section from SPLASH-2's *radix* benchmark [24]. The SPLASH-2 benchmark suite was extensively optimized over a decade ago by experts in both parallel programming and multi-processor hardware, and has been used to evaluate parallel performance of multi-processor and multi-core machines ever since.

```
534     BARRIER(...);
535     if (MyNum != (...)) {
540       while ((offset & 0x1) != 0) {  ...  }
549       while ((offset & 0x1) != 0) {  ...  }
557       for (i = 0; i < radix; i++) {  ...  }
560     } else {
562     }
566     while ((offset & 0x1) != 0) {  offset=...  }
575     for(i = 0; i < radix; i++) {  ...  }
578     while (offset != 0) {
579       if ((offset & 0x1) != 0) {
582         for (i = 0; i < radix; i++) {  ...  }
585       }
589     }
590     for (i = 1; i < radix; i++) {  ...  }
594     if ((MyNum == 0) || (stats)) {  ...  }
598     BARRIER(...);
```

Figure 1: Code for sample parallel section from *radix*. All non-control-flow statements are removed.

This parallel section begins and ends with barriers, where each thread waits for all others to arrive before proceeding further. Any load imbalance will result in threads arriving at the end barrier (line 598) at different times, forcing early-arriving threads to wait (i.e., be idle) until the longest-running thread arrives.

Within the parallel section, each thread uses its private *MyNum* and *offset* values to decide which part of the parallel computation it should perform. Depending of these values, the threads may have different execution times, either by executing different code (due to differences in control flow) or by taking different amounts of time to execute the same code (due to differences in how the executed code interacts with the hardware). In Figure 1, there are several examples of possible control flow differences: if-then-else blocks at lines 535, 579, and 594 may cause only a subset of threads to execute the if-path (and for line 560, other threads to execute the else-path); loops at lines 540, 549, 557, 575, 578, 582, and 590 could all execute a different number of iterations for different threads. Nesting of loops and conditionals (e.g., line 582) can compound these differences. Additionally, threads may have differences in interacting with the system, such as branch predictor performance (some threads might have more predictable branch decision patterns than others) or cache performance (e.g., threads that access data already in the cache may have more cache hits). These differences are too numerous to point out even in our small example code because every branch, jump, load, store, etc. instruction in the compiled code may be, at least in theory, a potential source of these performance differences.

It may seem that a trained programmer can inspect the source code to identify potential causes of imbalance and repair them. However, this is a very labor-intensive and error-prone endeavor because of the sheer number of potential causes, and because of the complexity of understanding each cause and determining whether or not each is responsible for imbalance (and then repairing those that are).

An actual example of the threads' execution times for one dynamic instance of this parallel section is shown in Figure 2. The shaded part of each bar represents the useful execution time of each thread, normalized to the overall execution time of the parallel section. The white part of each bar represents the thread's waiting time at the end of the parallel section (line 598).

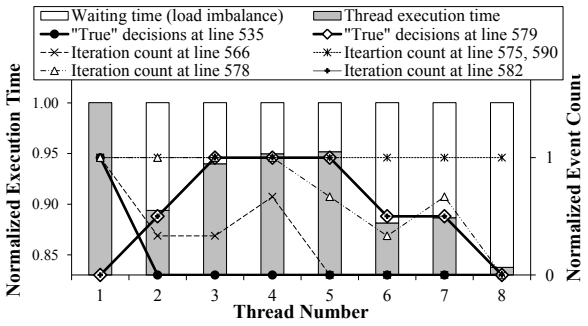

Figure 2: Execution time, imbalance, and thread behavior of key points in Figure 1. Points that cause load imbalance are shown with thicker lines.

Our discussion of possible causes of imbalance included two types of control flow causes—iteration counts of loops and decision counts of if-then-else blocks. These event counts are also shown in Figure 2, with each event's counts normalized to the maximum count for that event among all threads.

From visual inspection of the graph, the decision at line 535 appears to cause imbalance between the first thread and the others. The differences in "true" decisions at line 579 appears to account for the remaining imbalance (note how well the useful execution time tracks this factor for threads 2 through 8). The loops at lines 575 and 590 have identical iteration counts in all threads, and thus cannot be causing any imbalance—code inspection reveals the same insight, because the value of *radix* is constant and the same for all threads. The loops at lines 566 and 578 do produce different iteration counts in different threads but this does not seem to correspond to actual imbalance. Finally, loop iteration counts at lines 540, 549, and 557 (not shown) have the same relationship with the imbalance that decision count at line 535 has, and loop iteration count at line 582 has the same behavior as the decision count for line 579.

Our LIME framework performs this kind of analysis automatically and quantitatively. For each thread in each parallel section, LIME measures execution time and various *event counts*. The event counts are dynamic decision counts for all static control flow decision points (control flow events), as well as dynamic counts for each static code location that causes machine-interaction events[1] (hardware events). Using this data, LIME's analysis framework determines how much imbalance exists, which control flow decisions and machine interaction events are related to the imbalance, and assigns scores that help programmers decide which cause of the imbalance to "attack" first.

Our initial implementation of LIME includes two different profiling environments. The first implementation uses a cycle-accurate hardware simulator called SESC [20] that can be relatively easily extended to collect any desired machine-interaction event count; however, since it performs detailed simulation of a computer system, it is very slow and can only be used for parallel sections that execute quickly (e.g., with carefully designed small input sets). To overcome the speed limit, we implemented LIME with Pin [16], which is fast but can only accurately collect data for analysis of control flow causes of imbalance. The simulator-based implementation was designed to let us experiment with collecting different events, and the Pin-based one to let us test LIME on larger input sets. For practical use, a purpose-built profiler could be employed to collect control flow events and key hardware performance counters more efficiently.

The analysis part of the framework processes profiling data from either profiling environment. It first clusters together events whose counts are highly correlated to each other. The purpose of this step is to group together events that seem to be related to the same potential cause of imbalance. For example, this step puts the decision count from line 535 and the iteration counts from lines 540, 549, and 557 in the same cluster because they are linearly related to each other (they have zero counts in all threads but one, so one of these event counts is equal to a constant times the execution count of another). Similarly, the iteration count from line 582 and the number of cache misses at lines 580–585 are in the same cluster as the "true" decision count from line 579 (this would not be true if threads had differing cache miss rates for lines 580–585).

Next, LIME finds the "leader" of each cluster. The purpose of this step is to identify the event that corresponds to "introducing" a potential cause of imbalance. In our two example clusters, the "true" decision counts from lines 535 and 579 are found to be the leaders of their respective clusters.

The next step in LIME uses multiple regression to find which cluster leaders are related to the imbalance in a statistically significant way, and to find the strength of that relationship. In our example, the "true" decision counts from lines 535 and 579 are the only cluster leaders to have a statistically significant relation to the load imbalance.

Finally, LIME ranks and reports the cluster leaders that are related to the imbalance. The report includes the score, the location in the code, and the corresponding cause of imbalance. In our example, LIME reports only two causes, both with relatively high scores: (1) threads take different paths at line 535, and (2) threads have different biases ("true" vs. "false" decision) for the if-then-else at line 579.

The following section provides a detailed description of the analysis framework. Subsequent sections describe the two profiling implementations, and an experimental evaluation of LIME's accuracy with examples that provide more intuition about how LIME works and how its results can be used by the programmer to reduce load imbalance.

3. LIME ANALYSIS FRAMEWORK

LIME's analysis starts with data gathered by one of our profiling implementations (see Section 4). The profiling data consists of 1) per-thread control flow event counts for each edge in the static control flow graph and 2) per-thread hardware event counts for each static instruction that can cause such an event.

3.1 Causality Analysis for Hardware Events

Before entering the main analysis routine, LIME conducts preprocessing on collected hardware events in order to establish causal relationships between related events. For example, an L1 cache miss can occur only when a memory access instruction is executed, an L2 cache miss can occur only when an L1 cache miss happens, etc. If the dynamic count for a particular hardware event differs among threads, this hierarchy among events allows us to split the "blame" for the difference between that event itself and the events that must precede it. For instance, when threads have different numbers of L1 misses at a particular static instruction, this may due to some threads executing that instruction more times (and thus having more L1 miss opportunities) or due to how the application interacts with the L1 cache. In the preprocessing step, LIME removes from subordinate hardware events (e.g. L1 misses) the contribution from their "superior" events (e.g. instruction's execution count) using the Gram-Schmidt process [7], leaving each event count only with the event's own contribution to variations among threads. This adjusted hardware event count is used instead of the naïve one throughout the LIME analysis.

3.2 Hierarchical Clustering

The second step in the LIME analysis is to cluster related events together. There are two commonly used clustering al-

[1]Among machine-interaction events, we only experimented with cache misses because we expected them to be the only machine-interaction event that plays a significant role in creating load imbalance. As will be shown in Section 5.4, this turned out to not always be true. However, LIME's analysis treats all hardware events in the same way and we expect it to readily extend to other events (as long as they can be counted efficiently).

gorithms: hierarchical clustering and K-means clustering [9]. We use hierarchical clustering because it can automatically find an appropriate number of clusters for a given separation principle (clustering threshold) between clusters, whereas K-means yields a predetermined number of clusters. For the same reason, many projects on workload characterization [3, 11, 19] rely on hierarchical clustering to find benchmarks that have similar behavior.

Hierarchical clustering is performed in steps. Each step merges the two clusters that are "closest" according to a distance metric. Clustering ends when the distance between the two closest clusters is larger than a preset threshold.

In LIME, each event (control flow event or hardware event) is initially a cluster. We then compute a proximity matrix in which an element (i, j) represents the distance between 'cluster i' and 'cluster j'. At each step, we merge the two closest clusters and update the proximity matrix accordingly. The distance metric LIME uses is Pearson's correlation between the event's per-thread counts, because it effectively captures similarity in how the event count behaves in different threads. For linkage criteria, we used average linkage (UPGMA [17]), but other methods (single/complete-linkage) produced similar results.

LIME stops clustering when the largest correlation is <0.9. This threshold value provides the best results in most parallel sections we tested. The exceptions are *fmm* and *fluidanimate*, where we used threshold values of 0.8 and 0.6, respectively. A poorly chosen threshold affects the usefulness of the report—too-high of a threshold prevents merging of correlated event clusters, while too-low of a threshold results in clusters that contain unrelated events.

Clustering provides several benefits for further analysis:

- It results in a major reduction in the number of subjects for further analysis.
- It gathers highly co-linear events into one cluster, which improves accuracy of regression[2] in Section 3.5.
- It helps identify significant decision points in the program structure (e.g., branches where control flow differs between threads), as we explain in Section 3.4.

3.3 Classification of Clusters

Clusters are classified into two types: those that contain control flow events (control-flow clusters), and those with only hardware events (hardware event clusters).

For hardware event clusters, the absence of highly correlated control flow events indicates that different threads suffer the hardware events differently for the same code. Therefore, if any load imbalance is eventually attributed to the cluster, all the hardware events in the cluster are reported as contributing to that portion of the imbalance.

3.4 Finding Cluster Leaders

Within a control-flow cluster, control flow events are typically interdependent. To improve the usefulness of reported results, for each cluster, LIME discovers a *leader node*—a control flow instruction that steers program execution into the cluster. Intuitively, if the cluster is related to the imbalance, the leader node represents the code point where this imbalance is introduced.

[2]Statistical regression works poorly with collinear vectors.

Leader nodes are important because they are the decision points that change thread execution characteristics. They are the points in the program of most interest to the programmer: by inspecting the code that affects the leader node's decision, the programmer can typically find the high-level reason for the imbalance.

To formally define a leader node, assume a control flow graph of a program has vertices V and edges E. The leader node of a cluster C is a vertex v in the control flow graph such that all incoming edges to v *except backedges* have source vertices outside the cluster, i.e., $v \in C$ such that $\forall(s, v) \in E,\ s \notin C$.

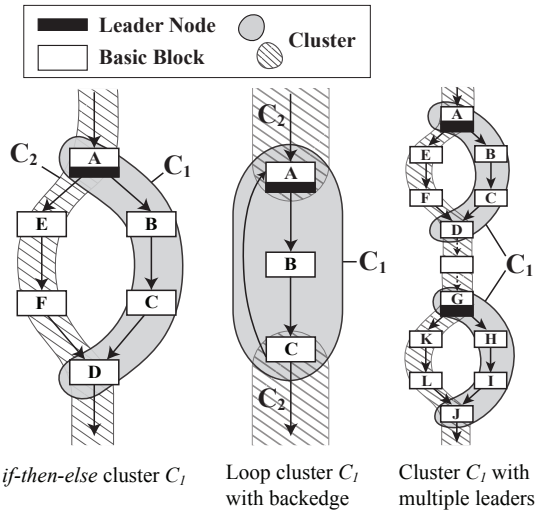

Figure 3: Example of clusters and leader nodes

Examples of typical clusters and their leader nodes are shown in Figure 3. The leftmost example shows a cluster whose leader is an *if*-statement. The middle example shows why a backedge restriction is needed in the leader definition—it allows a loop cluster to have a leader (the loop entry point). The rightmost example shows a cluster with two leaders (A and G)—this typically occurs when the same control flow decision is made in more than one code point.

After finding leader nodes for each cluster, each leader node is assigned a score according to its significance in creating load imbalance. First, LIME computes, for each edge, the Pearson's correlation coefficient between that edge and the execution time. The score of a leader node is the *difference* in this correlation between the node's incoming and outgoing edges (again, ignoring backedges). For example, if n threads have execution times $T = (t_1, t_2, \cdots, t_n)$ and a leader node v has incoming edge counts $ie_{v_1}, ie_{v_2}, \cdots, ie_{v_i}$, and outgoing edge counts $oe_{v_1}, oe_{v_2}, \cdots, oe_{v_j}$, the score s_v of the leader node v is

$$s_v = \max_x\big(corr\left(oe_{v_x}, T\right)\big) - \max_y\big(corr\left(ie_{v_y}, T\right)\big) \quad (1)$$

where $corr(a, b)$ is the Pearson's correlation coefficient between vectors a and b.

Intuitively, score s_v measures the amount of correlation the leader node incurs in regard to the overall imbalance. A high score means that the node converts events that are unrelated to load imbalance into events that are highly correlated to the imbalance.

3.5 Multiple Regression

Multiple regression analysis [6] is a statistical technique that estimates the linear relationships between a dependent variable and one or more independent variables. In LIME, the dependent variable is the vector of per-thread execution times, and the independent variables are the clusters. If a cluster appears to be a good predictor of execution time, we can infer with high confidence that the cluster is responsible for the differences in execution time between threads.

For the regression analysis, we need to combine event counts of all events in the cluster so that the cluster behaves like a single event. For this, LIME uses averaged Z-scores [13] of events in a cluster. A Z-score standardizes all events to have the same average and same variation, so that all events carry equal weight in determining the cluster's overall "event count", regardless of the actual absolute event counts for each event. For each event with a mean per-thread dynamic count μ and a standard deviation σ, for each per-thread dynamic count x, the Z-score is $z = \frac{x-\mu}{\sigma}$. After this step, regression proceeds by treating each cluster as a single "event" whose per-thread "event counts" are the cluster's per-thread Z-scores.

During regression analysis, LIME excludes clusters that are statistically redundant or insignificant in building a regression model to explain the execution time. We use the forward selection method [6] to choose which clusters to include in the model. The method selects clusters based on their unique contribution to the variance in the dependent variable (execution time). LIME iteratively adds the selected clusters to its regression model until none of the remaining clusters is statistically significant (determined by F-test [15]).

Multiple regression analysis computes a standardized coefficient, β_C, for each cluster C, which represents how sensitive execution time is to that cluster. That is, the regression computes the values of β_{C_i} to best fit $T \approx \Sigma\beta_{C_i} \cdot C_i$, where T is the vector of per-thread execution times and C_i is the vector of averaged Z-scores for cluster i across all events in the cluster. LIME uses the β_C values as a measure of a cluster's importance for load imbalance as follows.

For each control-flow leader node and hardware event, LIME computes a *final score*—an estimate of how responsible that node/event is for load imbalance. The score is based on regression results and, for control-flow clusters, on leader node importance scores. For a leader node v_i in control-flow C_j, the final score fs_{v_i} is

$$fs_{v_i} = \beta_{C_j} \cdot s_{v_i} \quad (2)$$

where s_{v_i} is the score (from Equation 1) for leader node v_i, and β_{C_j} is the standardized coefficient (from regression) for cluster C_j. For a hardware event (e.g., cache miss) h_i in a hardware-event cluster C_j, the final score fs_{h_i} is equal to β_{C_j}.

Figure 4 shows an example of computing final scores for two control-flow clusters. Cluster C_1 has one leader node A, while cluster C_2 has two leader nodes A and G. Note that node A is a leader in both clusters.

In this example, node G from cluster C_2 has the highest final score (i.e., a leader node score of 0.82 and a β_{C_2} of 0.91 combine to give a score of 0.7462). Therefore, we conclude that cluster C_2 is important in explaining the imbalance, and that node G is the prime suspect for introducing the imbalance.

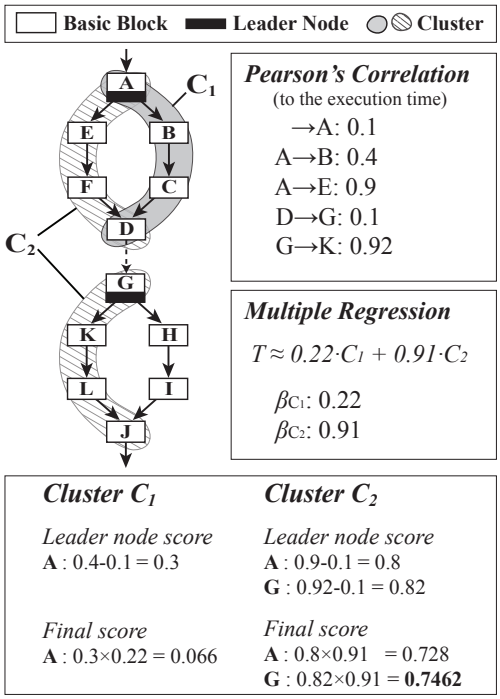

Figure 4: Example of final score computation

3.6 Reporting to the Programmer

In general, a program runs a static parallel section multiple times during its dynamic execution; LIME analysis operates on each dynamic instance independently because imbalance and other characteristics may vary across instances. Since programmers prefer feedback on static code, LIME then combines results from all dynamic instances of each parallel section using a weighted average, with load imbalance of a dynamic instance serving as its weight. For each leader node in a control-flow cluster, and for each hardware event (e.g., cache misses at static code location Y) in a hardware-only cluster, LIME presents to the programmer its static code location, type of event, and weighted score.

4. PROFILING IMPLEMENTATION

The LIME framework consists of two parts: (1) collection of profiling data, and (2) analysis of the data. Our contribution is primarily in the LIME analyzer, and our goal is to demonstrate the utility of our novel analysis techniques. We built the analyzer in C++ using the armadillo linear algebra library [21].

There is a large body of prior work on profilers. Our framework can make use of any data profiler that can collect the required control flow and hardware event counts. For our prototype tool, we implemented two different versions of the profiler.

Our first profiler is built on Pin [16], a binary instrumentation tool. With this profiler, we can collect control flow events, but no hardware events. We instrument synchronization points such as barriers to identify parallel sections in our program. We also instrument all branch instructions, and gather edge (control flow event) counts using a hashmap structure. When branch instruction b is executed and the previous branch was p, we increment the entry for edge (p, b).

Benchmark	Suite	Parallel Section	Type	Input Size	Imbalance			
					8 Core	16 Core	32 Core	64 Core
LU	SPLASH-2	lu.c:604	Barrier	1024×1024 matrix, 64×64 block	16.1%	35.5%	92.3%	92.3%
volrend	SPLASH-2	main.C:267	Barrier	head	15.3%	25.8%	38.8%	46.9%
fmm	SPLASH-2	fmm.C:283	Barrier	16,384 particles	10.6%	13.7%	22.9%	26.2%
barnes	SPLASH-2	code.C:715	Barrier	16,384 particles	10.2%	11.6%	13.5%	16.7%
canneal	PARSEC	annealer_thread.cpp:88	Barrier	10000 swaps/step, 32 steps	3.8%	6.9%	12.7%	15.2%
fluidanimate	PARSEC	pthreads.cpp:793	Barrier	35K particles	8.8%	15.3%	19.5%	13.3%
blackscholes	PARSEC	blackscholes.c:374	T. Join	4,096	0.2%	1.1%	7.0%	12.7%
water-sp	SPLASH-2	interf.C:205	Barrier	512 molecules	1.7%	6.6%	10.7%	12.0%
radix (small)	SPLASH-2	radix.C:497	Barrier	2,097,152 integers (small input)	0.2%	0.5%	3.4%	5.7%
swaptions	PARSEC	HJM_Securities.cpp:271	T. Join	16 swaptions, 5000 simulations	0.0%	1.2%	1.9%	3.3%
radix	SPLASH-2	radix.C:497	Barrier	4,194,304 integers	0.0%	0.1%	0.2%	3.3%
ocean	SPLASH-2	slave2.C:812	Barrier	514×514 grid	1.3%	1.3%	1.5%	1.5%
fft	SPLASH-2	fft.C:623	Barrier	4,194,304 data points	0.1%	0.7%	1.5%	1.5%
streamcluster	PARSEC	streamcluster.cpp:706	Barrier	4,096 points	1.1%	1.4%	1.4%	1.1%
radiosity	SPLASH-2	rad_main.C:810	Barrier	room	0.1%	0.3%	0.4%	0.6%
Average					4.6%	8.1%	15.2%	16.8%

Table 1: Description of applications and parallel sections used in our experiments. Parallel section denotes the location of the *pthread_barrier_wait* or *pthread_join* call that delimits the parallel section. Imbalance is the average percentage of idle time threads spend in the parallel section.

Since dynamic instrumentation severely distorts execution time, we use instruction count as an approximation to the execution time. When a dynamic parallel section ends (i.e., a thread enters a barrier), recorded data is exported to output files for later processing.

Our second profiler is built on SESC [20], a cycle-accurate computer system simulator. This simulator functionally emulates an application and uses the instruction stream to drive a detailed model of the caches, memory, and processor cores, including the key microarchitectural structures (e.g., the instruction queue and reorder buffer, which allow for out-of-order execution). This simulator is routinely used by computer architects to evaluate architectural and microarchitectural proposals. We use the simulator to collect cache miss event counts and control flow edge counts simultaneously. Event recording and data exporting is very similar to our Pin-based tool (e.g., control flow edges are counted with a hashmap). The primary difference is that our SESC-based tool can accurately measure execution cycles and hardware event counts, but is much slower than the Pin-based version.

5. EXPERIMENTS AND RESULTS

In this section, we present our experimental setup and the output of our LIME analysis framework, and then verify the output to show that LIME reports imbalance-related performance bugs accurately.

5.1 Experimental Setup

We tested LIME using data gathered from 15 parallel sections in multithreaded applications from SPLASH-2 [24] and PARSEC [4] benchmark suites. For runs with the SESC-based profiler, we simulate a typical multi-core general purpose processor with cores at 1GHz, 16KB each of data and instruction cache per core, and 4MB of shared cache. Since the accuracy of LIME depends on the number of threads (i.e., sample points), we run our analysis on a varying number of cores (8, 16, 32, and 64) to verify that our scheme can find performance bugs accurately across a wide range of available cores. In all configurations, we run one application thread on each core.

Table 1 summarizes the location and type of each parallel section. We sort the sections in order of decreasing imbalance on 64 cores. The tested parallel sections cover a wide range of imbalance, from almost no imbalance (radix for 8 cores) to over 90% imbalance (LU for 64 cores).

5.2 Simulator-Based Profiling

Table 2 summarizes the results of LIME using the SESC-based profiler. The "Rpt" columns show the number of "important" events (those with a report score greater than 0.1) for both control flow events and cache miss events. The "Score" column shows the highest score reported among control flow and among cache miss events. The larger of the two highest scores is shown in bold font, to emphasize the event type that is more important according to LIME.

For all but two of the parallel sections, LIME is able to draw a consistent and clear conclusion on the most important event type. For the top nine sections, control flow events cause the load imbalance, while for the next four, cache miss events cause the imbalance. For the bottom two parallel sections, LIME fails to find a consistent cause of imbalance; this is because, in our current profiling implementation, we only collected control flow and cache miss event counts, which do not cause the imbalance for these benchmarks. We note that this is not a limitation of the LIME analysis framework, but rather a limitation of the even count profiling implementation. Section 5.4 explains the true cause of imbalance for these two parallel sections and how it affects LIME.

LIME consistently reports a small number of important events; this is important because it means the programmer should be able to inspect the spots in the source code that LIME reports. On average, for benchmarks with imbalance from control flow, it reports 1.3 important control flow instructions per benchmark. For benchmarks with imbalance from cache misses, on average LIME reports 2.1 important code points per benchmark.

5.3 Pin-Based Profiling

The limitations of the Pin-based implementation are that it introduces significant distortion in thread execution times and that it cannot accurately capture all the hardware interaction events that might be causing load imbalance. However, it can gather control flow edge information at speeds that allow "real-world" problem sizes.

To circumvent execution time distortion, in Pin-based pro-

Benchmark	8 Cores				16 Cores				32 Cores				64 Cores			
	Ctrl flow		Cache miss		Ctrl flow		Cache miss		Ctrl flow		Cache miss		Ctrl flow		Cache miss	
	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score	Rpt	Score
LU	2 (8)	**0.97**	0 (1)	0.01	1 (4)	**0.94**	0 (3)	0.05	1 (1)	**0.88**	3 (9)	0.14	1 (1)	**0.72**	3 (9)	0.29
volrend	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–
fmm	1 (2)	**0.83**	4 (20)	0.12	1 (3)	**0.51**	0 (11)	0.06	1 (5)	**0.50**	0 (25)	0.06	1 (6)	**0.36**	0 (21)	0.04
barnes	1 (1)	**1.00**	0 (0)	–	1 (1)	**0.86**	0 (3)	0.05	1 (5)	**0.89**	0 (3)	0.01	3 (3)	**0.64**	0 (17)	0.07
canneal	1 (4)	**0.24**	0 (18)	0.04	1 (4)	**0.45**	0 (21)	0.05	1 (2)	**0.39**	1 (12)	0.21	1 (2)	**0.32**	1 (16)	0.12
fluidani.	1 (4)	**0.69**	0 (3)	0.03	1 (3)	**0.20**	0 (3)	0.03	1 (1)	**0.39**	1 (15)	0.15	1 (1)	**0.19**	0 (12)	0.02
water-sp	1 (1)	**0.93**	0 (0)	–	1 (1)	**0.92**	0 (0)	–	1 (1)	**1.06**	0 (0)	–	1 (1)	**0.86**	0 (0)	–
streamcls.	3 (3)	**0.50**	0 (0)	–	3 (3)	**0.50**	0 (0)	–	3 (3)	**0.50**	0 (0)	–	3 (3)	**0.50**	0 (0)	–
radiosity	2 (2)	**1.00**	0 (0)	–	2 (2)	**0.99**	0 (1)	0.03	2 (2)	**0.95**	0 (1)	0.03	2 (2)	**0.78**	0 (1)	0.06
blackschls.	0 (0)	–	4 (4)	**0.49**	0 (0)	–	0 (0)	–	0 (0)	–	8 (19)	**0.92**	0 (0)	–	9 (14)	**1.00**
radix (small)	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (1)	**1.00**
radix	0 (0)	–	1 (1)	**0.99**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (2)	**0.99**	0 (0)	–	1 (1)	**1.01**
ocean	0 (0)	–	1 (1)	**0.97**	0 (0)	–	1 (3)	**0.98**	0 (0)	–	1 (1)	**1.00**	0 (0)	–	1 (3)	**1.06**
swaptions	0 (0)	–	0 (0)	–	0 (0)	–	0 (1)	0.07	0 (0)	–	2 (8)	**0.41**	0 (0)	–	0 (5)	0.08
fft	0 (0)	–	2 (2)	**1.03**	0 (0)	–	3 (3)	**0.61**	0 (0)	–	4 (4)	**0.81**	0 (1)	0.04	2 (2)	**0.37**

Table 2: Results of LIME analysis. The Rpt columns show the number of reported events with score greater than 0.1. The number of all reported events is listed in parentheses. The Score columns give the reported score for the highest-scored event of each type.

filing the instruction count is used as a proxy for execution time. This eliminates the possibility of identifying hardware-interaction events as causes of imbalance, but this profiler does not collect those events anyway. Further, the instruction count may not accurately represent the execution time. We validate that this profiler is useful in quickly identifying control flow sources of load imbalance. If hardware events trigger additional imbalance, a more expensive simulator-based approach can be used.

Overall, the control flow results of Pin-based profiling are very similar to those from the simulator. For brevity, Table 3 shows LIME results for only five parallel sections for "real" inputs (too large for simulation), including one from a benchmark (PLSA from bioParallel benchmark [10]) that is infeasible to run in simulation. Also shown are simulation-size inputs for three benchmarks for comparison, with scores from simulator-based profiling shown in parentheses. The profiling is done on an Intel Quad Core Xeon server using 8 threads (four cores, each with support for two threads).

Benchmark	Input Size	Slowdown	Ctrl Flow	
			Rpt	Score
LU	1K×1K matrix	21.1×	3 (10)	**0.97** (0.97)
	16K×16K matrix	23.9×	3 (6)	**0.93**
barnes	16,384 particles	84.1×	1 (6)	**0.96** (1.00)
	1,048,576 particles	88.7×	2 (5)	**0.97**
streamcluster	4,096 points	23.0×	2 (2)	**0.75** (0.50)
	1 million points	4.2×	2 (2)	**0.75**
radiosity	largeroom	62.6×	2 (3)	**1.00**
PLSA	100K sequence	127.0×	1 (13)	**1.36**

Table 3: Results with Pin-based profiling.

5.4 Verifying Reported Cache Misses

While we have already shown that our framework consistently identifies a programmer-manageable number of causes of load imbalance, we now verify that our framework (with the SESC profiler) *correctly* identifies the causes of load imbalance, starting with cache misses. To verify that reported cache miss events are indeed causing load imbalance, one possible approach would be to try to reorganize the data structures and algorithms in each application, re-evaluate parallel performance, and check if load imbalance has been

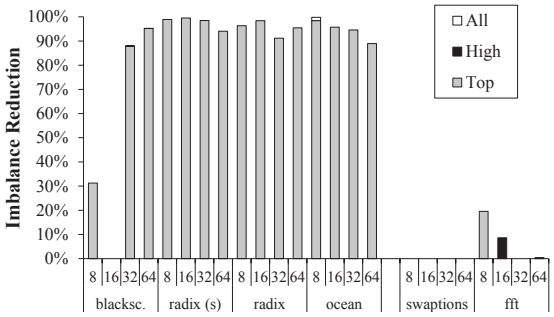

Figure 5: Imbalance reduction of each parallel section when reported cache misses are eliminated.

reduced. However, we lack domain expertise and resources to make such extensive changes in so many applications. Further, the results would highly depend on how well we understood each application, data structure, and algorithm.

Instead, we use cycle-accurate simulation to artificially eliminate the reported cache misses, while leaving all other aspects of the execution intact. To "erase" misses from reported memory access instructions, we override the simulated cache behavior for that instruction to make each dynamic instance of that instruction into a cache hit. If the reported cache misses are indeed the source of the imbalance problem, this modified execution should have a dramatically reduced load imbalance.

Figure 5 shows the results of this simulation for applications in which LIME reports cache misses as the main cause of load imbalance. The shaded portion of each bar represents the imbalance removed when only "erasing" the cache misses caused by the highest-scoring instructions for each application. The black portion represents additional imbalance removed when also "erasing" misses from all memory access instructions reported with final scores above 0.1. The white portion is the additional imbalance removed when erasing all cache misses reported by LIME as statistically significant causes of load imbalance.

For the first four applications in Figure 5, load imbalance is reduced dramatically when LIME-reported cache misses

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x4018b4	0.880	lu.C:668 (lu)

Reported source lines in *lu.C*

```
668   if (BlockOwner(I, J) == MyNum) {   /* parcel out blocks */
669       B = a[K+J*nblocks];
670       C = a[I+J*nblocks];
671       bmod(A, B, C, strI, strJ, strK, strI, strK, strI);
672   }
```

```
556   long BlockOwner(long I, long J)
557   {
558       return((I + J) % P);   // P: number of threads
559   }
```

Figure 6: LIME report for *LU*.

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x4068ec	0.999	render.C:38 (Render)

Reported source lines in *render.C*

```
31   Render(int my_node)        /* assumes direction is +Z */
32   {
33       if (my_node == ROOT) {
34           Observer_Transform_Light_Vector();
35           Compute_Observer_Transformed_Highlight_Vector();
36       }
37       Ray_Trace(my_node);
38   }
```

main.C

```
298   Render(my_node);
300   if (my_node == ROOT) {
307       WriteGrayscaleTIFF(outfile, image_len[X], ... );
310       WriteGrayscaleTIFF(filename, image_len[X], ... );
312   }
```

Figure 7: LIME report for *volrend*.

are "erased," except for configurations that do not have significant load imbalance to begin with (8 and 16 core configurations for blackscholes). It is important to note that, in all these simulations, we end up "erasing" misses from only 39.6% of all dynamic memory accesses, and the observed imbalance reduction is *not* caused simply by a dramatic reduction in execution time—in fact, the speedup in these runs mostly comes from reducing imbalance (making the slowest threads finish faster), with little performance benefit for the fastest threads.

Recall from Section 5.2 that LIME did not find a consistent cause of imbalance for the last two applications because neither control flow nor cache misses are the true cause. Further investigation reveals that imbalance is actually caused by cores having different success in getting access to the L1–L2 on-chip (back-side) bus when servicing cache misses—in these applications, this bus has high utilization and the bus arbitrator seems to be favoring some cores at the expense of others. We confirm this by simulating a higher-bandwidth bus (without "erasing" any misses) and observing a 95% imbalance reduction. However, unlike cache miss (and many other) events that can easily be attributed to particular instructions, such attribution for bus contention events is an open problem that is beyond the scope of this paper.

5.5 Verifying Reported Control Flow Causes

To verify that LIME correctly reports control flow events that cause imbalance, we use a different methodology than for cache misses—control flow events cannot be "erased" without affecting many other aspects (including correctness) of program execution. Thus, in all parallel sections where LIME reported control flow code locations as a significant cause of imbalance (Table 2), we manually confirm that the location and nature of the problem was correct. For lack of space, we only describe this analysis for three examples, selected to both illustrate different types of imbalance causes and to only require brief code fragments for explanation.

5.5.1 LU

Among the parallel sections used in this paper, *LU* has the most imbalance—the last-arriving thread takes over ten times longer than the first-arriving thread.

For the 32-core configuration, our framework reports only one static instruction, shown in Figure 6, as statistically significant cause of imbalance (at line 668). In the other three configurations, LIME also reports line 668 as the top-scoring (by a large margin) cause of imbalance.

The corresponding code point for the top-scoring instruction is also shown in Figure 6. This is an *if*-statement that parcels out blocks to threads using function *BlockOwner*, which returns the summed block coordinates modulo number of threads, P. Intuitively, this method of assigning blocks to threads should produce a balanced load. However, values of I and J vary in a range determined by the program's inputs—they may be small and/or not a multiple of P, causing uneven distribution of blocks to threads. For example, when I and J take values 1 through 15, the 225 blocks are distributed among 32 threads and ideally each thread should get about 7 blocks. The actual assignment using $(I+J)\%P$ turns out to assign only one block to threads 2 and 30, 2 blocks to threads 3 and 29, etc., giving 15 blocks to thread 16.

When line 558 in the *BlockOwner* function is changed to "return $(J\%Ncols)+(I\%Nrows)\times Ncols$;", where Ncols and Nrows are 8 and 4, respectively, for a 32-core configuration, the assignment of blocks to threads becomes more balanced and results in eliminating 61% of the original load imbalance. This confirms that imbalance was indeed introduced at the code point reported by LIME.

5.5.2 Volrend

Our second verification example is from *volrend*, which has up to 46.9% imbalance on 64 cores. The LIME report for 32 cores, summarized in Figure 7, suggests with high confidence that the return point of the function *Render* is the source of imbalance. At the first glance, this looks like a false report, but a closer inspection reveals that the problem is the mapping of the instruction to the line of source code because the compiler obfuscated the situation via inlining and other optimizations. When we examine the code in a disassembler, the reported control flow instruction is actually the *if*-statement at line 300 (immediately after the callsite for *Render*). This *if*-statement assigns extra work to the main thread. When compiled without optimizations, LIME reports the *if*-statement in line 300 correctly.

5.5.3 Barnes

Our third verification example is from *Barnes*, with 13.5% imbalance on the 32-core configuration. The LIME report for 32 cores, summarized in Figure 8, says the control flow edge from line 116 in *grav.C* to line 112 accounts for the imbalance. This edge corresponds to the recursive function call to *walksub*—*Barnes* implements the Barnes-Hut approach for the N-body problem, and *walksub* recursively traverses

Report from our analysis

No.	Address	Score	Code point (func.)
1	0x405470	0.893	grav.C:116 -> grav.C:112
2	0x40548c	0.030	grav.C:113 (walksub)
3	0x4055cc	0.024	grav.C:136 -> grav.C:114

Reported source lines in *grav.C*

```
105    void walksub(nodeptr n, real dsq, long ProcessId)
106    {
107       nodeptr* nn;
                ⋮
112       if (subdivp(n, dsq, ProcessId)) {  // First branch in walksub
113          if (Type(n) == CELL) {
114             for (nn = Subp(n); nn < Subp(n) + NSUB; nn++) {
115                if (*nn != NULL) {
116                   walksub(*nn, dsq / 4.0, ProcessId);
117                }
118             }
```

Figure 8: LIME report for *barnes*.

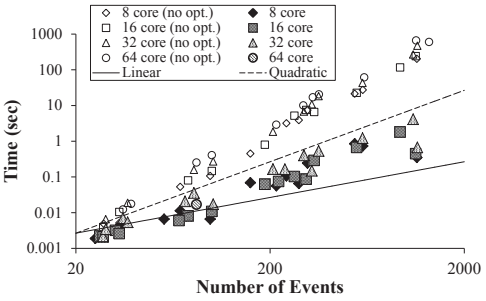

Figure 9: Clustering time vs. number of events. Each point represents clustering time of one parallel section. Note the logarithmic scale markings on each of the axes.

the primary data structure, a tree. Since LIME reports the tree traversal is imbalanced, this suggests that the tree itself is imbalanced. Further investigation shows that 86.5% of the total imbalance comes from the first dynamic instance of the parallel section, because the tree is skewed at the beginning of the run. The first Barnes-Hut iteration rebalances the tree and the load imbalance decreases.

5.6 Scalability of LIME

While LIME ran fast enough for this study, two possible concerns are: (1) what happens to analysis time as the core count increases, and (2) what happens to analysis time as event count (i.e., program size) increases?

A single-threaded implementation of LIME analysis took an average of 1.4, 1.8, 3.2, and 5.0 seconds to analyze the parallel sections for 8, 16, 32, and 64 cores, respectively, on a 2.67GHz Intel Quad Core Xeon processor with 12GB memory. This time includes clustering, leader selection, and regression. Since the analysis time grows in sub-linear proportion to the core count, we expect that a parallel implementation of LIME analysis will have a favorable scaling trend as the number of cores (for both application execution and analysis) increases.

The most time consuming part of LIME is the clustering part of the analysis—it accounts for over 76% of the average analysis time when no optimization is applied. A naive implementation of our hierarchical clustering method has asymptotic complexity of $O(n^3)$ where n is the number of event counts—to create a near-constant number of clusters, LIME does $O(n)$ merges, and for each merge it recomputes all $O(n^2)$ entries in the new cluster proximity matrix. This would be a major problem for applying LIME to real applications. To accelerate the clustering algorithm, we can cache the proximity matrix and perform bulk clustering. When merging two clusters, only the proximity values involving those two clusters become useless. Therefore, we re-use (cache) the proximities and compute only $O(n)$ new values. The "bulk clustering" optimization merges multiple clusters per step—as we compute the proximity matrix, we track clusters that are very close to each other and merge all of them at once. In this study, we use a proximity threshold of 0.99 for bulk clustering. Bulk clustering reduces the number of merges, but runs the risk of producing less precise clustering. We did not experience any imprecise clustering in our experiments.

Figure 9 compares the speed of the clustering implementation with and without our acceleration technique using a log–log plot. While our optimized implementation is still $O(n^2)$ (bulk clustering does not reduce the asymptotic complexity), in practice it shows near-linear scaling with n. This demonstrates that LIME is scalable with program size. Further optimizations are certainly possible but we leave them for future experimentation and fine-tuning.

6. RELATED WORK

Performance debugging has long been studied in distributed systems. Much work focuses on finding the causal trace, or the trace with the longest path through a distributed system; this is analogous to the slowest thread in our study. The causal trace naturally tells programmers where to focus optimization efforts. LIME shares the same goal as this work, but works in a different domain. Among the related literature, the performance debugging method proposed by Aguilera et al. [1] draws our attention—they use a black box approach that finds the causal trace without any knowledge of the system, and a signal processing technique called convolution to infer causal relationships. LIME collects profile data without a programmer's intervention, and uses clustering and regression analysis to infer causal relations between events and load imbalance.

A number of tools exist to detect and measure parallel overheads and inefficiency (i.e., idleness). One recent example is from Tallent et al. [22]. Their goal is to pinpoint where parallel bottlenecks occur, and classify bottlenecks as overhead or idleness. Our work is largely orthogonal to this—once a programmer knows that a problem exists, they can use our framework to help find the cause for the bottlenecks they have detected. Tallent et al. also have work on analyzing lock contention [23]. This has a lot in common with our work since they try to detect the cause of lock contention and identify the lockholder to blame. Our work focuses on barriers rather than locks and provides more direct information about where in the code the problem arises.

There is also significant work on trying to optimize performance and energy in the presence of load imbalance. Thrifty Barrier [14] predicts how much slack (i.e., idle time) each thread will have using a history-based predictor, and saves power by putting non-critical threads into a sleep state. Meeting Points [5] uses a different predictor that counts thread deviation at checkpoints called meeting points. It delays non-critical threads using dynamic voltage and frequency scaling to save power. It also attempts to accelerate

the critical thread by prioritizing it. The Thread Criticality Predictor [2] similarly predicts thread criticality based on adjusted cache miss rates, and prioritizes threads based on criticality. While automatically detecting and prioritizing critical threads requires no programmer effort, it can only reduce load imbalance by a limited amount. Instead, we help programmers find and permanently fix imbalance problems in applications regardless of how severe the problems are; this can (and usually does) have a dramatic performance impact.

7. CONCLUSIONS

This paper addresses a major problem in achieving good performance on multi-core processors: load imbalance. The existence of load imbalance is relatively easy to detect, but it is often very challenging to determine the cause of the problem and the point in the source code where the problem is introduced. We propose LIME, a framework that automates this process. LIME first uses profiling to collect counts of control flow events and hardware events for the different threads. It then uses clustering and regression to identify the small set of events that introduce significant amounts of imbalance. We show that LIME provides accurate and useful feedback to programmers on a set of popular parallel benchmarks. We also show that our LIME prototype runs fast enough to be used on large programs.

8. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants 0447783 and 0903470, and by the Semiconductor Research Corporation under contract 2009-HJ-1977.

9. REFERENCES

[1] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *9th Symp. on Operating Systems Principles (SOSP)*, 2003.

[2] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *36th Int'l. Symp. on Computer Architecture (ISCA)*, 2009.

[3] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *2008 Int'l. Symp. on Workload Characterization*, 2008.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[5] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *17th Int'l Conf. on Parallel Arch. and Compilation Techniques (PACT)*, 2008.

[6] M. Efroymson. Multiple regression analysis. In *Mathematical Methods for Digital Computers*, pages 191–203. Wiley, 1960.

[7] G. H. Golub and C. F. Van Loan. *Matrix computations*. John Hopkins Univ. Press, 1996.

[8] Intel Corp. Intel threading building blocks 3.0, 2010. `http://www.intel.com/software/products/tbb/`.

[9] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[10] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. In *12th Int'l. Symp. on High Performance Computer Architecture (HPCA)*, 2006.

[11] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comput.*, 55(6):769–782, 2006.

[12] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *34th Int'l. Symp. on Computer Architecture (ISCA)*, 2007.

[13] R. Larsen and M. Marx. *An Introduction to Mathematical Statistics and Its Applications*. Pearson, 2000.

[14] J. Li, J. F. Martinez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *10th Int'l. Symp. on High Perf. Computer Architecture (HPCA)*, 2004.

[15] R. G. Lomax. *Statistical Concepts: A Second Course*. Lawrence Erlbaum Associates, 2007.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[17] M. J. Norusis. *PASW Statistics 18 Statistical Procedures Companion*. Pearson, 2010.

[18] OpenMP Architecture Review Board. Openmp application program interface. Technical report, 2010. `http://openmp.org/wp/openmp-specifications/`.

[19] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *34th Int'l. Symp. on Computer Architecture (ISCA)*, 2007.

[20] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[21] C. Sanderson. Armadillo library, 2010. `http://mloss.org/software/view/176/`.

[22] N. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *14th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[23] N. Tallent, J. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *15th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[24] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture*, 1995.