STATIC LOCK ALLOCATION

by

Richard L. Halpert

School of Computer Science
McGill University, Montréal

April 2008

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

The allocation of lock objects to critical sections in concurrent programs affects both performance and correctness. Traditionally, this allocation is done manually by the programmer. Recent work explores automatic lock allocation, aiming primarily to minimize conflicts and maximize parallelism by allocating locks to individual critical sections. We investigate several modes of lock allocation, using *connected components* (groups) of interfering critical sections on a critical section *interference graph* as the basis for allocation decisions. Our allocator uses thread-based side effect analysis which is built from several pluggable component analyses. It benefits from precise points-to and may happen in parallel information. Thread-local object information provides a small improvement over points-to analysis alone. Our framework minimizes the restrictions on input programs, dealing gracefully with nesting and deadlock, and requiring only simple annotations identifying critical sections. Legacy programs using synchronized regions can be processed without alteration. We find that dynamic locks do not broadly improve upon identical allocations of static locks, but allocating several dynamic locks in place of a single static lock can significantly increase parallelism in certain situations. We experiment with a range of small and large Java benchmarks on 1 to 8 processors, and find that a singleton allocation is sufficient for five of our benchmarks, and that a static allocation with Spark points-to analysis is sufficient for another two. Of the other five benchmarks, two require the use of all phases of our analysis, one depends on using the lockset allocation, and two benchmarks proved too complex to be automatically transformed to satisfactory performance.

Résumé

L'allocation des verrous aux sections critiques des programmes construits avec des processus indépendants affecte la performance et la validité de ces programmes. Traditionnellement, cette allocation est faite manuellement par le développeur. La littérature récente rapporte des résultats pour l'allocation automatique des verrous. Le but est de minimiser les conflits et de maximiser le parallélisme avec une allocation de verrous aux sections critiques individuelles. Nous étudions plusieurs méthodes d'allocation de verrous, en utilisant des *composantes connectés* des sections critiques qui interfèrent sur un *graphe d'interférence* des sections critiques. Notre technique d'allocation utilise une analyse d'effets secondaires qui considère chaque processus indépendant individuellement. Notre analyse d'effets secondaires est construit d'un ensemble d'analyses composantes interchangeable. La technique d'allocation peut profiter d'une analyse des pointeurs précises et d'une analyse ≪may happen in parallel≫. Nous avons trouvé que la disponibilité des informations qui sont spécifiques à chaque processus indépendant peut améliorer la qualité des résultats par rapport aux résultats de l'analyse avec seulement l'analyse des pointeurs. Notre système demande des pré-requis minimes des programmes qu'il analyse, peut bien analyser les situations d'emboîtement et d'interblocage, et nécessite seulement quelques annotations simples pour identifier les sections critiques. Les logiciels patrimoniales qui portent des régions synchronisés peuvent être analysés directement. Nous avons trouvé que les verrous dynamiques, en général, ne réussissent pas à améliorer la performance au-dessus de celle des verrous statiques. Par contre, l'allocation de quelques verrous dynamiques au lieu d'un seul verrou statique peut augmenter le parallélisme dans certaines situations. Nous avons évalué notre analyse sur une gamme de programmes pour étudier com-

parée. Les programmes sont écrits en Java et notre gamme comprend des programmes qui varient en taille. Nous avons trouvé que l'allocation ≪singleton≫ suffit pour cinq de nos programmes, et que l'allocation statique avec l'analyse pointeur Spark suffit pour deux autres programmes. Il reste cinq autres programmes dans notre gamme, dont deux ont besoin de toutes nos analyses, un requiert l'allocation ≪lockset≫, et deux autres demeurent trop complexes pour la transformation automatique avec performance acceptable.

Acknowledgments

Acknowledgements

I would like to thank my thesis advisor, Clark Verbrugge, for guiding my research efforts. I would also like to thank Chris Pickett for his advice and relentless pursuit of perfection for this research. Clark and Chris both provided editorial advice, and contributed to an earlier work from which this thesis has evolved. I would also like to thank Patrick Lam for his invaluable assistance translating the abstract of this thesis, and my examiners Clark Verbrugge and Laurie Hendren for their numerous suggested improvements.

This research was funded by the Natural Sciences and Engineering Research Council of Canada and the IBM Toronto Centre for Advanced Studies.

Finally, I'd like to thank Leah for her support and companionship during the course of my studies.

Contents

Appendices

List of Figures

List of Tables

Chapter 1
Introduction

Many shared memory parallel programming languages use some version of critical sections to synchronize accesses to shared data. Achieving concurrency and scalability in these programs requires *lock allocation*: mapping locks to critical sections. Traditionally, lock allocation is a manual process susceptible to programmer error, where mistakes may lead to deadlock, livelock, data races, or performance degradation. Often, these bugs and bottlenecks are extremely difficult to find, due to the relative subtlety of the coding errors that cause them, and the difficulty of debugging parallel behavior. The simple examples shown in Figure 1.1 are not so far fetched; in more complete programs these mistakes would not be obvious, and some language features actually hide the locking object from the programmer.

Automatic lock allocation relieves programmers of this burden by guaranteeing desirable properties of the code, such as correctness, good performance, freedom from deadlock, or some subset of the generally accepted properties of software transactions: atomicity, consistency, and isolation[1]. One possible set of guarantees is that of *pessimistic transactions*, in which lock allocation is performed to ensure that critical sections behave atomically [MZGB06, HFP06]. A related technique is *optimistic transactions*, in which lock allocation is not performed (a single global lock is assumed), but speculative execution is used to achieve concurrency [LR06]. While lock

[1]The related property of durability is not normally associated with software transactions - it applies to database transactions.

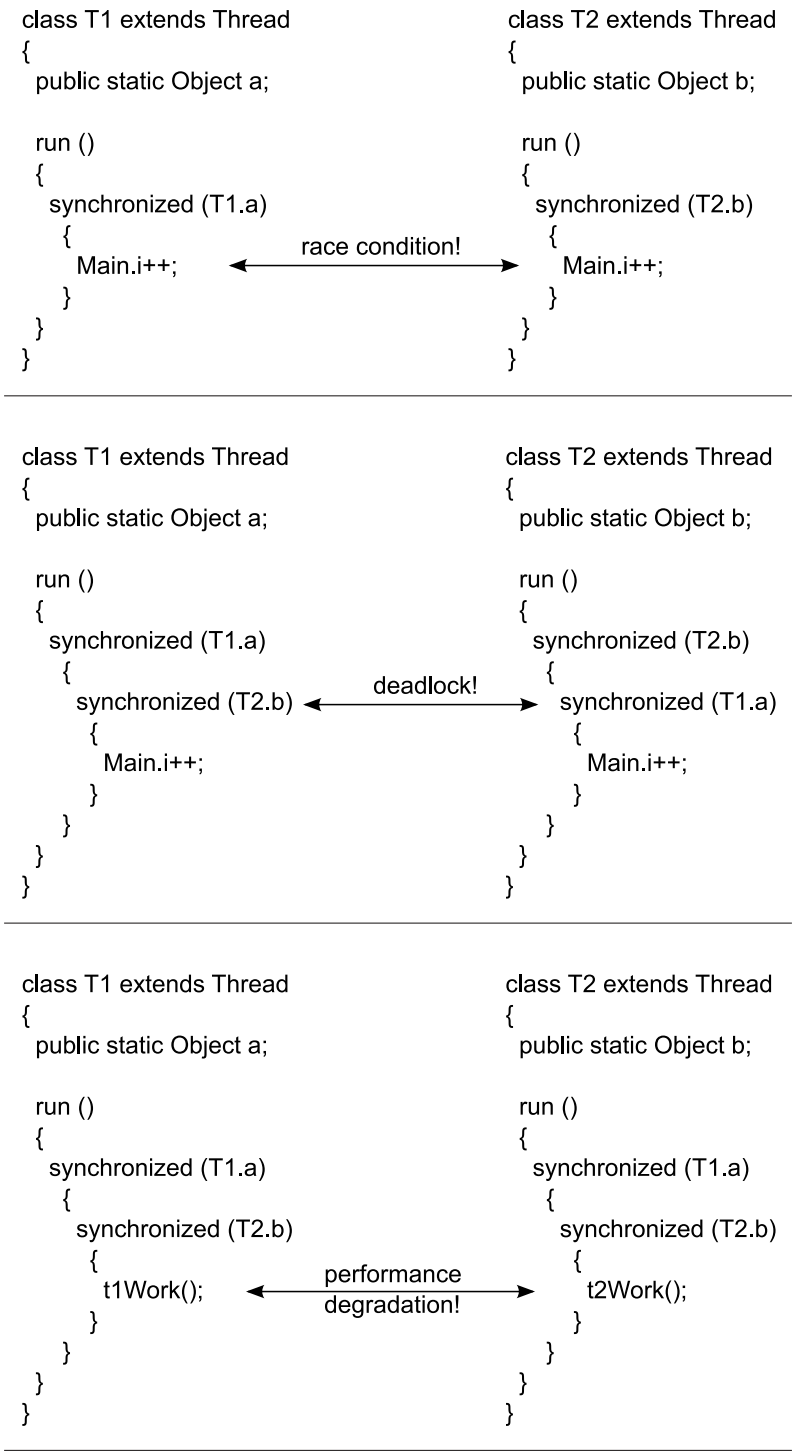


Figure 1.1: Three program excerpts with threading problems.

allocation seeks to simplify the parallel programming paradigm without sacrificing performance, speculative techniques like optimistic transactions actually aim to improve performance.

1.1 Technique

Recent work on lock allocation has focused on finding *optimal* allocations, where locking overhead is balanced against the benefit of increased parallelism [SZG05, ZSZ+06, ZSZ+07, HFP06, EFJM07]. Zhang [ZSZ+06, ZSZ+07] 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. Heuristics may thus be required for practical use of lock allocation.

We present *ALOCS*, the Automated Lock Object Compiler System, which aims to remove the burden of lock allocation from the programmer without sacrificing performance. Our goal in writing ALOCS is to improve lock-allocator technology by using analyses and transformations that mimic or improve upon the decisions made by seasoned programmers without precluding practical usage. We choose to develop ALOCS *without* optimal or heuristic MLA solutions. Specifically, ALOCS generates a *critical section interference graph* and allocates locks on a per-*graph-component* basis, using tunable granularity. ALOCS is especially targeted at novice programmers whose programs with manually allocated locks might have unnecessary correctness or performance issues. We use existing benchmarks with existing locks to test the quality of our allocations, and we find that for many of these benchmarks, our approach is able to allocate locks with similar runtime performance to the original program.

Our results suggest that parallel programs often exhibit simple concurrent behaviour, and that good solutions can be obtained using straightforward program analyses. Of course, techniques such as MLA may sometimes be able to improve on the runtime performance achieved by our *component-based lock allocation*. However, in this instance our analyses would still play two valuable roles: 1) they reduce the size of the MLA problem by initially dividing the interference graph into components,

thereby decreasing the cost of optimal allocation; and 2) they provide a "next best" solution that may suffice when MLA is too expensive.

1.2 Top-Down and Bottom-Up Approaches

Our design is primarily top-down in that we first conservatively assume all critical sections potentially interfere, then use compiler analyses to refine the solution and expose groups of potentially interfering critical sections and the data on which they interfere, from which we construct an *interference graph*. Finally, we assign locks based on interference graph components. This contrasts with the more bottom-up approaches used by McCloskey [MZGB06], Hicks [HFP06], and Emmi [EFJM07], which associate locks with individual data manually or automatically, and then use a subset of these locks to transform critical sections. The approach used by Sreedhar and Zhang [SZG05] is similar to ours, but moves immediately to MLA and KLA [ZSZ+06, ZSZ+07] without considering the particular data in need of locking.

In order to combine the benefits of top-down and bottom-up approaches, we use a *lockable reference analysis* that is similar to the bottom-up techniques used by others, but benefits from being provided with high-level information from the interference graph. This analysis effectively bridges the two approaches by allowing locking decisions to be made at the component scope with both high- and low-level information available.

1.3 Features

ALOCS includes several features intended to increase its practical usability. It is flexible with respect to locking disciplines: it can allocate *dynamic*, per-data structure locks, it allows for use of implicit condition variables, and it requires only that the programmer indicate the locations of critical sections. It permits true nested synchronization; it does not require any type of *two-phase locking*, in which all locks are acquired before any are allowed to be released.

Certain features of ALOCS are intended to improve its value as a research platform. Unlike most related work, ALOCS provides the option to assign exactly one lock for the whole program, to assign exactly one lock per critical section or to assign multiple locks. The last situation is called *locksets*, a set of locks to be acquired before and released after the execution of a critical section. Using multiple locks per critical section is often desirable, though it can introduce extra overhead if it's not possible for concurrency to be further increased.

In the general case, lock allocation poses deadlock concerns, requiring the construction of an ordering among lock acquisitions to break Coffman's circular wait condition [CES71]. Some related work constructs a total ordering of lock acquisitions, and enforces early acquistion of locks in order to satisfy that order, while our work constructs only the minimal partial ordering necessary to prove the absence of circular acquisitions. When no such order exists, our allocator inserts additional locks or merges components to create one.

Another benefit of our work, which is shared by other pessimistic concurrency solutions, is that programs transformed by our lock allocator can be run on existing, unmodified Java Virtual Machines.

Finally, we provide comprehensive compile-time and run-time data for a variety of small and large Java benchmarks. We investigate the utility of applying different static analyses and allocation strategies to the lock allocation problem, and compare the resulting allocations qualitatively and quantitatively. This thorough treatment allows us to suggest the best general-purpose configuration, and to identify areas where future work will likely lead to significant improvements.

1.4 Contributions

We make the following specific 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.

- An implementation of a *thread-local objects* analysis that improves side effect information, and an implementation of 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 six small and six large Java benchmarks, various configurations of contributing analyses and allocation strategies, and complete 1 to 8-way scalability data. Component-based allocation often recovers the original program performance.

- Publicly available code, benchmarks, build scripts and analysis scripts to allow third-party verification of our results. Our code has been integrated into the Soot project to encourage its use in future work. The Soot project is available under the GNU Lesser General Public License. See Appendix B for more information.

The remainder of this thesis is organized as follows: in Chapter 2, we discuss related work; in Chapter 3, we present the design of our system and detail the compiler analyses used (Section 3.2 describes, in order, the activities carried out by our lock allocator); in Chapter 4, we evaluate our lock allocator statically; in Chapter 5, we evaluate our lock allocator dynamically; and in Chapter 6, we conclude and suggest future work.

Chapter 2
Related Work

ALOCS builds on the work of many others. For analysis and transformation we use the Soot Java bytecode compiler framework [VR00]. Our allocator depends on the built-in class hierarchy analysis (CHA) [DGC95], context-insensitive subset-based points-to analysis (Spark) [Lho03], side effect analysis [Lho03], and portions of the may happen in parallel (MHP) analysis [Li04]. Our research is closely related to existing work on thread-sensitive analyses and lock allocation, as well as work on synchronization elimination, static race detection, and optimistic concurrency.

2.1 Foundations

Vallée-Rai introduced the Soot Java bytecode compiler framework [VR00], a framework for the analysis, optimization, and manipulation of Java bytecode. Soot provides compilation and code generation, and provides access to Java bytecode in various intermediate formats. ALOCS operates on the Jimple intermediate format, a three-address stackless representation of the bytecode. Soot provides a framework for writing inter- and intraprocedural static analyses for these intermediate formats, with which many stages of ALOCS were written. Additionally, Soot contains many existing static analyses, some of which are heavily used by ALOCS.

2.1.1 Points-to Analysis

Andersen [And94] developed a context-insensitive subset-based points-to analysis for C, which Lhoták adapted to handle Java and OO features. Lhoták's version is implemented in Soot in the default configuration of his Spark points-to analysis framework [Lho03]. We use Spark, unaltered, as an essential component of ALOCS.

2.1.2 Side-Effect Analysis

Lhoták also provides a Side Effect Analysis as a client of Spark. Lhoták's Side Effect Analysis computes, for each instruction, abstract sets of locations read and written by the instruction. These location sets include static fields, array elements, and instance fields with an associated points-to set for the instance. Lhoták opts to store the computed side effect set for each instruction, but compute on-demand the side effect set for each callsite using the stored sets of the callsite's transitive targets. This technique balances memory and computation requirements to allow the side effect analysis to scale.

We alter Lhoták's side effect analysis by replacing the existing representation of a side effect set with a more precise and more easily manipulated version. These new side effect sets allow a larger variety of standard set computations, which are essential for generating the critical section interference graph.

2.1.3 May Happen in Parallel Analysis

Naumovich *et al.* present an algorithm for computing MHP information for concurrent Java programs [NAC99], and Li provides an implementation in Soot [Li04]. 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 [SFW+05]. Barik proposes a scalable alternative to Naumovich's analysis for Java [Bar05], and Agarwal *et al.* extend it to support X10 [ABSS07]. An uncommon 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. This technique was used for static race detection by Naik *et al.* in [NAW06].

2.2 Thread Sensitivity

Chang and Choi [CC04] and also Sălcianu and Rinard [SR01] 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 [NR06]. We improve thread-sensitivity using a *thread-local objects* (TLO) analysis that identifies thread-local reads and writes inside critical sections. Ruf [Ruf00] and Aldrich *et al.* [ASCE03] 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 [vPG03].

2.3 Synchronization Elimination

We find that synchronization elimination is an inherent consequence of our lock allocation technique. More importantly, we find that elimination of unnecessary locks improves the ability of a lock allocator to resolve the features of a multithreaded program's structure, and thus to assign locks. As such, it is worth revisiting existing synchronization elimination and lock coarsening techniques in the context of lock allocation.

Aldrich *et al.* [ASCE03] present techniques for two kinds of synchronization elimination: thread-local, and enclosed lock. They also state that may happen in parallel analysis can be used for synchronization elimination. Our findings support this suggestion, and we find that this particular step to avoid inserting unnecessary synchro-

nization improves the precision of our critical section interference graph, which can in turn improve our lock allocation.

Others use escape analysis [Bla99][BH99] and specialized points-to analyses [CGS+99][WR99] to find and remove thread-local object synchronization. We similarly use a heap-partitioning thread-local objects analysis to avoid inserting unnecessary synchronization. Although Aldrich *et al.* find little runtime benefit of thread-local synchronization, we find that, as with MHP, TLO can improve our critical section interference graph in beneficial ways.

2.4 Static Race Detection

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 [NAW06]. 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 [NA07]. 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 ρ, and whether there is some lock l that is always held for each access to ρ [PFH06]. Abadi *et al.* present a type-based system for Java programs that depends on annotations to detect races [AFF06]. 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 [FF05]. 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 [MPA05].

2.5 Lock Allocation

There is a large body of recent work on lock allocation. Several manual and automatic techniques have been proposed to address the limitations of most popular programming environments. Table 2.1 compares the most relevant recent work on lock allocation.

McCloskey *et al.* introduce *pessimistic atomic sections* and provide a tool to convert them automatically to lock-based code [MZGB06]. They require manually inserted annotations that associate locks with shared data, in addition to annotations that identify atomic sections. Pessimistic atomic sections can be nested; the locking requirements are flattened and applied to the outer atomic section. *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. If no such order can be found, the program is rejected as potentially containing deadlock. Lock acquisitions are then placed before the statements requiring them, potentially being moved earlier in the code in order to respect the lock order. One limitation is that a *two-phase locking* discipline is required, such that once any 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 [VTD06].

Hindman and Grossman [HG06] propose a source-to-source translation for atomicity, employing fine-grained locking, but not ahead-of-time deadlock avoidance. They instead log write operations to detect deadlock at runtime, and roll back atomic sections when it occurs.

Hicks *et al.* also convert atomic sections to pessimistic transactions [HFP06], using the same compiler analysis framework as their static race detection tool [PFH06]. Locks are associated with abstract memory locations identified by a pointer analysis

Table 2.1: Related work on lock allocation.

Related Work						
Work(s)	[MZGB06]	[HFP06]	[SZG05, ZSZ+06] [ZSZ+07]	[EFJM07]	[CCG08]	[HPV07], This
First Author(s)	McCloskey	Hicks	Sreedhar, Zhang	Emmi	Cherem	Halpert
Language(s)	C	C	OpenMP	C, Java	C/C++	Java
Compiler Analysis						
Pointer	yes	yes	yes	yes	yes	yes
Thread Local Objects	no	yes	no	no	no	yes
May Happen in Parallel	no	no	yes	no	no	yes
Locking Discipline						
Allow Nesting	yes	yes	no	yes	yes	yes
Require At-Once Acq/Rel	no	yes	yes	no	yes	no
Require 2-Phase Locking	yes	yes	yes	yes	yes	no
Allow Locksets	yes	yes	yes	yes	yes	yes
Features						
Definite Dynamic Locks	yes	no	no	yes	yes	yes
Indefinite Dynamic Locks	yes	no	no	no	no	yes
Allow Condition Variables	yes	no	yes	no	no	yes
Lock Choice						
Require Data Annotations	yes	no	no	no	no	no
Heuristics	no	yes	yes	no	no	yes
MLA (ILP)	no	no	yes	yes	no	no
Results						
Small Input	yes	no	yes	yes	yes	yes
Large Input	yes	no	yes	yes	yes	yes
Static Results	yes	no	yes	yes	yes	yes
Runtime Results	yes	no	yes	no	yes	yes

to create locksets that protect critical sections. They allow nesting of critical sections by flattening locking requirements: all locks are acquired and released at the beginning and end of *outer* 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 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. It may also greatly complicate the lock coalescence technique.

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 [SZG05, ZSZ+06, ZSZ+07]. 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. Our *interference graph* is a straightforward translation of their concurrency graph where all edges indicate interference on some set of static and/or dynamic memory locations, 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. This algorithm relies on the fact that their representation of interference between a pair of critical sections is a binary, all or none value. 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, limiting 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 [SZG05].

Emmi *et al.* also examine the problem of lock allocation [EFJM07]. 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. However, they disallow the use of dynamic locks to protect l-values for which a must-alias relation cannot be found, which could potentially have a limiting effect on parallelism. They also note that more precise compiler analysis is complementary to their work.

Cherem, Chilimbi, and Gulwani [CCG08] present a formal framework for *abstract lock schemes*, which is a technique for representing and analyzing locks of different granularities. They develop an intermediate representation transformation system and a runtime library to implement their framework. Cherem, Chilimbi, and Gulwani prove the soundness of their framework, and that the Cartesian product of any two sound lock schemes is a sound lock scheme. This suggests the easy introduction of schemes derived from new analyses.

2.6 Optimistic Concurrency

Finally, we see lock allocation in general as complementary to optimistic concurrency and transactional memory, an active field of research [LR06]. 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 [MT02], and Welc *et al.* demonstrate a software implementation in a Java virtual machine [WHJ06].

Chapter 3
Design

We present the Automated Lock Object Compiler System, ALOCS, a system for automatic lock allocation. ALOCS is designed as an optional module of the Soot Java bytecode optimization and analysis framework [VR00]. It is comprised of several whole-program static data flow analyses that support a pipeline of data transformations. The contributing analyses are described in Section 3.1, and the transformation stages are described in Section 3.2.

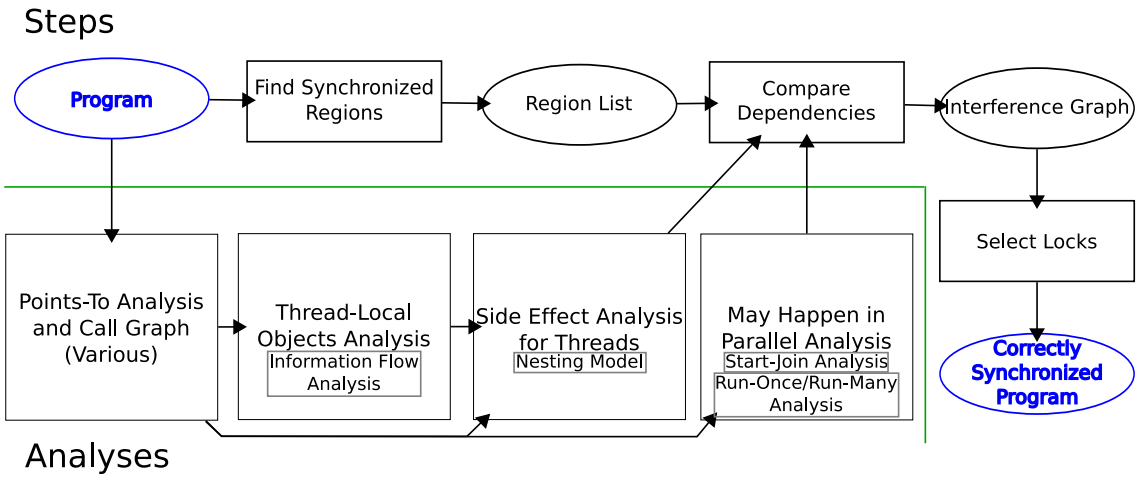


Figure 3.1: Analysis pipeline.

An overview of ALOCS is shown in Figure 3.1. The core of the system is a representation of the input program as a *critical section interference graph*, de-

scribed in Section 3.2.4, which is enabled by our *thread-based side effect* analysis, described in Section 3.1.3. Combined, these two technologies allow Component-Based Lock Allocation [HPV07], which we extend here with the addition of locksets, a finer-grained lock allocation. Our implementation of locksets builds upon those of McCloskey *et al.* [MZGB06], Hicks *et al.* [HFP06], Sreedhar, Zhang, *et al.* [SZG05, ZSZ+06, ZSZ+07], and Emmi *et al.* [EFJM07] by allowing arbitrary nesting of critical sections without increasing the duration for which locks are held.

3.1 Analyses

We perform a set of whole-program data flow analyses which collectively describe the input program with sufficient precision to allocate locks. Most of these analyses are optional, as described in Section 3.2, and in it's simplest form, our lock allocator performs only the thread-based side effect analysis described in Section 3.1.3 and one of the points-to analyses described in Section 3.1.1.

3.1.1 Points-To Analysis

Points-to analysis is a whole program analysis which approximates the set of memory locations to which a given pointer may refer. Points-to analysis is closely intertwined with call graph construction because points-to information can resolve virtual method calls, which in turn may remove some pointer operations from the reachable call graph. Some points-to analyses use an initial rough approximation of the call graph, which is later refined, while others build the call graph simultaneously with the points-to analysis.

ALOCS employs points-to analysis to:

- Determine thread-types for may happen in parallel analysis.
- Find receiving objects for side effect analysis.
- Determine which dynamic locks may be aliased.

ALOCS uses the program call graph by:

- Traversing it in search of channels of information flow for thread-local objects analysis.
- Determining reachable methods for each thread in may happen in parallel analysis.
- Traversing it to propagate run-many status in run-once-run-many analysis.
- Determining reachable methods for a given instruction in thread-based side effect analysis.
- Traversing it to search for relevant uses in lockable reference analysis.
- Determining reachable methods from each critical section for deadlock detection.

We employ SPARK, from [Lho03], which is a highly efficient and flexible points-to analysis framework. In its default configuration, its most precise, it is context-insensitive, field-sensitive, and it builds the call graph on-the-fly. SPARK is also capable of emulating Variable Type Analysis [SHR+00] and Rapid Type Analysis [BS], the former of which we use as an example of a less-precise points-to analysis for our experiments.

3.1.2 Thread Local Objects Analysis

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 3.1.3. TLO classifies all fields, parameters, and local variables as either *thread-local* or *thread-shared*, where any memory location that may be accessed by more than one thread is thread-shared and all others are thread-local. TLO uses an initial classification based on the privacy properties of fields and methods, as described in Section 3.1.2, and uses *information flow* analysis, as described in Section 3.1.2, to determine how to propagate this information throughout the program.

TLO accepts a set of thread classes T that implement the `Runnable` interface, and analyzes each t in T independently to determine which fields in the thread may hold thread-shared values.

In the next sections, we discuss the stages of TLO. These are initial classification of fields and parameters, information flow analysis, analysis of the thread class, propagation, and reporting/results.

Initial Classification of Fields and Parameters

If a field is private to class t, its value is hidden from other threads unless explicitly shared by one of the methods of t. Likewise, if a method is private to class t, it's parameters are hidden from other threads unless a shared value is explicity passed in to it. We therefore include a simple field/method access finder in ALOCS to determine which fields and methods are `private` *or could be* `private`, by not being accessed outside of their declaring class. We call all of these fields and methods *private*. The field/method access finder is a whole-program analysis.

Initially, *private* fields of t and the parameters of *private* methods of t are classified as thread-local, and all other fields and parameters are classified as thread-shared.

Information Flow Analysis

The primary enabling technology of the TLO analysis in ALOCS is *information flow* analysis (IFA). 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 [Ahm06].

Given a method to analyze, 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 data structures manipulated by the method: parameters, fields, statics, and the return value. Our analysis distinguishes between syntactically different values but is type-based and ignores points-to information. The edge-generation rules are listed in Appendix A.

Figure 3.2 illustrates the relationship between a method and the data structures it can access. In this figure, method `foo` in class `C` takes two parameters (`5` and `O` are shown as examples), and returns an `Obj` (shown here being assigned to a variable `result`). `foo` also accesses the two instance fields, `my_object` and `my_int`, and two static fields, `G._object` and `H.an_int`. `foo` can *only access these data structures* and *data structures reachable through them*. IFA conservatively determines the relationships between these data structures, and optionally *data structures reachable through them*. The latter option is significantly more costly in time.

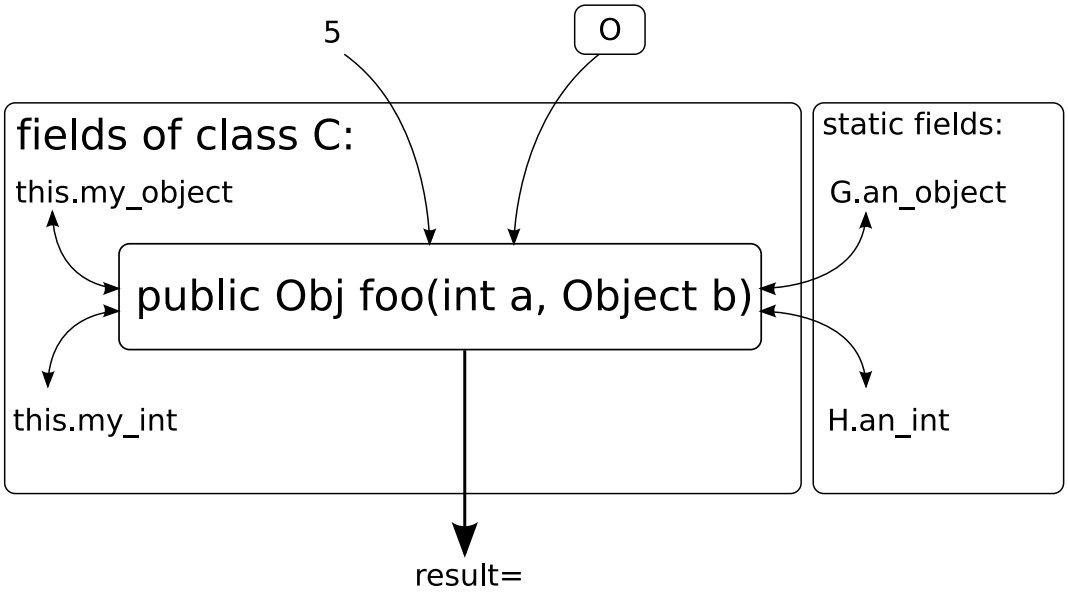


Figure 3.2: A method and the data it can access.

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. The first two cases

are illustrated in Figure 3.2 where `5`, `0` and `result` are connected to `a`, `b`, and the return value, respectively. The full callsite show in Figure 3.2 might take this form: "`c.foo(5,0);`", and the accesses within `foo` to `this.my_object` and `this.my_int` would be connected to the callee as `c.my_object` and `c.my_int`. If no summary exists for a target method of some callsite, 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.

```
public Location getLocation()
{
        Location retval = new Location();
        synchronized(location)
        {
                // get a snapshot of the current location
                retval.forwardLocation = location.forwardLocation;
                retval.lateralLocation = location.lateralLocation;
                retval.roadSegment = location.roadSegment;
        }
        return retval;
}
```

Listing 3.1: Java code for rotary.Car.getLocation() in the *traffic* benchmark.

A code sample and associated information flow graph are shown in Figures 3.1 and 3.3. The code sample is taken from the *traffic* benchmark, which is a simulation of cars navigating around a traffic circle. Each node in the graph represents either a local variable (`[$]rX`, `lock`, `this`), a field read on the current object (`this.location`), a field read on an object pointed to by a local variable (`.roadSegment`, `.forward`, `.lateral`), a field write (`Location.roadSegment`), or a special value. `@this: rotary.Car` represents the current object, `new rotary.Location` represents a newly created object, and `ReturnValue` represents the return value of the method. The value `sourceof<retval>`

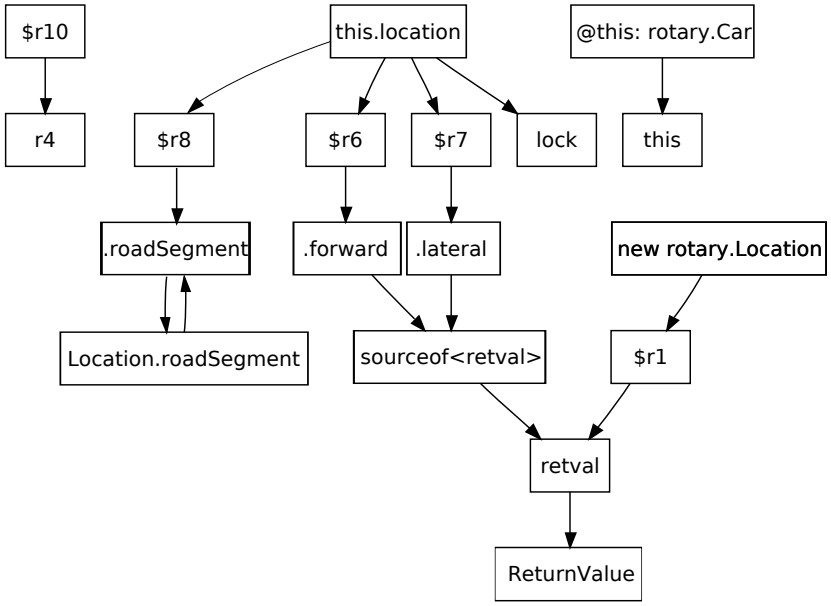


Figure 3.3: Information flow graph for rotary.Car.getLocation() from *traffic*.

is a special node in the information flow graph that indicates that values flowing into it are in fact being stored in the object pointed to by `retval`, in this case the new rotary.Location object.

Figure 3.4 shows the information flow summary that is generated from the information flow graph of Figure 3.3. Local variable nodes have been collapsed, non-escaping and new-object flow has been removed[1], and `sourceof<?>` nodes have been resolved (in this case, a no-op).

Analysis of the Thread Class

After the initial classification, TLO queries IFA to obtain an information flow summary for each method in the thread 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 continues until a fixed point is reached.

[1] New-object flow is ignored as an optimization for the thread-local objects analysis: TLO works by propagating thread-shared status between fields and method inputs/outputs, and new objects cannot become thread-shared until they are stored in a field or returned.

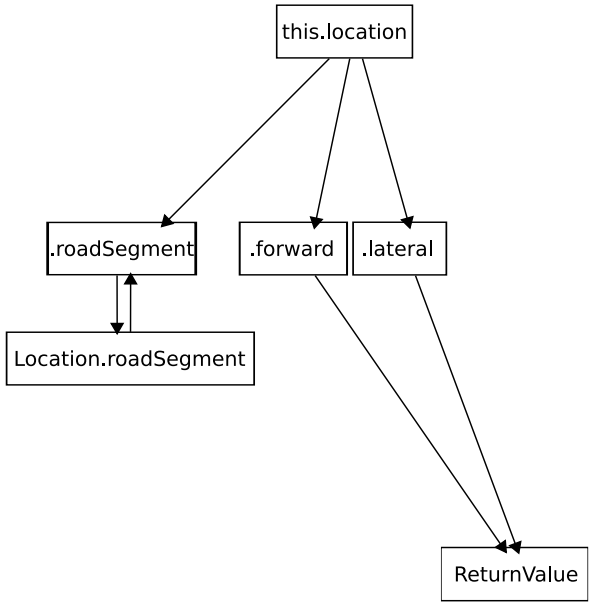


Figure 3.4: Information flow summary for rotary.Car.getLocation() from *traffic*.

TLO Propagation

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.

Reporting TLO Results

When queried about a value u for any method 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.

Figure 3.5 shows an example of a summary of TLO findings for a specific thread

Local fields:
<rotary.Driver: boolean enter>
<rotary.Driver: int noCollide>
<rotary.Driver: rotary.StateActionHistory history>
<rotary.Driver: rotary.ReinforcementLearner learner>
<rotary.Driver: boolean collision>
<rotary.Driver: int changeLanes>
<rotary.Driver: double speedTolerance>
<rotary.Driver: double laneChangeLookahead>
<rotary.Driver: double laneChangeHeadway>
<rotary.Driver: double laneChangeGain>
<rotary.Driver: double spasticity>
<rotary.Driver: java.util.Vector adjacentCars>
<rotary.Driver: java.util.Vector adjacentCarLocations>

Local inner fields:
<rotary.Car: rotary.Driver driver>
<rotary.DriverValueFunction: double[] change>
<rotary.DriverValueFunction: double[] exit>
<rotary.DriverValueFunction: double[] any>
<rotary.DriverValueFunction: double[] accel>
<rotary.DriverValueFunction: double[] enter>
<rotary.ReinforcementLearner: int[] rewards>
<rotary.DriverValueFunction: double[] decel>
<rotary.RoadSegment: rotary.RoadSegment prevExitRoadSegment>
<rotary.RoadSegment: java.lang.String exitLeadsTo>
<rotary.RoadSegment: rotary.RoadSegment exitRoadSegment>

Shared fields:
<rotary.Driver: boolean exit>
<rotary.Driver: boolean done>
<rotary.Driver: rotary.Acceleration acceleration>
<rotary.Driver: java.lang.String destination>
<rotary.Driver: rotary.Car car>
<rotary.Driver: double desiredSpeed>
<rotary.Driver: rotary.Car nextCar>
<rotary.Driver: rotary.Location nextCarLocation>
<rotary.Driver: rotary.Car nextCarBeside>
<rotary.Driver: rotary.Location nextCarBesideLocation>
<rotary.Driver: rotary.Car prevCarBeside>
<rotary.Driver: rotary.Location prevCarBesideLocation>

Shared inner fields:
<rotary.Car: rotary.Location location>
<rotary.RoadSegment: rotary.RoadSegment nextRoadSegment>
<rotary.RoadSegment: rotary.RoadSegment mergeRoadSegment>
<rotary.Location: rotary.RoadSegment roadSegment>

Figure 3.5: Summary of TLO results for rotary.Driver from *traffic*.

class, `rotary.Driver`. The fields of the Driver class are all categorized as Local or Shared. Inner fields, the fields of the objects manipulated by the Driver class and its callees, are separately categorized as Local or Shared. When queried about the fields of these objects, TLO will report that they are thread-shared if either the object is thread-shared or the field is a Shared inner field.

3.1.3 Thread-Based Side Effect Analysis

We extend Lhoták's side effect analysis as implemented in Soot [Lho03] 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 3.1.2, 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 allocations differ from pessimistic transactions in the work of others in that our nesting model allows inner critical sections to acquire and release locks independently of their parent critical sections, and requires they provide their own complete synchronization. Accordingly, TBSE 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 because they impact on precision. We assume that static initializers do not contain or affect critical sections. However, efficient inclusion of static initializers in our analysis would be possible with the availability of precise data on class resolution points.

In addition to making side effect analysis thread-aware, TBSE also includes improvements to the computability of the generated side effect sets. While the existing

side effect analysis in Soot loses precision when its sets are subjected to set operations, TBSE employs sets that retain full precision when standard set operations are performed. Although this makes set operations more expensive, the benefit to the quality of output is drastic for lock allocation.

3.1.4 May Happen in Parallel Analysis

May happen in parallel (MHP) analysis is a technique for determining what parts of a program are possibly allowed to execute together in time from different threads. If two sections of a program are found to never execute in parallel, it may be possible to remove synchronization between them regardless of any data dependences.

Some implementations of MHP perform extensive analysis of possible thread classes in order to generate per-statement MHP information. Our implementation is a context-insensitive and lock-oblivious adaptation of Li's MHP analysis in Soot [Li04]. Our version focuses primarily on thread starts and joins, and uses reachable methods as a simple descriptor for the execution of each thread class. It first uses a *run-once, run-many* analysis to find singleton threads. It then uses a *start-join* analysis to further categorize the remaining threads as *run-one-at-a-time* or *run-many-at-a-time*.

Our MHP analysis is *lock-oblivous* 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.

Run-Once, Run-Many Analysis

The *run-once, run-many* analysis is used to determine which statements are provably run just once. The 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* [Li04]. The initial approximation is that all statements are 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

propagate run-many status until a fixed point is reached.

Start-Join Analysis

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 its receiver object is singleton (has a points-to set with only one object, whose allocation site is a run-once statement) then the thread class of this start statement can be categorized as run-once. Otherwise, t is conservatively assumed to be run-many. As an implementation detail, T is input to the TLO analysis in Section 3.1.2.

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 if run. 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.* [SFW+05].

Method-Level MHP

The MHP analysis finally reports that pairs of methods reachable from two or more different threads 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 [Li04]. 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

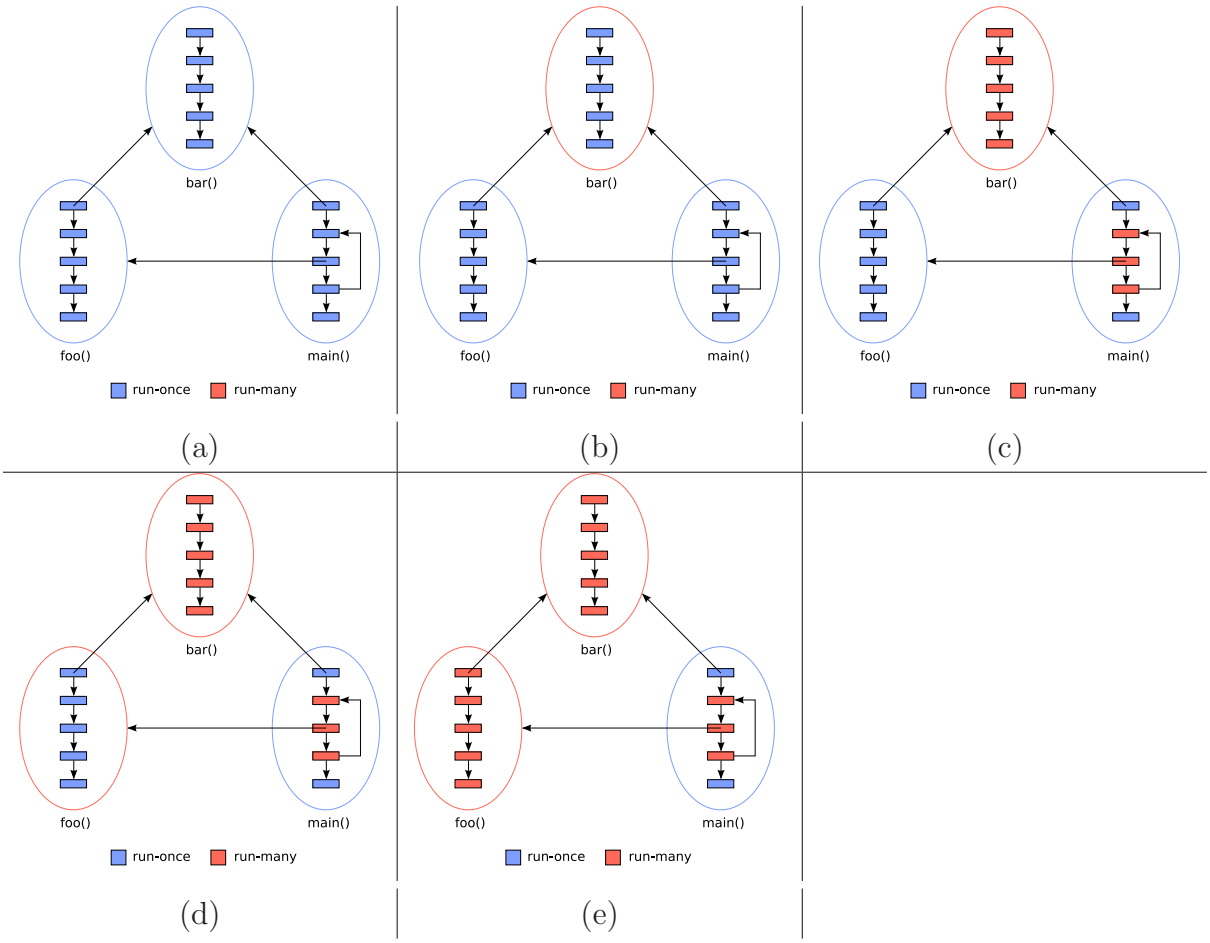


Figure 3.6: Run-Once, Run-Many Analysis.
It is initially assumed that all methods and statements are run-once in (a). In (b), `bar()` is found to be run-many because it is called from two different locations. `bar()`s statements are therefore run-many in (c). Also, the statements within the loop in `main()` are found to be run-many in (c). In (d), since the statement calling `foo()` is run-many, `foo()` itself is run-many as well. Finally, in (e), the statements of `foo()` are found to be run-many.

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. The primary limitation of our analysis is that it ignores threads implicitly started by the JVM and Java class libraries.

3.1.5 Lockable Reference Analysis

Lockable reference analysis (LRA) is used to obtain a list of all of the references which, if locked prior to the execution of critical section S, would guarantee that S executes atomically with respect to other critical sections, inner critical sections notwithstanding[2]. LRA takes as a parameter a set E of all of the side effects of S that might interfere with other critical sections; references whose uses in the given code do not overlap with E are not included in the returned list. The parameter E is a way of applying the results of TBSE to the calculation of LRA, providing a link between the top-down approach of the former, and the bottom-up approach of the latter.

Each reference found must be accessible at the beginning of the section of code as a local reference, or it must be possible to create a local reference to it by inserting some set of assignment statements. This analysis assumes that the objects in question are not altered by any other thread during the execution of S.

LRA examines each Jimple statement s in S. It is a backwards data flow analysis built on the existing Soot framework. If the side effect set e of statement s intersects with E, then the points-to set e' of each value v in s is compared against the objects in E. If e' intersects with the objects in E, then v is considered a *lockable reference*, and is added to the flow set of lockable references u.

The lockable reference set for a callsite is calculated by recursively invoking LRA on all possible target methods, and merging the resulting lists. When back edges in the recursion are encountered, LRA aborts with "lost objects."

Each lockable reference u is assigned a unique value number. When an assignment statement a is encountered, the r-value is assigned the value number that the l-value

[2]See Section 3.1.3 for information about the ALOCS nesting model.

currently has, and the l-value is then assigned a new value number to indicate that its value prior to a is unknown. If the r-value already had a value number, then the two value numbers are merged to indicate that the r-value and l-value from after this point are actually known to be the same. Figure 3.7 illustrates this analysis for a simple example program.

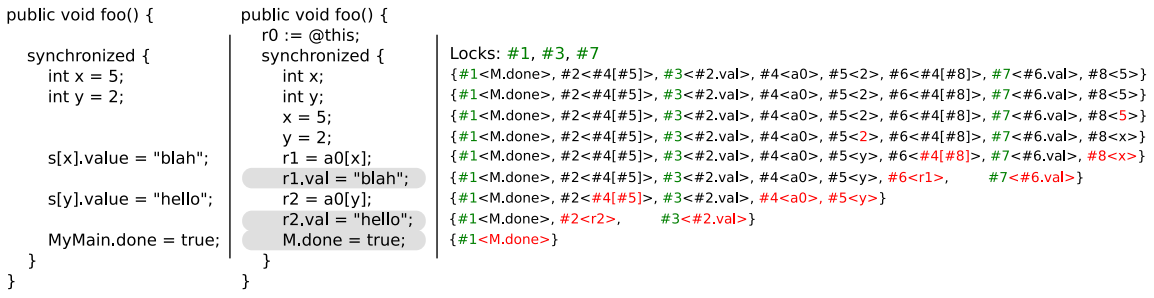


Figure 3.7: Example of Lockable Reference Analysis.
A simple example program and its translation to Jimple code is shown in the left two panes, and the flow sets generated for each statement by Lockable Reference Analysis are shown in the right pane. The shaded Jimple statements are those whose (write) side effects intersect the set of side effects known to require locking, and their left-side value numbers are the lockable references.

`Local`, `Constant`, and `StaticFieldRef` Jimple expressions are handled normally, but certain types of r-values require special considerations. `InstanceFieldRef`s and `ArrayRef`s require that the base (and index) be added to u and assigned a value number as well. The base and index of the references are then replaced with a dummy node that refers to the value number instead of the value, so that they may be reconstructed at the beginning of S. `CastExpr`s require that the value being cast be unpackaged and added to u bare. Finally, expressions that create new objects are not added to u, but a warning is logged that a "lost object" is being ignored. This behavior is preferable for LRA because new object references do not need to be locked. Any other r-value is treated as untrackable, and LRA aborts with "lost objects."

Lockable reference analysis returns a list containing one value from U, the flowset

at the beginning of S, for each value number. To simplify the use of the results, `IdentityRef`s (Jimple statements that associate a local variable with `this` or with method parameters) are preferred over other `Ref`s, which are preferred over any other value in the construction of the result set.

LRA suffers from two significant limitations: It cannot track arithmetic, and it would be prohibitively expensive to analyze the class library. The former often causes lost objects within for loops, while the latter limitiation often causes lost objects when utilizing Collection classes. The effects of the latter limitation could be mitigated by including a special-case model of the Collection classes.

3.2 Pipeline

Our lock allocator uses the analyses described in Section 3.1, reformulates the results as an Interference Graph, chooses appropriate locks according to the given granularity, corrects possible deadlock conditions, and finally transforms the input program accordingly. The lock allocator chooses locks to produce a race-free, deadlock-free program. Variations on the analyses used, the stages run, and the desired granularity affect the resulting lock allocation considerably. The effects of the configuration are studied in Chapter 4 and Chapter 5.

3.2.1 Input Programs

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 [MPA05].

In its current incarnation, the lock allocator uses synchronized regions (or methods) to represent the critical sections. Any original lock allocation is discarded. 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 allowable, provided the input program retries condition variables after waking up from a call to `wait()`. After lock allocation, these calls are redirected to the (innermost) 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()` can have deadlock implications, as described in Section 3.2.6.

3.2.2 Finding Critical Sections

Working with critical sections requires not only extensive record keeping regarding the location and extent of each section, but also details of the interrelationships between them. The first stage of our pipeline finds critical sections and records the preparation, start, ends, and contents of each. It also identifies in-method nesting, and, if analyzing locks instead of allocating them, records the existing object of synchronization. This information is all stored in a *LockRegion* record.

In the Jimple intermediate representation, synchronized regions from valid Java source code will take the form shown in Figure 3.8. Each critical section consists of the following:

- An *entermonitor* statement.
- A *beginning* statement, which is the first statement of the body of the synchronized region.
- A *last* statement, which is the last statement of the body of the synchronized region.
- An *after* statement, which is the first statement executed after control flow falls through the end of the synchronized region.

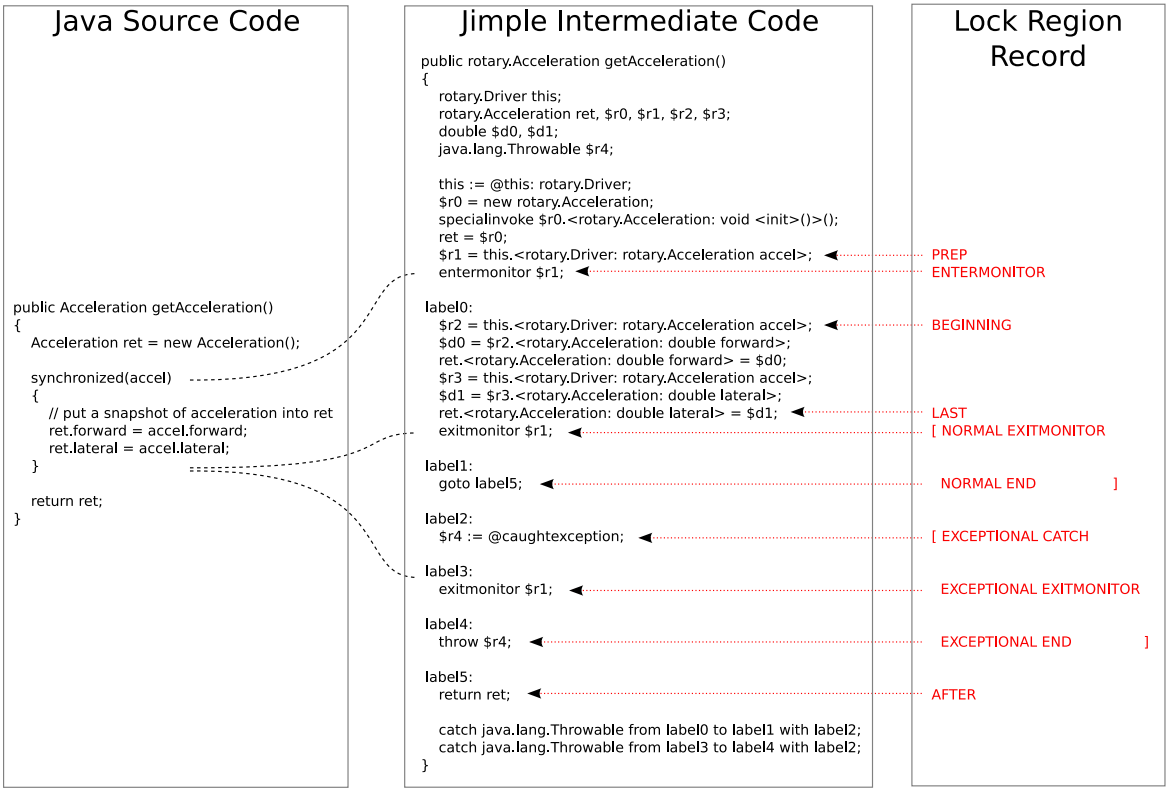


Figure 3.8: Anatomy of a synchronized region in Java and Jimple.

- An *exceptional catch* statement, *exceptional exitmonitor* statement, and *exceptional end* statement. These are the statements executed if an exception is thrown during execution of the body of the synchronized region. The *exceptional end* statement rethrows the caught exception.

- Optionally, a *preparatory* statement, which creates a local reference to the object being locked. A statement is only considered preparatory if the reference it creates is used only by the *entermonitor* statement.

- Optionally, one or more *early end* statements, which is any `goto` or `return` statement that exits the body of the synchronized region. The statement just before each *early end* is an *early exitmonitor* statement.

- Optionally, a *normal exitmonitor* statement, which is the statement executed when control flow falls through the *last* statement. The *normal exitmonitor* statement is followed by the *normal end* statement, which is a goto statement directed at the *after* statement.

The three types of exitmonitor/end statements are used to ensure that the lock is relinquished prior to exiting the synchronized region. The method containing the synchronized region will contain entries in the exception handling table for dealing with exceptions thrown within the body of the synchronized region - these entries are identified and replaced during the transformation phase of ALOCS, so they are not stored in the LockRegion object.

Our analysis is not guaranteed to work with `.class` files generated from sources other than Java, nor with Java bytecode that has been obfuscated.

For reporting purposes, each critical section is given a unique name derived from its containing method and its location within that method. These names are stable between runs of ALOCS so long as new methods have not been added to the program being analyzed.

3.2.3 Generating Read/Write Sets

A critical section's potential interaction with other critical sections can be characterized by its side effects. A pair of critical sections whose side effects do not overlap cannot interfere with each other at runtime. Therefore, they need not share a common lock. Our lock allocator generates approximated sets of side effects in order to identify potential interactions and guaranteed non-interactions.

We employ the thread-based side effect analysis described in Section 3.1.3 to generate these sets. Optionally, TBSE can in turn use TLO to improve the precision of its results. During critical section discovery, the non-transitive side effects of the statements contained within each critical section are calculated and stored in separate read and write sets. Transitive side effects are not calculated until after discovery is complete, so that the effects of nested critical sections can be properly taken into account. Each of a critical section's invoke statements is inspected by TBSE, and the results added to the existing read and write sets.

3.2.4 Constructing the Interference Graph

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 share a data dependence and might conflict at runtime, and a self loop indicates that two or more threads compete for the same critical section.

ALOCS constructs the interference graph using the critical section read and write sets. 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 $(\mathit{read}(v_i) \cap \mathit{write}(v_j)) \cup (\mathit{write}(v_i) \cap \mathit{read}(v_j)) \cup (\mathit{write}(v_i) \cap \mathit{write}(v_j)) \neq \emptyset$. 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.

The interference graph provides not only a structure for the analysis of critical section behavior, but also an illustrative visualization of program behavior. An example of an interference graph is shown in Figure 3.9. This is the graph that is constructed

for the *traffic* benchmark using dynamic locks, Spark, MHP, and TLO. The graph has been enhanced with a visualization of an associated lock allocation (in this case, the dynamic allocation). Each boxed cluster is a locked component of the graph, while each lone node is an unlocked component. The grayed nodes represent unreachable code. The third locked component from the left uses a static lock named “lockobj3”, while each of the other components uses a dynamic lock of the indicated type. The dotted edge lines with labels ”D#” indicate dynamic locks, and the solid lines with labels ”S#” indicate static locks.

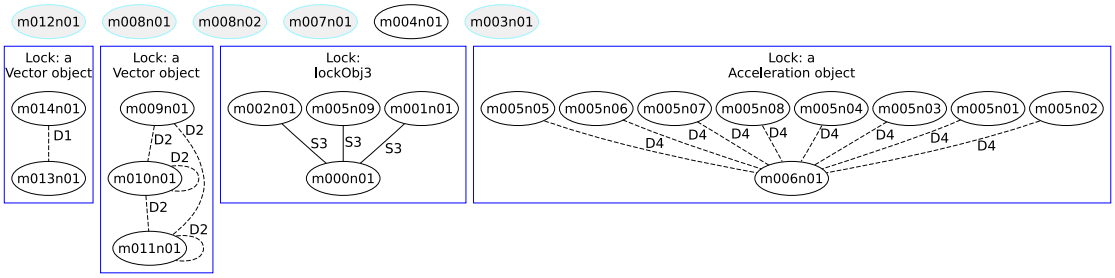


Figure 3.9: Interference graph for *traffic* using dynamic locking, Spark, MHP, TLO.

Applying MHP

Any pair of critical sections that cannot run in parallel cannot interfere with each other at runtime, so they need not share a common lock. This principle is applied by using the results of the may happen in parallel analysis described in Section 3.1.4 to rule out pairs of critical sections. For each such pair, the edge between them, if it exists, is removed from the interference graph. The effect of this application on the interference graph can be drastic for programs containing identifiable singleton threads.

Analyzing Existing Locks

Optionally, rather than analyzing the potential interferences between critical sections, our lock allocator can examine any previously existing locking structure in the pro-

gram. In this case, an interference graph is constructed based on the original locks in the program, with edges representing locks that are type compatible. Edges between locks whose points-to sets do not intersect are grayed. This is a very conservative approximation of the locking structure in the original program. The program is not transformed, lock allocation is not performed, and the interference graph is printed as the only output.

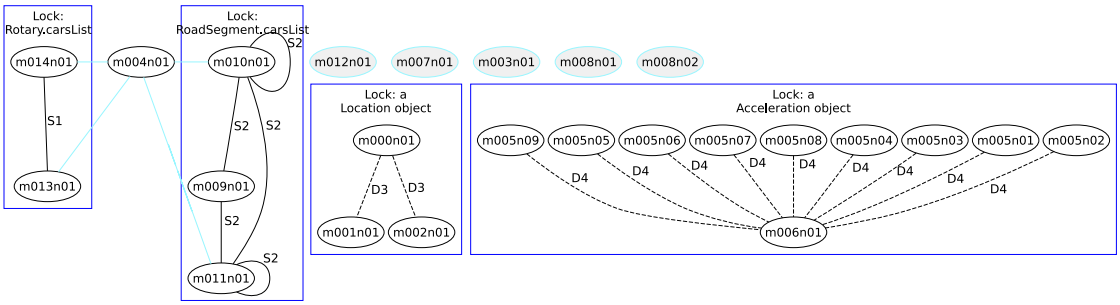


Figure 3.10: Lock analysis graph for *traffic*.

This non-allocating mode is useful, when analyzing legacy programs, for comparing manually written locking schemes against generated allocations. An example of a lock analysis interference graph is shown in Figure 3.10. This is the graph for the *traffic* benchmark. The graph shows the same four locked components that were found by the lock allocator in Figure 3.9, but with several small differences.

- The lock analysis finds that the locks in the first and second locked components are in fact static objects, which are the same objects that the allocator used local references to. The allocator conservatively estimated that these local references might not always refer to the same static object.

- Critical section `m005n09` appears in the fourth locked component instead of the third. This is a known locking defect in *traffic*: there is a data race between `m005n09` and `m000n01`, and there is no contention between `m005n09` and `m006n01`. The allocator correctly groups `m005n09` with the third locked component.

- The third locked component is found to use dynamic locks. This is a result of the previously discussed locking error in the original program.
- There are faded edges between `m004n01` and the first and second locked components. This is because the lock in `m004n01` is type-compatible with the locks in the first and second locked components, but the points-to analysis found that in fact these critical sections never lock the same objects. The faded edges are shown to indicate that the programmer may have believed that these critical sections share a lock.

3.2.5 Finding and Choosing Lock Objects

The completed interference graph yields a set of *locked components* which contain potentially interfering critical sections, and a set of *unlocked components* which are isolated critical sections without self-loops. Component-based lock allocation assigns locks per locked component, which may be *static*, instantiated once per program run, or *dynamic*, instantiated once per protected data structure. ALOCS allocates locks in one of these four 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 the entire locked component to static allocation if necessary.
- Lockset: A set of dynamic and/or static locks for each critical section, reverting the entire locked component to static allocation if necessary.

All four 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(s). Each synchronized method is replaced with an unsynchronized one containing a synchronized block. Calls to `wait()` and `notify()` are redirected to the new lock object (or innermost lock object) for the immediately

enclosing critical section. We insert `public static Object` fields to provide lock objects for all locks in the Singleton and Static granularities, and for any locks for which existing program objects are not available in the Dynamic and Lockset granularities.

Singleton Lock Allocation

Singleton allocation is trivial: the interference graph is ignored, and the same static lock is assigned to every critical section in the program. The composition of our analysis pipeline has no bearing on this naïve allocation.

Static Lock Allocation

Static allocation *does* depend on the interference graph, but the lock assignment process is straightforward: each locked component is assigned a different static lock, and synchronization is removed from unlocked components.

Dynamic Lock Allocation

Dynamic allocation builds on static allocation by allowing the use of a dynamic lock for locked components if one can be found. For a given locked component, each critical section must be *dynamically lockable*: it must have one object available on entry to which all reads and writes of the critical section are performed. The key difference between dynamic allocation and lockset allocation is that dynamic allocation uses a maximum of one lock per critical section.

To find such an object, the lockable reference analysis described in Section 3.1.5 is used on each critical section in the locked component. For a given critical section, if the list of lockable references that is returned is either empty (indicating that a complete list could not be found) or larger than one object, then the entire locked component reverts to the use of a static lock. Otherwise, each critical section is locked using the one object in its list.

This type of locking is called *a dynamic lock* because each critical section in the component locks the one object that it is working with. If two of the critical sections

attempt to read/write the same object, both will need to acquire the same lock to do so. This sufficiently guarantees non-interference.

Unlike static locks, dynamic locks require deadlock considerations within the locked component because a dynamic lock actually represents more than one possible runtime lock.

Lockset Allocation

Lockset allocation further builds on dynamic allocation by allowing the use of multiple lockable references as locks. If the lockable reference analysis succeeds for every critical section in the locked component, then locksets are used. Otherwise the entire locked component reverts to the use of a static lock.

A critical section's lockset initially contains all of the objects found by the lockable reference analysis. Like dynamic locks, locksets require deadlock considerations within the locked component, but they also require deadlock considerations within each lockset because the locksets contain more than one runtime lock. Additionally, some order must be selected for the locks in a lockset.

3.2.6 Detecting and Correcting Deadlock

The stages 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 [CES71]. Our lock allocator abides by a policy of minimal perturbance 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 other factors, 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 visible orderings. An ordering is considered visible unless all

of the critical sections associated with it share a static lock with the critical section currently under review. Critical sections that do share a static lock are prevented by that lock from happening in parallel, which is why their orderings are not visible to each other. 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.

For the static and dynamic allocations, when a deadlock avoidance edge is inserted, the locked components of the two vertices it joins are merged, and lock allocation restarts. For the lockset allocation, when a deadlock avoidance edge is inserted, a shared static lock is added to the beginning of the locksets of the two vertices it joins. The new static lock is ordered before the other locks in the locksets, so that those acquisitions cannot happen in parallel, and will no longer be visible to each other.

Reordering Locksets

For the lockset allocation, the result of deadlock detection and correction is a partial ordering P of all of the locks in the program. No specific ordering is required for the locks within each lockset, except that it must respect the order in P, and that the selected orders for locksets visible to each other cannot introduce new sources of deadlock. The order of locks in each lockset is constructed according to P, but chosen arbitrarily where P does not specify an order. Each arbitrarily chosen order is then added to P so that it will not be violated by the orders generated for other locksets.

Wait/Notify Deadlock

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 merging locks 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 for the singleton, static, or dynamic allocations.

One benchmark exhibits this behavior for the lockset allocation: in *xalan*, a static lock L is split into two dynamic locks l and m in both a method W calling `wait()` and a method N calling `notify()`. The two calls are associated to the lock m, but N is never able to reach its call because it cannot acquire l.

Detecting this type of deadlock is left as future work.

3.2.7 Transforming the Program

Like finding synchronized regions, inserting synchronized regions requires careful record-keeping and technique. ALOCS's transformation phase is responsible for inserting `public static Object` fields for use as static locks, and for creating synchronized regions for the lock(s) of each critical section. The transformer reuses existing code structures whenever possible, and inserts new ones if necessary.

The references in a lockset are those returned by LRA, and may be placeholders for a real reference that must be generated (called *lockable reference reconstruction*). The transformation stage is responsible for generating the correct lockable reference. Placeholder references are used for array references and field references, and contain a dummy value in place of the base and index. The dummy values contain a value number whose value is the real base/index. The transformation stage looks up the value numbers, and replaces the dummy value with the real value. In some cases, the real value may be another array reference or field reference, which may itself contain dummy values that need to be replaced. The statements required to reconstruct a lockable reference are inserted prior to the *entermonitor* statement of the lock region, as well as one statement that copies the final reference into a local variable. The same process is used to reconstruct dynamic locks as for lockset locks.

Static locks are treated differently; new `public static Object` fields are inserted into the program and assigned a singleton object. A statement is inserted before the *entermonitor* statement of the lock region to make a local variable reference to the public static Object.

Every critical section has an associated LockRegion record that catalogues its existing code structures. The first lock inserted for a given critical section uses that

LockRegion to determine which code structures may be reused, and which must be inserted and where. For subsequent locks, a new LockRegion is calculated from the previous one, as shown in Figure 3.11. The body of the LockRegion is kept the same (*beginning* and *last* remain the same), while the *after* statement and *early end* statements are redirected such that the machinery (*normal end*, *exceptional end*, etc.) of the new lock region is inserted inside the old one. LockRegion statements that are null must be created and inserted in locations relative to the non-null LockRegion statements. For example, a new *preparatory* statement and *entermonitor* statement must be inserted before the *beginning* statement, and new *exceptional catch, exitmonitor, end* statements must be inserted before the *after* statement.

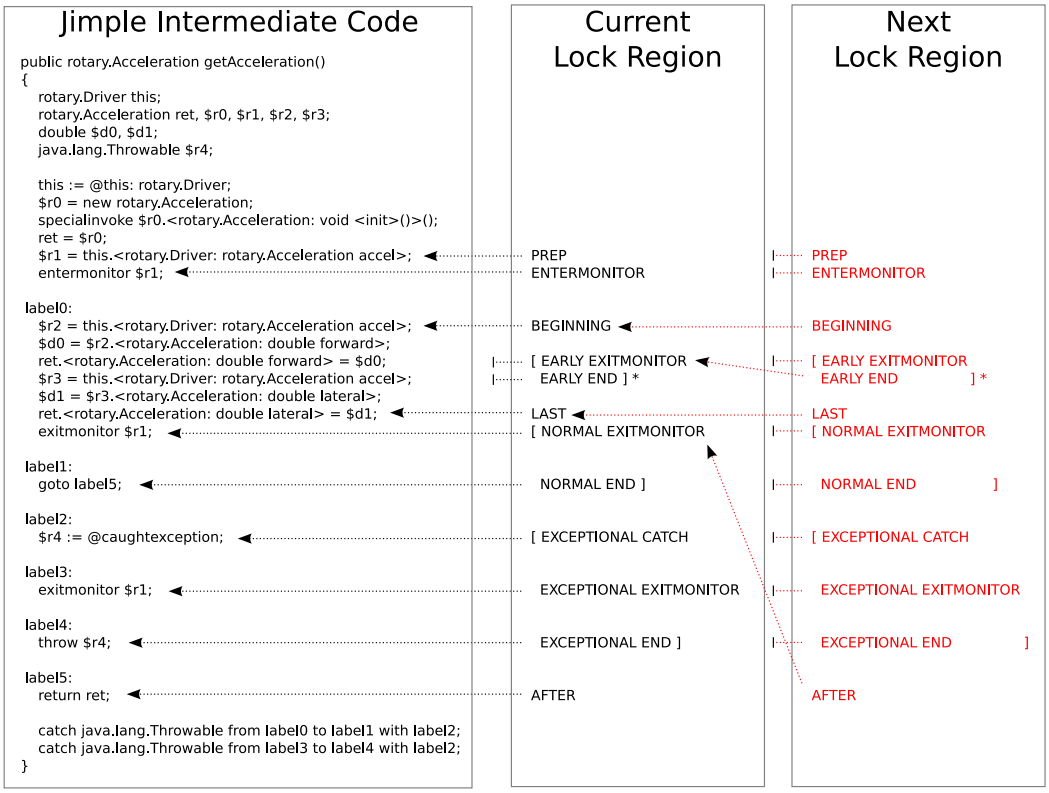


Figure 3.11: Calculating a nested lock region.

3.2.8 Output

Because our allocator is integrated into Soot, it can output the transformed program in a variety of formats. The default is `.class` files. The output program is compliant with the Java Bytecode Specification, and should run in any mature Java Virtual Machine.

Chapter 4
Compile-time Results

In this chapter we present compile-time results for our lock allocator. We transform twelve different benchmarks using twenty different configurations of our analysis pipeline and output phase for each. We present a set of interference graphs demonstrating the effects of each analysis and output phase, and we present complete allocation information for each benchmark for all pipeline configurations. Chapter 5 presents the runtime results for these experiments. Our benchmarks are described in Table 4.1.

4.1 Interference Graph Evolution

Figure 4.1 shows the significant differences between points-to analyses. While CHA and VTA find similarly conservative interference graphs, Spark finds a much more precise graph which is obscured by spurious edges in the others.

Though not necessarily as effective for all benchmarks, Figure 4.2 illustrates the significant gain in precision that MHP can sometimes provide. Note that the choice of points-to analysis is still the more significant factor in the resulting graphs. This benchmark's threads are mostly categorized as run-once or run-one-at-a-time, which allows edges between threads of the same type to be pruned. This causes a considerable "flattening" of the graph if the other analyses (like points-to) have already provided a sufficiently well-resolved graph.

Table 4.1: 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
heavy	24	4 car threads and 4 driver threads 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 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_{peak} through $2\times N_{peak}$ threads perform middleware operations	SPEC
jbb2005	187	CPU_{max} through $2\times CPU_{max}$ threads perform middleware operations	SPEC

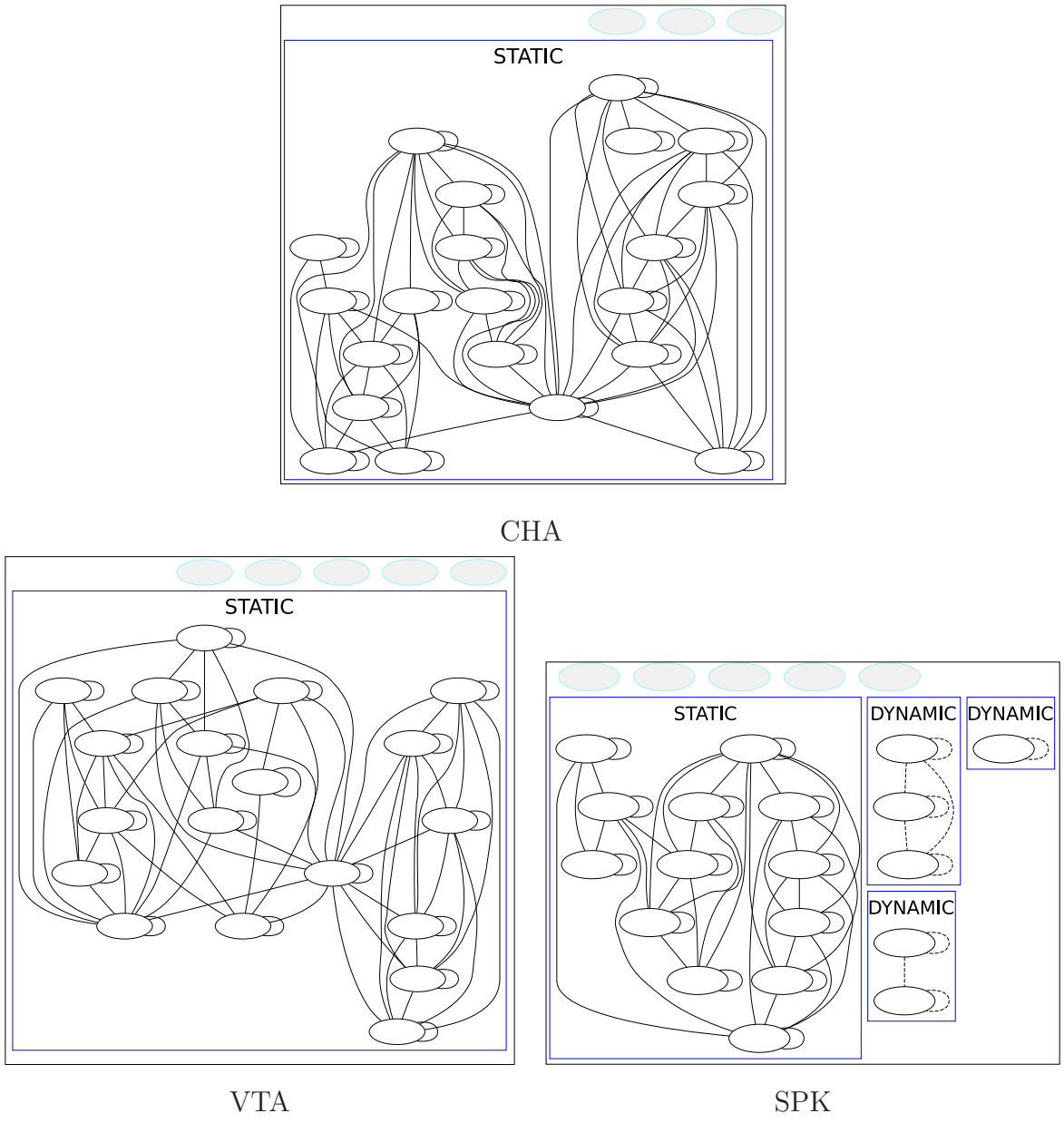


Figure 4.1: Lock allocations for *traffic* with three different points-to analyses.

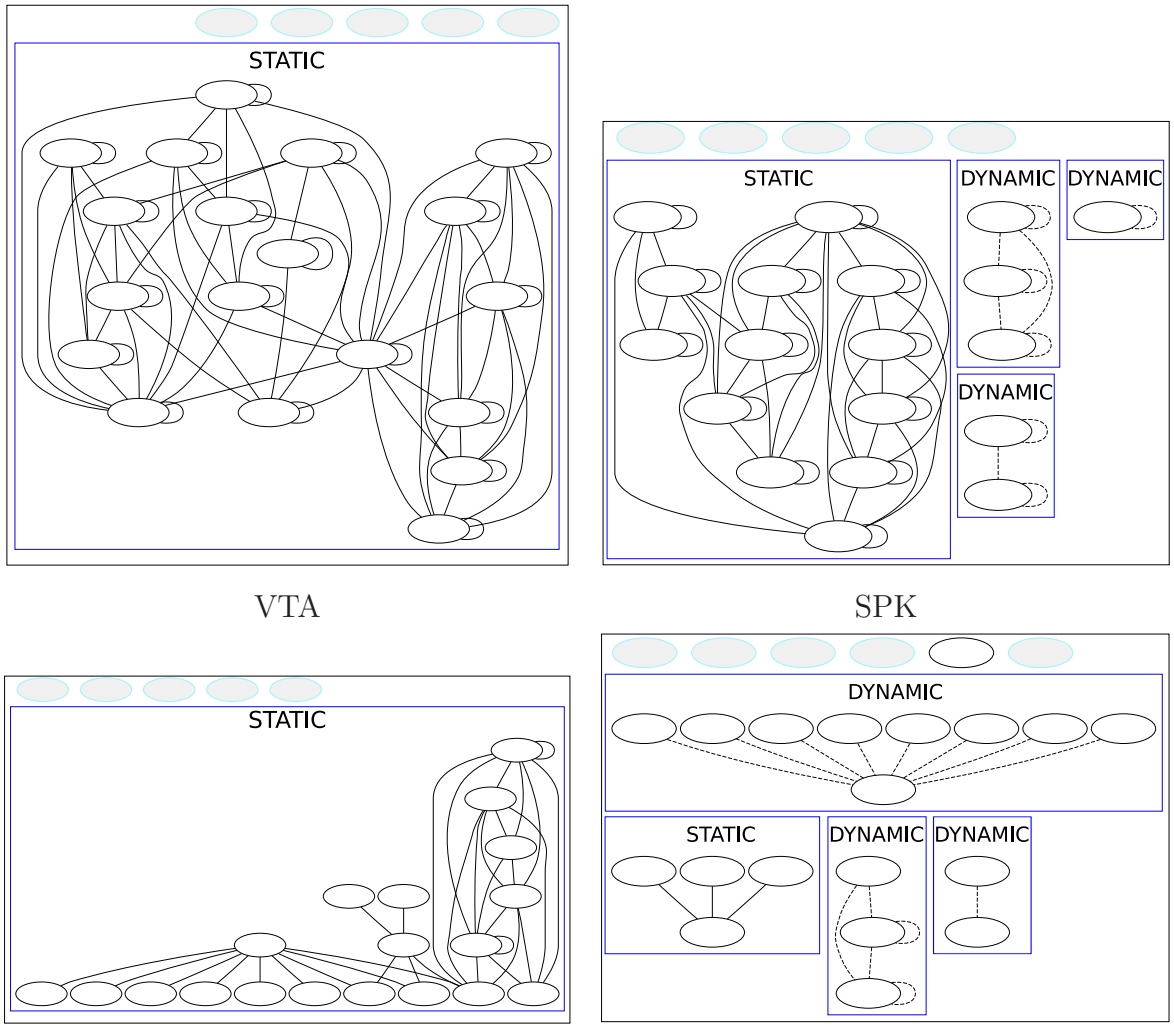


Figure 4.2: Lock allocations for *traffic* with the addition of MHP.

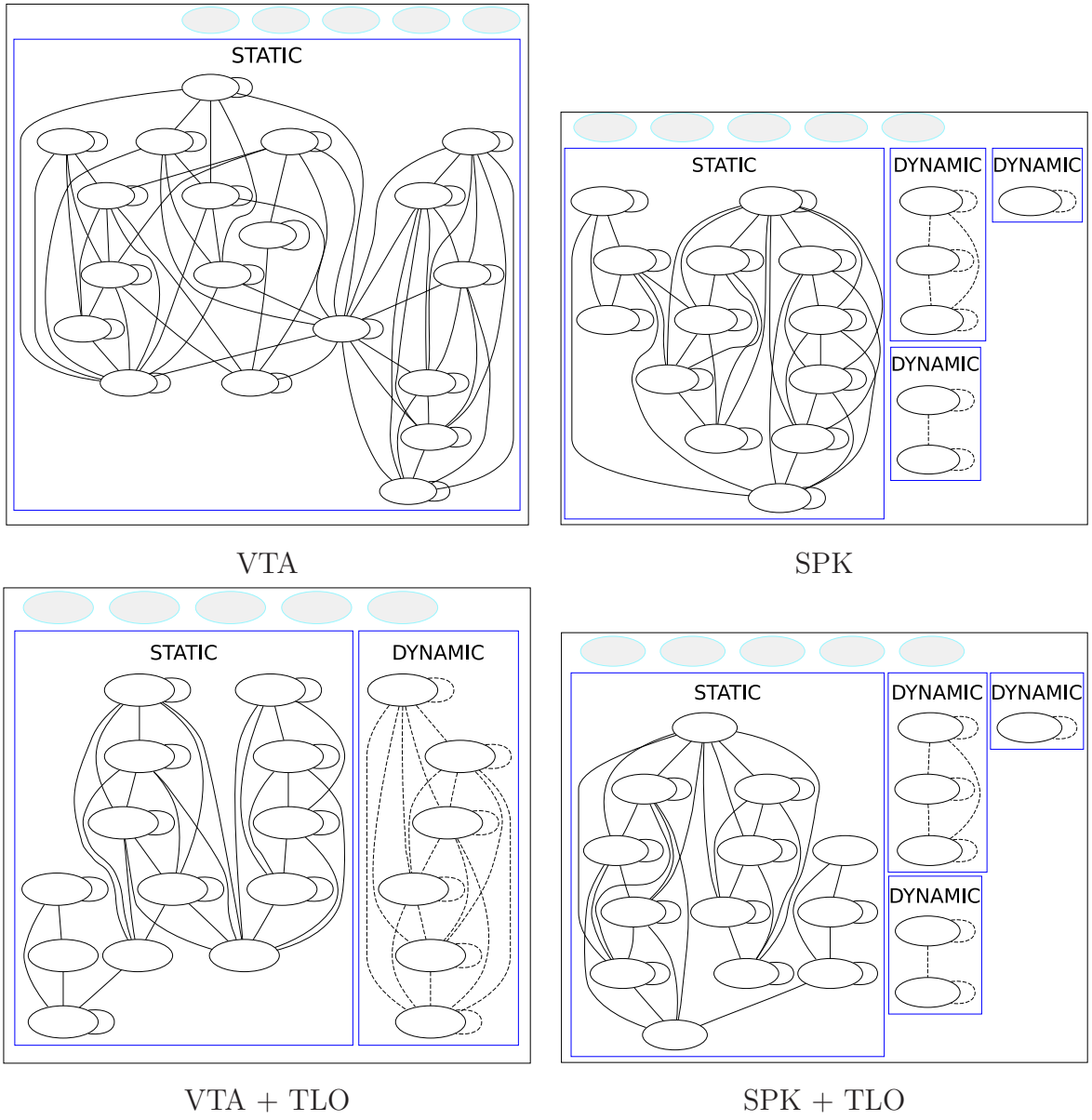
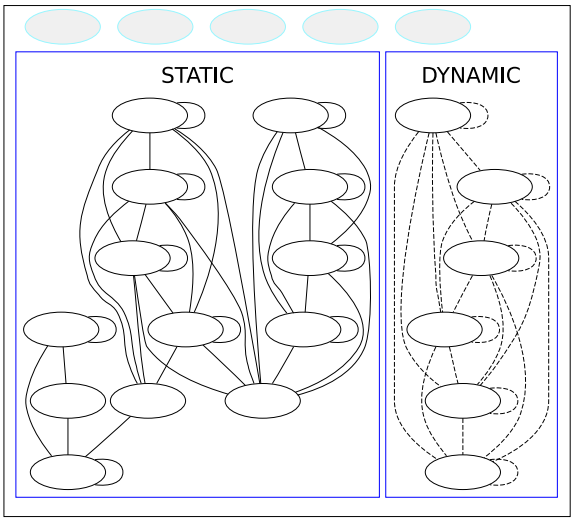
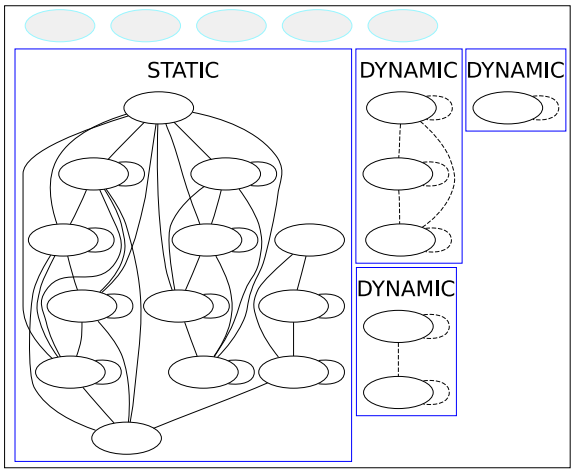


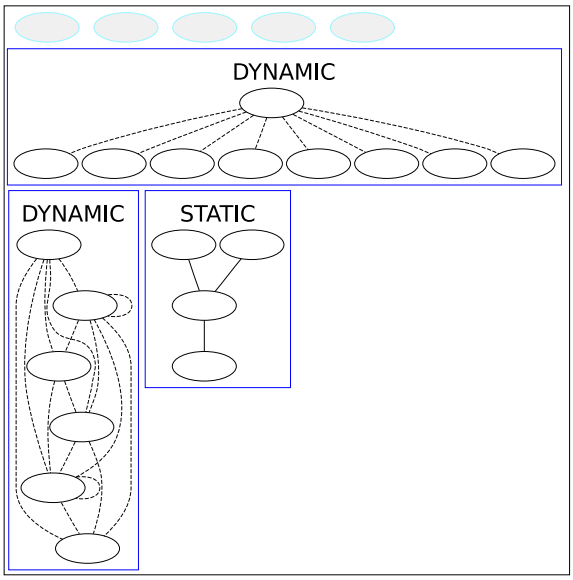
Figure 4.3: Lock allocations for *traffic* with the addition of TLO (without MHP).



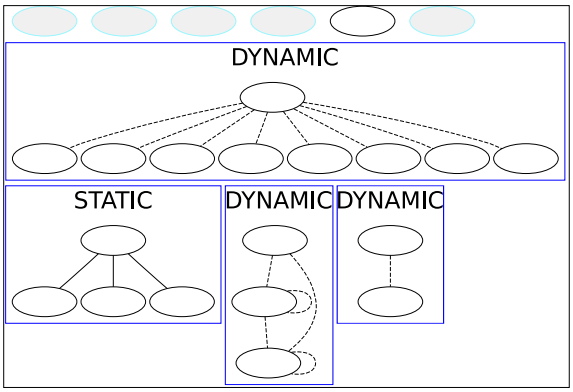
VTA + TLO



SPK + TLO



VTA + TLO + MHP



SPK + TLO + MHP

Figure 4.4: Lock allocations for *traffic* with the addition of both TLO and MHP.

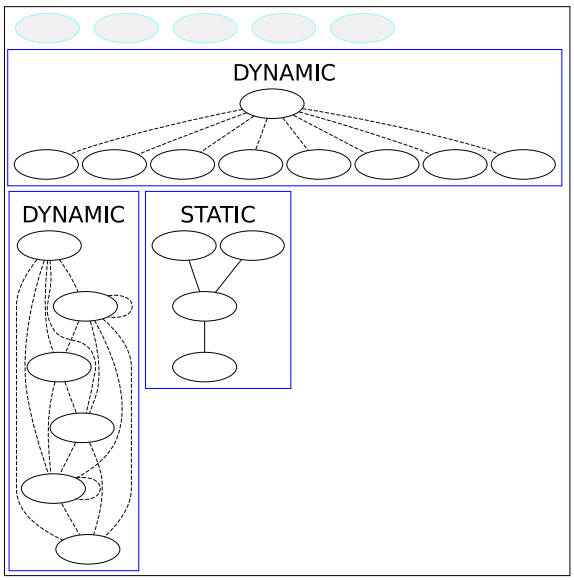
Figure 4.3 demonstrates that while TLO can have a significant effect on some interference graphs, those underpinned by a strong points-to analysis might already be too precise to benefit from TLO. However, we will later see that TLO can sometimes have a meaningful affect on Spark-based graphs. TLO's effectiveness against VTA-based graphs is undeniable, as it clearly recovers some of the precision of Spark over VTA. Again note that the choice of points-to analysis is still the more significant factor in the resulting graphs. In Figure 4.4, we see that the combination of MHP and TLO can sometimes be more precise than either is alone.

Figure 4.5 shows interference graphs with TLO and MHP using each of the three allocation types: static, dynamic, and lockset. These graphs illustrate that the components of the graph remain the same regardless of allocation - only the assigned locks change. A partial exception to this rule is that occasionally the lockable reference analysis assigns locks in a way that allows independence between parts of a component (see Figure 4.5). However, for the sake of analysis, allocation, deadlock avoidance, and output, the components are as indicated in the graph.

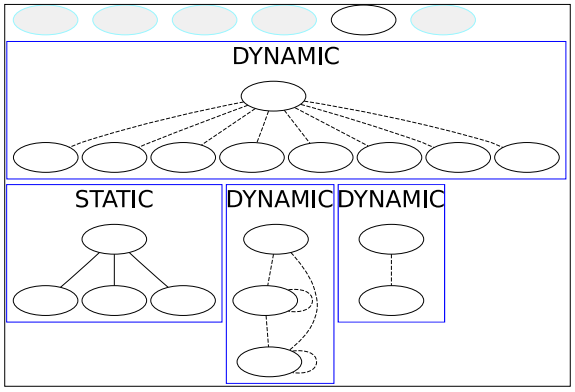
The static and dynamic graphs contain identical sets of edges. In the static case, those edges represent singleton objects to be used as locks. In the dynamic case, some of these edges instead represent local references to objects. The lockset case is different in that there may be more or fewer edges between nodes. These edges represent the locks of the locksets. Some pairs of nodes that share a static lock in the static and dynamic allocations share several dynamic locks in the lockset allocation. Similarly, some sets of nodes that share a static lock in the static and dynamic allocations share several partially or non-overlapping locks in the lockset allocation. In both situations, the lockset allocation potentially allows greater parallelism.

4.2 Interference Graph Characteristics and Allocations

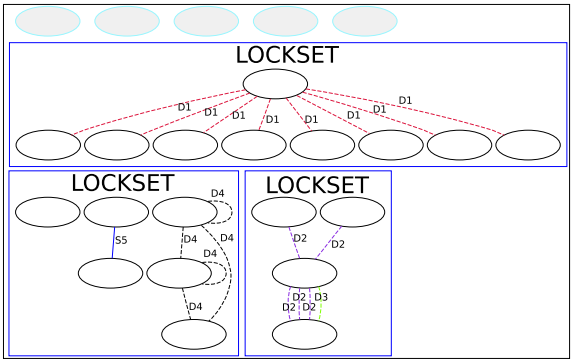
The complete set of allocations generated for each benchmark are shown in Tables 4.2 to 4.13 along with characteristics of the interference graphs from which they are generated. We discuss the changes resulting from using Spark (instead of VTA), from adding MHP, from adding TLO, and from using the lockset allocation (instead



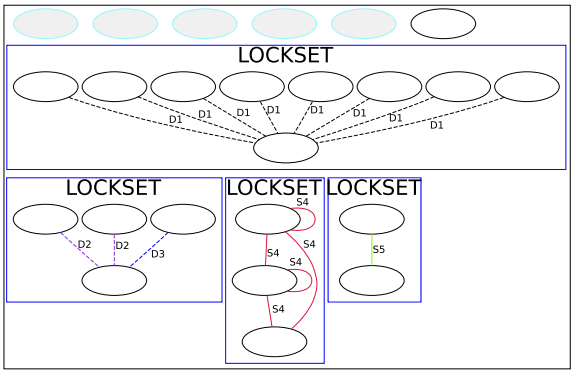
VTA + TLO + MHP (dynamic allocation)



SPK + TLO + MHP (dynamic allocation)



VTA + TLO + MHP (lockset allocation)



SPK + TLO + MHP (lockset allocation)

Figure 4.5: Lock allocations for *traffic* with and without locksets.

of the static or dynamic allocations).

We include a single measure of analysis time (single-threaded) 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*. The total interference graph edge weight for *hsqldb* is at least an order of magnitude larger than any of the other benchmarks, which means that hsqldb contains an unusually large number of fields for TLO to track. The per-field cost of our implementation is clearly suboptimal, but this is not an intrinsic characteristic of our algorithm. For the other benchmarks, we also find that TLO is more expensive than MHP, but not prohibitively so.

The gray rows of the *graph characteristics* column illustrate the effect of introducing new pipeline components on interference graph evolution. $|V|$ is constant for each benchmark, corresponding to the number of critical sections in Table 4.1, 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 $\frac{|V|+1}{2}$, a suitable graph complexity metric. The *weight* of any edge $e \in E$ is the number of fields involved in its contributing read/write set, and total graph weight, $\sum weight(e)$, indicates the size of the dynamic lock allocation problem.

The white rows of the *graph characteristics* column illustrates the effect that interference graph quality has on lock allocation. A connected set of vertices is a *locked* component, and an isolated vertex with no self loop is an *unlocked* component. The allocator assigns either one static or dynamic lock to each component, or a set of locks to each component for the lockset allocation. The static allocation includes only statically locked components. For the dynamic allocation, dynamic locks are used where they can be found. For the lockset allocation, multiple locks are used where they can be found. A set of N locked components C is shown as N:[$|C_1| \ldots |C_N|$]. For example, the *spk-** configurations of *roller* have three statically locked components with two critical sections each, shown as 3:[2 2 2]. For lockset components, more than one lock may be used for some critical sections. These components are listed as a fraction: the total number of locks for all critical sections over the number of critical sections.

For *sync*, MHP removes a static component and significantly reduces total graph

Table 4.2: Static results for *sync*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**12s**	**16**	**0**	**0**	**0**
singleton allocation			1	1:[16]	-	-	0
VTA			**77s**	**16**	**68**	**4.250**	**608**
static allocation			8	2:[2 8]	-	-	6
dynamic allocation			8	1:[8]	1:[2]	-	6
lockset allocation			8	1:[8]	-	1:[2]	6
VTA		**X**	**75s**	**16**	**4**	**.250**	**4**
static allocation			15	1:[2]	-	-	14
dynamic allocation			15	0	1:[2]	-	14
lockset allocation			15	0	-	1:[2]	14
VTA	**X**	**X**	**77s**	**16**	**4**	**.250**	**4**
static allocation			15	1:[2]	-	-	14
dynamic allocation			15	0	1:[2]	-	14
lockset allocation			15	0	-	1:[2]	14
SPK			**59s**	**16**	**66**	**4.125**	**606**
static allocation			9	3:[1 1 8]	-	-	6
dynamic allocation			9	1:[8]	2:[1 1]	-	6
lockset allocation			9	1:[8]	-	2:[1 1]	6
SPK		**X**	**61s**	**16**	**2**	**.125**	**2**
static allocation			16	2:[1 1]	-	-	14
dynamic allocation			16	0	2:[1 1]	-	14
lockset allocation			16	0	-	2:[1 1]	14
SPK	**X**	**X**	**62s**	**16**	**2**	**.125**	**2**
static allocation			16	2:[1 1]	-	-	14
dynamic allocation			16	0	2:[1 1]	-	14
lockset allocation			16	0	-	2:[1 1]	14

Table 4.3: Static results for *pcmab*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**10s**	**2**	**0**	**0**	**0**
singleton allocation			1	1:[2]	-	-	0
VTA			**76s**	**2**	**4**	**2.000**	**13**
static allocation			1	1:[2]	-	-	0
dynamic allocation			1	1:[2]	0	-	0
lockset allocation			1	0	-	1:[$\frac{4}{2}$]	0
VTA		**X**	**74s**	**2**	**4**	**2.000**	**13**
static allocation			1	1:[2]	-	-	0
dynamic allocation			1	1:[2]	0	-	0
lockset allocation			1	0	-	1:[$\frac{4}{2}$]	0
VTA	**X**	**X**	**74s**	**2**	**4**	**2.000**	**12**
static allocation			1	1:[2]	-	-	0
dynamic allocation			1	1:[2]	0	-	0
lockset allocation			1	0	-	1:[$\frac{3}{2}$]	0
SPK			**55s**	**2**	**4**	**2.000**	**13**
static allocation			1	1:[2]	-	-	0
dynamic allocation			1	1:[2]	0	-	0
lockset allocation			1	0	-	1:[$\frac{4}{2}$]	0
SPK		**X**	**57s**	**2**	**4**	**2.000**	**13**
static allocation			1	1:[2]	-	-	0
dynamic allocation			1	1:[2]	0	-	0
lockset allocation			1	0	-	1:[$\frac{4}{2}$]	0
SPK	**X**	**X**	**58s**	**2**	**4**	**2.000**	**12**
static allocation			1	1:[2]	-	-	0
dynamic allocation			1	1:[2]	0	-	0
lockset allocation			1	0	-	1:[$\frac{3}{2}$]	0

Table 4.4: Static results for *roller*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**11s**	**6**	**0**	**0**	**0**
singleton allocation			1	1:[6]	-	-	0
VTA			**76s**	**6**	**20**	**3.333**	**318**
static allocation			2	2:[2 4]	-	-	0
dynamic allocation			2	1:[4]	1:[2]	-	0
lockset allocation			2	1:[4]	-	1:[2]	0
VTA		**X**	**76s**	**6**	**16**	**2.666**	**240**
static allocation			2	2:[2 4]	-	-	0
dynamic allocation			2	1:[4]	1:[2]	-	0
lockset allocation			2	1:[4]	-	1:[2]	0
VTA	**X**	**X**	**76s**	**6**	**9**	**1.500**	**12**
static allocation			3	3:[2 2 2]	-	-	0
dynamic allocation			3	2:[2 2]	1:[2]	-	0
lockset allocation			3	2:[2 2]	-	1:[2]	0
SPK			**60s**	**6**	**12**	**2.000**	**166**
static allocation			3	3:[2 2 2]	-	-	0
dynamic allocation			3	2:[2 2]	1:[2]	-	0
lockset allocation			3	2:[2 2]	-	1:[2]	0
SPK		**X**	**59s**	**6**	**10**	**1.666**	**126**
static allocation			3	3:[2 2 2]	-	-	0
dynamic allocation			3	2:[2 2]	1:[2]	-	0
lockset allocation			3	2:[2 2]	-	1:[2]	0
SPK	**X**	**X**	**59s**	**6**	**9**	**1.500**	**12**
static allocation			3	3:[2 2 2]	-	-	0
dynamic allocation			3	2:[2 2]	1:[2]	-	0
lockset allocation			3	2:[2 2]	-	1:[2]	0

Table 4.5: Static results for *mtrt*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**26s**	**6**	**0**	**0**	**0**
singleton allocation			1	1:[6]	-	-	0
VTA			**426s**	**6**	**11**	**1.833**	**238**
static allocation			3	1:[4]	-	-	2
dynamic allocation			3	1:[4]	0	-	2
lockset allocation			3	1:[4]	-	0	2
VTA		**X**	**410s**	**6**	**1**	**.166**	**97**
static allocation			6	1:[1]	-	-	5
dynamic allocation			6	1:[1]	0	-	5
lockset allocation			6	1:[1]	-	0	5
VTA	**X**	**X**	**411s**	**6**	**1**	**.166**	**31**
static allocation			6	1:[1]	-	-	5
dynamic allocation			6	1:[1]	0	-	5
lockset allocation			6	1:[1]	-	0	5
SPK			**78s**	**6**	**11**	**1.833**	**238**
static allocation			3	1:[4]	-	-	2
dynamic allocation			3	1:[4]	0	-	2
lockset allocation			3	1:[4]	-	0	2
SPK		**X**	**78s**	**6**	**1**	**.166**	**97**
static allocation			6	1:[1]	-	-	5
dynamic allocation			6	1:[1]	0	-	5
lockset allocation			6	1:[1]	-	0	5
SPK	**X**	**X**	**82s**	**6**	**1**	**.166**	**31**
static allocation			6	1:[1]	-	-	5
dynamic allocation			6	1:[1]	0	-	5
lockset allocation			6	1:[1]	-	0	5

weight, and Spark separates two simlar but distinct locks into different components. While none of the analyses significantly simplifies the already simple interference graph for *pcmab*, TLO does allow one unnecessary critical section to be unlocked. For *roller*, TLO has the most dramatic effect on total graph weight, and either SPK or TLO is sufficient to split one component into two. In contrast, Spark and TLO have little effect on the interference graph and allocations of *mtrt*. However, MHP reduces graph weight and density, thereby identifying several critical sections that do not require locking.

Both *hsqldb* and *xalan* benefit from the use of Spark and MHP. Notably, the lockset allocation is able to replace a static component with a lockset component for both.

lusearch and *traffic* both benefit from the introduction of Spark and MHP, with *lusearch* depending more on the former, and *traffic* on the latter. *lusearch* is further improved by TLO, while *traffic* is further improved by lockset allocation.

Spark has an unusually small affect on the weight and density of *jbb2005*'s graph, and likewise splits fewer components than usual. MHP, on the other hand, has an especially drastic affect on the weight and density of the graph, and also on the total number of components that are locked. More typically, *heavy* is significantly improved by Spark, though not by MHP and TLO. Lockset allocation replaces the largest static component with multiple dynamic locks.

Lockset allocation breaks up a component of *bank*, but none of the other analyses improve the allocation, and none of them improve the interference graph weight or density. In contrast, for *jbb2000*, all of the analyses split and remove components.

Most benchmarks exhibit reduced graph density and weight as more sophisticated analyses are introduced, although not necessarily at every stage. In general, switching from VTA to Spark has the largest effect, and including MHP has the second largest effect.

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 weight should intuitively allow for the number of dynamic locks to increase, in practice these compete with an increasing number of unlocked components.

Table 4.6: Static results for *hsqldb*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**134s**	**269**	**0**	**0**	**0**
singleton allocation			1	1:[269]	-	-	0
VTA			**664s**	**269**	**10694**	**39.754**	**371212**
static allocation			95	10:[1 1 1 1 1 2 2 2 2 171]	-	-	85
dynamic allocation			95	5:[1 1 1 2 171]	5:[1 1 2 2 2]	-	85
lockset allocation			95	3:[1 2 171]	-	7:[1 1 1 1 2 2 2]	85
VTA		**x**	**846s**	**269**	**10476**	**38.944**	**370237**
static allocation			97	9:[1 1 1 2 2 2 2 6 164]	-	-	88
dynamic allocation			97	4:[1 2 6 164]	5:[1 1 2 2 2]	-	88
lockset allocation			97	4:[1 2 6 164]	-	5:[1 1 2 2 2]	88
SPK			**361s**	**269**	**6191**	**23.014**	**190534**
static allocation			121	7:[1 1 1 1 2 4 145]	-	-	114
dynamic allocation			121	4:[1 1 4 145]	3:[1 1 2]	-	114
lockset allocation			121	1:[145]	-	6:[$\frac{7}{4}$ 1 1 1 1 2]	114
SPK		**x**	**405s**	**269**	**5201**	**19.334**	**160842**
static allocation			130	5:[1 1 2 4 136]	-	-	125
dynamic allocation			130	2:[4 136]	3:[1 1 2]	-	125
lockset allocation			130	1:[136]	-	4:[$\frac{7}{4}$ 1 1 2]	125
SPK	**x**	**x**	**29768s**	**269**	**4585**	**17.044**	**134385**
static allocation			135	6:[1 1 2 2 4 130]	-	-	129
dynamic allocation			135	3:[2 4 130]	3:[1 1 2]	-	129
lockset allocation			135	2:[2 130]	-	4:[$\frac{7}{4}$ 1 1 2]	129

Table 4.7: Static results for *xalan*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**84s**	**73**	**0**	**0**	**0**
singleton allocation			1	1:[73]	-	-	0
VTA			**582s**	**73**	**249**	**3.410**	**12771**
static allocation			52	4:[1 1 2 21]	-	-	48
dynamic allocation			52	4:[1 1 2 21]	0	-	48
lockset allocation			52	1:[21]	-	3:[$\frac{4}{2}$ 1 1]	48
VTA		**X**	**557s**	**73**	**195**	**2.671**	**11224**
static allocation			55	2:[2 18]	-	-	53
dynamic allocation			55	2:[2 18]	0	-	53
lockset allocation			55	1:[18]	-	1:[$\frac{4}{2}$]	53
VTA	**X**	**X**	**1330s**	**73**	**166**	**2.273**	**7751**
static allocation			57	3:[1 2 16]	-	-	54
dynamic allocation			57	2:[2 16]	1:[1]	-	54
lockset allocation			57	1:[16]	-	2:[$\frac{4}{2}$ 1]	54
SPK			**155s**	**73**	**6**	**.082**	**22**
static allocation			72	3:[1 1 2]	-	-	69
dynamic allocation			72	3:[1 1 2]	0	-	69
lockset allocation			72	0	-	3:[$\frac{4}{2}$ 1 1]	69
SPK		**X**	**156s**	**73**	**3**	**.041**	**15**
static allocation			72	1:[2]	-	-	71
dynamic allocation			72	1:[2]	0	-	71
lockset allocation			72	0	-	1:[$\frac{4}{2}$]	71
SPK	**X**	**X**	**183s**	**73**	**3**	**.041**	**15**
static allocation			72	1:[2]	-	-	71
dynamic allocation			72	1:[2]	0	-	71
lockset allocation			72	0	-	1:[$\frac{4}{2}$]	71

Table 4.8: Static results for *lusearch*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**45s**	**88**	**0**	**0**	**0**
singleton allocation			1	1:[88]	-	-	0
VTA			**179s**	**88**	**287**	**3.261**	**2645**
static allocation			60	4:[1 1 2 28]	-	-	56
dynamic allocation			60	3:[1 1 28]	1:[2]	-	56
lockset allocation			60	1:[28]	-	3:[1 1 2]	56
VTA		**X**	**181s**	**88**	**250**	**2.840**	**2318**
static allocation			62	2:[2 26]	-	-	60
dynamic allocation			62	1:[26]	1:[2]	-	60
lockset allocation			62	1:[26]	-	1:[2]	60
VTA	**X**	**X**	**264s**	**88**	**210**	**2.386**	**1911**
static allocation			64	3:[2 2 23]	-	-	61
dynamic allocation			64	1:[23]	2:[2 2]	-	61
lockset allocation			64	1:[23]	-	2:[2 2]	61
SPK			**92s**	**88**	**86**	**.977**	**774**
static allocation			74	6:[1 1 1 1 6 10]	-	-	68
dynamic allocation			74	5:[1 1 1 6 10]	1:[1]	-	68
lockset allocation			74	3:[1 6 10]	-	3:[1 1 1]	68
SPK		**X**	**94s**	**88**	**73**	**.829**	**628**
static allocation			74	3:[1 6 10]	-	-	71
dynamic allocation			74	3:[1 6 10]	0	-	71
lockset allocation			74	3:[1 6 10]	-	0	71
SPK	**X**	**X**	**122s**	**88**	**59**	**.670**	**444**
static allocation			77	4:[1 2 6 6]	-	-	73
dynamic allocation			77	3:[1 6 6]	1:[2]	-	73
lockset allocation			77	3:[1 6 6]	-	1:[2]	73

Table 4.9: Static results for *traffic*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**16s**	**24**	**0**	**0**	**0**
singleton allocation			1	1:[24]	-	-	0
VTA			**79s**	**24**	**129**	**5.375**	**290**
static allocation			6	1:[19]	-	-	5
dynamic allocation			6	1:[19]	0	-	5
lockset allocation			6	0	-	1:[$\frac{40}{19}$]	5
VTA		**X**	**78s**	**24**	**66**	**2.750**	**186**
static allocation			6	1:[19]	-	-	5
dynamic allocation			6	1:[19]	0	-	5
lockset allocation			6	0	-	1:[$\frac{40}{19}$]	5
VTA	**X**	**X**	**103s**	**24**	**52**	**2.166**	**148**
static allocation			8	3:[4 6 9]	-	-	5
dynamic allocation			8	1:[4]	2:[6 9]	-	5
lockset allocation			8	0	-	3:[$\frac{8}{4}$ 6 9]	5
SPK			**62s**	**24**	**89**	**3.708**	**146**
static allocation			9	4:[1 2 3 13]	-	-	5
dynamic allocation			9	1:[13]	3:[1 2 3]	-	5
lockset allocation			9	0	-	4:[$\frac{34}{13}$ 1 2 3]	5
SPK		**X**	**88s**	**24**	**32**	**1.333**	**66**
static allocation			10	4:[2 3 4 9]	-	-	6
dynamic allocation			10	1:[4]	3:[2 3 9]	-	6
lockset allocation			10	0	-	4:[$\frac{5}{4}$ 2 3 9]	6
SPK	**X**	**X**	**90s**	**24**	**32**	**1.333**	**66**
static allocation			10	4:[2 3 4 9]	-	-	6
dynamic allocation			10	1:[4]	3:[2 3 9]	-	6
lockset allocation			10	0	-	4:[$\frac{5}{4}$ 2 3 9]	6

Table 4.10: Static results for *jbb2005*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum$ ***weight***(e)
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**42s**	**187**	**0**	**0**	**0**
singleton allocation			1	1:[187]	-	-	0
VTA			**436s**	**187**	**710**	**3.796**	**3068**
static allocation			108	12:[1 1 1 1 1 1 1 2 2 2 2 76]	-	-	96
dynamic allocation			108	2:[2 76]	10:[1 1 1 1 1 1 1 2 2 2]	-	96
lockset allocation			108	1:[76]	-	11:[$\frac{4}{2}$ 1 1 1 1 1 1 1 2 2 2]	96
VTA		**X**	**461s**	**187**	**119**	**.636**	**131**
static allocation			152	6:[2 2 2 3 8 24]	-	-	146
dynamic allocation			152	3:[3 8 24]	3:[2 2 2]	-	146
lockset allocation			152	1:[24]	-	5:[$\frac{4}{3}$ $\frac{9}{8}$ 2 2 2]	146
VTA	**X**	**X**	**553s**	**187**	**25**	**.133**	**29**
static allocation			176	3:[2 4 8]	-	-	173
dynamic allocation			176	3:[2 4 8]	0	-	173
lockset allocation			176	0	-	3:[$\frac{11}{4}$ $\frac{3}{2}$ $\frac{9}{8}$]	173
SPK			**138s**	**187**	**560**	**2.994**	**2414**
static allocation			110	14:[1 1 1 1 1 1 1 1 2 2 2 2 2 6 69]	-	-	96
dynamic allocation			110	3:[2 6 69]	11:[1 1 1 1 1 1 1 1 2 2 2]	-	96
lockset allocation			110	2:[6 69]	-	12:[$\frac{4}{2}$ 1 1 1 1 1 1 1 1 2 2 2]	96
SPK		**X**	**137s**	**187**	**75**	**.401**	**85**
static allocation			156	7:[2 2 2 2 4 8 18]	-	-	149
dynamic allocation			156	3:[4 8 18]	4:[2 2 2 2]	-	149
lockset allocation			156	1:[18]	-	6:[$\frac{7}{4}$ $\frac{9}{8}$ 2 2 2 2]	149
SPK	**X**	**X**	**444s**	**187**	**21**	**.112**	**25**
static allocation			177	3:[2 3 8]	-	-	174
dynamic allocation			177	2:[3 8]	1:[2]	-	174
lockset allocation			177	0	-	3:[$\frac{7}{3}$ $\frac{9}{8}$ 2]	174

Table 4.11: Static results for *heavy*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**16s**	**24**	**0**	**0**	**0**
singleton allocation			1	1:[24]	-	-	0
VTA			**78s**	**24**	**129**	**5.375**	**290**
static allocation			6	1:[19]	-	-	5
dynamic allocation			6	1:[19]	0	-	5
lockset allocation			6	0	-	1:[$\frac{40}{19}$]	5
VTA		**X**	**103s**	**24**	**128**	**5.333**	**286**
static allocation			6	1:[19]	-	-	5
dynamic allocation			6	1:[19]	0	-	5
lockset allocation			6	0	-	1:[$\frac{40}{19}$]	5
VTA	**X**	**X**	**108s**	**24**	**101**	**4.208**	**215**
static allocation			7	2:[6 13]	-	-	5
dynamic allocation			7	1:[13]	1:[6]	-	5
lockset allocation			7	0	-	2:[$\frac{17}{13}$ 6]	5
SPK			**61s**	**24**	**89**	**3.708**	**146**
static allocation			9	4:[1 2 3 13]	-	-	5
dynamic allocation			9	1:[13]	3:[1 2 3]	-	5
lockset allocation			9	0	-	4:[$\frac{34}{13}$ 1 2 3]	5
SPK		**X**	**89s**	**24**	**88**	**3.666**	**142**
static allocation			9	4:[1 2 3 13]	-	-	5
dynamic allocation			9	1:[13]	3:[1 2 3]	-	5
lockset allocation			9	0	-	4:[$\frac{34}{13}$ 1 2 3]	5
SPK	**X**	**X**	**89s**	**24**	**79**	**3.291**	**125**
static allocation			9	4:[1 2 3 13]	-	-	5
dynamic allocation			9	1:[13]	3:[1 2 3]	-	5
lockset allocation			9	0	-	4:[$\frac{14}{13}$ 1 2 3]	5

Table 4.12: Static results for *bank*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum \textbf{\textit{weight}}(e)$
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**20s**	**8**	**0**	**0**	**0**
singleton allocation			1	1:[8]	-	-	0
VTA			**98s**	**8**	**12**	**1.500**	**12**
static allocation			4	2:[2 4]	-	-	2
dynamic allocation			4	1:[4]	1:[2]	-	2
lockset allocation			5	0	-	3:[2 2 2]	2
VTA		**X**	**99s**	**8**	**12**	**1.500**	**12**
static allocation			4	2:[2 4]	-	-	2
dynamic allocation			4	1:[4]	1:[2]	-	2
lockset allocation			5	0	-	3:[2 2 2]	2
VTA	**X**	**X**	**96s**	**8**	**12**	**1.500**	**12**
static allocation			4	2:[2 4]	-	-	2
dynamic allocation			4	1:[4]	1:[2]	-	2
lockset allocation			5	0	-	3:[2 2 2]	2
SPK			**77s**	**8**	**12**	**1.500**	**12**
static allocation			4	2:[2 4]	-	-	2
dynamic allocation			4	1:[4]	1:[2]	-	2
lockset allocation			5	0	-	3:[2 2 2]	2
SPK		**X**	**81s**	**8**	**12**	**1.500**	**12**
static allocation			4	2:[2 4]	-	-	2
dynamic allocation			4	1:[4]	1:[2]	-	2
lockset allocation			5	0	-	3:[2 2 2]	2
SPK	**X**	**X**	**79s**	**8**	**12**	**1.500**	**12**
static allocation			4	2:[2 4]	-	-	2
dynamic allocation			4	1:[4]	1:[2]	-	2
lockset allocation			5	0	-	3:[2 2 2]	2

Table 4.13: Static results for *jbb2000*.

ANALYSIS PIPELINE			GRAPH CHARACTERISTICS				
PTA	**TLO**	**MHP**	**time**	$\|V\|$	$\|E\|$	$\frac{\|E\|}{\|V\|}$	$\sum$ ***weight***(e)
			total	statically locked	dynamically locked	lockset locked	unlocked
NA			**42s**	**241**	**0**	**0**	**0**
singleton allocation			1	1:[241]	-	-	0
VTA			**441s**	**241**	**460**	**1.908**	**2578**
static allocation			200	12:[1 1 1 1 1 1 2 2 2 2 5 34]	-	-	188
dynamic allocation			200	3:[2 5 34]	9:[1 1 1 1 1 1 2 2 2]	-	188
lockset allocation			200	1:[34]	-	11:[$\frac{4}{2}$ $\frac{8}{5}$ 1 1 1 1 1 1 2 2 2]	188
VTA		**X**	**482s**	**241**	**407**	**1.688**	**2353**
static allocation			209	6:[1 1 1 2 2 31]	-	-	203
dynamic allocation			209	1:[31]	5:[1 1 1 2 2]	-	203
lockset allocation			209	1:[31]	-	5:[1 1 1 2 2]	203
VTA	**X**	**X**	**1395s**	**241**	**141**	**.585**	**808**
static allocation			223	5:[1 1 1 1 19]	-	-	218
dynamic allocation			223	1:[19]	4:[1 1 1 1]	-	218
lockset allocation			223	1:[19]	-	4:[1 1 1 1]	218
SPK			**105s**	**241**	**163**	**.676**	**1304**
static allocation			225	3:[2 4 13]	-	-	222
dynamic allocation			225	3:[2 4 13]	0	-	222
lockset allocation			225	2:[4 13]	-	1:[$\frac{4}{2}$]	222
SPK		**X**	**109s**	**241**	**150**	**.622**	**1171**
static allocation			227	2:[3 13]	-	-	225
dynamic allocation			227	2:[3 13]	0	-	225
lockset allocation			227	1:[13]	-	1:[$\frac{5}{3}$]	225
SPK	**X**	**X**	**478s**	**241**	**44**	**.182**	**169**
static allocation			233	3:[1 3 7]	-	-	230
dynamic allocation			233	2:[3 7]	1:[1]	-	230
lockset allocation			233	1:[7]	-	2:[$\frac{5}{3}$ 1]	230

Over twelve benchmarks, Spark has an impact on the lock allocations of nine, MHP on eight, and TLO on six. The lockset allocation converts static components to lockset components in five benchmarks.

Ten 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, have data dependence 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 [BKMS98, KKO02, RD06]. Unlocked components are often found in instrumentation and harness code associated with benchmark suites, or in library code, which tends to be very conservatively locked, and only partially used. The primary benefit of synchronization elimination is a more fine-grained allocation for the remaining locked components.

Chapter 5
Runtime Results

In this chapter we present runtime results for our lock allocator. Our goal is to demonstrate the extent to which this work is able to remove the burden of lock allocation from the programmer without sacrificing performance. We aim to match the performance of normal, manually assigned allocations, and to beat the performance of allocations that are incorrect or naive. Our strategy is to take legacy programs, eliminate the lock information, assign locks automatically using ALOCS, and compare the resulting program to the original.

We transform twelve different benchmarks using twenty different configurations of our analysis pipeline and allocation phase for each, and we test the performance and scalability of each by running them on one to eight processors of an 8-way x86_64 machine.

We find that runtime results for these twelve benchmarks validate our approach of top-down, component-based lock allocation, and demonstrate the necessity of thread-specific analyses like TLO and MHP. We additionally find that there is still progress to be made in the analysis of certain particularly complex benchmarks. Finally, we posit that the absence of solid evidence for the necessity of dynamic and lockset allocations is an indication that the available multithreaded Java benchmarks are not representative of the large class of casual and/or desktop Java programmers for whom lock allocation would be most relevant. We also find that most of the benchmarks do not scale well up to eight processors, which may be due more to benchmark workloads

than to their code architectures.

All experiments were run on an 8-way machine with two 1.6 GHz Intel Xeon 5310 (4-core x86_64) processors and 1 GB RAM. It runs Debian GNU/Linux 4.0 (Etch), with Sun's 64-bit 1.5.0_10 JVM.

5.1 Experimental Procedure

We use twelve different multithreaded benchmarks from various sources, as shown in Table 4.1. The first six are smaller and contention heavy. The four Sable benchmarks were developed internally: *pcmab* stresses thread scheduling, producer/consumer thread coordination, and AspectJ 1.5 performance, *roller* simulates a race for seats on a roller coaster by having threads race to synchronize without performing any real work, and *traffic* simulates a car driving around a rotary with high contention between the car and driver threads. A fourth Sable benchmark, *heavy*, simulates several cars travelling around a rotary. It uses the same codebase as *traffic*, but uses an adaptation to allow multiple cars to ignore each other while navigating the rotary. This is necessary because the workload of *traffic* is non-deterministic if cars interact with each other[1]. *bank* is a micro benchmark derived from Doug Lea's ATApplet [Lea99], in which each thread makes a random account transaction then calls `Thread.yield()` one million times. For these five benchmarks, we measure the time of one program run. Our sixth benchmark, *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. We measure only the former, because our lock allocator converts all synchronizations to synchronized blocks (object synchronizations), which causes the method synchronization portion of the benchmark to be invalid.

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

[1]Thread non-determinism can cause *traffic* drivers to make different navigation decisions from one run to the next. For example, thread unfairness could cause one car to catch up to another, and a driver might choose to change lanes if another car is observed at a close enough distance.

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[BGH+06], 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_{peak}, the iteration with peak throughput and N threads, through $2{\times}N_{peak}$. For *jbb2005*, we were unable to use the official metric that averages all points from CPU_{max} through $2{\times}CPU_{max}$ threads, due to memory limitations of our 8-core machine. We instead limited the benchmark to using 1 to 8 threads, as larger numbers would require more memory than was available. We find that *jbb2005*'s performance degrades after four cores, which is surely a result of our limitation on the workload.

We perform twenty different experiments on each benchmark. A *control* experiment processes all class files with Soot but does not perform lock allocation. This is done to factor out any performance effects that the use of Soot might have on the benchmarks. The *singleton* allocation uses a single static lock. There remain eighteen allocation experiments: static, dynamic, and lockset variants of six different analysis pipeline configurations. These configurations are: 1) VTA only, which uses VTA as the only input to TBSE, 2) VTA with MHP, 3) VTA with MHP and TLO, 4) Spark only, 5) Spark with MHP, 6) Spark with MHP and TLO.

Each benchmark was transformed twenty times with Soot, once for *control*, and nineteen times for the allocation experiments. We use a 2.0 GHz machine for the transformations with a 1 GB heap, except for those including TLO on *hsqldb*, which require an 8 GB heap.

The results of running these transformations are shown in Table 4.2 to Table 4.13, in Chapter 4. Here, we present the results of measuring runtime performance. Each benchmark is run 50 times per allocation, per processor configuration, with a 1 GB heap, except for *jbb2000* and *jbb2005* which run only 5 times due to their excessive

length and relative stability.

Figures 5.5 through 5.14 show runtime performance for each configuration normalized against the control configuration on one processor. For *sync*, *jbb2000*, and *jbb2005*, we report changes in throughput rather than inverse execution time.

Our source code is available in revision 3043 of Soot [VR00]. In the interest of encouraging external repeatability we are also making available our benchmarks, run scripts, raw data, and data processing scripts at
http://www.sable.mcgill.ca/~rhalpe/thesis.

5.2 Performance Results

Figure 5.2 to Figure 5.14 present runtime performance data for all combinations of benchmark, experiment, and number of cpus. The figures show the performance of each configuration as relative speedup versus the control configuration on one processor. The line representing the control configuration is equivalent to the scalability graph for the original version of that benchmark. Note that a more rigorous evaluation of scalability might show performance normalized against a lockless or single-threaded version of the benchmark on one processor, in which case $speedup \leq \#Processors$ would be expected to hold.

The following sections are a categorization of benchmarks based on the lock allocation strategies that are sufficient for good performance. We loosely define good performance for an allocation against the control experiment as having a similar or more ideal scalability curve with a small ($< 10\%$) drop in performance at any data point, or as consistently outperforming the control experiment. We discuss each benchmark individually, noting the factors that contribute to its categorization.

Figure 5.1 shows the key for all of the graphs. It can be summarized as:

- The control and singleton lines are black (marks are '+' and 'x' respectively).
- VTA marks are solid, Spark (spk) marks are hollow.
- Static allocation (sta) marks are round, dynamic (dyn) allocation marks are square, lockset (set) allocation marks are triangular.

- Baseline analyses are pink (lines with small dashes), with MHP is green (lines with long dashes), and with TLO is blue (lines with medium dashes).

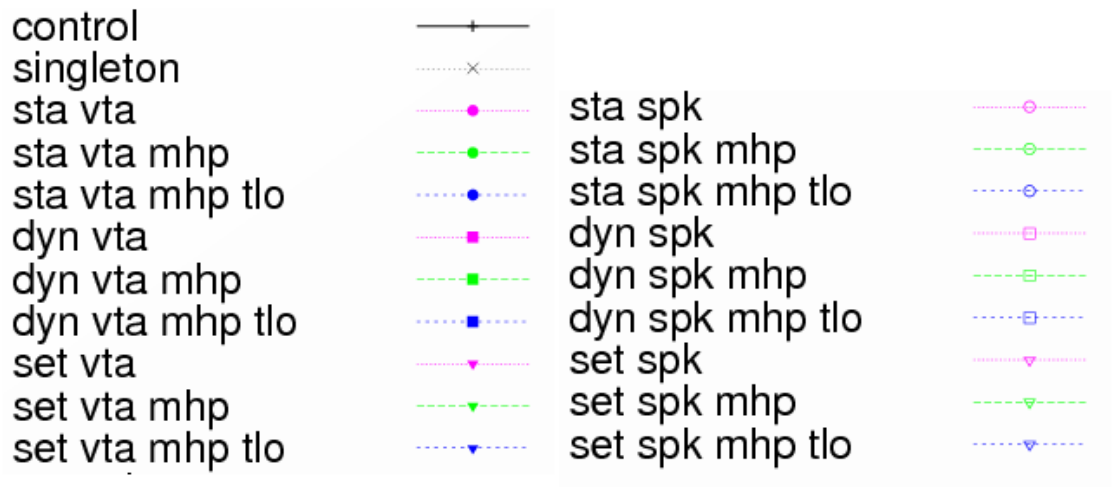


Figure 5.1: Key to performance graphs.

5.2.1 Benchmarks with Underlying Threading Problems

The *sync* (Figure 5.2) and *pcmab* (Figure 5.3) benchmarks demonstrate that programs with underlying threading problems can show counterintuitive behavior with regards to lock allocation. *sync* tends to prefer lock objects with low overhead, because it has no real workload, and no real parallelism. *pcmab* tends to prefer coarser granularity locking because it hides a starvation problem of the original program.

sync

The *sync* (Figure 5.2) benchmark's workload is completely dominated by the cost of synchronization. As a result, the simplest allocations, which carry the least synchronization overhead, perform the best. Its performance is measured in synchronizations per second, which generally decreases with the addition of more processors due to increased contention, so *sync* shows negative scalability.

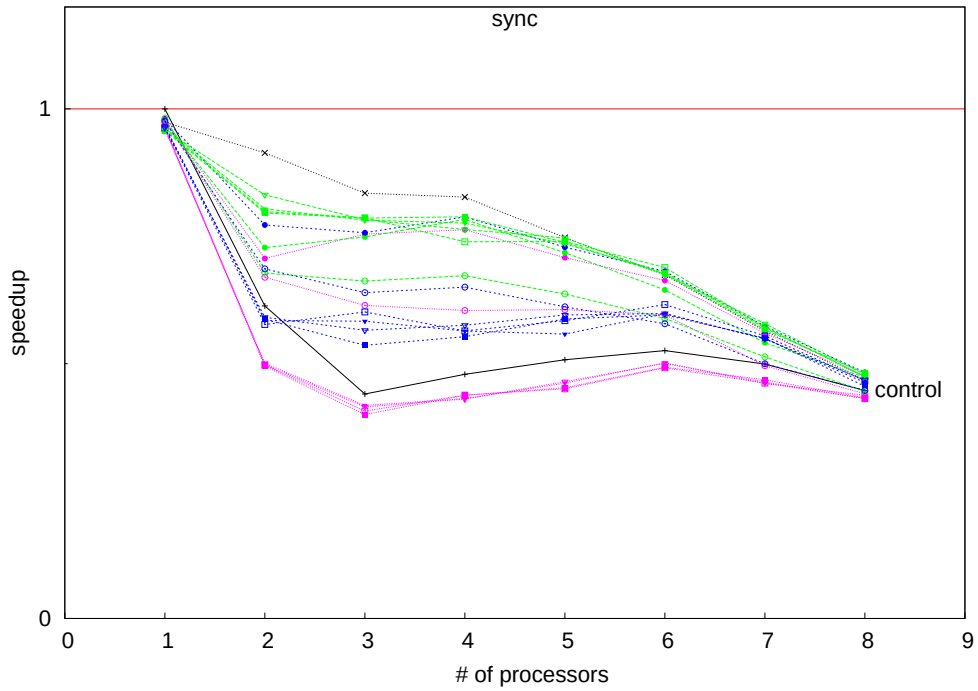


Figure 5.2: Relative speedup for *sync*.

Although all configurations perform similarly for one and eight processors, performance varies considerably for two to seven processors. Note that *sync* uses eight threads, which probably accounts for the convergence at eight processors. The control configuration, along with the basic dynamic and lockset configurations, drop sharply in throughput when a second or third processor core is added, and recover slightly thereafter, peaking at six cores. Dynamic and lockset configurations using TLO, and static Spark configurations perform slightly better, but the best performing configurations are those using just MHP (no TLO), or static locks with VTA. These are the configurations that result in the simplest allocations by removing unnecessary locks and using static locks rather than dynamic ones. It is notable that the singleton allocation outperforms all others for this benchmark.

pcmab

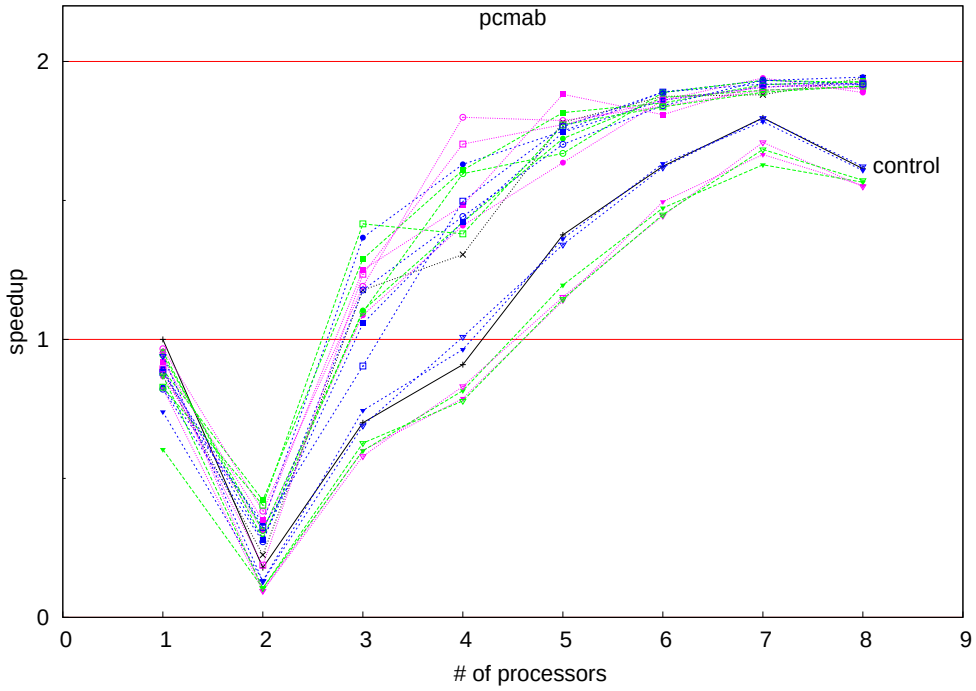


Figure 5.3: Relative speedup for *pcmab*.

pcmab (Figure 5.3) suffers from thread starvation problems because it is a polling solution to the classic producer-consumer problem. Coarse-grained lock allocations tend to lessen the starvation problem, and as a result, the static and dynamic allocations all outperform the original program and the lock set allocations by a small margin for three to eight cores. Note in Figure 5.4 that the variance is very large for this benchmark, which suggests that the benchmark is heavily affected by the nondeterminism of thread scheduling. This is a typical effect of partial starvation problems.

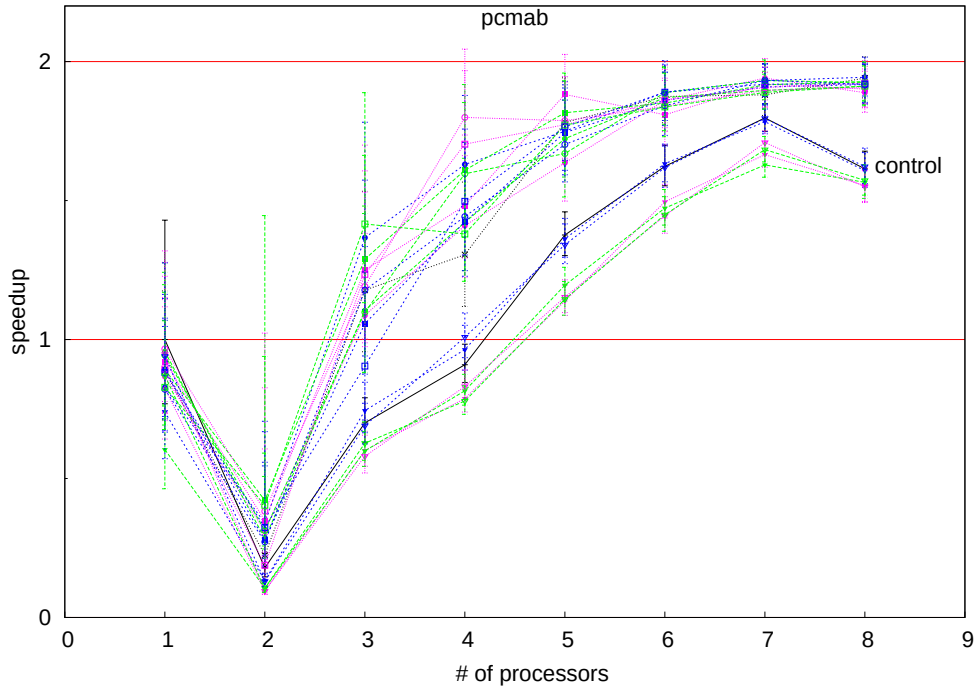


Figure 5.4: Relative speedup for *pcmab* with 95% confidence intervals.

5.2.2 Lock-Indifferent Benchmarks

Three benchmarks, *roller*, *mtrt*, and *hsqldb*, are examples of programs where the locking scheme does not play a significant role in performance. All of these benchmarks exhibit uniform behavior across allocations. For each of them, even the singleton al-

location performs similarly to the original allocation. Such programs do not depend on any particular pipeline stage for performance.

roller

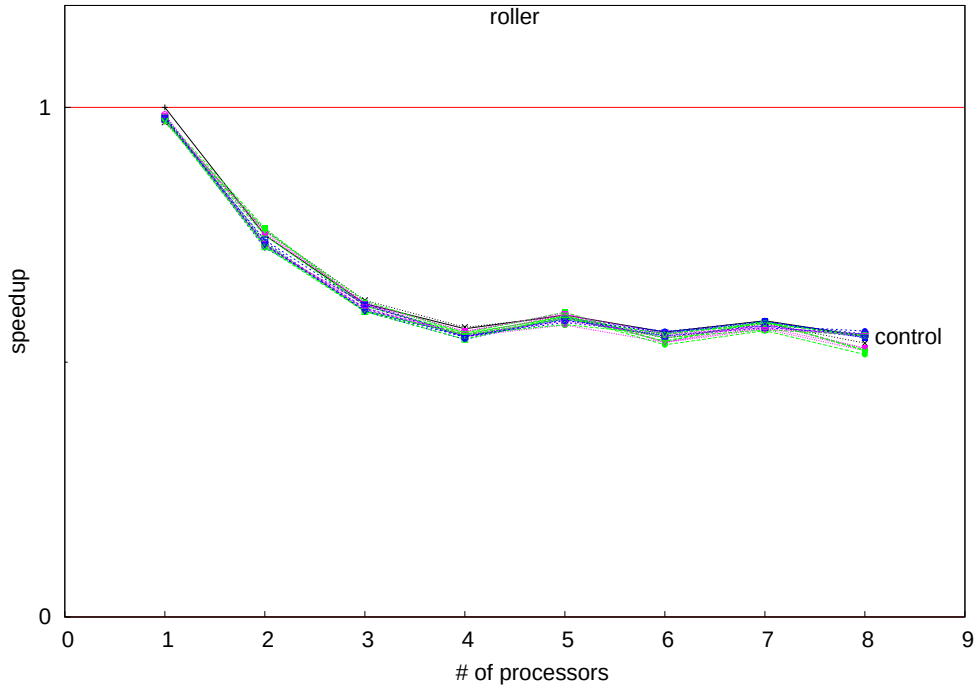


Figure 5.5: Relative speedup for *roller*.

The workload of *roller* consists entirely of a synchronization race[2] and associated accounting. The benchmark is essentially impervious to locking strategy changes because the accounting portion of the workload is serialized (by design) for all strategies, and dominates the running time. *roller* performs essentially no work outside of synchronization itself, so its scalability graph (Figure 5.5) is characterized by the increasing performance cost of increasing contention.

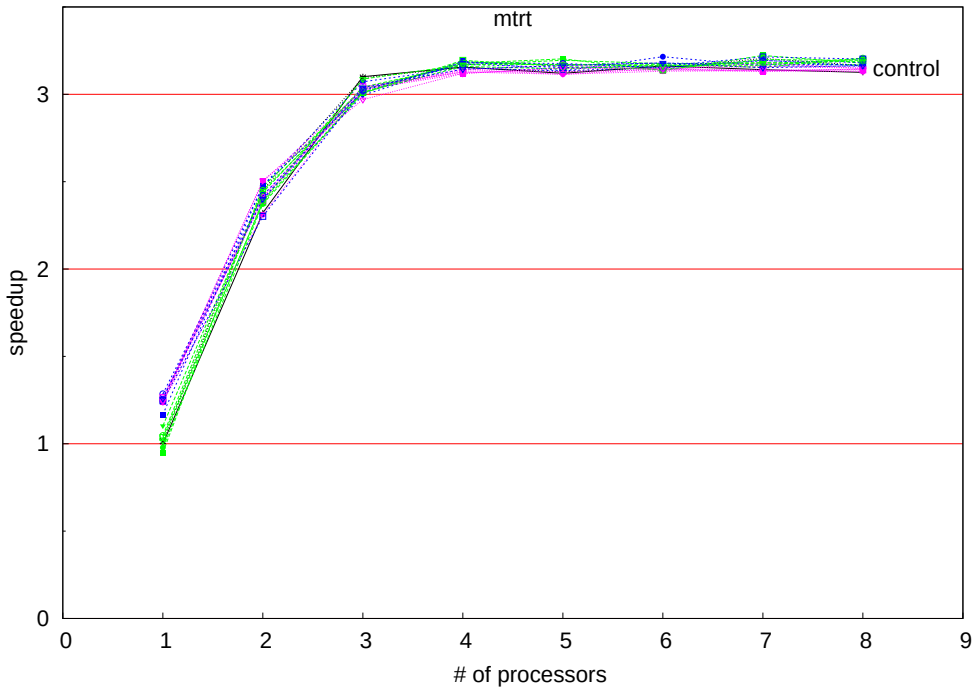


Figure 5.6: Relative speedup for *mtrt*.

mtrt

From the perspective of lock allocation, *mtrt* is the least interesting of the benchmarks because locking is not an intrinsic part of its algorithm. It is embarassingly parallel, simply forking two threads and later joining them, with a small amount of locking added as insurance against an unlikely data race. Since the locked portions of *mtrt* are dominated in running time by the unlocked portions, the lock allocation matters little.

The performance graph is shown in Figure 5.6. Note that this two-threaded benchmark shows statistically significant improvement with the addition of a third core. This may be an artifact of the machine architecture of two processors with four cores each, and the choice of core order, which is to alternate between physical processors when adding cores. As a result, the three-core configuration offers two colocated cores for the two active threads, which offers cache benefits that may be the cause for the observed performance improvements.

hsqldb

Like *roller* and *mtrt*, all automatic allocations for *hsqldb* show similar scalability and performance to the control. However, unlike those other benchmarks, *hsqldb*'s automatic allocations do exhibit an observable, albeit small, performance degradation from the control. This degradation is on the order of 5%, and is fairly stable across cpu configurations.

Despite the large number of worker threads, this benchmark does not scale at all past three processors under any configuration.

5.2.3 Points-to Dependent Benchmarks

Two of the DaCapo benchmarks, *xalan*, and *lusearch*, illustrate the essential nature of points-to analysis to ALOCS. Like many production programs, these two benchmarks use large, widely deployed, conservatively synchronized code libraries to perform their

[2]Not a data race: threads compete to claim resources, but do so under synchronization.

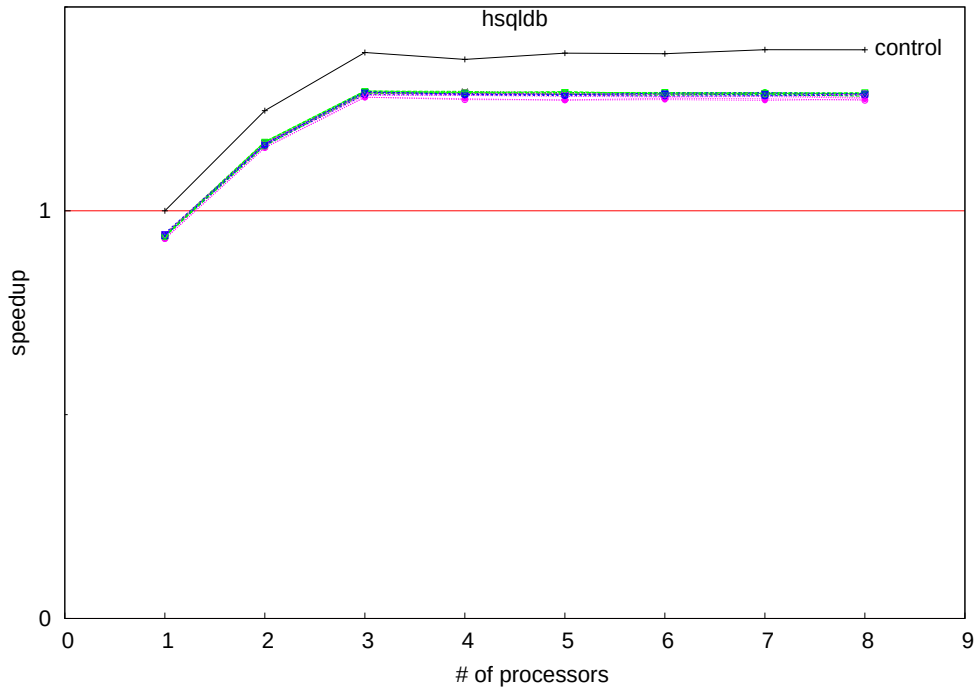


Figure 5.7: Relative speedup for *hsqldb*.

workload. These two benchmarks require only the use of a good context-insensitive points-to analysis (Spark) to find lock allocations that provide good performance.

For many of the other benchmarks used here, points-to analysis plays an important role even if it is not the sole deciding factor in allocation performance.

xalan

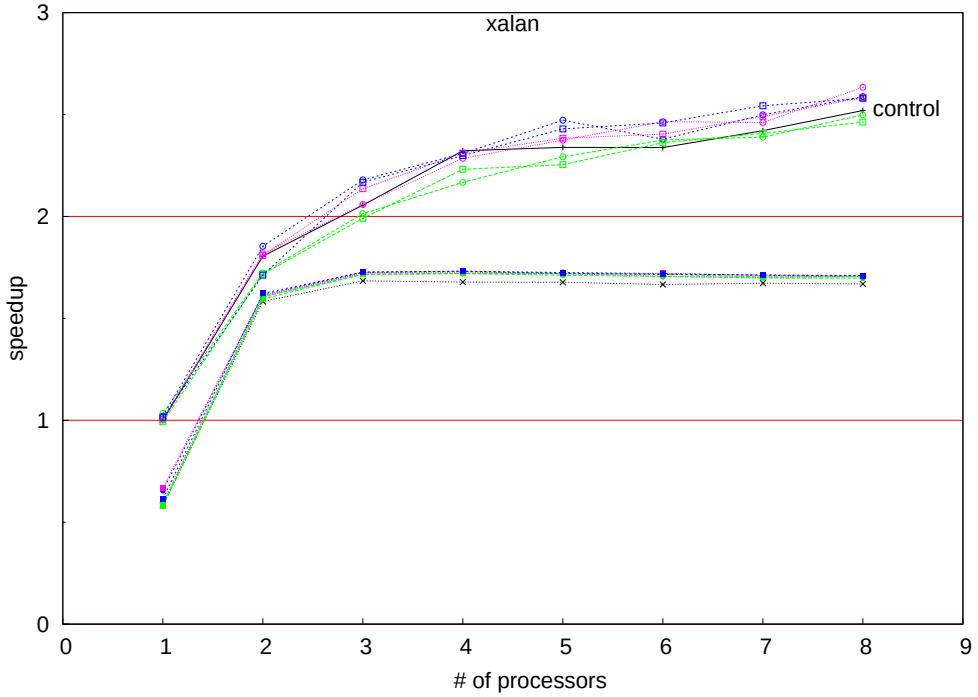


Figure 5.8: Relative speedup for *xalan*.

xalan (Figure 5.8) performance is separated into two categories: those allocations using Spark, and the singleton allocation plus those using VTA. The latter category peaks in performance around two cores, and is flat thereafter, whereas the former category scales modestly all the way to eight cores, like the original program. The use of precise points-to analysis is clearly essential to good performance.

It is interesting to note that the pathological behavior we reported on Xalan in [HPV07] was not present in these experiments. Both the hardware and the software

differs between the two sets of experiments, which may account for the differences.

lusearch

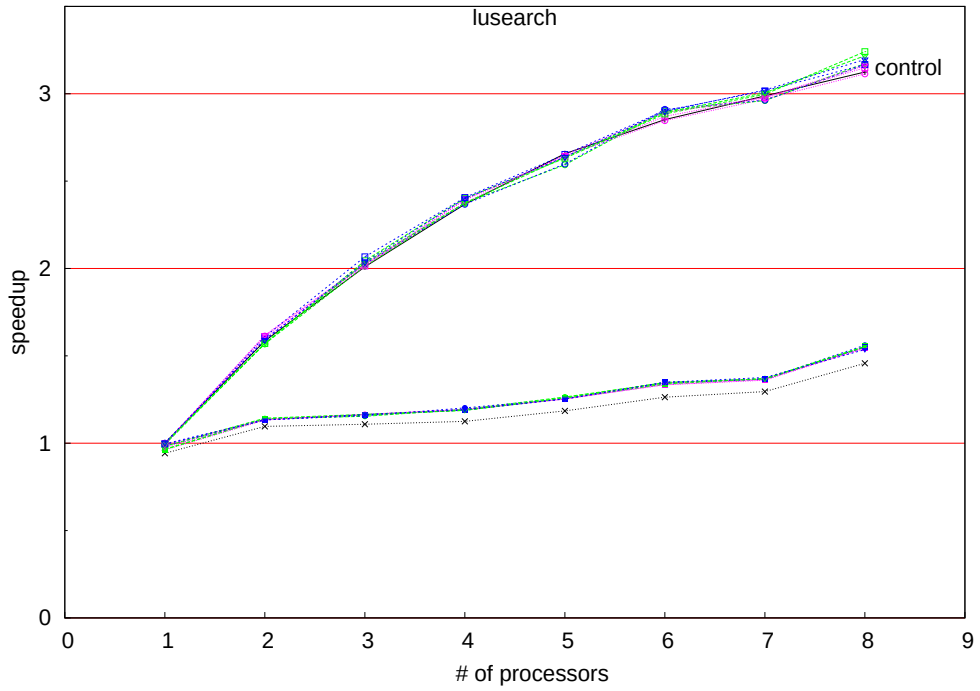


Figure 5.9: Relative speedup for *lusearch*.

Like *xalan*, *lusearch* (Figure 5.9) demonstrates a clear division between VTA and Spark allocations, with control performing like the Spark allocations, and singleton performing like the VTA allocations.

All three DaCapo benchmarks with multithreaded workloads show only modest scalability for even the control experiment. This may be an indication that the default workloads for these benchmarks do not fully take advantage of the high-profile, widely distributed libraries on which they are built, rather than an indication of scalability problems in the libraries themselves.

5.2.4 Thread Analysis Dependent Benchmarks

Benchmarks like *traffic* and *jbb2005*, where locking is heavily used to control thread cooperation, depend on the thread analyses, MHP and TLO, to produce allocations that perform well. These benchmarks also differentiate between transformation stages much more than those already presented, though that differentiation does not entirely match intuitive expectations.

traffic and *jbb2005* are examples of programs that require every available tool in order to be properly analyzed. Only with the complete combination of spark, mhp, and tlo, do these benchmarks (nearly) match the performance of the originals. Performance improves with the addition of each analysis.

traffic

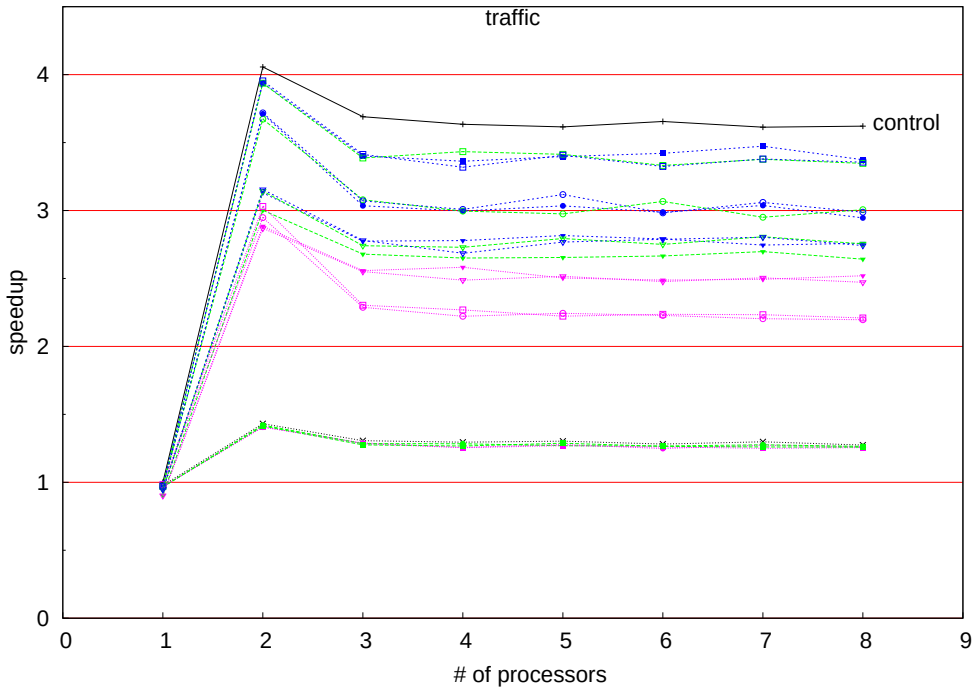


Figure 5.10: Relative speedup for *traffic*.

traffic simulates a single car and driver navigating around a rotary. It contains two active and one dormant thread during the bulk of the workload, and as a result, performance peaks for all allocations at two processors and remains flat after three.

In general, VTA-based allocations do not perform well. There are notable exceptions, however, in that VTA-based allocations using either TLO or the lockset transformation perform nearly as well as their Spark-based equivalents. A reasonable conclusion, and one supported by similar results for *jbb2005*, is that TLO and the lockset analysis both recover some of the information lost by using VTA instead of Spark. In other cases, such as *xalan* and *lusearch*, these analyses are not sufficient to match the performance of allocations using Spark, so this relationship does not necessarily apply in the general case.

The best performing allocations for *traffic* are those using the dynamic locking transformation, followed by those using static locking, and finally by those using locksets. This is one of the only benchmarks where locksets perform significantly worse than their static and dynamic equivalents.

jbb2005

The performance of allocations for *jbb2005* depends heavily on the use of thread analysis stages. Allocations lacking MHP and TLO do not perform well, showing negative scalability accross all core counts. Allocations using MHP perform well, especially those based on Spark. The Spark- and MHP- based allocation using dynamic locks performs especially well, nearly matching the performance of the control. However, all TLO-based allocations (which all include MHP as well) outperform all non-TLO allocations, and match the control performance. Clearly, for *jbb2005*, TLO resolves the thread-independence of a crucial data structure access that may be unnecessarily locked by allocations that do not use TLO.

5.2.5 Lockset Dependent Benchmarks

Although rare amongst benchmarks and legacy software, some programs require multiple different locks per critical section in order to perform optimally. Legacy software

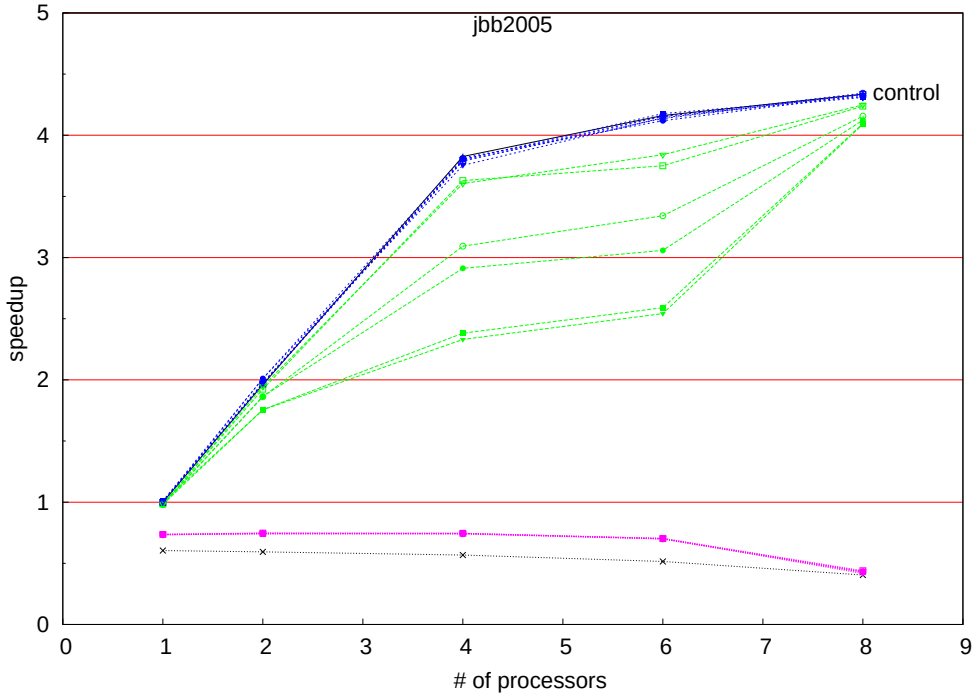


Figure 5.11: Relative speedup for *jbb2005*.

tends to require exactly one lock per critical section because it is designed to do so in order to maximize performance under the limitations of Java synchronized regions. Benchmarks tend not to require more than one lock per critical section because they tend to be either very simple, or based on legacy software.

We believe that naive solutions to many locking problems are likely to require this type of critical section splitting to allow maximal performance, and that in less widely used or higher-complexity programs than those comprising our benchmark suite, this splitting would not have been done manually. This is especially likely as multithreaded programming becomes necessary, even for novice programmers, to take advantage of increasing core counts.

heavy

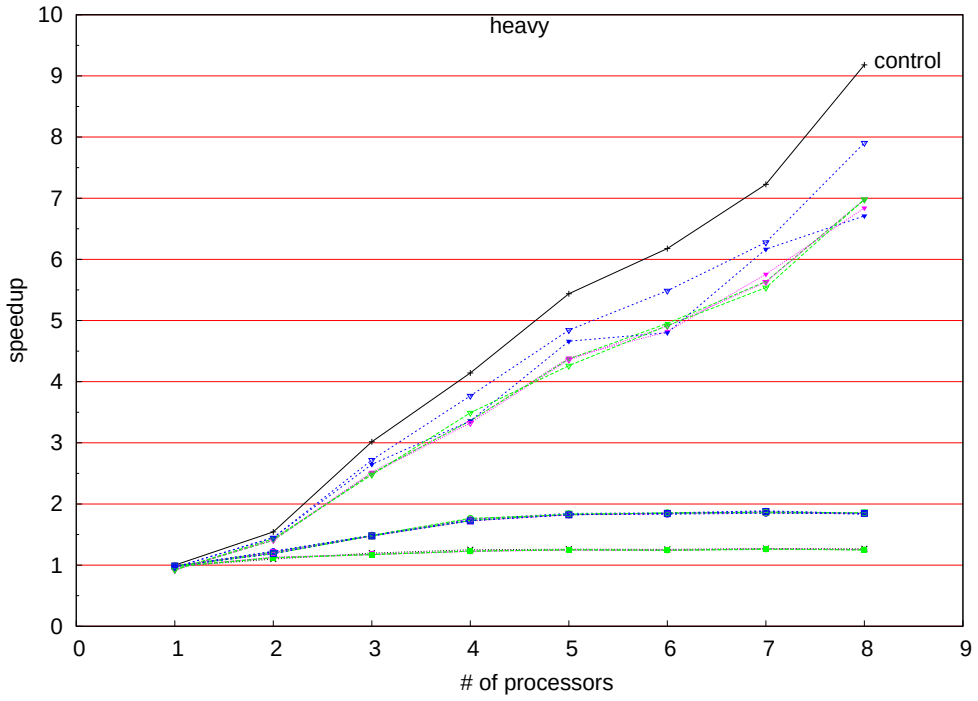


Figure 5.12: Relative speedup for *heavy*.

heavy (Figure 5.12) is an adaptation of the *traffic* benchmark that allows the use

of a more realistic workload. Four cars navigate the traffic circle at a time, but the workload is deterministic because the cars ignore each others' presence. As a result, while traffic scales only to two processors, heavy demonstrates that the code base scales at least up to eight.

traffic improves when MHP and TLO are enabled because the improvements they find in the allocation are necessary to allow its two cooperative threads to work together effectively. However, in the case of *heavy*, those same allocations fail to allow multiple pairs of cooperative threads to work independently enough to perform optimally. For this, the lockset allocation is needed.

heavy's thread pairs share the same data as the thread pair in *traffic*, and all of *heavy*'s threads share a small amount of global data. However, most of the data used by each pair of threads is needed only by that pair. In one important critical section, both shared and pair-local data is accessed[3]. The static and dynamic output phases both assign a static lock to this critical section, thereby serializing it amongst all threads. The lockset output phase correctly assigns two different locks to protect these two different pieces of data, thereby allowing increased parallelism at the cost of having to acquire two locks instead of one. For *heavy*, this cost is miniscule in comparison to the benefits.

Our results show that the lockset output phase results in allocations that mirror the scalability of the control, while none of the static, dynamic, or singleton allocations do. As our results for *traffic* would make us expect, the allocation using Spark, MHP, and TLO performs the best.

Among the other allocations, as with *traffic*, those using VTA with TLO and those using Spark perform better than the singleton and others.

5.2.6 Stubborn Benchmarks

Some benchmarks contain certain structures or a level of complexity that the analyses and output phases presented here cannot address. While some programs may require

[3]In the original program, a single dynamic lock is used that does not protect all of the data being accessed, and leaves the shared data susceptible to a data race

greater precision of the points-to analysis, others might depend on more precise thread analyses or different deadlock avoidance techniques. Two of these cases were observed in our benchmark suite.

bank

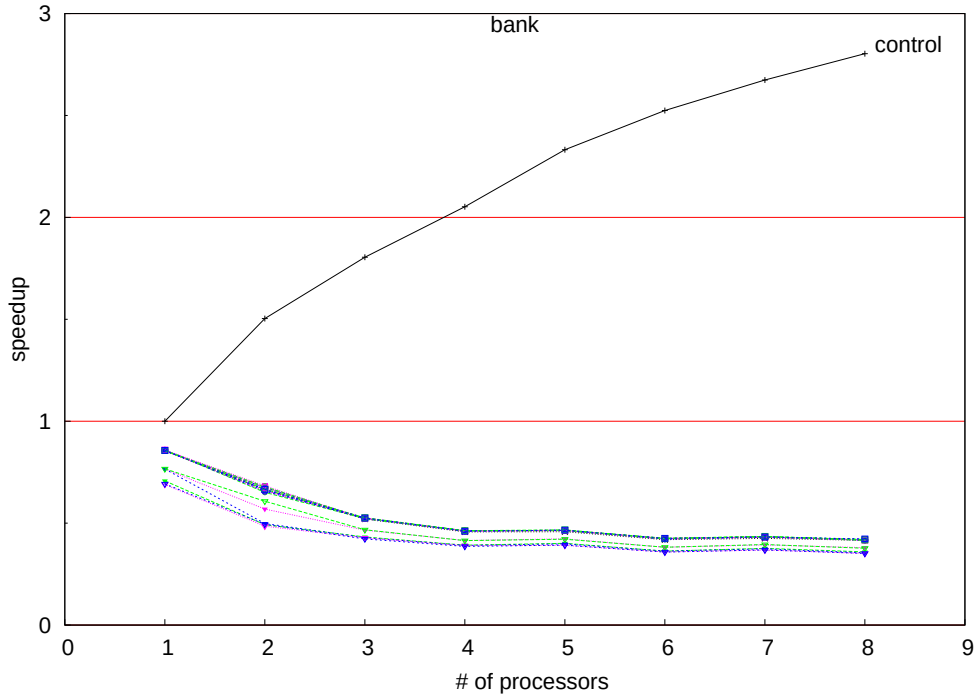


Figure 5.13: Relative speedup for *bank*.

Bank is a simple benchmark that employs a nonstandard locking technique. ALOCS detects potential deadlock in this benchmark, and corrects it by the addition of a static lock. The benchmark author, Doug Lea, avoids the same potential deadlock by the use of a hand-programmed test-and-set boolean, which equates roughly to the use of a dynamic lock. Unfortunately, ALOCS is unable to determine that this code prevents the deadlock, because it does not take volatile variable value testing into consideration as a method of locking.

As a result of the inserted static lock, all allocations suffer from poor scalability.

An alternative approach to deadlock avoidance could remedy this problem. It is possible to choose an arbitrary order between the two locks involved in the deadlock, and add a lock acquisition in the appropriate place to observe that order. This technique is employed by some other lock allocators. For this work, we opted to prevent deadlock without altering the boundaries of critical sections in this manner.

jbb2000

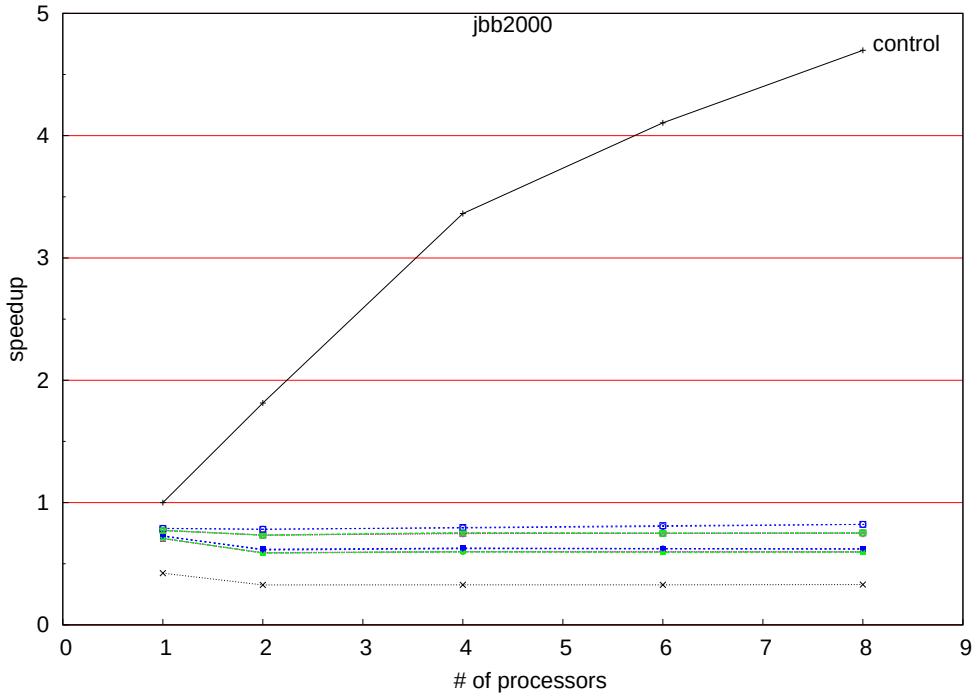


Figure 5.14: Relative speedup for *jbb2000*.

All automatic allocations for this benchmark suffer from poor performance, though the use of Spark over VTA offers improvement, and the use of TLO provides a further benefit. Unlike the similar *jbb2005*, *jbb2000* does not show any improvement from the application of MHP. It is possible that this difference alone accounts for the discrepancy in the success of automatic lock allocation. However, it could very well be other factors that are responsible for the differences. *jbb2005* is a rewrite of *jbb2000*

that is considered to be more idiomatic of Java. It may be the use of arrays in *jbb2000* instead of collections that results in an obscured interference graph, and thus poorly performing allocations. The relevant portions of the interference graphs of *jbb2000* and *jbb2005* are shown in Figure 5.15. Note that the primary component of *jbb2000* is more highly interconnected than that of *jbb2005*, and it lacks the two-tier arrangement that characterizes the addition of MHP information to a well-resolved interference graph (like the VTA allocation versus the SPK allocation of *traffic* in Figure 4.4).

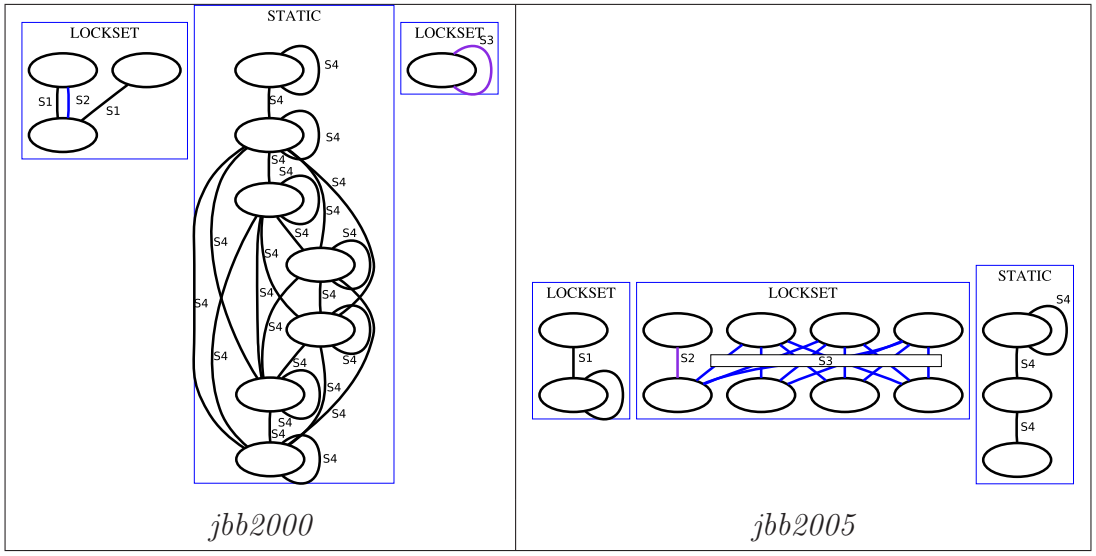


Figure 5.15: Interference graphs of *jbb2000* and *jbb2005* (locked components only).

5.3 Performance Observations

In general, we find that Spark with the two thread structure analyses, MHP and TLO, and the lockset allocation offers the best balance of performance across all benchmarks. Other than *sync* and *pcmab*, for which coarser allocations offer the benefit of hiding other problems, this particular configuration succeeds in generating an allocation in the range of those that perform best for every benchmark except *traffic*. For *traffic*, this configuration degrades approximately 25%, which may or

may not be an acceptable price to pay for automatic allocation. Nonetheless, if we were to suggest a single combination to use for a general purpose lock allocator, it would be this one.

While stubborn benchmarks may demonstrate the limitations in the ability of our implementation of lock allocation to find a suitable solution for every program, and while benchmarks with underlying problems may demonstrate the limitations of the ability of lock allocation in general to solve the world's threading problems, the eight remaining benchmarks prove the usefulness of this technique. Furthermore, we believe that our approach is a viable one even for the few stubborn benchmarks presented here, and that future refinements of our implementation will yield progress on these, and other programs like them.

Four mainstays of Java benchmarking, *mtrt*, *hsqldb*, *xalan*, and *lusearch*, plus the simple benchmark *roller*, all prove the effectiveness of the combination of good points-to analysis with component-based allocation. More complex benchmarks like *traffic*, *jbb2005*, and *heavy* illustrate the value of thread structure analyses such as MHP and TLO.

Although no benchmark here conclusively demonstrates the value of our dynamic locking output phase, it should be noted that the only benchmark with a compelling need for dynamic locking, *heavy*, coincidentally also requires that one critical section be split between two dynamic locks, and thus happens to require the lockset allocation. We intend to actively seek out new and existing benchmarks that can offer more parallel and complex workloads for further study. Finally, we believe that *heavy* is representative of a large category of new Java programs that use significant multithreading, but that are not written by experts, and that are likely to contain locking missteps that cause a need for the lockset allocation. These sorts of programs and their programmers stand to benefit the most from automatic lock allocation.

Chapter 6
Conclusions and Future Work

In this chapter, we present our conclusions regarding this work and lock allocation in general, and we suggest some future work that might increase the quality and usefulness of our system.

6.1 The ALOCS System

Automatic techniques like ours take the considerable burden of choosing locks out of the hands of the programmer. We find that while programs with a very complex architecture or with tricky manual synchronization are not good candidates for automatic techniques, most programs are.

Our lock allocator, ALOCS, is built around a thread-based side effect analysis that generates an interference graph. Pluggable points-to, may happen in parallel, and thread local objects analyses offer different levels of graph quality, and different allocation phases choose the level of locking granularity. Conceptually, our analysis is top-down, in that we start with the conservative whole-program assumption that all critical sections interfere, and refine this solution by application of different analyses to expose groups of interfering critical sections. However, our lockable reference analysis is similar to portions of the bottom-up analyses of others' work on lock allocation. Locks are allocated on a per-*component* basis, which provides good runtime performance without the use of optimal or heuristic MLA solutions.

ALOCS puts a minimum of restrictions on input programs, dealing gracefully with nesting, deadlock, different granularities, and condition variables, the combination of which allows existing software to be analyzed without alteration. Furthermore, our framework outputs standard Java class files, allowing programs transformed by our lock allocator to be run on existing, unmodified Java Virtual Machines.

6.2 Empirical Evaluation of ALOCS

Runtime results for a majority of benchmarks validate the effectiveness of our approach, with ten of twelve benchmarks recovering or exceeding original performance. Just one benchmark, *jbb2000*, gives evidence that there's still room for improvement of the analysis phase of ALOCS.

We find that the most important phase in lock allocation is points-to analysis, with Spark providing significant improvement over VTA for half of our benchmarks. MHP and TLO together offer improvement for four benchmarks. We judge MHP to be a better investment of analysis time, as it generally runs considerably shorter than TLO.

Our benchmark suite does not demonstrate a pressing need for dynamic locking, but we remain convinced that dynamic locks are relevant for real programs. One benchmark, *traffic*, does improve somewhat from the use of dynamic locks. In contrast to dynamic locking, there is compelling evidence that locksets are sometimes necessary. The benchmark *heavy* is likely representative of a larger category of software in its need of locksets to repair an inefficient critical section.

We conclude that the combination of Spark, MHP, TLO, and locksets is the best pipeline for general-purpose lock allocation. Although locksets do not always provide the best performance, they are generally within a small margin of it. More importantly, there are programs like the benchmark *heavy*, where locksets scale an order of magnitude better than static or dynamic locking.

6.3 Analysis of Lock Allocation

Our results suggest that parallel programs often exhibit simplistic concurrent behaviour, and that good solutions can be obtained using straightforward program analyses. Although we recommend the use of our entire analysis pipeline, we acknowledge that for most of the benchmarks here, this is not necessary. It may, however, be possible to recognize within ALOCS when lockset allocation is not necessary, or to otherwise improve the quality of the allocations generated.

Optimal allocation approaches may be able to improve on the runtime performance of component-based allocation for some benchmarks.

6.4 Future Work

There are several possible avenues for future work on ALOCS and component-based lock allocation in general. Beyond the usual array of design implementation improvements, we believe that ALOCS can be made to work together with other development tools and areas of research.

As with many still-immature technologies, lock allocation may be best offered as an online system which provides information to programmers within an IDE without forcing them to accept a completely automatic lock allocation.

Alternatively, component-based lock allocation could be used in the context of an allocator based on an optimal allocation approach. Our top-down approach can reduce the size of the optimization input problem, thereby decreasing the cost of optimal allocation; and it can provide a "next best" solution that may suffice when optimal solutions are too expensive.

An even more promising possibility is to integrate ALOCS as an optional optimization step for an optimistic concurrency system. The information generated by ALOCS could be used to avoid doing unnecessary accounting on speculatively locked regions or transactions that are guaranteed to be non-conflicting. Furthermore, in those cases where conflicts are possible, ALOCS could provide ahead-of-time information to the optimistic system about which fields could possibly be involved in those

conflicts. This type of integration could potentially reduce the overhead of optimistic systems quite substantially.

In the short-term, we plan to improve the quality of the analyses in ALOCS, improve its transformation phase, and offer alternative means of deadlock avoidance. There are numerous optimizations possible that might shorten analysis time. We also intend to study the impact of context-sensitive points-to analysis on the quality of allocations, and consider the possibility of reformulating our thread-local objects analysis as a type of context applied to a points-to analysis.

Appendix A
Definitions for Selected Flow Analyses

In this appendix, we present flow analysis definitions for several selected analyses.

A.1 Information Flow Analysis

1. What: Sets of pairs of values <source, sink> where source is in {Ref} and sink is in {Local, Ref}
2. Definition: A source flows to a sink if it is assigned to that sink, if it flows to a value that is assigned to that sink, or if it flows to a field of that sink[1]
3. Direction: forward
4. Confluence Operator: union
5. Flow Equations: see below
6. Initial Sets: entry flow = {}, initial flow = {}

Flow Equations:

1. Gen(return a;) = {<a, returnref>}

[1] except assignments of the form *this.somefield*, in which case only the first two rules apply.

2. Gen(a = @identityref;) = {<@identityref, a>}
3. Gen(a = b;) = {<b, a>}
4. Gen(this.f = b;) = {<b, this.f>}
5. Gen(a.f = b;) = {<b, typeof(a).f>}
6. Gen(a[e] = b;) = {<b, a>, <b, sources(a)>}
7. Gen(a = b[e];) = {<b, a>}
8. Gen(a = this.f;) = {<this.f, a>}
9. Gen(a = b.f;) = {<b, a>, <typeof(b).f, a>}
10. Gen(a = b op c;) = {<b, a>, <c, a>}
11. Gen(a = op b;) = {<b, a>}
12. Gen(a = b.f();) = {<retval(b.f()), a>, IFA(b.f())}

A.2 Lockable Reference Analysis

1. What: Sets of pairs of <variable, value#>, where variable is in {Local, Ref}, and value# is an integer.
2. Definition: A variable is assigned a value if it is "used".
3. Direction: backwards
4. Confluence Operator: union, with equivalent value#s merged.
5. Flow Equations: see below
6. Initial Sets: entry flow = {}, initial flow = {}

`Generate a new value# when a "used" value is found`

a.b = 5;	val#(a) := new value number

`Track a value# through an identity assignment`

a = @identityrefb;	val#(@identityrefb):= val#(a) val#(a):= `nothing`

`Track a value# through a constant assignment`

a = 5;	IFDEF val#(5) THEN ∀ x ∈ {x: val#(x)=val#(5)}, val#(x):=val#(a) ELSE val#(5):= val#(a) val#(a):= nothing

`Track a value# through a Local assignment`

a = b;	IFDEF val#(b) THEN ∀ x ∈ {x: val#(x)=val#(b)}, val#(x):=val#(a) ELSE val#(b):= val#(a) val#(a):= nothing [2]

`Track a value# through assignment from a static field`

a = B.f;	IFDEF val#(B.f) THEN ∀ x ∈ {x: val#(x)=val#(B.f)}, val#(x):=val#(a) ELSE val#(B.f):= val#(a) val#(a):= nothing

`Safely remove a value# through new object assignment`

a = new Object;	val#(a):= nothing

`Track a value# to 2 value#s through assignment from a field`

a = b.f;	IFDEF val#(b.f) THEN ∀ x ∈ {x: val#(x)=val#(b.f)}, val#(x):=val#(a) ELSE val#(b.f):= val#(a) val#(b):= new integer Replace b.f with b'.f val#(a):= nothing

[2] In these equations, 'a' represents any lvalue which already has an assigned value number.

`Track a value# to 3 value#s through assignment from an element`	
	IFDEF val#(d[e]) THEN ∀ x ∈ {x: val#(x)=val#(d[e])}, val#(x):=val#(a)
	ELSE val#(d[e]):= val#(a)
	val#(d):= new integer
a = d[e];	val#(e):= new integer
	Replace d with d' in d[e]
	Replace e with e' in d[e]
	val#(a):= nothing

Appendix B
Public Availability of Implementation, Benchmarks, and Scripts

---

In the interest of encouraging external repeatability, we are making available our implementation, benchmarks, run scripts, raw data, and data processing scripts.

Our implementation has been developed as part of the Soot project. The version used here is Soot SVN revision 3043. Soot is available at http://www.sable.mcgill.ca/soot/, with a publicly readable subversion repository at https://svn.sable.mcgill.ca/soot/soot/trunk

All other materials are available at http://www.sable.mcgill.ca/~rhalpe/thesis.

Appendix C
Code Map

In this appendix, we present a mapping from Soot revision 3043 source packages to the sections of this thesis, since many of the analyses described here exist in Soot under different names.

Class	Section
soot.jimple.toolkits.pointer.CodeBlockRWSet	3.1.3
soot.jimple.toolkits.infoflow.InfoFlowAnalysis	3.1.2
soot.jimple.toolkits.infoflow.ClassInfoFlowAnalysis	3.1.2
soot.jimple.toolkits.infoflow.SmartMethodInfoFlowAnalysis	3.1.2
soot.jimple.toolkits.infoflow.UseFinder	3.1.2
soot.jimple.toolkits.infoflow.LocalObjectsAnalysis	3.1.2
soot.jimple.toolkits.infoflow.ClassLocalObjectsAnalysis	3.1.2
soot.jimple.toolkits.infoflow.SmartMethodLocalObjectsAnalysis	3.1.2
soot.jimple.toolkits.infoflow.CallLocalityContext	3.1.2
soot.jimple.toolkits.infoflow.FakeJimpleLocal	3.1.2
soot.jimple.toolkits.thread.transaction.TransactionAnalysis	3.2.2
soot.jimple.toolkits.thread.transaction.LockRegion	3.2.2
soot.jimple.toolkits.thread.transaction.Transaction	3.2.2
soot.jimple.toolkits.thread.transaction.TransactionAwareSideEffectAnalysis	3.1.3
soot.jimple.toolkits.thread.transaction.TransactionVisibleEdgesPred	3.1.3
soot.jimple.toolkits.thread.transaction.TransactionGroup	3.2.4
soot.jimple.toolkits.thread.transaction.TransactionDataDependency	3.2.4
soot.jimple.toolkits.thread.transaction.LocksetAnalysis	3.1.5
soot.jimple.toolkits.thread.transaction.DeadlockAvoidanceEdge	3.2.6
soot.jimple.toolkits.thread.transaction.NewStaticLock	3.2.6
soot.jimple.toolkits.thread.transaction.TransactionBodyTransformer	3.2.7
soot.jimple.toolkits.thread.AbstractRuntimeThread	3.1.4
soot.jimple.toolkits.thread.ThreadLocalObjectsAnalysis	3.1.2

soot.jimple.toolkits.thread.mhp.findobject.AllocNodesFinder	3.1.4
soot.jimple.toolkits.thread.mhp.findobject.MultiCalledMethods	3.1.4
soot.jimple.toolkits.thread.mhp.findobject.MultiRunStatementsFinder	3.1.4
soot.jimple.toolkits.thread.mhp.StartJoinAnalysis	3.1.4
soot.jimple.toolkits.thread.mhp.StartJoinFinder	3.1.4
soot.jimple.toolkits.thread.mhp.MhpTester	3.1.4
soot.jimple.toolkits.thread.mhp.UnsynchronizedMhpAnalysis	3.1.4
soot.jimple.toolkits.scalar.EqualUsesAnalysis	3.1.4
soot.jimple.toolkits.callgraph.TransitiveTargets	3.1.3
soot.jimple.toolkits.callgraph.ReachableMethods	3.1.1

Bibliography

[ABSS07] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna 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*, San Jose, California, USA, March 2007, pages 183–193.

[AFF06] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 28(2):207–255, March 2006.

[Ahm06] Ahmer Ahmedani. Information flow in a Java intermediate language. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, August 2006.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, May 1994.

[ASCE03] Jonathan Aldrich, Emin Gün Sirer, Craig Chambers, and Susan J. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, 47(2-3):91–120, May 2003.

[Bar05] Rajkishore 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*, October 2005, volume 4339 of *LNCS: Lecture Notes in Computer Science*, pages 152–169.

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben 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*, Portland, Oregon, USA, October 2006, pages 169–190.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. 1999, volume 34, pages 35–46.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI'98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, Montreal, Quebec, Canada, June 1998, pages 258–268.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999, volume 34, pages 20–34. ACM, New York, NY, USA.

[BS] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. pages 324–341.

[CC04] Byeong-Mo Chang and Jong-Deok Choi. Thread-sensitive points-to analysis for multithreaded Java programs. In *ISCIS'04: Proceedings of the 19th International Symposium on Computer and Information Sciences*, October 2004, volume 3280 of *LNCS: Lecture Notes in Computer Science*, pages 945–954.

[CCG08] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *CSUR: ACM Computing Surveys*, 3(2):67–78, June 1971.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1999, pages 1–19.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95: Proceedings of the 9th European Conference on Object-Oriented Programming*, August 1995, volume 952 of *LNCS: Lecture Notes in Computer Science*, pages 77–101.

[EFJM07] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL'07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007, pages 291–296.

[FF05] Cormac Flanagan and Stephen N. Freund. Automatic synchronization correction. In *SCOOL'05: Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[HFP06] Michael Hicks, Jeffrey S. Foster, and Polyvios 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.

[HG06] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *MSPC'06: Proceedings of the 2006 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, San Jose, California, October 2006, pages 82–91.

[HPV07] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, September 2007, pages 353–364.

[KKO02] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA'02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming,*

Systems, Languages, and Applications, Seattle, Washington, USA, November 2002, pages 130–141.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2nd edition, November 1999.

[Lho03] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, February 2003.

[Li04] Lin Li. A practical MHP information computation for concurrent Java programs. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, August 2004.

[LR06] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, December 2006.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Long Beach, California, USA, January 2005, pages 378–391.

[MT02] José F. Martínez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, USA, October 2002, pages 18–29.

[MZGB06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL'06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, USA, January 2006, pages 346–358.

[NA07] Mayur Naik and Alex 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*, 2007, pages 327–338.

[NAC99] Gleb Naumovich, George S. Avrunin, and Lori 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*, Toulouse, France, September 1999, pages 338–354.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006, pages 308–319.

[NR06] Mangala Gowri 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, November 2006.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael 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*, June 2006, pages 320–331.

[RD06] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *OOPSLA'06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, October 2006, pages 263–272.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *PLDI'00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000, pages 208–218.

[SFW+05] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP'05: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, USA, June 2005, pages 2–13.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, October 2000, volume 35, pages 264–280.

[SR01] Alexandru Sălcianu and Martin 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*, Snowbird, Utah, United States, June 2001, pages 12–23.

[SZG05] Vugranam C. Sreedhar, Yuan Zhang, and Guang 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.

[vPG03] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, USA, June 2003, pages 115–128.

[VR00] Raja 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/`.

[VTD06] Mandana Vaziri, Frank Tip, and Julian 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*, Charleston, South Carolina, USA, January 2006, pages 334–345.

[WHJ06] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, Nantes, France, July 2006, volume 4067 of *LNCS: Lecture Notes in Computer Science*, pages 148–173.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. November 1999, pages 187–206.

[ZSZ+06] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang 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, April 2006.

[ZSZ+07] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R. Gao. Optimized lock assignment and allocation: A method for exploiting concurrency among critical sections. In *PPoPP'07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, California, USA, March 2007, pages 146–147. Short paper from poster session.