McGill University
School of Computer Science
Sable Research Group

Component-Based Lock Allocation

Sable Technical Report No. 2007-3

Richard L. Halpert Christopher J. F. Pickett Clark Verbrugge

{rhalpe,cpicke,clump}@sable.mcgill.ca

May 17th, 2007

Abstract

The choice of lock objects in concurrent programs can affect both performance and correctness, a burden of complexity for programmers. Recently, various automated lock allocation and assignment techniques have been proposed, each aiming primarily to minimize the number of conflicts between critical sections. However, practical performance depends on a number of important factors, including the nature of concurrent interaction, the accuracy of the program analyses used to support the lock allocation, and the underlying machine hardware. We introduce *component-based lock allocation*, which starts by analysing data dependences and automatically assigns lock objects with tunable granularity to *groups* of interfering critical sections. Our experimental results show that while a single global lock is usually suboptimal, high accuracy in program analysis is not always necessary to achieve generally good performance. Our work provides empirical evidence as to sufficient strategies for efficient lock allocation, and details the requirements of an adaptable implementation.

1 Introduction

Achieving good concurrency in a program involves selecting appropriate locks to guard individual critical sections. Automatic techniques for lock allocation remove this complex concern from the programmer, and can also be the basis of the implementation of transactional languages [14]. The problem of determining an *optimal* set of locks to guard given code with critical sections has been considered, with the *Minimum Lock Assignment* (MLA) optimization problem being NP-hard, and k-bounded lock assignment (KLA) being NP-complete [10, 14, 26, 31]. Heuristics are thus 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, and find that for most benchmarks this can achieve the same runtime performance as the original program with a manually specified lock allocation. Integration with an MLA solver may be beneficial for cases where performance is sub-optimal; however, many concurrent programs practically exhibit simplistic concurrent behaviour, so good solutions can be achieved with straightforward program analyses.

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 use heuristics to assign locks to interference graph components. This contrasts with more *bottom-up* approaches used by McCloskey [18], Hicks [14], and Emmi [10], which associate locks with individual data, either manually or automatically, and then use a subset of these locks to transform critical sections. The overall approach used by Sreedhar et al. has some similarity [26] to ours, but does not perform the key intermediate refinement of allocating locks per graph component rather than per critical section, instead moving immediately to MLA and KLA as later developed by the same authors [31].

Our technique is highly flexible with respect to locking disciplines as well. We do not require explicit data annotations, nested synchronization is permitted, we allocate dynamic locks if possible, and we allow for use of condition variables. We do not require *locksets*, which acquire and release all locks at the beginning and end of an outer critical section, nor do we require *two-phase locking*, in which all locks are released before others are acquired. Instead we use static analyses to detect when nested locking might lead to deadlock, and default to a conservative solution that prevents cyclic dependences; related work is careful to construct a total ordering, and either approach breaks one of Coffman's four conditions necessary for deadlock [8].

1.1 Contributions

We make the following specific contributions:

- A *component-based* lock allocator for Java that assigns either static or dynamic locks to groups of interfering critical sections. This depends on accurate construction of a *critical section interference graph* using a *thread-based side effect* analysis, and contrasts with previous work that focuses on assigning locks to individual critical sections. Additionally, our allocator is more flexible with respect to the locking disciplines it supports than prior work.
- Synchronization elimination is a trivial consequence of our approach. A component containing an isolated critical section that does not interfere with itself does not require synchronization, and our data show that many such components exist.
- We provide useful enhancements to *may-happen-in-parallel* (MHP) analysis for Java, which we use to improve interference graphs by pruning false edges. MHP performance can be a concern, and we ensure fast results by using a *lock oblivious* approach, essentially ignoring the effect of locks. Although this reduces information quality, our *start-join analysis* allows us to provide a coarse categorization of thread behaviour to this simple MHP analysis that achieves sufficient accuracy to make good lock allocation decisions.
- We provide experimental data for a wide range of Java benchmarks using various permutations of our analysis pipeline. We find that in many cases, our component-based allocator matches the performance of the original program.

2 Related Work

We use Soot as a Java bytecode compiler framework [28], depending on the built-in class hierarchy analysis (CHA) [9], context-insensitive points-to analysis [15], and may-happen-in-parallel (MHP) analysis [16] as a starting point. Naumovich *et al.* were the first to present an algorithm for computing MHP information for concurrent Java programs [22], and the one provided by Soot is similar. Barik proposed a scalable alternative to Naumovich's analysis for Java [5], and Agarwal *et al.* later extended it to support X10 [2].

Our points-to analysis, while thread-insensitive, provides input to a *thread-context-sensitive* side effect analysis that models the heap using thread-local and shared partitions. A similar analysis for Java was developed by Chang and Choi, and we build on that work by demonstrating a concrete application [7]. Our analyses work towards determining interferences between critical sections, which might also benefit from an interprocedural thread-sensitive slicing analysis for Java [21]. We derive thread-sensitivity using a *thread-local objects* (TLO) analysis that detects when certain reads and writes inside critical sections are thread-local, and find that this improves lock allocation considerably. This is particularly interesting when compared to the work of Sălcianu and Rinard who found that thread-local object information was ineffective for synchronization *removal* [24], and Aldrich *et al.* who found it statically effective for multithreaded programs, but not such that runtime performance was affected [3].

One problem closely related to lock allocation is static race detection, in which significant advances have been achieved recently. 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 [20]. Naik and Aiken later define *conditional must not alias analysis* as the problem of concluding whether two objects are

aliased from the hypothesis that two other objects are not aliased [19]. In the context of static race detection, this involves determining whether two guarded memory regions are aliased based on the knowledge that the lock objects guarding them are not aliased. 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 ρ, and whether there is some lock l that is always held for each access to ρ [23]. Abadi *et al.* present a type-based system for Java programs that depends on annotations to detect races [1]. They use a tool to infer these annotations automatically, and then feed them to a fixed point computation that identifies and removes the incorrect ones using their 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 [11]. These techniques all find memory accesses that are not properly synchronized, whereas the lock allocation problem examines properly synchronized memory accesses and provides a less conservative solution. Our requirement that the input program be correctly synchronizable is precisely defined by the Java Memory Model [17].

work	language	compiler analysis			locking discipline						allocation		input		results	
		pointer	TLO	MHP	data	nesting	2-phase	locksets	dynamic	CVs	heuristics	MLA	small	large	AOT	runtime
[18]	C	some	no	no	yes	yes	yes	no	yes	yes	no	no	yes	yes	yes	yes
[14]	C	yes	yes	$\approx$	no	yes	yes	yes	no	no	yes	no	no	no	no	no
[26, 31]	OpenMP	yes	no	yes	no	no	yes	yes	no	yes	yes	yes	yes	no	yes	some
[10]	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

Table 1: Related work on lock allocation.

Recently there has also been a fair amount of work directly related to lock allocation, and we compare the significant differences in Table 1. McCloskey *et al.* introduce *pessimistic atomic sections* and provide a tool to convert them automatically to more efficient lock-based code [18]. In addition to requiring user identification of atomic sections, they also require annotations associating locks with all shared data. A whole-program analysis detects the use of shared data inside atomic sections and a transformation ensures that the right locks are acquired according to a global total ordering. Pessimistic atomic sections can be nested, and the transformation algorithm is provably sound. One limitation is that a strict *two-phase locking* discipline is required, such that for a given atomic section, once a lock is released, 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 if the annotations are correct [29].

Hicks *et al.* also convert atomic sections to pessimistic transactions [14], using the same compiler analysis framework as for their static race detection tool [23]. An analysis of *continuation effects* upon thread creation is roughly comparable to MHP analysis. Locks are associated with abstract memory locations identified by a pointer analysis, and the correct locks acquired and released at the beginning and end of atomic sections. They make two improvements, first by eliminating synchronization on thread-local data, and second by coalescing locks that are always acquired and released together. The first improvement is comparable to our *thread-local objects* (TLO) analysis. The second, lock coalescence, is a heuristic for minimizing the number of locks. They do not permit dynamic locks, whereas the annotations used by McCloskey *et al.* do, and they note that maintaining a global ordering with dynamic locks might require runtime support. They also require that all locks in a lockset be acquired and released only at the outer boundaries of composed atomic sections.

Sreedhar, Zhang *et al.* propose a framework for dataflow and concurrency analysis of parallel programs, and use it specifically to study the problem of assigning locks so as to maximize concurrency and minimize serialization overhead [26, 31]. They compute a *concurrency relation* that is comparable to our MHP analysis and use it for data flow problems, in particular pointer analysis and lock allocation. Like us, they

depend on concurrency information to identify critical sections with intersecting read/write sets that are actually independent. They construct a concurrency graph with either an interfering or a non-interfering edge between two critical section vertices, and compute a *minimum lock assignment* (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. We use a straightforward translation of their concurrency graph in which all edges indicate interference and non-interfering edges are removed. They also provide a *k-lock allocation* (KLA) algorithm for bounding the number of locks in exchange for serialization overhead, an obvious corrollary of k-colouring as used by register allocation. They formulate MLA and KLA as *integer linear programming* (ILP) problems, and heuristic solutions for a range of randomly generated inputs are then compared 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 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. The work has practical importance because OpenMP *only* supports a single global lock, and despite a lack of good benchmarks enables a 2x improvement in one function of UA in the NAS parallel benchmark suite.

Emmi *et al.* have also examined the problem of lock allocation [10]. They build directly on McCloskey's work by eliminating the requirement for annotations protecting shared data, and in most cases can automatically infer the same protections for his AOLServer benchmark. Like Zhang *et al.* they depend on ILP for allocation, and clearly explain 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 AOLServer, a realistic benchmark. They do consider dynamic locks in some detail, using an *accessed-before relation* derived from temporal analysis of critical sections to avoid deadlock, 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, an active field of research [12, 13, 25, 30]. Our analysis could be used to reduce the overhead of optimistic concurrency by providing information about interferences between critical or atomic sections to the runtime system.

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 such that there exists some allocation of locks that results in correct synchronization, as defined by the Java Memory Model [17]. Any original lock allocation is discarded, and the lock allocator chooses locks that guarantee a race-free, deadlock-free program. This allows for newly written software to ignore lock allocation, and for legacy applications to benefit from automatic correction of unsafe manual allocations. Additionally, both classes of program benefit from unnecessary synchronization removal, and experimentally, legacy programs containing fine-grained manual allocations provide a basis for evaluation of lock allocation strategies.

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, we redirect these calls 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. Uses of `wait()` and

`notify()` require deadlock considerations, 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*. Interference edges between two critical sections indicate 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.

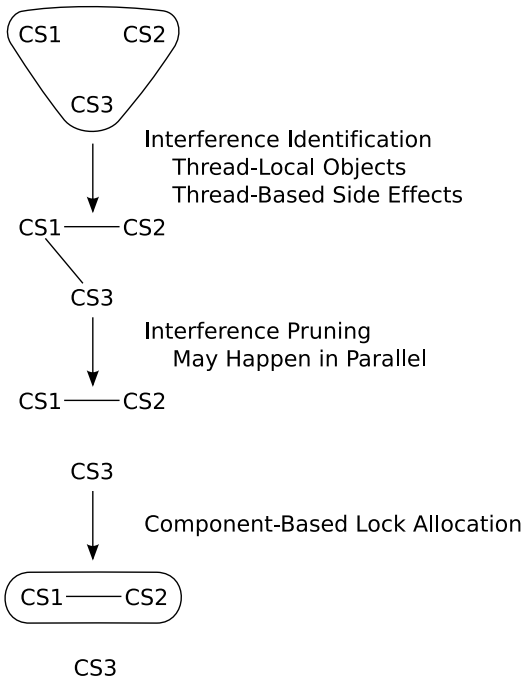

Figure 1: Analysis pipeline.

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

This yields a set of *locked components* which contain interfering critical sections, CS1–CS2 in the example, and *unlocked components* which are isolated critical sections without self-loops, CS3 in the example. *Component-based lock allocation* proceeds to allocate a static or dynamic lock to each locked component, locking only CS1–CS2 in the example and removing unnecessary synchronization from CS3. The result is a correctly synchronized program that is less conservatively synchronized than the original.

3.1 Information Flow Analysis

Our lock allocator uses a flow-insensitive, context-sensitive, interprocedural *information flow analysis* (IFA) as the basis of a *thread-local objects analysis* (TLO) as described in Section 3.2. Given a pair of named memory locations, IFA approximates whether the value stored in one location is derived from the value

stored in the other location. For the purpose of detecting relevant thread interactions, this analysis considers only explicit information flow resulting from direct assignment or arithmetic operations. As used by the lock allocator, the current analysis is unsound because implicit information flow is not included, although a refined version could provide sound behavior.

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. Each assignment statement generates an edge in the graph, as does each return statement. 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. Of course, multiple values may be accessed by a single `GETFIELD` instruction; the mapping from values to nodes determines the precision of the analysis.

At callsites, summaries are retrieved and combined for all possible target methods. The 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 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. If a method is already under consideration by the interprocedural recursion when a summary is requested, a simple conservative summary is used instead. Also, in order to ensure reasonable runtime at only a small cost to precision, internal library calls always use a simple conservative summary.

3.2 Thread-Local Objects Analysis

After initial points-to analysis and call graph construction, our lock allocator performs a *thread-local objects analysis* (TLO) that serves to improve the precision of a later *thread-based side effect analysis* (TBSE), as described in Section 3.3. TLO classifies all fields as either *thread-local* or *thread-shared*, where any field whose value is possibly accessed by more than one thread at a time is considered thread-shared. It uses IFA to propagate this information throughout the program, which includes every local variable. It can then be safely assumed that reads from and writes to thread-local fields do not need to be synchronized between threads.

TLO is called with a list of thread classes that have type `Runnable`, each of which is analysed independently to determine which fields may hold thread-shared values. Initially, any field accessed directly outside of the `Runnable` via `GETFIELD` or `PUTFIELD` is classified as thread-shared, and all other fields are classified as thread-local. Each method in the `Runnable` is classified as external if it is ever called from outside of the class, otherwise internal. The parameters of each internal method are initially classified as thread-local, and the parameters of each external method are initially classified as thread-shared.

After this initial classification, IFA is used to generate the information flow summary for each method of the `Runnable`. Whenever IFA indicates that a thread-shared value flows to a thread-local field, the classification of that field is changed to thread-shared. This propagation continues until a fixed-point is reached.

Next, a *locality context* containing the classification of each field and parameter is created for each callsite in the methods of the `Runnable`, and IFA provides an information flow graph for each target method of those callsites. The locality context of each callsite in each target is determined from the information flow graph for that target and the locality context of the caller. If a target has already been analysed, then the locality context of the caller is merged with the existing locality contexts of its callsites. This interprocedural propagation continues until a fixed point is reached.

Finally, when queried, TLO will report that a given value in a given method is thread-local if it is thread-local

in all `Runnable` classes that call the method. Otherwise it will report that it is thread-shared.

3.3 Thread-Based Side Effect Analysis

We extend the side effect analysis present in Soot 2.2.3 [15] to a *thread-based side effect analysis* (TBSE), computing a list of `<field, object>` pairs that may be read or written by individual critical sections. These lists are used directly to create the interference graph. TBSE is based on points-to analysis, and incorporates a thread-local objects analysis as described in Section 3.2, a critical section nesting model, and side effect approximations for library methods.

By default, Soot computes the side effects of statements using a simple set of flow equations and the output of its points-to analysis. TBSE analysis alters these equations to ignore side effects involving thread-local objects. Additionally, TBSE significantly alters side effect calculations for method calls. For each method call, instead of considering all transitively reachable methods and accounting for their side effects, our modified analysis excludes internal library calls and static initializers. Our nesting model allows for inner locks to be released independent of outer locks, and accordingly TBSE also excludes the side effects of any method called inside a critical section.

We exclude static initializers because they impact on the precision of our analysis. Although ignoring static initializers as such is unsound, we could safely force them to execute before the code affected by our transformations. Internal library calls are similarly excluded because deep library call chains also impact on the precision of our analysis. Importantly, this excludes many static library fields. However, we do provide a treatment of library interface calls that assumes all receiver fields will be read and written.

We construct an interference graph using the information provide by TBSE. A vertex v is created for every critical section, and interference edges e are inserted between every pair of nodes 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 data dependences for any two critical sections, and when non-empty is stored for the corresponding interference graph edge as its *contributing read/write set*.

3.4 May-Happen-in-Parallel Analysis

After constructing the interference graph from thread-based side effect results, we use a *may-happen-in-parallel* (MHP) analysis to prune false positive edges from the interference graph. Our implementation is a context-insensitive and lock-oblivious adaptation of the analysis available in Soot [16]. It first uses a *run-once, run-many analysis* to categorize calls to `Thread.start()` 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-oblivous* in that it ignores `MONITORENTER` and `MONITOREXIT`, synchronized `INVOKE*`, and calls to any form of `wait()` and `notify()`. This allows us to determine which critical sections actually require locks to prevent parallel execution, and contributes to unnecessary synchronization elimination. This 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 reads the call graph in conjunction with the body of each method, and marks each statement and the method itself as either *run-once* or *run-many*. 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, calls to `Thread.start()` are used to identify and categorize distinct thread classes. If the invocation statement is run-once, then the corresponding thread class is run-once. If the statement is run-many but the points-to analysis determines that a unique `Thread` object reaches the call, then the corresponding thread class is also run-once. Otherwise, the call creates a run-many thread class. These thread classes are actually input to the TLO analysis described in Section 3.2.

In the case of run-many thread classes, a *start-join analysis* searches the method for a matching call to `Thread.join()`. Next, local must-alias analysis is performed to ensure that any apparent match is guaranteed to join the same thread. Then a post-dominator analysis determines if the match always runs. If a match is found, and the containing method is not reentrant and may not happen in parallel, then the corresponding `Thread.start()` is labeled *run-one-at-a-time*, otherwise *run-many-at-a-time*.

The MHP analysis finally reports that any pair of methods that might be executed by two or more different threads may happen in parallel. For this classification, each run-many-at-a-time thread class is treated as two different threads, and all other thread classes 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 is somewhat complicated, and differs from that of Li and Verbrugge [16] in several ways. Their analysis creates a whole-program control flow graph in order to correctly analyse synchronization and wait/notify statements and provide lock sensitivity. Our lock-oblivious analysis works more quickly than their lock-sensitive implementation because it does not need to create this whole-program CFG. It also works for a wider variety of programs: they require that every virtual method call can be statically resolved. The primary limitation of our analysis is that it ignores threads implicitly started by the JVM and Java class libraries.

3.5 Lock Allocation

Lock allocation begins after the interference graph has been pruned using MHP information. Identification of locked and unlocked graph components is straightforward, and our allocator assigns locks to locked 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 dynamically allocated set of lock objects for each locked component, reverting to static allocation if necessary.

All three granularities replace any previously existing lock objects used by critical sections, and guarantee that every critical section is protected by an appropriate lock object. Synchronized methods are replaced with unsynchronized wrappers around synchronized blocks, and calls to `wait()` and `notify()` are redirected to the new lock object protecting 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. For static allocation, each locked component is assigned a different static lock.

For dynamic allocation, the interference graph is also partitioned into locked components. Several tests are performed on each component to determine if a dynamic lock can be used. First, if the union of the contributing read/write sets for all interference edges in the component, as defined in Section 3.3, contains fields from different object types, then a dynamic lock cannot be used.

Next, a deeper analysis determines if all contributing read/write sets for each critical section can be protected by a single dynamic lock. For each critical section, and for every path inside the section, every contributing read/write must involve a single object accessible at its entry point. A local flow-sensitive must-alias analysis is used to determine if all contributing reads/writes on a path involve the same object. The safety of this analysis depends on our input requirement that all shared state be protected by some critical section, and our guarantee of critical section isolation semantics. A local *common predecessor alias analysis* determines if all objects accessed over all paths are reachable from a single object reference at the beginning of the section. If these two conditions are met, then the object found for each critical section may be used as the lock, and we classify this as *dynamic allocation*. Otherwise, a static lock is assigned to the entire component.

3.6 Deadlock Avoidance

The analyses discussed so far have focused on freedom from data races. However, another condition for correct synchronization is absence of deadlock, and we ensure this by breaking cyclic lock acquisitions [8]. Our lock allocator abides by a policy of minimal perturbance to any problematic lock allocation when ensuring a deadlock-free program. We allow an initial lock allocation to proceed without regard to deadlock, detect possible cycles in the partial ordering implied by critical section nesting, and correct deadlock by adding *deadlock avoidance edges* to the interference graph and reallocating the locks. This technique is well suited to the current design of our allocator, and also to the use of finer allocations involving multiple locks per critical section, which we intend to address in our future work.

Deadlock detection involves examining pairs of critical sections. If a pair is nested, then we assign a partial ordering to their locks. This occurs regardless of whether the locks are static or dynamic. The ordering is then compared to all previously identified partial orders, and if no violation is found then detection proceeds to the next pair.

When a violation is found, it indicates a source of potential deadlock. The order that it violates was stored along with a list of nested critical sections that induced it. A *deadlock avoidance edge* is added between the outer critical section of the failed partial order and the outer critical sections of the violated order. Then, lock allocation is restarted using this new interference graph.

Deadlock avoidance edges are treated like interference edges for static fields, and must be protected by a static lock. This static lock guarantees the absence of deadlock in the final allocation. These edges work for both static and dynamic allocations.

Our deadlock detection algorithm handles typical nested locking deadlock; however, it does not handle nested wait/notify deadlock. This situation can arise 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 requiring that calls to `notifyAll()` be reachable regardless of the locks guarding a call to `wait()`. This type of deadlock is no more difficult to detect than the type that we already handle; however, at present, we do not detect it. In practice, none of the benchmarks we experimented with exhibited this behavior.

4 Experimental Results

In this section, we evaluate the effectiveness of several different configurations of our lock allocator by using it to transform multithreaded benchmarks from five different sources. Table 2 lists the benchmarks and their properties. We use benchmarks from version 2006-10-MR2 of DaCapo at the default input size [6], and SPEC JVM98 at input size 100 [27]. Notably, we did not include most of the JavaGrande suite of parallel benchmarks. Our tests revealed that this suite makes little use of critical sections, and heavy use of volatile variables. This makes it ill-suited for benchmarking current lock allocators, but a good candidate for exploring compiler assisted synchronization outside of critical sections.

name	Critical Sections	Description	Source
TrafficSim	24	a car and driver pair navigate around a rotary	Internal
RollerCoaster	6	7 passengers compete for seats	Internal
PCMAB	2	25 producers and consumers connect via an aspect	Internal
TestBench	3	3 threads increment a private and a shared counter	Internal
BankBench	8	8 threads transfer funds between two accounts	Doug Lea
JGFSyncBench	16	4 threads increment a counter	Java Grande
mtrt	6	2 threads render a raytraced image	SPECjvm98
lusearch	44	32 threads search a large index for 3500 words	DaCapo
hsqldb	4	25 threads run transactions against a banking app	DaCapo
xalan	4	8 threads perform XSL transforms	DaCapo

Table 2: Benchmarks.

4.1 Lock Allocations

Table 3 shows the effect of different analysis pipeline configurations on interference graph construction and lock allocation for each benchmark. Only dynamic allocation is shown for each configuration because the static allocation can be determined from it by simply replacing each dynamic lock with a static one. The graph characteristics column shows how the interference graph evolves with the introduction of new components into the pipeline. The $|V|$ column is naturally constant for each benchmark, corresponding to the number of critical sections in Table 2, and it indicates the size of the interference graph construction problem. The $|E|$ column is the number of edges in the final interference graph, and the $\frac{|E|}{|V|}$ column is graph density, which ranges from 0 to $|V|^2$ and is a suitable measure of graph quality. The weight of any edge $e \in E$ is the number of fields involved in its contributing read/write set, and therefore the weight summed over all edges indicates the size of the lock allocation problem for dynamic locks.

Most of the benchmarks show reduced graph density as more detailed analyses are introduced, although not necessarily at every stage. It is clear from the graph density and total edge weight metrics that the introduction of MHP has the most profound effect on the interference graph. The graph components column illustrates how improvements to the interference graph lead to improvements in lock allocation. A *graph component* is either a connected set of vertices which will be *locked*, or an isolated vertex which will be left *unlocked*. Our lock allocator assigns one lock, either static or dynamic, to each connected component. As the graph density decreases, the total number of graph components increases, leading to a larger number of non-overlapping locks, a larger number of eliminated locks, and the possibility for greater parallelism. A lower graph density also contains fewer edges that could restrict the availability of dynamic locks.

Most the benchmarks have several isolated vertices in at least one analysis configuration. Each isolated vertex represents a critical section that does not require a lock, either because it lies in dead code, interferes

benchmark	analysis pipeline			graph characteristics				graph components			
	points-to	TLO	MHP	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum weight(e)$	total	locked		unlocked
									static	dynamic	
TrafficSim	CHA			24	101	4.20	128	7	3	1	3
	SPK			24	95	3.95	107	9	3	1	5
	SPK	X		24	81	3.37	90	9	3	1	5
	SPK		X	24	38	1.58	44	9	3	0	6
	SPK	X	X	24	32	1.33	38	10	3	1	6
RollerCoaster	CHA			6	16	2.66	27	2	1	1	0
	SPK			6	16	2.66	27	2	1	1	0
	SPK	X		6	16	2.66	27	2	1	1	0
	SPK		X	6	14	2.33	22	2	1	1	0
	SPK	X	X	6	14	2.33	22	2	1	1	0
PCMAB	CHA			2	4	2.00	5	1	1	0	0
	SPK			2	4	2.00	5	1	1	0	0
	SPK	X		2	4	2.00	4	1	1	0	0
	SPK		X	2	4	2.00	5	1	1	0	0
	SPK	X	X	2	4	2.00	4	1	1	0	0
TestBench	CHA			3	5	1.66	5	2	1	1	0
	SPK			3	5	1.66	5	2	1	1	0
	SPK	X		3	4	1.33	4	2	1	0	1
	SPK		X	3	5	1.66	5	2	1	1	0
	SPK	X	X	3	4	1.33	4	2	1	0	1
BankBench	CHA			8	29	3.62	29	3	1	1	1
	SPK			8	20	2.50	20	4	1	1	2
	SPK	X		8	20	2.50	20	4	1	1	2
	SPK		X	8	20	2.50	20	4	1	1	2
	SPK	X	X	8	20	2.50	20	4	1	1	2
JGFSyncBench	CHA			16	68	4.25	222	8	2	0	6
	SPK			16	68	4.25	222	8	2	0	6
	SPK	X		16	68	4.25	222	8	2	0	6
	SPK		X	16	4	.25	4	15	1	0	14
	SPK	X	X	16	4	.25	4	15	1	0	14
mtrt	CHA			6	16	2.66	95	3	1	0	2
	SPK			6	10	1.66	62	4	2	0	2
	SPK	X		6	10	1.66	50	4	2	0	2
	SPK		X	6	1	.16	1	6	1	0	5
	SPK	X	X	6	1	.16	1	6	1	0	5
lusearch	CHA			44	173	3.93	271	10	3	0	7
	SPK			44	81	1.84	131	29	4	1	24
	SPK	X		44	80	1.81	123	29	3	1	25
	SPK		X	44	65	1.47	102	30	2	0	28
	SPK	X	X	44	65	1.47	95	30	2	0	28
hsqldb	CHA			4	6	1.50	8	3	3	0	0
	SPK			4	6	1.50	8	3	3	0	0
	SPK	X		4	6	1.50	8	3	3	0	0
	SPK		X	4	2	.50	2	4	2	0	2
	SPK	X	X	4	2	.50	2	4	2	0	2
xalan	CHA			4	8	2.00	11	2	2	0	0
	SPK			4	8	2.00	11	2	2	0	0
	SPK	X		4	8	2.00	11	2	2	0	0
	SPK		X	4	3	.75	4	3	1	0	2
	SPK	X	X	4	3	.75	4	3	1	0	2

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

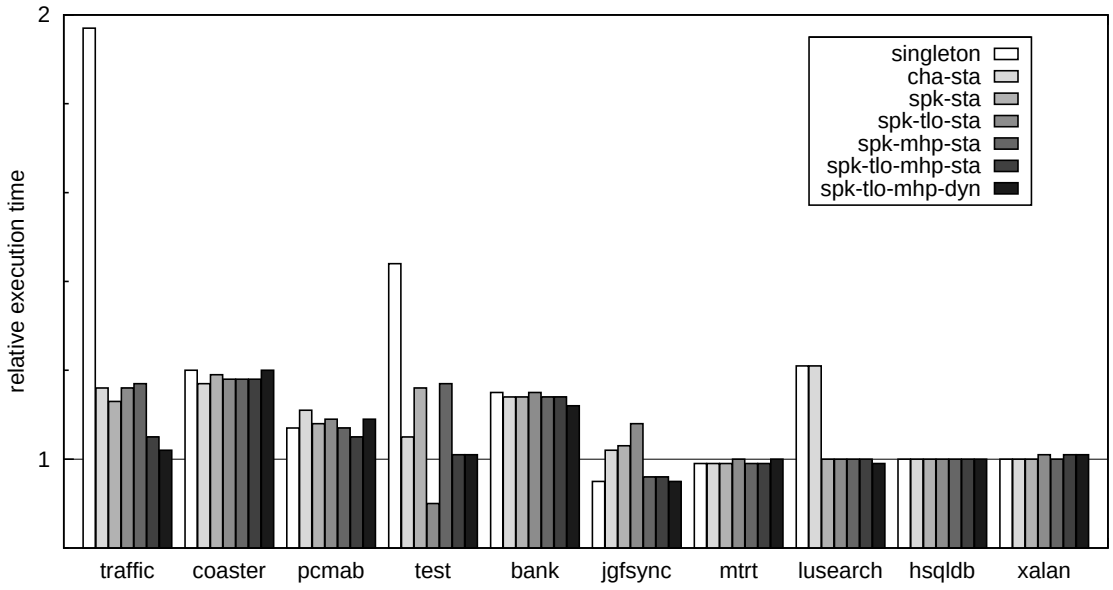

Figure 2: Dual-core x86_64 performance.

only with critical sections with which it cannot occur in parallel, or has no thread-visible side effects. In the latter two cases, the removal of the critical section will reduce the locking overhead of the program. However, the effect of removing a single unneeded lock is generally expected to be small, due to optimizations in most JVMs for uncontended locks [4].

4.2 Performance

Figure 2 shows the relative performance of each benchmark for each configuration on a dual-core x86_64 AMD machine. These configurations match the ones used in Table 3, with *cha* (Class Hierarchy Analysis [9]) or *spk* (Spark [15]) for points-to analysis, *singleton*, *sta*tic, or *dyn*amic lock allocation, and with or without *mhp* and *tlo* analyses.

Seven out of ten benchmarks examined achieved performance within 5% of the original lock allocation. For five of those, good performance was achieved without ever using a dynamic lock, and for four, it was done with nothing more than a context-insensitive points-to analysis. No benchmark exhibited more than a 20% performance hit using the most naïve static lock allocation, and clearly singleton lock allocation is not generally a viable strategy. Furthermore, of the five external benchmarks, only Doug Lea's banking benchmark proved difficult to allocate locks for, still exhibiting reduced parallelism even with our best allocation. This benchmark and several internal benchmarks may benefit from the increased parallelism of using multiple locks per locked component. These results suggest that practical concurrency in popular benchmark suites is not typically improved by sophisticated lock allocation strategies.

An interesting contrast arises when the performance results are compared to the interference graph characteristics. Although Table 3 showed that there is often a large reduction in interference graph edges when MHP analysis is added to the pipeline, for the majority of benchmarks, there is little to no performance improvement from the addition of MHP. Consideration of this result reveals that the edges removed represent pairs of critical sections that cannot execute in parallel, so the performance impact of having those critical

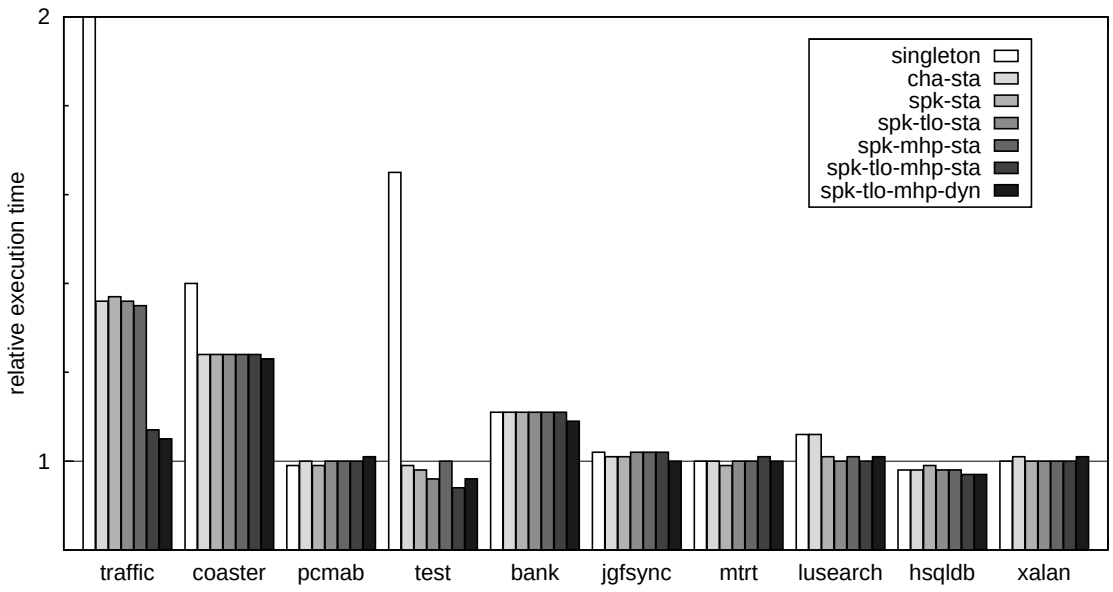

Figure 3: 4-way i686 HT performance.

sections share or not share a lock is negligible. However, MHP does serve a valuable purpose in thinning the interference graph, reducing the size of the lock allocation problem and allowing the key improvements from other stages to take effect.

Figure 3 shows the same configurations run on a four-processor Intel i686 machine with hyperthreading, for a total of 8 virtual cores. The poor scalability of using a singleton static lock is apparent on this machine for the benchmarks that make heavy use of locks at runtime. This is particularly true of the traffic simulation, which contains many non-interfering critical sections whose execution becomes serialized, and the roller coaster, whose threads spend most of their time waiting for locks, and none of it doing real work. Those benchmarks that do perform significant real work experienced better relative performance on this platform.

These results clearly illustrate that the minimum analysis necessary to achieve good performance rises for some benchmarks and falls for others when moving between hardware platforms, and that in general, coarse allocation strategies are worth exploring in order to focus more sophisticated optimal solutions.

5 Conclusions and Future Work

A top-down, component-based approach to automatic lock allocation has the advantage of generating coarse solutions rather than finely grained sets of locks associated with each critical section. Experimental data shows that this approach corresponds rather well with the behaviour of most benchmarks. Aggressively optimal solutions as identified by others [10, 31] are still important of course, and the high quality interference graphs created by our allocator can be used as input. Nevertheless, the simple concurrent behaviour of real benchmarks allows for coarse solutions that focus on key aspects of concurrent interaction to be generally effective in practice.

We have based our design on a consideration of how interference components are formed and what kinds of analysis data would most help separate them. Deeper analysis of the structure and patterns found in

interference graphs would help identify further program properties that may be of use in expanding the scope of our lock allocation, and additional static analyses could be used to relax the restrictions on input programs.

Finally, we assume that there exists a lock allocation for our input programs that will leave them correctly synchronized. However, we are also interested in reusing our compilation infrastructure to support static race detection [1, 19, 20, 23] and correction [11] when this is not the case.

Acknowledgements

This research was funded by the Natural Sciences and Engineering Research Council of Canada, le Fonds Québécois de la Recherche sur la Nature et les Technologies, and the IBM Toronto Centre for Advanced Studies.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 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: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193, Mar. 2007.

[3] J. Aldrich, E. G. Sirer, C. Chambers, and S. J. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, 47(2-3):91–120, May 2003.

[4] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI'98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.

[5] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *LCPC'05: Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, volume 4339 of *LNCS: Lecture Notes in Computer Science*, pages 152–169, Oct. 2005.

[6] 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. Eliot, 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: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[7] B.-M. Chang and J.-D. Choi. Thread-sensitive points-to analysis for multithreaded Java programs. In *ISCIS'04: Proceedings of the 19th International Symposium on Computer and Information Sciences*, volume 3280 of *LNCS: Lecture Notes in Computer Science*, pages 945–954, Oct. 2004.

[8] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *CSUR: ACM Computing Surveys*, 3(2):67–78, June 1971.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95: Proceedings of the 9th European Conference on Object-Oriented Programming*, volume 952 of *LNCS: Lecture Notes in Computer Science*, pages 77–101, Aug. 1995.

[10] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL'07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 291–296, Jan. 2007.

[11] C. Flanagan and S. N. Freund. Automatic synchronization correction. In *SCOOL'05: Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages*, Oct. 2005.

[12] B. Goetz. Optimistic thread concurrency: Breaking the scale barrier. Technical Report AWP-011-010, Azul Systems, Inc., Mountain View, CA, USA, Jan. 2006.

[13] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.

[14] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *TRANSACT'06: Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[15] 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.

[16] L. Li and C. Verbrugge. A practical MHP information analysis for concurrent Java programs. In *LCPC'04: Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, volume 3602 of *LNCS: Lecture Notes in Computer Science*, pages 194–208, Sept. 2004.

[17] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, Jan. 2005.

[18] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL'06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 346–358, Jan. 2006.

[19] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL'07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 327–338, 2007.

[20] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, June 2006.

[21] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to Java. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 28(6):1088–1144, Nov. 2006.

[22] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *ESEC/FSE'99: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 338–354, Sept. 1999.

[23] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, June 2006.

[24] A. Sălcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP'01: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 12–23, June 2001.

[25] N. Shavit and D. Touitou. Software transactional memory. In *PODC'95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[26] 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, Computer Architecture and Parellel Systems Laboratory, University of Delaware, Newark, Delaware, USA, July 2005.

[27] Standard Performance Evaluation Corporation. SPEC JVM Client98 benchmark suite, June 1998. `http://www.spec.org/jvm98/`.

[28] R. Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, July 2000.

[29] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 334–345, Jan. 2006.

[30] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *LNCS: Lecture Notes in Computer Science*, pages 148–173, July 2006.

[31] 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, Computer Architecture and Parellel Systems Laboratory, University of Delaware, Newark, Delaware, USA, Apr. 2006.