

# Component-Based Lock Allocation

Richard L. Halpert      Christopher J. F. Pickett      Clark Verbrugge  
School of Computer Science, McGill University  
Montréal, Québec, Canada  
{rhalpe, cpicke, clump}@sable.mcgill.ca

## Abstract

*The allocation of lock objects to critical sections in concurrent programs affects both performance and correctness. Recent work explores automatic lock allocation, aiming primarily to minimize conflicts and maximize parallelism by allocating locks to individual critical section interferences. We investigate component-based lock allocation, which allocates locks to entire groups of interfering critical sections. Our allocator depends on a thread-based side effect analysis, and benefits from precise points-to and may happen in parallel information. Thread-local object information has a small impact, and dynamic locks do not improve significantly on static locks. We experiment with a range of small and large Java benchmarks on 2-way, 4-way, and 8-way machines, and find that a single static lock is sufficient for mtrt, that performance degrades by 10% for hsqldb, that jbb2000 becomes mostly serialized, and that for lusearch, xalan, and jbb2005, component-based lock allocation recovers the performance of the original program.*

## 1. Introduction

Achieving concurrency and scalability in parallel programs requires *lock allocation*: mapping locks to critical sections. Traditionally, lock allocation is a manual process susceptible to programmer error, and may lead to deadlock, livelock, data races, or performance degradation. Automatic lock allocation relieves programmers of this burden, and provides an implementation of *pessimistic transactions*, in which critical sections behave atomically [13, 21]. This contrasts with *optimistic transactions*, which default on lock allocation and assume a single global lock, but exploit speculative execution to achieve concurrency [15]. Recent work examines *optimal* lock allocation, and proves that the *minimum lock allocation* (MLA) optimization problem is NP-hard, and the corresponding *k-bounded lock allocation* (KLA) decision problem is NP-complete [11, 13, 30, 36, 37]. Heuristics may thus be required for practical use.

We focus on improving the quality of lock allocations possible *without* either optimal or heuristic MLA solutions. Specifically, we allocate locks on a *per-component* basis. We find that for many benchmarks this achieves the same runtime performance as the original program with a manu-

ally specified lock allocation. Our results suggest that parallel programs often exhibit simplistic concurrent behaviour, and that good solutions can be obtained using straightforward program analyses. Of course, MLA may be able to improve on the runtime performance achieved by component-based allocation. However, in this instance our analyses still play two roles: 1) they reduce the size of the MLA input problem, thereby decreasing the cost of optimal allocation; and 2) they provide “next best” solutions that may suffice when MLA is too expensive.

Our design is essentially top-down in that we first conservatively identify interfering critical sections, then use compiler analyses to refine the solution, and finally assign locks to interference graph components. This contrasts with the more bottom-up approaches used by McCloskey [21], Hicks [13], and Emmi [11], which associate locks with individual data, either manually or automatically, and then use a subset of these locks to transform critical sections. The approach used by Sreedhar and Zhang [30] is similar to ours, but moves immediately to MLA and KLA [36, 37]. It does not explore the intermediate refinement of allocating locks per graph component rather than per critical section, rejected for its limiting effect on parallelism [36].

Our lock allocator is also flexible with respect to locking disciplines. It only requires annotations in the form of synchronized blocks or methods, it permits nested synchronization, it allocates *dynamic* or per-data structure locks if possible, and it allows for use of condition variables. It does not require *locksets*, which acquire and release all locks at the beginning and end of an outer critical section, nor does it require *two-phase locking*, in which all locks are acquired before any are released. Although using multiple locks per critical section is not *a priori* undesirable, it introduces deadlock concerns, and related work constructs a total ordering among lock acquisitions to break Coffman’s circular wait condition [9]. Our allocator uses static analysis to detect when nested locking might lead to deadlock and merges components with cyclic dependences.

### 1.1. Contributions

- A *component-based* lock allocator for Java that assigns locks to groups of interfering critical sections. This depends on precise construction of a *critical section interference graph* using a *thread-based side effect* analysis.

- Automatic synchronization elimination, a trivial consequence of the approach. A component containing an isolated critical section that does not interfere with itself does not require synchronization, and results show that many such components exist.
- A *thread-local objects* analysis that improves side effect information, and a relaxed *lock-oblivious* form of *may happen in parallel* analysis for Java, which we use to prune false interference graph edges.
- Experimental data for five small and six large Java benchmarks, various analysis configurations, and 2-way, 4-way, and 8-way machines. Component-based allocation often recovers the original program performance.

## 2. Related Work

For analysis and transformation we use the Soot Java bytecode compiler framework [32]. Our allocator depends on the built-in class hierarchy analysis (CHA) [10], context-insensitive subset-based points-to analysis (Spark) [17], and may happen in parallel (MHP) analysis [18].

Naumovich *et al.* present an algorithm for computing MHP information for concurrent Java programs [25], and Li provides an implementation in Soot [18]. As a refinement to her *run-once* and *run-many* categorization of thread behaviour, we further categorize run-many threads as either *run-one-at-a-time* or *run-many-at-a-time* using a *start-join* analysis; Sura *et al.* discover similar information in their analysis of thread structure for sequentially consistent compilation [31]. Barik proposes a scalable alternative to Naumovich's analysis for Java [6], and Agarwal *et al.* extend it to support X10 [2]. The unique feature of our analysis is that it is *lock-oblivious*: it ignores the impact of mutual exclusion and thereby overestimates MHP information. This provides better scalability and is appropriate for a lock allocator that discards existing allocations.

Chang and Choi [8] and also Sălcianu and Rinard [29] present thread-sensitive points-to analyses for Java. Our points-to analysis, while thread-insensitive, provides input to a *thread-based side effect* (TBSE) analysis that models the heap using thread-local and thread-shared partitions. Our use of TBSE for interference identification might benefit from the interprocedural thread-sensitive slicing analysis for Java by Nanda and Ramesh [24]. We improve thread-sensitivity using a *thread-local objects* (TLO) analysis that identifies thread-local reads and writes inside critical sections. Ruf [27] and Aldrich *et al.* [4] find TLO information statically effective for synchronization elimination in Java, but not such that multithreaded runtime performance is significantly affected. Praun and Gross provide a related *object use graph* (OUG) and use it to check for conflicting and non-conflicting object accesses [34]. It is worth revisiting all synchronization elimination techniques in the context of lock allocation.

One problem closely related to lock allocation is static race detection. Naik *et al.* detect races in Java programs using a staged analysis that refines the set of memory access pairs potentially involved in a race until the number of false alarms is small [23]. Naik and Aiken later investigate a *conditional must not alias* analysis that concludes whether two objects are aliased from the hypothesis that two other objects are not aliased [22]. In the context of static race detection, their analysis determines whether two guarded memory regions are aliased given that the lock objects guarding them are not aliased, and reports a race if true. Pratikakis *et al.* detect races in C programs using a *consistent correlation* analysis that determines which locks are held when a thread accesses a memory location  $\rho$ , and whether there is some lock  $l$  that is always held for each access to  $\rho$  [26]. Abadi *et al.* present a type-based system for Java programs that depends on annotations to detect races [1]. A tool infers these annotations automatically, and they are input to a fixed point computation that removes the incorrect ones using a type-based race detector. Finally, a set of warnings is produced using the correct annotations. Flanagan and Freund also demonstrate that a constraint-based analysis can be used to insert synchronized operations and correct a program containing data races [12]. These techniques find memory accesses that are not properly synchronized, whereas lock allocation examines memory accesses inside critical sections and specifies objects to protect them. Our requirement that input programs be correctly *synchronizable* by our allocator is precisely defined by the Java Memory Model [19].

Table 1 compares recent work on lock allocation. McCloskey *et al.* introduce *pessimistic atomic sections* and provide a tool to convert them automatically to more efficient lock-based code [21]. They require annotations that associate locks with all shared data, in addition to annotations that identify atomic sections. Pessimistic atomic sections can be nested, and *dynamic locks* are permitted, namely dynamically allocated lock objects that guard dynamically allocated data structures. A whole-program analysis detects the use of shared data inside atomic sections, and a provably sound transformation ensures that the right locks are acquired according to a global total ordering. One limitation is that a *two-phase locking* discipline is required, such that once a lock is released for a given atomic section, no more locks can be acquired. In a related but significantly more radical technique, Vaziri *et al.* propose that *only* data be synchronized, and prove that lock operations can be safely inserted [33].

Hicks *et al.* also convert atomic sections to pessimistic transactions [13], using the same compiler analysis framework as their static race detection tool [26]. Locks are associated with abstract memory locations identified by a pointer analysis to create *locksets* that protect critical sections. Locksets restrict two-phase locking by acquiring and releasing all locks at the beginning and end of outer atomic sections. They make two improvements, first by eliminating

**Table 1. Related work on lock allocation.**

work	language	compiler analysis			locking discipline						allocation		input		results	
		pointer	TLO	MHP	data	nested	2-phase	locksets	dynamic	CVs	heuristic	MLA	small	large	AOT	RT
[21]	C	yes	no	no	yes	yes	yes	no	yes	yes	no	no	yes	yes	yes	yes
[13]	C	yes	yes	no	no	yes	yes	yes	no	no	yes	no	no	no	no	no
[30, 36, 37]	OpenMP	yes	no	yes	no	no	yes	yes	no	yes	yes	yes	yes	yes	yes	yes
[11]	C, Java	yes	no	no	no	yes	yes	no	yes	no	no	yes	yes	yes	yes	no
current	Java	yes	yes	yes	no	yes	no	no	yes	yes	yes	no	yes	yes	yes	yes

synchronization on thread-local data, and second by coalescing locks that are always acquired and released together. The first improvement is comparable to applying TLO information to the construction of an interference graph. The second is a lock minimization heuristic that eliminates redundant locking. They do not permit dynamic locks, and note that maintaining a total ordering among acquisitions with dynamic locks may require runtime support.

Sreedhar, Zhang, *et al.* develop a framework for data flow and concurrency analysis of parallel programs, and use it to allocate locks that maximize concurrency and minimize serialization overhead [30, 36, 37]. Their computation of a *concurrency relation* is comparable to MHP analysis and they apply it to data flow problems, in particular pointer analysis and lock allocation. They use this concurrency information to identify critical sections with intersecting read/write sets that are actually independent, and construct a *concurrency graph* with either an interfering or non-interfering edge between two critical section vertices. We use a straightforward translation of their concurrency graph where all edges indicate interference and non-interfering edges are removed.

They compute a *minimum lock allocation* (MLA) such that two vertices connected by an interfering edge have at least one lock in common, and two vertices connected by a non-interfering edge have no locks in common. They also provide a *k-bounded lock allocation* (KLA) algorithm for bounding the number of locks in exchange for serialization overhead, an obvious corollary of k-colouring as used by register allocation. They formulate MLA and KLA as integer linear programming (ILP) problems, and for a range of randomly generated inputs compare heuristic solutions with optimal ones provided by an industrial ILP solver. Limitations include that they disallow nested locking altogether, which impacts on the use of synchronized library code, and that they only allocate static locks. They also analyse OpenMP, and note that the interaction between aliasing and concurrency is more complicated for Java programs. However, they do provide a useful extension of data flow that considers the isolation semantics of critical sections, and describe support for condition variables and barriers in some detail, albeit for a structured subset of OpenMP. They claim that existing OpenMP programs often use *unnamed* critical sections, thus requiring a single global lock, and that the work therefore has practical importance [30].

Emmi *et al.* also examine the problem of lock allocation [11]. They build directly on McCloskey’s work by eliminating the requirement that annotations protect shared data. Like Zhang *et al.* they depend on ILP for allocation, clearly explaining how to set up MLA and KLA for 0–1 ILP while accounting for various refinements. Importantly, they find that optimal solutions are tractable for McCloskey’s larger AOLServer benchmark. They consider dynamic locks in some detail, avoiding deadlock by using an *accessed-before relation* derived from temporal analysis of critical sections, and favouring dynamic locks over static locks during allocation. They also note that more precise compiler analysis is complementary to their work.

Finally, we see lock allocation in general as complementary to optimistic concurrency and transactional memory, an active field of research [15]. Lock allocation could be used to reduce the overhead incurred by optimistic concurrency in a system that executes *uncontended* critical or atomic sections non-speculatively and without overhead. Although this model differs from most transactional memory proposals, which incur overhead for every transaction, Martínez and Torrellas do propose hardware for such a system [20], and Welc *et al.* demonstrate a software implementation in a Java virtual machine [35].

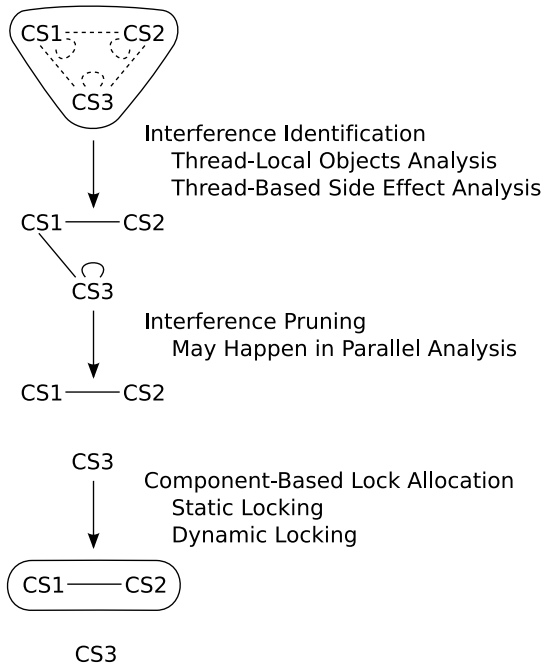
### 3. Design

Our lock allocator accepts compiled Java programs consisting of .class files. Input programs must not use volatiles, native code, or `java.util.concurrent` for thread synchronization, and must contain critical sections protecting all accesses to thread-shared state. It must be possible to specify a lock allocation that results in correct synchronization as defined by the Java Memory Model [19]. Any original lock allocation is discarded, and the lock allocator chooses locks to produce a race-free, deadlock-free program. This allows for newly written software to ignore the lock allocation problem altogether, and for existing programs to benefit from automatic correction of unsafe allocations. Existing programs containing fine-grained manual allocations also provide a basis for experimental evaluation of lock allocation strategies. Both classes of program undergo unnecessary synchronization elimination.

Any form of `Object.wait()`, `Object.notify()`, or `Object.notifyAll()` is safe, provided the input program retries condition variables after waking up from a call to

`wait()`. After lock allocation, these calls are redirected to the lock object protecting the immediately enclosing critical section. Additionally, calls to `notify()` are replaced with calls to `notifyAll()`, which guarantees that wakeup notifications reach their intended thread without being unsafely intercepted by some other waiting thread. Nested use of `wait()` and `notify()` may have deadlock implications, as described in Section 3.6.

We represent programs with a *critical section interference graph*  $G = (V, E)$ , where each  $v \in V$  is a critical section and each  $e \in E$  is an *interference*. An interference edge between two critical sections indicates that they might conflict at runtime, and a self loop indicates that two or more threads compete for the same critical section. Our initial approximation is a fully connected graph, which we refine through a series of compiler analyses.



**Figure 1. Analysis pipeline.**

An overview of our analysis pipeline and an example allocation are given in Figure 1. Initially, the input program contains a set of critical sections that we assume are safely protected by a singleton lock object; in the figure, these are critical sections CS1, CS2, and CS3. Interference information is computed by a *thread-based side effect* (TBSE) analysis, which in turn employs a *thread-local objects* (TLO) analysis, both of which depend on points-to information and a call graph. In the example, this reveals the CS1–CS2, CS1–CS3, and CS3–CS3 edges. False edges in the resultant interference graph are pruned by a *may happen in parallel* (MHP) analysis; consider the false CS1–CS3 edge and CS3–CS3 self loop in the example.

This yields a set of *locked components* which contain interfering critical sections, CS1–CS2 in the example, and a set of *unlocked components* which are isolated critical sections without self loops, CS3 in the example. *Component-based lock allocation* proceeds to allocate a lock to each locked component, locking only CS1–CS2 in the example and removing unnecessary synchronization from CS3. Locks may be *static*, instantiated once per program run, or *dynamic*, instantiated once per protected data structure. The result is a correctly synchronized program with a set of locks that is less conservative than a singleton allocation.

### 3.1. Information Flow Analysis

Our lock allocator uses a flow-insensitive, context-sensitive, interprocedural *information flow analysis* (IFA) as the basis of the thread-local objects analysis described in Section 3.2. Given a pair of memory locations  $a$  and  $b$ , IFA approximates whether the value stored in  $a$  is derived from the value stored in  $b$ . This analysis only considers *explicit* information flow resulting from direct assignment or arithmetic operations; a more accurate analysis would also consider *implicit* control-based information flow [3].

Given a method to analyse, IFA generates an *information flow graph* and an *information flow summary*. The graph nodes represent all values manipulated by the method, namely parameters, locals, fields, statics, and the return value. Every assignment or return statement generates an edge in the graph. The summary is derived from the graph by removing local variables and collapsing strongly connected components. It thus approximates all publicly accessible values manipulated by the method, namely parameters, fields, statics, and the return value. Our analysis distinguishes between syntactically different values but is type-based and ignores points-to information.

At a callsite, the summaries of all target methods are combined. This combined summary is merged with the current graph by connecting summary parameters to callsite arguments, the summary return value to the callsite return value, and the summary `this` object and fields to the callsite receiver local. If no summary exists for some target method, then a graph and summary are recursively constructed. A simple conservative summary is used when back edges in the recursion are encountered, and internal library calls also use a conservative summary to improve runtime.

### 3.2. Thread-Local Objects Analysis

After initial points-to analysis and call graph construction, our lock allocator performs a *thread-local objects* (TLO) analysis that serves to improve the precision of the thread-based side effect analysis described in Section 3.3. TLO classifies all fields as either *thread-local* or *thread-shared*, where any field that may be accessed by more than one thread is thread-shared and all others are thread-local. TLO uses IFA as described in Section 3.1 to propagate this

information throughout the program. TLO information contributes to interference graph quality because reads from and writes to thread-local fields do not interfere.

TLO accepts a set of thread classes  $T$  that implement the `Runnable` interface, and analyses each  $t \in T$  independently to determine which fields in the program may hold thread-shared values. Initially, any field of  $t$  accessed by a method external to  $t$  is classified as thread-shared, and all other fields are classified as thread-local. The parameters of each method in  $t$  are classified as thread-local unless the method is called externally, in which case they are thread-shared. After this initial classification, TLO queries IFA to obtain an information flow summary for each method in  $t$ . Whenever a summary indicates that a thread-shared value flows to a thread-local field, the classification of that field is changed to thread-shared. This propagation repeats until a fixed point is reached.

Next, a *locality context* is created for each method  $m$  in the call graph. A locality context contains a thread-shared or thread-local classification for each field and parameter in  $m$ . The previous classification of fields and parameters for each  $m$  in  $t$  provides an initial set of locality contexts. Shared values are propagated from locality contexts to callsites using the information flow graph of  $m$ , and then merged with the locality contexts of all target methods. This interprocedural propagation continues until a fixed point is reached.

When queried about some value  $u$  for any  $m$ , TLO starts from the locality context of  $m$  and traverses its information flow graph to find all information sources of  $u$ . It reports that  $u$  is thread-shared unless all sources are thread-local and thus  $u$  is thread-local in each  $t$  that calls  $m$ .

### 3.3. Thread-Based Side Effect Analysis

We extend Lhoták's side effect analysis as implemented in Soot [17] to a *thread-based side effect* (TBSE) analysis. TBSE computes sets of (*field*, *object*) pairs that may be read or written by individual critical sections, and these sets are used to create the interference graph. TBSE is a points-to analysis client, and incorporates TLO as described in Section 3.2, a critical section nesting model, and special handling of calls to library methods and static initializers.

Soot computes the side effects of statements using a simple set of data flow analyses and the output of its points-to analysis. TBSE alters these analyses to ignore side effects involving thread-local objects. Our nesting model allows inner locks to be released independently of outer locks, and accordingly TBSE also excludes the side effects of inner critical sections.

For application calls to library interface methods, TBSE assumes full side effects on receiver and parameter objects. However, the side effects of internal library calls are excluded because deep library call chains and side effects involving static fields both impact significantly on precision. TBSE also excludes the side effects of static initializers be-

cause they impact on precision. Although ignoring static initializers is unsound, they could safely be forced to execute before the code affected by our transformations.

Our allocator constructs an interference graph using the information provided by TBSE. A vertex  $v$  is created for every critical section, and an interference edge  $e$  is inserted between every pair of vertices  $v_i$  and  $v_j$  for which  $(read(v_i) \cap write(v_j)) \cup (write(v_i) \cap read(v_j)) \cup (write(v_i) \cap write(v_j)) \neq \emptyset$ . This union of intersections contains all of the data dependences between two critical sections. When non-empty, it is stored as the *contributing read/write set* of  $e$ .

### 3.4. May Happen in Parallel Analysis

After construction of the interference graph using thread-based side effect information, a *may happen in parallel* (MHP) analysis prunes false positive edges. Our implementation is a context-insensitive and lock-oblivious adaptation of Li's MHP analysis in Soot [18]. It first uses a *run-once*, *run-many* analysis to categorize thread classes as *run-once* or *run-many*. It then uses a *start-join* analysis to further categorize the run-many threads as *run-one-at-a-time* or *run-many-at-a-time*.

Our MHP analysis is *lock-oblivious* in that it ignores object synchronization, method synchronization, and calls to any form of `wait()` or `notify()`. This allows our lock allocator to determine which interfering critical sections require locks to *prevent* parallel execution, and contributes to unnecessary synchronization elimination. This simplification reduces the MHP analysis problem to first identifying and categorizing threads, and then for each thread determining the set of reachable methods.

The *run-once*, *run-many* analysis iterates over the call graph, and for each method marks each statement in the body of the method and the method itself as either *run-once* or *run-many* [18]. The initial approximation is *run-once*. Statements inside loops and inside *run-many* methods are categorized as *run-many*. Methods with incoming edges from multiple callsites and methods called from *run-many* statements are also categorized as *run-many*. These complementary analyses alternate until a fixed point is reached.

Next, invocation statements  $s$  calling `Thread.start()` are used to identify and categorize distinct thread classes  $t \in T$ . If  $s$  is *run-once*, then  $t$  is *run-once*. If  $s$  is *run-many* but the points-to set of the receiver contains one object and the allocation site of that object is *run-once*, then a singleton `Thread` object reaches  $s$  and  $t$  is also *run-once*. Otherwise,  $t$  is *run-many*. As an implementation detail,  $T$  is input to the TLO analysis in Section 3.2.

In the case of a *run-many* thread class  $t$ , a *start-join* analysis searches the method  $m$  containing the `start()` invocation  $s$  that identified  $t$  for invocation statements  $j$  calling `Thread.join()`. A local must alias analysis first filters out any  $j$  that is not guaranteed to join  $t$ . A post-dominator

analysis then filters out any  $j$  that does not post-dominate  $s$ . If some  $j$  exists after filtering, and  $m$  is not reentrant and does not happen in parallel with itself, then  $t$  is labelled *run-one-at-a-time*, otherwise *run-many-at-a-time*. If  $m$  is later found to happen in parallel with itself, then  $t$  is reclassified as *run-many-at-a-time*. Our run-one-at-a-time classification is comparable to the *single thread constraint* identified by Sura *et al.* [31].

The MHP analysis finally reports pairs of methods reachable from two or more different threads as may happen in parallel. For this classification, each run-many-at-a-time thread class is treated as two threads, and each other thread class as one. This information is used to prune edges between critical sections in the interference graph whose containing methods may not happen in parallel.

The above treatment differs in several ways from Li's work [18]. Her lock-sensitive analysis creates a whole-program control flow graph in order to analyse synchronization and wait/notify statements correctly. Our lock-oblivious analysis works more quickly than her implementation because it does not need to create a whole-program CFG. It also works for a wider variety of programs, because her construction of a whole-program CFG requires that every virtual method call be statically resolvable.

### 3.5. Lock Allocation

Lock allocation begins after the interference graph has been pruned using MHP information. Our allocator assigns locks to components at three different granularities:

- Singleton: A single static lock shared by all components.
- Static: A different static lock for each locked component.
- Dynamic: A different dynamic lock for each locked component, reverting to static allocation if necessary.

All three granularities remove all previously existing uses of lock objects by critical sections, and guarantee that every critical section in a locked component is protected by some new appropriate lock object. Synchronized methods are replaced with unsynchronized wrapper methods around synchronized blocks, and calls to `wait()` and `notify()` are redirected to the new lock object for the immediately enclosing critical section. We insert `public static Object` fields to provide lock objects for the singleton and static granularities.

Singleton allocation is trivial: the interference graph is ignored, and the same static lock is assigned to every critical section in the program, including those in unlocked components. The composition of our analysis pipeline has no bearing on this naïve allocation. Static allocation *does* depend on the interference graph, but the actual process is also straightforward: each locked component is assigned a different static lock, and synchronization is removed from unlocked components.

Dynamic allocation builds on static allocation. For a given locked component, each critical section must be *dynamically lockable*: it must have an object available on entry through which all reads and writes of the critical section are accessible. To determine if such an object exists, the allocator examines all statements in the critical section that generate elements of the contributing read/write sets of connected interference edges. Static field reads and writes immediately disqualify the critical section. For each non-static read and write, the accessed object is added to a set of objects that require locking. Any object that aliases all objects in the set is a suitable dynamic lock for the critical section. If there exists a dynamic lock common to all critical sections then it can safely protect the entire component, otherwise a static lock is required.

### 3.6. Deadlock Avoidance

The analyses discussed so far have focused on freedom from data races. However, another necessary condition for correct synchronization is the absence of deadlock, which can be ensured by breaking cyclic lock acquisitions [9]. Our lock allocator abides by a policy of minimal perturbation when performing deadlock detection and correction. It first allows an initial lock allocation to proceed without regard to deadlock, then detects violations of the partial ordering of acquisitions implied by critical section nesting, and finally corrects deadlock by adding *deadlock avoidance edges* to the interference graph and reallocating the locks.

During deadlock detection, the allocator examines all pairs of critical sections. If a pair is nested, it records the ordering of their locks and adds the *outer* critical section to a set of critical sections associated with that ordering. Any new ordering is then compared to all previous orderings. If a violation is found, it indicates potential deadlock, and a deadlock avoidance edge is inserted between the outer critical section of the new ordering and every critical section associated with the violated ordering. Lock allocation restarts after all pairs are examined if any edges were added.

A *deadlock avoidance edge* is treated like an interference edge for a static field, and must be protected by a static lock. This effectively merges any graph components that generate a cycle. This technique is suitable for component-based allocation, but we also intend to use it for fine-grained allocations involving multiple locks per critical section.

Our deadlock detection algorithm handles typical nested mutual exclusion deadlock. However, it does not handle nested wait/notify deadlock, which occurs if a thread waits on one lock while holding others, and the locks it holds prevent other threads from reaching the necessary call to `notifyAll()`. This type of deadlock can be avoided by inserting an edge between the outer critical sections preventing access to `notifyAll()` and the inner critical section containing the call to `wait()`. None of the benchmarks we experiment with exhibit this behavior.

**Table 2. Benchmarks.**

name	critical sections	description	source
pcmab	2	25 producers and 25 consumers connect via an aspect	Sable
roller	6	7 passenger threads compete for 7 seats in 1 roller coaster thread	Sable
traffic	24	1 car thread and 1 driver thread navigate together around a rotary	Sable
bank	8	8 threads transfer funds between two accounts	Doug Lea
sync	16	8 threads increment a counter, synchronized on an object or a method	Java Grande
mtrt	6	2 threads render a raytraced image	SPEC
hsqldb	269	20 threads run transactions against a banking application	DaCapo
lusearch	88	32 threads search a large index for 3500 words	DaCapo
xalan	73	8 threads perform XSL transforms	DaCapo
jbb2000	241	$N_{\text{peak}}$ through $2 \times N_{\text{peak}}$ threads perform middleware operations	SPEC
jbb2005	187	$\text{CPU}_{\text{max}}$ through $2 \times \text{CPU}_{\text{max}}$ threads perform middleware operations	SPEC

## 4. Experimental Evaluation

We use three different x86\_64 SMP machines for experimental evaluation. Our 2-way machine has one dual-core 2.0 GHz AMD Athlon 64 X2 3800+ processor, runs Ubuntu 6.06 with a Linux 2.6.15 kernel, and uses Sun's 64-bit 1.5.0\_06 JVM. Our 4-way machine has four 1.8 GHz AMD Opteron 844 processors, runs Ubuntu 7.04 with a Linux 2.6.20 kernel, and uses Sun's 64-bit 1.5.0\_06 JVM. Our 8-way machine has four dual-core 2.0 GHz AMD Opteron 870 processors and uses Sun's 64-bit 1.5.0\_12 JVM. For *jbb2000* it runs RHEL 4.0 with a Linux 2.6.9 kernel, and for all other benchmarks it runs 64-bit Windows Server 2003 Enterprise Edition. These environments are fairly heterogeneous and our results indicate both system-dependent behaviour and broader trends. Complementary experiments would investigate scalability on the same machine by disabling processors or using CPU affinity masks. We experimented briefly with a 16-way POWER5 machine running AIX 5.3, but found brittleness in the interaction of Soot, DaCapo, and IBM's 1.5.0 JVM.

We use eleven different multithreaded benchmarks from various sources, as shown in Table 2. The first five are smaller and contention heavy. The three Sable benchmarks were developed internally: *pcmab* stresses thread scheduling, producer/consumer thread coordination, and AspectJ 1.5 performance, and *roller* and *traffic* are both simulations of high contention situations. *bank* is a micro benchmark derived from Lea's ATApplet [16], in which each thread makes a random account transaction and then calls `Thread.yield()` one million times. For these four benchmarks we measure the time of one program run. *sync* is the only benchmark from the Java Grande Forum multithreaded benchmark suite that exercises lock-based synchronization. It reports two throughput metrics, object synchronizations per second and method synchronizations per second, and we measure them individually.

The next six are larger Java benchmarking standards. We include *mtrt* from SPEC JVM98, the only benchmark in the suite with a multithreaded workload, and measure

the time of the first iteration at input size 100. We use all three benchmarks with multithreaded workloads from version 2006-10-MR2 of DaCapo [7], namely *hsqldb*, *lusearch*, and *xalan*, and measure the time of the first iteration at the default input size. We use the `-xdeps` packaging of DaCapo suitable for Soot transformation, and note that extra care is required to analyse and run all application classes properly: *hsqldb* loads its main driver by reflection, and *xalan* is also contained in the JVM class libraries. Finally, we include SPEC JBB2000 and JBB2005, which run multiple fixed-length iterations internally, each increasing the number of threads by one. For *jbb2000*, we use the official metric that averages all points from  $N_{\text{peak}}$ , the iteration with peak throughput and  $N$  threads, through  $2 \times N_{\text{peak}}$ . For *jbb2005*, we use the official metric that averages all points from  $\text{CPU}_{\text{max}}$  through  $2 \times \text{CPU}_{\text{max}}$  threads.

We perform thirteen different experiments on each benchmark. An *unprocessed* experiment simply measures raw performance without transformation. A *control* experiment processes all class files with Soot but does not perform lock allocation. There remain eleven allocation experiments: *singleton*, a single static lock, and then both static (*sta*) and dynamic (*dyn*) variants of five different analysis pipeline configurations. These configurations are: 1) *cha*, which uses class hierarchy analysis (CHA) [10] to determine points-to information, and then our thread-based side-effect (TBSE) analysis; 2) *spk*, which uses Spark [17] to determine points-to information, and then TBSE; 3) *spk-tlo*, which is *spk* with thread-local objects (TLO) analysis enabled; 4) *spk-mhp*, which is *spk* with may happen in parallel (MHP) analysis enabled; and 5) *spk-tlo-mhp*, which is *spk* with both TLO and MHP enabled. CHA is included to test whether the more precise points-to information provided by Spark has an effect on allocations using TBSE alone. The TLO and MHP analyses depend on Spark, and we do not perform *cha-tlo*, *cha-mhp*, or *cha-tlo-mhp* experiments.

All benchmarks are transformed twelve times with Soot, once for *control*, and eleven times for the allocation experiments. We use a 2.0 GHz machine for the transformations with a 1 GB heap, except for *spk-tlo-\** on *hsqldb* which re-

quire an 8 GB heap, and show results in Table 3. For those experiments where the lock allocation differs, we also measure runtime performance. Each benchmark runs 50 times per differing allocation and per machine with a 1 GB heap, except for *jbb2000* and *jbb2005* which run only 5 times due to both their excessive length and relative stability. The *singleton* experiment for *jbb2005* requires that we disable bytecode verification, a peculiar transformation artifact that we do not investigate. We find a significant performance difference between *unprocessed* and *control*, and compare the performance of allocation experiments to *control* to exclude unrelated factors. Figure 2 shows normalized slowdowns against *control*, as well as 95% confidence intervals for the differences between means. These intervals are plotted as offsets against the *control* mean, and when they cross 1.0 the difference between means is not statistically significant. For *sync*, *jbb2000*, and *jbb2005*, we report changes in inverse throughput rather than execution time.

Our source code is available in revision 2995 of Soot [32]. In the interest of encouraging external repeatability we are also making available our benchmarks, run scripts, raw data, and data processing scripts.

#### 4.1. Static Results

Table 3 shows the effect of different analysis pipeline configurations on interference graph construction and lock allocation. We report only the dynamic allocation for each configuration, as the static allocation is identical save that all dynamic locks become static. We include a single measure of analysis time on a 2.0 GHz machine, and find it acceptable in all cases except one, generally under 2 minutes for small benchmarks and 5 minutes for large ones. The exception is for *spk-tlo-\** on *hsqldb*; some feature of our TLO algorithm or implementation does not scale well to this problem.

The *graph characteristics* column of Table 3 illustrates the effect that introducing new pipeline components has on interference graph evolution.  $|V|$  is constant for each benchmark, corresponding to the number of critical sections in Table 2, and indicates the size of the interference graph construction problem.  $|E|$  is the number of edges in the final interference graph, and  $|E|/|V|$  is *graph density*, ranging from 0 to  $|V|^2$ , a suitable graph quality metric. The *weight* of any edge  $e \in E$  is the number of fields involved in its contributing read/write set, and  $\sum weight(e)$  indicates the size of the dynamic lock allocation problem. Most benchmarks exhibit reduced graph density and summed edge weight as more sophisticated analyses are introduced, although not necessarily at every stage. Switching from CHA to Spark has the largest effect, and including MHP has the second largest effect.

The *graph components* column illustrates the effect that interference graph quality has on lock allocation. A connected set of vertices is a *locked* component, and an iso-

lated vertex with no self loop is an *unlocked* component. The allocator assigns one lock to each locked component, either *static* or *dynamic*. A set of  $N$  locked components  $C$  is shown as  $N:[|C_1| \dots |C_N|]$ . For example, the *spk-\** configurations of *roller* have three statically locked components with two critical sections each, shown as 3:[2 2 2].

Empirically, as graph density decreases, the *total* number of graph components increases monotonically towards  $|V|$ : locked components split into isolated locked and unlocked sub-components as internal edges are removed. Although lower graph density and summed edge weight should intuitively allow for the number of dynamic locks to increase, in practice these compete with unlocked components. For the seven allocations with dynamic locks, only *traffic* and *sync* increase dynamic lock count with more sophisticated analyses; *bank* remains constant, *hsqldb* decreases from seven to two, and *lusearch*, *jbb2000*, and *jbb2005* all decrease to zero. Over eleven benchmarks, Spark has an impact on the lock allocations of nine, MHP on eight, and TLO only on two, *hsqldb* and *jbb2005*.

Nine out of eleven benchmarks have unlocked components, which often account for a significant fraction of both  $|V|$  and the total number of components. These are isolated critical sections that either lie in dead code, interfere only with critical sections with which they may not happen in parallel, or do not read or write thread-shared data. In the second and third cases, synchronization elimination will reduce locking overhead, but the performance improvement is generally expected to be negligible. This is due to optimizations in all production and many research JVMs for uncontended and unshared locks [5, 14, 28]. The primary benefit of synchronization elimination is a more fine-grained allocation for the remaining locked components.

#### 4.2. Dynamic Results

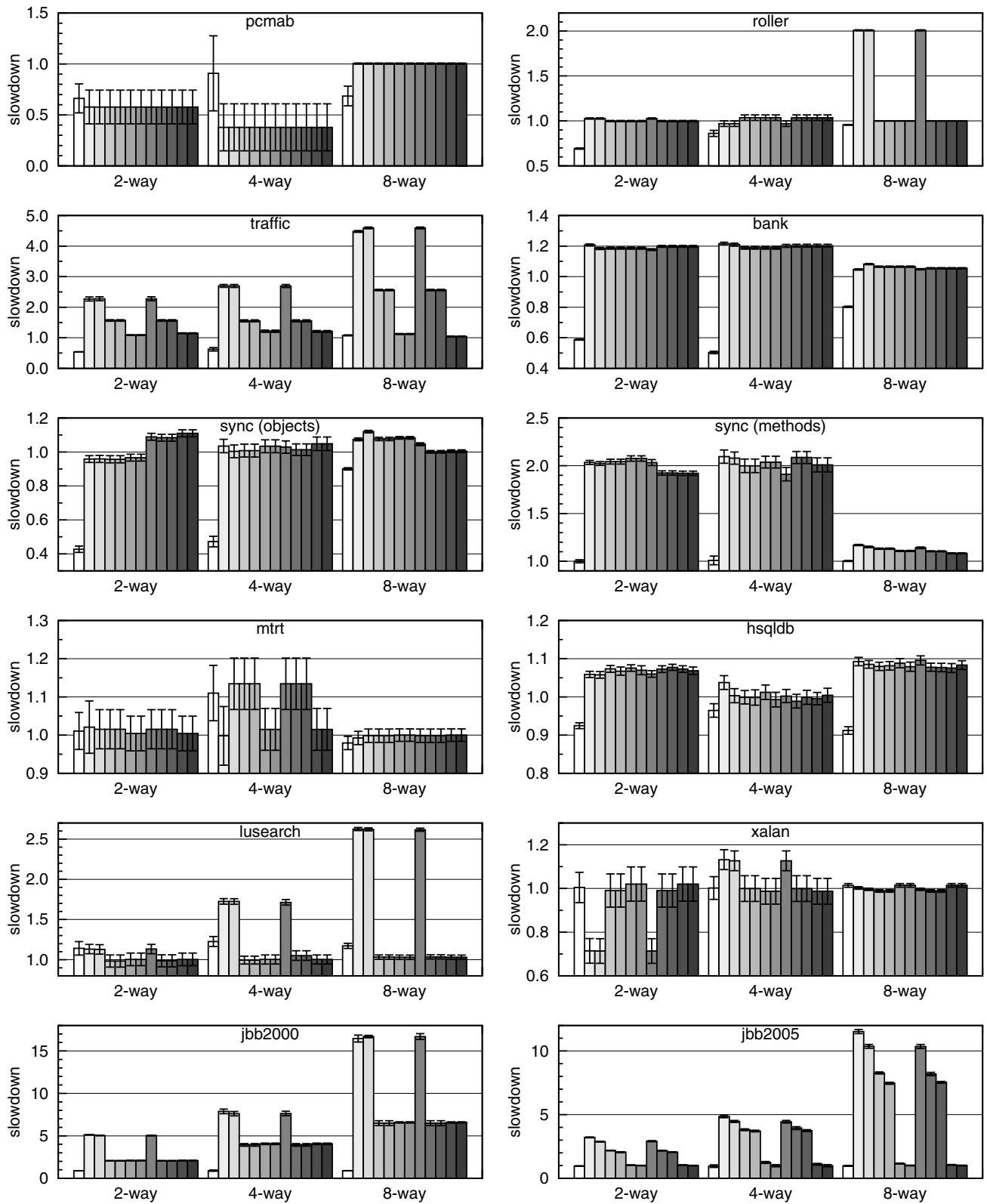
Figure 2 presents runtime performance data for all combinations of benchmark, experiment, and machine. We now discuss each benchmark in detail.

*pcmab*. All allocations are equivalent to *singleton*. The transformed version outperforms *control* on 2-way and 4-way machines due to a starvation problem in the original program, and variation in *control* contributes most of the large confidence interval. This is an example of coarse-grained locking actually improving on fine-grained locking that was not created artificially for this purpose. On an 8-way machine, the starvation problem is not present, which may be due to scheduler fairness differences or decreased contention for processor time.

*roller*. Performance is usually unaffected, even for the *singleton* and *cha-\** allocations where a single static lock protects all passenger seats. The portion of the workload that becomes serialized is small, and the portion that follows is already serialized in the original program. However, *singleton* and *cha-\** do not scale to an 8-way machine,

**Table 3. Effect of analysis pipeline configuration on interference graph and lock allocation.**

benchmark	analysis pipeline				graph characteristics				graph components			
	points-to	TLO	MHP	time (s)	V	E	$\frac{ E }{ V }$	$\sum weight(e)$	total	locked		unlocked
										static	dynamic	
pcmab	CHA			83	2	4	2.0	17	1	1:[2]	0	0
	SPK			77	2	4	2.0	17	1	1:[2]	0	0
	SPK	X		78	2	4	2.0	16	1	1:[2]	0	0
	SPK	X	X	76	2	4	2.0	16	1	1:[2]	0	0
roller	CHA			82	6	24	4.0	450	1	1:[6]	0	0
	SPK			76	6	12	2.0	230	3	3:[2 2 2]	0	0
	SPK	X		81	6	12	2.0	230	3	3:[2 2 2]	0	0
	SPK	X	X	78	6	10	1.7	174	3	3:[2 2 2]	0	0
traffic	CHA			87	24	147	6.1	486	4	1:[21]	0	3
	SPK			82	24	89	3.7	174	9	4:[1 2 3 13]	0	5
	SPK	X		85	24	81	3.4	163	9	4:[1 2 3 13]	0	5
	SPK	X	X	84	24	32	1.3	76	10	3:[2 3 4]	1:[9]	6
bank	CHA			79	8	29	3.6	29	3	1:[5]	1:[2]	1
	SPK			78	8	12	1.5	12	4	1:[4]	1:[2]	2
	SPK	X		75	8	12	1.5	12	4	1:[4]	1:[2]	2
	SPK	X	X	77	8	12	1.5	12	4	1:[4]	1:[2]	2
sync	CHA			81	16	68	4.2	1033	8	1:[8]	1:[2]	6
	SPK			79	16	66	4.1	1031	9	1:[8]	2:[1 1]	6
	SPK	X		82	16	66	4.1	1031	9	1:[8]	2:[1 1]	6
	SPK	X	X	76	16	2	.13	2	16	0	2:[1 1]	14
mrt	CHA			259	6	16	2.7	444	3	1:[4]	0	2
	SPK			113	6	16	2.7	411	3	1:[4]	0	2
	SPK	X		119	6	16	2.7	343	3	1:[4]	0	2
	SPK	X	X	114	6	1	.17	93	6	1:[1]	0	5
hsqldb	CHA			821	269	15030	56	703181	79	4:[1 1 2 190]	7:[7×1]	68
	SPK			287	269	6555	24	294011	119	4:[1 1 2 150]	2:[1 1]	113
	SPK	X		31032	269	6441	24	286845	123	4:[1 1 2 146]	2:[1 1]	117
	SPK	X	X	21780	269	5453	20	244225	128	2:[2 141]	2:[1 1]	124
lusearch	CHA			142	88	1193	14	32526	39	3:[1 1 49]	1:[2]	35
	SPK			130	88	86	.98	1307	74	5:[1 1 1 6 10]	1:[1]	68
	SPK	X		157	88	86	.98	1252	74	5:[1 1 1 6 10]	1:[1]	68
	SPK	X	X	160	88	73	.83	982	74	3:[1 6 10]	0	71
xalan	CHA			395	73	1195	16	136813	30	3:[1 1 44]	0	27
	SPK			194	73	6	.08	26	72	3:[1 1 2]	0	69
	SPK	X		200	73	6	.08	26	72	3:[1 1 2]	0	69
	SPK	X	X	198	73	3	.04	18	72	1:[2]	0	71
jbb2000	CHA			276	241	7482	31	81649	74	1:[167]	1:[2]	72
	SPK			149	241	163	.68	4737	225	3:[2 4 13]	0	222
	SPK	X		298	241	162	.67	4655	225	3:[2 4 13]	0	222
	SPK	X	X	296	241	150	.62	4533	227	2:[3 13]	0	225
jbb2005	CHA			276	187	3225	17	29376	51	1:[129]	11:[3×1 8×2]	39
	SPK			173	187	562	3.0	4150	110	3:[2 6 69]	11:[8×1 3×2]	96
	SPK	X		298	187	450	2.4	3906	132	3:[1 2 55]	9:[9×1]	120
	SPK	X	X	295	187	21	.11	27	177	3:[2 3 8]	0	174



**Figure 2. Runtime performance.** For each benchmark and machine, left to right bars are: 1) *unprocessed*; 2) *singleton*; 3–7) static *cha*, *spk*, *spk-tlo*, *spk-mhp*, and *spk-tlo-mhp*; 8–12) dynamic variants of 3–7.

possibly due to increased processor resources and therefore contention between threads.

*traffic*. 4-way *unprocessed* livelocks or nearly livelocks on startup for approximately 30% of all runs, and we discard those measurements; none of the other experiments experience this problem. Moving from CHA to Spark and including MHP both have a significant effect on performance, as suggested by static allocation data. MHP exposes a dynamic lock, which degrades 2-way performance but improves 8-way performance. Although there are only two threads, non-optimal 8-way performance is two-fold worse than non-optimal 2-way performance.

*bank*. This is the only benchmark where deadlock is detected. All allocations degrade performance, by roughly 20% on 2-way and 4-way machines, and 5% on an 8-way machine. In some contexts, these may be tolerable overheads. The slowdown is due to the lack of a suitable dynamic lock for the static component, even though one exists; this component is created by a deadlock avoidance edge. On an 8-way machine there is one thread per processor, which possibly contributes to the improved performance.

*sync*. For object synchronization, the throughput of all allocations is close to *control*. 2-way dynamic locking degrades performance but 8-way dynamic locking improves it; we note that *traffic* and *bank* exhibit similar behaviour. For method synchronization, our allocator actually converts the method under test to use object synchronization, and performance degrades on 2-way and 4-way machines, presumably because object synchronization is more expensive in the JVM. Although 8-way method synchronization is acceptable, we observe that the raw throughput is on the order of 20 to 30 times worse for all experiments.

*mtrt*. The *spk*-\**mhp*-\* allocations are functionally equivalent to the *control* program, except that synchronization is removed where possible. This benchmark is embarrassingly parallel and performance is independent of allocation on 2-way and 8-way machines. On the 4-way machine, the intermediate allocations without MHP reduce performance, but *singleton* allocation has no effect. TBSE alone only identifies two unlocked components, and performance changes may be due to JIT compilation idiosyncrasies.

*hsqldb*. Static analysis has difficulty, but performance is not significantly different between lock allocations, all of which have one large component and exhibit 5–10% slowdowns on 2-way and 8-way machines. Despite that *hsqldb* runs 20 threads, the raw execution speed does not scale from two to eight processors, suggesting an existing serialization.

*lusearch*. This application is embarrassingly parallel, and *singleton* and *cha*-\* conservatively limit parallelism where the original program has no contention. However, the more precise points-to information in *spk*-\* allows TBSE to regain the original performance.

*xalan*. This application is also embarrassingly parallel, as threads simply remove documents from a queue for processing, but results indicate subtle concurrency issues. On a

2-way machine, the *singleton* and *cha*-\* allocations outperform both *control* and *unprocessed* by 30%; however, this change is inverted on a 4-way machine, and on an 8-way machine they have no effect. In all cases, *spk*-\* matches original performance. Like *pcmab*, *xalan* has a built-in performance problem that coarse-grained lock allocation can correct, at least on 2-way machines.

*jbb2000*. *singleton* does not scale, generally degrading performance by a factor of  $2 \times \text{CPU}_{\max}$ . *spk*-\* roughly halve the degradation, but only such that throughput is 80–100% serialized. MHP improves allocations only marginally, and there is no effect on runtime behaviour.

*jbb2005*. Like *jbb2000*, *singleton* does not scale. Unlike *jbb2000*, each new addition to the analysis pipeline contributes until the original performance is regained. This is the only benchmark where TLO has an effect, although the effect of MHP is much greater. Dynamic locking improves *spk-mhp-dyn* on 4-way and 8-way machines, but *spk-tlo-mhp*-\* outperform it. Our hypothesis as to why *jbb2005* succeeds but *jbb2000* fails is that *jbb2005* is a refactored version of *jbb2000* that is more idiomatic of Java.

## 5. Conclusions and Future Work

This work presents a complete system for component-based lock allocation and experimentally identifies sufficient locking strategies. A single static lock only guarantees equal or better performance for *pcmab* and *mtrt*, although it does improve *xalan* by 30% on 2-way machines. Thread-based side effect analysis alone guarantees equal or better performance for *pcmab*, *roller*, *lusearch*, and *xalan*, but not for *mtrt*. It also improves *traffic*, *jbb2000*, and *jbb2005*. However, it depends on precise context-insensitive points-to information; CHA-based analyses are too imprecise, and this is predictable statically.

The inclusion of may happen in parallel information guarantees the performance of *mtrt*, and further improves *traffic* and *jbb2005*. However, this is not predictable statically, as MHP also affects the allocations of five other benchmarks. MHP always removes interference edges between two critical sections that may not happen in parallel. Therefore, it only affects performance when the removal also splits a larger component, and both sub-components are significantly contended at the same time. Thread-local object information guarantees the performance of *jbb2005*, but only in combination with MHP. Dynamic locking offers no improvement when other strategies are sufficient. For *traffic*, *bank*, and *sync* (*objects*), it affects performance negatively for small machines and positively for large ones.

*traffic*, *bank*, *sync*, *hsqldb*, and *jbb2000* are good benchmarks for future work, although only *jbb2000* exhibits intolerable overheads. We will consider heuristic and optimal MLA and KLA, converting object synchronization to method synchronization, and context-sensitive points-to analysis as potential improvements to our approach.

## Acknowledgements

This research is funded by NSERC and IBM CAS Toronto. We would like to thank the anonymous reviewers for their comments, and IBM for their 8-way hardware.

## References

- [1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28(2):207–255, Mar. 2006.
- [2] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP'07*, pages 183–193, Mar. 2007.
- [3] A. Ahmedani. Information flow in a Java intermediate language. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, Aug. 2006.
- [4] J. Aldrich, E. G. Sireer, C. Chambers, and S. J. Eggers. Comprehensive synchronization elimination for Java. *Sci. Comput. Program.*, 47(2-3):91–120, May 2003.
- [5] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI'98*, pages 258–268, June 1998.
- [6] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *LCPC'05*, volume 4339 of *LNCS*, pages 152–169, Oct. 2005.
- [7] 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'06*, pages 169–190, Oct. 2006.
- [8] B.-M. Chang and J.-D. Choi. Thread-sensitive points-to analysis for multithreaded Java programs. In *ISCI'04*, volume 3280 of *LNCS*, pages 945–954, Oct. 2004.
- [9] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *CSUR*, 3(2):67–78, June 1971.
- [10] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95*, volume 952 of *LNCS*, pages 77–101, Aug. 1995.
- [11] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL'07*, pages 291–296, Jan. 2007.
- [12] C. Flanagan and S. N. Freund. Automatic synchronization correction. In *SCOO'05*, Oct. 2005.
- [13] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *TRANSACT'06*, June 2006.
- [14] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA'02*, pages 130–141, Nov. 2002.
- [15] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
- [16] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2nd edition, Nov. 1999.
- [17] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, Feb. 2003.
- [18] L. Li. A practical MHP information computation for concurrent Java programs. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, Aug. 2004.
- [19] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL'05*, pages 378–391, Jan. 2005.
- [20] J. F. Martínez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X*, pages 18–29, Oct. 2002.
- [21] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL'06*, pages 346–358, Jan. 2006.
- [22] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL'07*, pages 327–338, 2007.
- [23] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI'06*, pages 308–319, June 2006.
- [24] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to Java. *TOPLAS*, 28(6):1088–1144, Nov. 2006.
- [25] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *ESEC/FSE'99*, pages 338–354, Sept. 1999.
- [26] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *PLDI'06*, pages 320–331, June 2006.
- [27] E. Ruf. Effective synchronization removal for Java. In *PLDI'00*, pages 208–218, June 2000.
- [28] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *OOPSLA'06*, pages 263–272, Oct. 2006.
- [29] A. Sălcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP'01*, pages 12–23, June 2001.
- [30] V. C. Sreedhar, Y. Zhang, and G. R. Gao. A new framework for analysis and optimization of shared memory parallel programs. Technical Report CAPSL-TM-063, CAPSL, U. Delaware, Newark, Delaware, USA, July 2005.
- [31] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP'05*, pages 2–13, June 2005.
- [32] R. Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, July 2000. <http://www.sable.mcgill.ca/soot/>.
- [33] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06*, pages 334–345, Jan. 2006.
- [34] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI'03*, pages 115–128, June 2003.
- [35] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *ECOOP'06*, volume 4067 of *LNCS*, pages 148–173, July 2006.
- [36] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Optimized lock assignment and allocation for productivity: A method for exploiting concurrency among critical sections. Technical Report CAPSL-TM-065, CAPSL, U. Delaware, Newark, Delaware, USA, Apr. 2006.
- [37] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Optimized lock assignment and allocation: A method for exploiting concurrency among critical sections. In *PPoPP'07*, pages 146–147, Mar. 2007. Short paper from poster session.