# MAMA: Mostly Automatic Management of Atomicity

Christian DeLozier Joseph Devietti Milo M. K. Martin
Computer and Information Science Department, University of Pennsylvania
{delozier, devietti, milom}@cis.upenn.edu

## Abstract

Correctly synchronizing the parallel execution of tasks remains one of the most difficult aspects of parallel programming. Without proper synchronization, many kinds of subtle concurrency errors can arise and cause a program to produce intermittently wrong results. The long-term goal of this work is to design a system that automatically synchronizes a set of programmer-specified, partially-independent parallel tasks. We present here our progress on the MAMA (Mostly Automatic Management of Atomicity) system, which can infer much but not all of this synchronization. MAMA provides a safety guarantee that a program either executes in a correctly atomic fashion or it deadlocks. Initial experiments indicate that MAMA can semi-automatically provide atomicity for a set of Java benchmarks while still allowing parallel execution.

## 1. Introduction

Despite decades of research, writing multithreaded programs that execute efficiently and correctly remains a challenging task [12]. Successful parallelization of non-trivial code continues to be an effort worthy of publication [18], though with increasingly parallel CPUs and GPUs in everything from cell-phones to servers, more non-research programmers are called upon to utilize these parallel hardware resources. New systems that simplify parallel programming are crucial to support these parallel programmers.

Coordinating parallel execution via synchronization is one of the key challenges of parallel programming [10], though there are many other important facets such as discovering parallelism, debugging, testing, and performance tuning. We focus in this work on specifying synchronization: a notoriously error-prone task, rife with opportunities for subtle errors like data races, atomicity violations [7], ordering violations [15], and deadlocks.

Our long-term goal is a system that automatically provides synchronization for a set of partially-independent parallel tasks, with those tasks specified by the programmer. We think this division of labor leverages well the complementary strengths of both humans and tools. Human creativity finds the parallelism within a program that is hard for machines to discover. Machine rigor automatically applies a generic synchronization protocol without the occasional lapse humans might introduce.

In this work, we present a partial result towards this larger goal: the MAMA (Mostly Automatic Management of Atomicity) system, which can infer much of the synchronization used in parallel programs. MAMA provides *atomicity* for parallel thread executions by ensuring that the actions of a thread appear, to other threads, to occur instantaneously. We draw a distinction between *atomicity* and *ordering* constructs (Section 2); our current MAMA prototype can infer the former but not the latter. We also find that atomicity constructs are far more common in our workloads.

MAMA works by conservatively over-synchronizing a program, providing safe parallel execution without any programmer interaction. This over-synchronization comes with two costs, however: liveness and performance. MAMA tackles these challenges with programmer help. When over-synchronization leads to deadlock, MAMA uses the information represented by the deadlock to guide programmers to precise code points where liveness can be restored. When over-synchronization leads to serialization, MAMA similarly identifies the code points responsible and guides programmers to reducing serialization.

MAMA is thus a hybrid system that relies on programmers for several key tasks: expressing the parallelism within a program, identifying ordering constructs, and identifying when MAMA's automatic synchronization is overly conservative. Nevertheless, by taking much of the burden of specifying atomicity constructs out of programmer's hands, we think that MAMA is a useful advance towards systems that handle synchronization autonomously.

In this paper, we describe the MAMA algorithm and its safety properties (Section 2), then describe how MAMA and human programmers collaborate on the task of managing the synchronization of a program (Section 3) and how our MAMA software prototype is implemented (Section 4). We evaluate our prototype implementation (Section 5) by removing the locking from existing multithreaded workloads to show that MAMA can run these programs safely and can recover some performance through parallelism.

## 2. Mostly Automatic Management of Atomicity

MAMA takes as input a program divided into partially-independent parallel tasks by a human programmer. These tasks represent the parallelism within a program (e.g., rays in a ray tracer), but lack synchronization to coordinate data sharing between tasks (e.g., updating an output buffer). This input program will not, in general, execute correctly on its own; it needs synchronization to prevent erroneous execution.

We decompose the task of specifying synchronization into two parts: specifying ordering constraints and atomicity constraints. *Ordering constraints* enforce a particular order between two events, as with constructs like barriers, thread fork/join and condition variables. *Atomicity constraints* specify ordering-agnostic logical mutual exclusion between two events as with constructs like locks or transactions [9]. Ordering and atomicity are not orthogonal concepts, e.g., ordering constructs are often implemented using atomicity constructs inside loops, and atomicity can be enforced in a cumbersome fashion using ordering constructs as well. Nevertheless, existing synchronization constructs can be readily characterized as providing either ordering or atomicity. This distinction of ordering versus atomicity has been noted before in classifying patterns of concurrency bugs [15].

Our current MAMA prototype infers atomicity constraints for a program, but relies on programmers to specify ordering constraints. Note that ordering constraints are often implicitly involved when expressing parallelism, e.g., with thread fork and join. Moreover, ordering constructs appear an order of magnitude less frequently in programs than atomicity constructs (Table 1).

### 2.1 Basic MAMA

Our MAMA prototype is based on a simple algorithm that provides generic but safe access to shared variables. On each access to

| $t_0$ | $t_1$ |
|---|---|
| write A | write A |

(a) deadlock due to data sharing

| $t_0$ | $t_1$ |
|---|---|
| read A | read B |
| read B | read A |

(b) deadlock due to read sharing

| $t_0$ | $t_1$ |
|---|---|
| read A | read A |
| write A | write A |

(c) deadlock with reader-writer, but not mutex, locks

| $t_0$ | $t_1$ |
|---|---|
| write A | write A |
| write A | write A |
| join $t_1$ | |
| write A | |

(d) deadlock with thread join

| $t_0$ | $t_1$ |
|---|---|
| write A | write B |
| write B | write A |

(e) deadlock AMA cannot safely resolve

**Figure 1.** Code examples illustrating potential deadlocks MAMA can resolve safely (a-d) and only with programmer help (e).

a shared variable $v$, an atomic section $A_v$ begins for that shared variable. For this prototype of MAMA, atomic sections are enforced using locks, though other implementations (e.g., transactional memory) are also possible. In our lock-based implementation of MAMA, we first associate a mutex lock with every variable in the program. As a thread $t_0$ is executing, before $t_0$ reads or writes a shared variable $v$, $t_0$ first acquires $v$'s lock $l_v$. If $t_0$ is able to acquire $l_v$, or if $t_0$ already holds $l_v$ from a previous access, then $t_0$ proceeds to its next access. Otherwise, $t_0$ becomes blocked on $l_v$ if $l_v$ is held by another thread $t_1$. In this basic version of the algorithm, locks are never released.

This basic algorithm provides the following safety property: under MAMA, a program executes in a correctly atomic fashion or it deadlocks. One can think of MAMA as over-approximating the atomicity that a program actually requires, as locks are always acquired before a shared variable is accessed and never released. Although MAMA uses fine-grained (per-variable) locking, by never releasing locks it over-approximates any usage of coarse- or fine-grained locking. Another way to think of MAMA is that it enforces atomic execution for each thread, introducing waiting whenever inter-thread communication threatens to violate atomicity.

Our distinction between ordering and atomicity constructs is crucial for our safety property to hold. By assuming that ordering constraints are provided, we are free to execute the threads of the program in any order that preserves atomicity, and the MAMA algorithm accordingly enforces atomicity but not ordering.

With this basic version of MAMA, deadlocks frequently arise. We discuss below several cases in which we can break deadlocks without compromising MAMA's key safety property, and how to handle the remaining deadlocks that do not fall into these cases.

### 2.2 Breaking deadlocks

With the basic version of MAMA, deadlocks can easily arise when threads share data, as shown in Figure 1a. Suppose that $t_0$ first acquires $l_A$; then $t_1$ will never be able to make progress since $t_0$ will never release $l_A$. However, a thread $t$ needn't hold onto locks after it exits – $t$'s entire execution has occurred atomically and releasing locks at exit cannot jeopardize this atomicity. Thus, we release locks at thread exit, allowing the program in Figure 1a to execute without deadlock.

To enable more concurrency without sacrificing safety we can also generalize the basic algorithm in Section 2.1 to use reader-writer locks instead of mutex locks. With this extension, the type of lock being acquired (read/write) depends on the kind of access a thread is about to perform. Using reader-writer locks allows the program in Figure 1b to execute without deadlock as both threads can acquire read locks on locations $A$ and $B$ concurrently.

Reader-writer locks can also, however, introduce deadlocks into a program that was deadlock-free with mutex locking. Consider the program in Figure 1c. With reader-writer locks, both threads can complete their reads and then stall trying to upgrade to a write lock. With mutex locks, the execution of each thread will be serialized but progress is preserved. Thus, neither locking scheme has inherently superior progress properties with MAMA. For our prototype, we have adopted reader-writer locks as the default because they admit more concurrency.

Other deadlocks that involve an ordering constraint can also be broken safely in an automatic way. Recall that MAMA requires a programmer to specify the ordering constraints for a program via barriers, thread fork/join, condition variables, etc.

Consider the program in Figure 1d, in which $t_1$ can get stuck in a deadlock before its first write to $A$, if $t_0$ is at the join statement and thus already holds $l_A$. Since $t_0$ is joining with $t_1$, however, $t_0$ will never be able to make progress until $t_1$ does. The ordering constraint expressed by the thread join thus *necessitates* that atomicity be broken to ensure progress. Here we again leverage the correctness of the ordering constraints provided by the programmer, specifically relying on the fact that they preserve forward progress. This fact allows us to break atomicity at principled times without violating safety. Any ordering constraint, expressed via thread join, condition variable waits, etc. can be used to break atomicity at principled times to preserve progress.

### 2.3 Remaining deadlocks

Our evaluation shows that few deadlocks arise on our benchmarks when using the extensions described above (Section 5). Nevertheless, the possibility of deadlock is always present with MAMA, as with the program in Figure 1e. Next, we describe how we leverage programmer intuition to handle the remaining deadlocks.

## 3. Bringing the Programmer into the Loop

After applying the MAMA extensions described above, some programs may still encounter deadlocks (Figure 1e). Moreover, even programs that continue to make progress may execute in an extremely serial fashion, negating the performance benefits of parallelism. In these situations, the MAMA system can identify deadlocks and serialization and provide suggestions to the programmer about how to improve performance.

### 3.1 Annotations for liveness

One useful aspect of the deadlocks that MAMA encounters are that they arise precisely at the point in the execution at which the atomicity of a thread's execution must be broken to maintain forward progress, i.e., the point at which a thread must release a lock to allow the program to continue executing. All threads, program counters, and shared variables involved in the deadlock are readily identifiable during the deadlock detection process. While the MAMA system cannot automatically determine whether this early lock release is safe or not – such a release certainly violates the thread's atomicity, but many programs do not require entire threads to execute atomically – we can use the precise deadlock state to generate a specific report for the programmer identifying locations at which atomicity must be broken to maintain forward progress.

To preserve progress, at least one lock release must be added to the code manually. We use a special function `MAMA_release()` to

| | | Atomicity | Ordering | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Benchmark | LoC | `synchronized` | `volatile` | `wait()` | `notify()` | `run()` | `join()` | Barrier | Total |
| crypt | 314 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 2 / 14 | 2 / 14 | 0 / 0 | 4 / 28 |
| lufact | 461 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 1 / 7 | 1 / 7 | 1 / 23,952 | 3 / 23,966 |
| lusearch | 124,105 | 440 / 1,327,734 | 21 / 1,157,045 | 18 / 64 | 27 / 64 | 1 / 7 | 1 / 7 | 0 / 0 | 28 / 1,157,187 |
| matmult | 187 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 1 / 7 | 1 / 7 | 0 / 0 | 2 / 14 |
| moldyn | 487 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 1 / 7 | 1 / 7 | 1 / 2,424 | 3 / 2,438 |
| montecarlo | 1,165 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 1 / 7 | 1 / 7 | 0 / 0 | 2 / 14 |
| pmd | 60,062 | 15 / 322 | 2 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 3 / 8 |
| series | 180 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 1 / 7 | 1 / 7 | 0 / 0 | 2 / 14 |
| sor | 186 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 1 / 7 | 1 / 7 | 1 / 1,600 | 3 / 1,614 |
| sunflow | 21,970 | 43 / 770 | 0 / 0 | 0 / 0 | 0 / 0 | 2 / 14 | 2 / 14 | 0 / 0 | 4 / 28 |
| xalan | 172,300 | 107 / 4,448,917 | 0 / 0 | 6 / 8 | 8 / 1,704 | 1 / 7 | 1 / 7 | 0 / 0 | 16 / 1,726 |

**Table 1.** Static and dynamic synchronization in benchmarks. Cells are listed as static / dynamic. PMD does not explicitly launch threads because it uses Futures (1 static / 8 dynamic).

allow a programmer to tell the runtime system that the lock $l_A$ for a shared variable $v$ should be released at a particular code point.

The deadlock reports generated by MAMA give a programmer a concrete starting point at which to begin analyzing the code. We have found anecdotally that this precise starting point is extremely useful for guiding a programmer to a few key lines within a large code base. Liveness annotations can be added to the code incrementally, as future deadlock reports will identify remaining issues.

### 3.2 Annotations for performance

The same annotations used for liveness can also be used to improve performance. Releasing locks early is the primary way of reducing the serialization of a program running under MAMA, allowing more parallelism and faster execution times. Unlike liveness annotations, deadlocks cannot drive the placement of performance annotations. Instead, we extend the MAMA system to identify when the execution is being serialized and suggest the placement of lock releases to increase parallelism. Serialized execution can be identified as many threads being blocked waiting for the same lock. Once MAMA identifies serialized execution, it can suggest where locks should be released to increase parallelism.

*Variable initialization* can lead to unnecessary serialization. Under this pattern, one thread writes to a variable once and then the variable becomes read-only and is read by many threads. This pattern can easily cause deadlock, as the initializing thread maintains a write-lock on the variable for its entire execution. MAMA can detect the pattern of a large number of read attempts paired with a single write and automatically downgrade the writer's lock to a read-lock. This downgrade preserves safety but may induce future deadlocks if some thread attempts to acquire a write-lock.

## 4. Prototype and Experimental Setup

Using the Roadrunner [8] framework, we developed a runtime system to dynamically apply the MAMA algorithm to Java applications. RoadRunner's dynamic instrumentation adds overhead but enables us to gather preliminary indications of the effectiveness of the MAMA algorithm. Our current prototype instruments application code only, not library code. MAMA associates a reader-writer lock with each program variable. On each variable access, MAMA requires that the accessing thread either already owns or acquires the lock for the given variable. As discussed in Section 2, MAMA detects deadlocks in order to provide liveness hints to the programmer. MAMA uses a distributed deadlock detection algorithm [4] to detect and break deadlocks.

We evaluated MAMA on benchmarks from the Java Grande [20] and DaCapo [3] benchmark suites. From the DaCapo suite, only avrora, lusearch, jython, pmd, sunflow, tomcat, and xalan run under RoadRunner's baseline instrumentation. We removed jython from our suite because it did not display significant parallelism. Under MAMA instrumentation, we are currently debugging states in avrora and tomcat under which all threads `wait()` with no threads left to `notify()`. The number of lines of code in each of these programs are shown in Table 1. We ran all of the benchmarks on a 32-core/64-thread machine with four Intel Xeon E7-4820 2.0 GHz sockets and 128 GB RAM. For the parallel experimental results, all benchmarks were run using 8 threads and pinned to a single socket to avoid the performance overheads of data-sharing across multiple sockets. Runtime performance overheads were measured using Java's `currentTimeMillis()`, and memory overheads were measured at the high water mark using the `jvisualvm` tool provided by the JDK. We ran RoadRunner using fine-grained field and array tracking (one shadow variable per field and one shadow variable per array element). For `crypt`, `lufact`, `sor`, `montecarlo`, `sunflow`, and `xalan`, we used coarse-grained array tracking but chunked the arrays into 64 buckets to reduce the runtime and memory overheads of fine-grained tracking on large arrays. We validated that the benchmarks executed correctly using the built-in validation mechanisms of the Java Grande and DaCapo benchmarks For the performance evaluation, we averaged five runs of each benchmark.

To gain confidence that the MAMA algorithm works correctly on programs without atomicity constructs, we removed the locking from our benchmarks. All `synchronized` blocks were automatically removed by modifying RoadRunner to not insert `MONITOR_ENTER` and `MONITOR_EXIT` bytecodes. We also manually removed uses of Java's `ReentrantReadWriteLock`, Java atomics, and concurrent data structures. For example, we replaced the `PriorityBlockingQueue` used by `sunflow` with a non-concurrent `PriorityQueue`. After this synchronization removal (but prior to applying MAMA), `sunflow` and `xalan` produced incorrect output, though the other benchmarks produced correct results on multiple trial runs.

We ran each benchmark 40 times with randomly-inserted sleeps before variable accesses to explore uncommon thread schedules. This exposed an atomicity violation in `lusearch`'s library code as MAMA does not currently instrument library code. We believe that adding support for library code in a future version of MAMA will resolve this issue.

## 5. Preliminary Results

We evaluated both the effectiveness and performance of the MAMA algorithm on various benchmarks. First, we observed how effective the MAMA algorithm was at avoiding deadlocks and at breaking deadlocks when they occurred. We also recorded the types of deadlocks that were broken for forward progress. In some cases, a deadlock may occur inside a former critical section, and we observed how often these events occurred in real programs. Second,

| Benchmark | Safe | Liveness | Performance |
|---|---|---|---|
| crypt | 5,250,330 | 0 / 0 | 0 / 0 |
| lufact | 4,240,434 | 1 / 2,977 | 4 / 12,583,386 |
| lusearch | 250 | 0 / 0 | 4 / 43 |
| matmult | 700,405 | 0 / 0 | 0 / 0 |
| moldyn | 2,019,626 | 3 / 178 | 0 / 0 |
| montecarlo | 647,279 | 0 / 0 | 28 / 143,362 |
| pmd | 3,442 | 0 / 0 | 4 / 1,915,602 |
| series | 15 | 0 / 0 | 0 / 0 |
| sor | 4,508,422 | 1 / 4,058 | 0 / 0 |
| sunflow | 262,448 | 1 / 1 | 3 / 27,948 |
| xalan | 19,908 | 0 / 0 | 0 / 0 |

**Table 2.** Characterization of the deadlocks that occur (and are broken) in the benchmarks under MAMA. "Safe" deadlocks are deadlocks in which one of the threads is joined, waiting on a condition variable, at a barrier, or exited. "Liveness" deadlocks are broken to allow the program to make progress. "Performance" deadlocks are prevented by releasing locks early based on common patterns found in the benchmarks. In the "Liveness" and "Performance" columns, the static count is on the left, and the dynamic count is on the right.

we measured the performance overheads of the MAMA algorithm to determine how the performance overheads could be reduced in future implementations.

## 5.1 Benchmark Synchronization Characteristics

Table 1 details the static and dynamic synchronization present in the benchmarks used in our study. For our workloads ordering constructs typically occur at least an order of magnitude less frequently, both statically and dynamically, than do atomicity constructs. Lu et al. [15] also found, in mining the bug tracking databases of several large open-source parallel code bases, that atomicity violations were about twice as common as ordering violations. We believe that relieving programmers from the burden of specifying atomicity constructs is an important and useful goal.

## 5.2 Effectiveness

To evaluate the effectiveness of MAMA, we applied the algorithm to multiple parallel benchmarks and recorded where deadlocks occurred in the target program. Figure 2 details the deadlocks that occurred during the execution of the benchmark suite under MAMA. The majority of deadlocks occurred while one of the threads was either joined, waiting on a condition variable, at a barrier, or exited. Thus, most deadlocks could be broken with confidence that MAMA was not breaking the atomicity required by the program.

`lufact`, `moldyn`, `sor`, and `sunflow` required annotations for liveness. Despite the number of dynamic deadlock breaks that were required for these benchmarks, the number of static annotations to perform these deadlock breaks is just six across all benchmarks. In `lufact`, `sor`, and `moldyn`, deadlocks occur despite these benchmarks not having any `synchronized` blocks in the original code because these benchmarks all use barriers for synchronization. In each of these benchmarks, it is safe to break the deadlocks that occur because the overlapping reads and writes are synchronized by the barrier.

In `sunflow`, applying MAMA to the benchmark resulted in a deadlock inside a former critical section because the program checks a shared variable for null before writing to it. When two threads read the shared variable before writing to it, an upgrade deadlock occurs in the critical section. Although this is potentially a problem with using reader-writer locks in MAMA, we found that this situation did not frequently occur in these benchmarks. In many cases, the upgrade deadlock, as shown in Section 2.2, is prevented by a write to a separate shared variable that serializes the threads

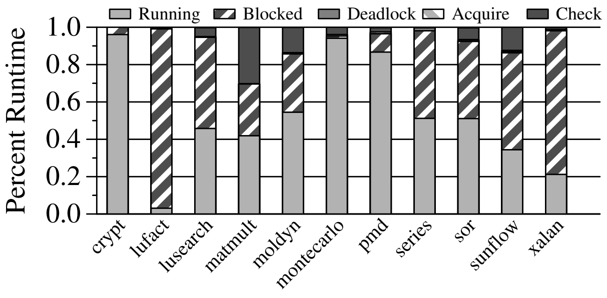

**Figure 3.** Percentage runtime for various routines and states in MAMA.

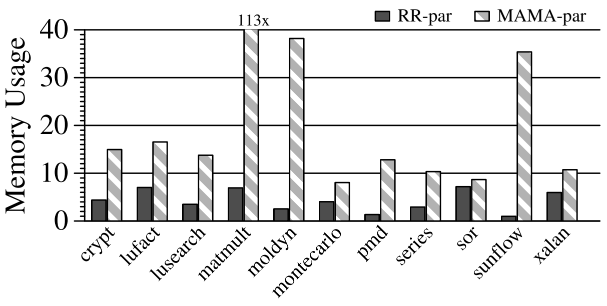

**Figure 4.** Normalized high water mark memory usage parallel RoadRunner and parallel MAMA, normalized to JVM execution

prior to the read and subsequent write of the shared variable. As shown below, the write to B prevents the upgrade deadlock that would otherwise occur due to the read and write to A.

| $t_0$ | $t_1$ |
|---|---|
| write B | write B |
| read A | read A |
| write A | write A |

In some cases, we explicitly broke the atomicity guarantees of MAMA in order to allow increased parallel execution. In `lufact`, `lusearch`, `montecarlo`, `pmd`, and `sunflow`, we identified shared counter variables that were updated atomically, requiring that the locks for these variables be released early to allow the threads to execute in parallel. In `montecarlo` and `sunflow`, we also identified locks that were acquired for static initialization and could thereafter be downgraded to read-shared.

## 5.3 Parallelism

We evaluated MAMA's parallel execution (MAMA-par) against a few different baselines: parallel RoadRunner execution (RR-par), serialized RoadRunner execution (RR-ser), and serialized MAMA execution (MAMA-ser). We compare MAMA to serialized baselines to verify whether MAMA can indeed exploit the parallelism in each workload, and whether MAMA exploits enough parallelism to overcome its locking overheads. By comparing the difference between RR-par and RR-ser with the difference between MAMA-par and MAMA-ser, we can determine whether or not MAMA preserves the potential parallel speedup in each benchmark.

The results of our evaluation are shown in Figure 2. We note that, on average, RoadRunner incurs approximately 6x overhead over the uninstrumented programs. Due to the overheads of locking on every variable access, MAMA is never faster than the RR-par baseline. Nevertheless, MAMA-par is capable of exploiting parallelism in many benchmarks. Compared to the RR-ser baseline, MAMA-par is competitive in many cases and performs better than RR-ser on `lusearch`, `montecarlo`, and `series`. In these cases, MAMA-par overcomes its locking overheads with parallelism.

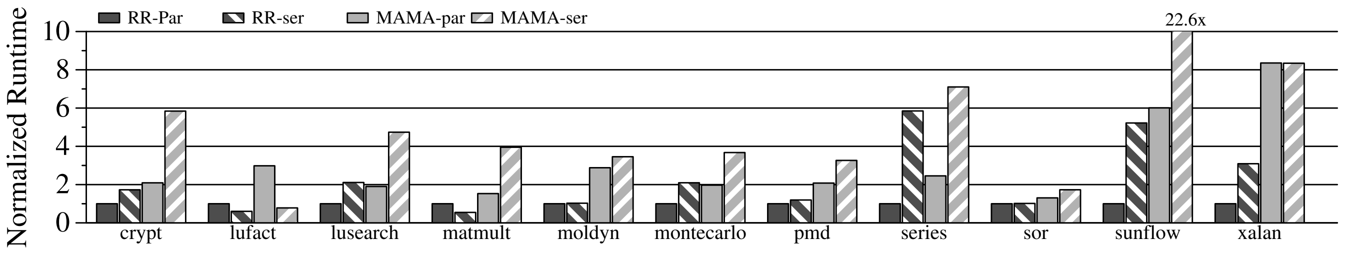

**Figure 2.** Runtime of parallel RoadRunner, serialized RoadRunner, parallel MAMA, and serial MAMA, normalized to parallel RoadRunner

Finally, we compared MAMA-par to MAMA-ser to measure the amount of parallelism in the execution of the benchmarks under MAMA. In almost all cases, MAMA-par handily outperforms MAMA-ser. There are two exceptions. `lufact` does not scale well with eight threads under RoadRunner's instrumentation (RR-par is slower than RR-ser). `xalan` does not exhibit parallelism under MAMA, even with early lock breaking, though there is clearly parallelism within the workload. More investigation is necessary to determine how to unlock `xalan`'s parallelism.

Most of the performance overheads of MAMA stem from two sources: testing locks for ownership and serialization due to contested locks. Figure 3 shows the summed performance counters for all threads in each benchmark. In general, the deadlock detector is run infrequently and only when threads are blocked. Thus, the overheads of deadlock detection are negligible. On every variable access, MAMA must check to see if the corresponding lock is already held. In the case of read-sharing, MAMA must check to ensure that the thread is one of the read owners of the lock. Recording the read owners of a lock is necessary to allow deadlocks to be broken at runtime. However, this overhead might be reduced by simply denoting that some thread had read ownership of a lock rather than explicitly recording which thread held ownership. For some benchmarks, such as `xalan`, contended locks cause the program's execution to be serialized. For these benchmarks, more investigation is necessary to find ways to allow multiple threads to execute under MAMA while still preserving the atomicity of the program as much as possible.

### 5.4 Memory Usage

We also evaluated the memory overheads of MAMA on our benchmark suite. MAMA requires a reader-writer lock for every shared variable in the program, which can lead to high memory overheads, as shown in Figure 4. The memory overheads for MAMA range from 8x on `montecarlo` to 113x on `matmult`, as compared to the uninstrumented Java baseline (without RoadRunner). Although the array chunking optimization (Section 4) reduces `matmult`'s memory overheads to just 9.8x, it also results in serialized execution for this workload. More adaptive array chunking could alleviate this time-space tradeoff. MAMA's memory overheads could possibly be further reduced by using a more compact reader-writer lock or by avoiding the need to record all of the current readers of the lock.

### 5.5 Memory consistency

The MAMA programming model has many subtle interactions with memory consistency, as programs written with MAMA in mind will not have any notion of "critical sections" to constrain compiler and hardware reordering of memory operations. Compiler optimizations could thus reorder accesses in ways not intended by the programmer, and MAMA's safety property would apply only to the already-broken compiled program.

Memory consistency concerns can be avoided by using a compiler that preserves sequential consistency (SC), e.g., by limiting optimizations to thread-private variables [17]. The locking at every variable access introduced by the MAMA runtime system enforces SC dynamically, thus preserving an end-to-end SC guarantee.

## 6. Related Work

MAMA draws inspiration from work in several areas, including automatic parallelization, program synthesis, data-centric synchronization and concurrency bug detection.

**Automatic parallelization** schemes use static analysis [2] or hardware support [21] to extract coarse-grained parallelism from sequential code. Automatic parallelization presents a very friendly sequential programming model, but the burden of sequential semantics has limited the amount of parallelism that can be exploited. MAMA takes an alternate approach by relying on programmers to find parallelism, which eases the burden on the runtime system.

**Transactional memory** also relates to MAMA as another mechanism for providing atomicity for parallel programs [9]. On their own, transactions do not accomplish the same task as MAMA because the programmer must still denote atomic sections. However, transactions may provide an alternate mechanism for implementing MAMA, rather than using locks.

**Program synthesis** techniques have been employed to automatically synthesize concurrent programs given the desired program's specification by searching a large space of possible programs for one that satisfies the specification. This work shares MAMA's goal of making parallel programming simpler by freeing programmers from the need to specify synchronization. Prior work has shown that mutual exclusion algorithms [1] can be automatically discovered in this way. The PSketch project [22] leveraged partial programs written by a human ("sketches") to narrow the search space, allowing several concurrent data structures to be synthesized.

Other projects have synthesized the synchronization for existing, but under-synchronized, concurrent programs based on a correctness specification. Recent work in this space has shown how to synthesize minimal atomicity constraints for concurrent algorithms [25], how to insert memory fences to make concurrent data structures safe for relaxed consistency models [13], and how to add ordering constraints to nondeterministic programs to ensure they execute deterministically [19].

Because synthesis techniques rely on heavyweight verification techniques like SMT solving to discover where synchronization is required, they do not scale to larger programs. Most of the work in this space focuses on concurrent algorithms and data structures of limited scope, such as work-stealing queues. [19] infers deterministic synchronization for the JavaGrande workloads we use, though only a "program fragment" from each benchmark is considered in their analysis. While program synthesis techniques are more limited in scope than MAMA's dynamic analysis, synthesis is ultimately deeply complementary to MAMA: synthesis can find safe, highly-concurrent implementations for performance-sensitive pieces of code while MAMA provides a more general approach to handle the rest of the code base.

**Data-centric synchronization** (DCS) schemes allow synchronization to be specified for data declarations, instead of with code-centric annotations like critical sections. Since each piece of data is declared only once in a program, the number of DCS annotations is generally small. The DCS system then enforces the desired synchronization constraints at runtime.

In the Atomic Sets work [23, 24], a programmer groups variables together into atomic sets. Accesses to variables within an atomic set are guaranteed to happen atomically. A compiler analyzes the annotated program to conservatively insert lock acquires that provide the required atomicity. Lock releases are performed at the end of the function in which the lock acquire was inserted. [23] shows that Atomic Sets can support the synchronization required by many classes in the Java Collections Framework. Colorama [5] and Data Coloring [6] provide hardware support for an Atomic Sets-like programming model, and demonstrate that the performance overhead for such a scheme is modest on several full benchmark programs.

MAMA's policy of acquiring a lock for a variable before any access to the variable is inspired by these DCS schemes. However, by holding onto locks until deadlock, MAMA 1) provides a stronger safety guarantee that does not break atomicity at arbitrary function returns and 2) obviates the need to specify atomic sets since MAMA's fine-grained locking over-approximates arbitrarily coarse-grained locking (modulo deadlocks).

MAMA is also motivated by work on **concurrency bug detection**. Many schemes have been proposed to infer what sections of program execution should be atomic without relying on existing program synchronization, as that existing synchronization may be buggy. SVD [26] uses data and control dependences to infer atomic sections and then verifies that these inferred sections execute serializably; unserializable execution is often indicative of a bug. AVIO [16] identifies pairs of accesses to a single variable that execute in a non-atomic fashion as potential atomicity violation bugs. MUVI [14] generalizes AVIO to handle multi-variable atomicity violations, inferring "atomic sets" of variables and thus sets of accesses which should be performed atomically. These bug detection schemes are useful for identifying bugs in existing parallel programs, but, unlike MAMA, they neither provide safety guarantees nor function as a replacement for user-specified synchronization.

Finally, other systems have attempted to statically infer atomic sections, mainly for code that relies on transactional memory for atomicity [11]. A static approach to inferring atomic sections is complementary to MAMA's dynamic approach. In the future, we hope to combine static methods with our runtime system to improve the performance and programmability of MAMA.

## 7. Conclusions

MAMA provides a starting point for future exploration into automatically providing atomicity for parallel programs. Our results suggest that a programming model where parallel execution is safe by default and requires annotations only in liveness and performance-critical situations may be viable given additional exploration and optimization.

## References

[1] Y. Bar-David and G. Taubenfeld. Automatic Discovery of Mutual Exclusion Algorithms. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 305–305, July 2003.

[2] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Trans. on Electronic Computers*, EC-15(5):757–763, Oct. 1966.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, Oct. 2006.

[4] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, 1987.

[5] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural Support for Data-Centric Synchronization. In *HPCA*, pages 133–144, Feb. 2007.

[6] L. Ceze, C. von Praun, C. Caşcaval, P. Montesinos, and J. Torrellas. Concurrency Control with Data Coloring. In *MSPC*, pages 6–10, 2008.

[7] Cyrille Artho, Klaus Havelund, and Armin Biere. High-Level Data Races. *Journal on Software Testing, Verification & Reliability*, 13(4): 220–227, 2003.

[8] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, pages 1–8, 2010.

[9] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, May 1993.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 1st edition, 2008.

[11] S. Kempf, R. Veldema, and M. Philippsen. Compiler-Guided Identification of Critical Sections in Parallel Code. In *CC*, pages 204–223, 2013.

[12] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[13] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic Synthesis for Relaxed Memory Models. In *PLDI*, pages 429–440, June 2012.

[14] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 103–116, Oct. 2007.

[15] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, Mar. 2008.

[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, page 3748, Oct. 2006.

[17] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-preserving Compiler. In *PLDI*, pages 199–210, June 2011.

[18] L. A. Meyerovich and R. Bodik. Fast and Parallel Webpage Layout. In *WWW*, pages 711–720, Apr. 2010.

[19] V. Raychev, M. Vechev, and E. Yahav. Automatic Synthesis of Deterministic Concurrency. In *SAS*, June 2013.

[20] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *Proceedings of SC2001*, pages 8–8, Nov. 2001.

[21] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *ISCA*, pages 414–425, June 1995.

[22] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching Concurrent Data Structures. In *PLDI*, pages 136–148, June 2008.

[23] M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-oriented Language. In *POPL*, pages 334–345, Jan. 2006.

[24] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A Type System for Data-centric Synchronization. In *ECOOP*, pages 304–328, 2010.

[25] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided Synthesis of Synchronization. In *POPL*, pages 327–338, Jan. 2010.

[26] M. Xu, R. Bodík, and M. D. Hill. A Serializability Violation Detector for Shared-memory Server Programs. In *PLDI*, pages 1–14, June 2005.