A PRACTICAL MHP INFORMATION COMPUTATION FOR CONCURRENT JAVA PROGRAMS

by

Lin Li

School of Computer Science

McGill University, Montreal

August 2004

A thesis submitted to McGill University

in partial fulfillment of the requirements of the degree of

Master of Science

Copyright © 2004 by Lin Li

Abstract

With the development of multi-processors, multi-threaded programs and programming languages have become more and more popular. This requires extending the scope of program analysis and compiler optimization from traditional, sequential programs to concurrent programs.

Naumovich *et al.* proposed *May Happen in Parallel* (MHP) analysis that determines which program statements may be executed concurrently. From this information, compiler optimization improvements, as well as analysis data on potential program problems such as *dataraces* can be analyzed or discovered.

Unfortunately, MHP analysis has some limitations with respect to practical use. In this thesis we present an implementation of MHP analysis for Java that attempts to address some of the practical implementation concerns of the original work. We describe a design that incorporates techniques for aiding a feasible implementation and expanding the range of acceptable inputs.

The MHP analysis requires a particular internal representation in order to run. By using a combination of techniques, we are able to compact that representation, and thus significantly improve MHP execution time without affecting accuracy. We also provide experimental results showing the utility and impact of our approach and optimizations using a variety of concurrent benchmarks. The results show that our optimizations are effective, and allow more and larger benchmarks to be analyzed. For some benchmarks, our optimizations have very impressive results, speeding up MHP analysis by several orders of magnitude.

Résumé

Dans la foulée du développement des multiprocesseurs, les programmes en chapelet et les languages de programmation ont acquis une grande popularité. Cette dynamique exige d'étendre la portée de l'analyse des programmes et de l'optimisation des compilateurs, pour les faire passer de programmes séquentiels classiques à des programmes concurrents.

Naumovich *et al.* ont proposé l'analyse *May happen in Parallel* (MHP), qui détermine les instructions pouvant s'exécuter en parallèle ou concurremment. À partir de cette information, les améliorations á l'optimisation des compilateurs, de même que les données d'analyse sur les problèmes de programme potentiels comme l'accès concurrent d'unités d'exécution, peuvent être analysées ou découvertes.

Malheureusement, l'analyse MHP comporte des limites sur le plan pratique. Dans cette thèse, nous présentons la mise en œuvre de l'analyse MHP pour Java visant à aborder certaines préoccupations entourant la mise en œuvre du travail original. Nous définissons un concept qui incorpore des techniques pour favoriser une mise en œuvre efficiente et étendre la portée des intrants acceptables.

L'analyse MHP exige une représentation interne afin de fonctionner. En utilisant une combinaison de techniques, nous sommes en mesure de synthétiser cette représentation, et ainsi d'améliorer considérablement le délai d'exécution de l'analyse MHP sans en compromettre l'exactitude. Nous présentons également des résultats de recherche expérimentale démontrant l'utilité et l'incidence de notre approche et de nos optimisations en utilisant un éventail de bancs d'essai. Les résultats démontrent que nos optimisations sont efficientes et qu'elles permettent d'analyser des bancs d'essai plus imposants. Pour certains bancs d'essai, nos optimisations ont produit des réultats

impressionnants, accélérant l'analyse MHP selon plusieurs ordres de grandeur.

Acknowledgements

First of all, I would like to express my sincere gratitude to my thesis supervisor, Professor Clark Verbrugge, for his guidance and support throughout the course of this research work. This work would never have materialized without his insight and Advice. Professor Verbrugge dedicates a lot to his students and his profession; and he contributed a great deal of his time, effort and thought to the work presented in this dissertation. His constant financial support enabled me to concentrate on research work.

I am very grateful to Professor Laurie Hendren. She gives first rate compiler courses and many students have learned state-of-art compiler knowledge from her lectures. Her insight into compiler technology and interesting lectures benefitted me a lot. I highly respect her. Her cheerful nature and enthusiasm make the Sable Research Group full of fun.

I also would like to thank other members of Sable Research Group. Special thanks go to Ondřej Lhoták. His in-depth knowledged of the compiler and *points-to analysis*, and his consistent help and advice have been invaluable. Bruno Dufour is always glad to help, and I thank him for reviewing my abstract in French. The discussions with Feng Qian, Navindre Umanee, and Chris Pickett have been helpful to my research work. Thanks go to all the other members of the Sable Research Group, especially, John Jorgensen, David Bélanger, Marc Lanctot, Jennifer Lhoták, Nomair Naeem, Ahmer Ahmedani, Lin Wang, and Dayong Gu for their help and creating a pleasant working environment in the Sable Research Group.

Last, but not the least, I would like to thank my family for their continuous support and encouragement.

Contents

Abstract i

Résumé ii

Acknowledgements iv

Contents v

List of Figures viii

List of Tables ix

1 Introduction and Contributions 1

1.1 Motivation 1

1.2 Contributions 3

1.3 Thesis Organization 4

2 Related Work 5

3 Java Thread and Concurrency Model 9

3.1 Java Thread States 9

3.2 Java Lock Model 11

3.2.1 High-level Constructs for Synchronization 11

3.2.2 Wait and Notification 11

3.2.3 Bytecode-level Implementation of Synchronization 12

4 May Happen in Parallel (MHP) Analysis **14**
4.1 Limitations and Requirements 14
4.2 Parallel Execution Graph 15
4.3 A Worklist Flow Analysis Algorithm 18
4.3.1 Overview 18
4.3.2 Terminology 19
4.3.3 Worklist Version of MHP Flow Analysis 21
4.4 Discussion 25

5 MHP computing in the Context of Soot **27**
5.1 Soot Framework 27
5.2 MHP Computing in the Context of Soot 28
5.2.1 Jimple 28
5.2.2 Intra-procedural Analysis 32
5.2.3 Inter-procedural Analysis 33
5.3 MHP Analysis in Soot 35

6 Design and Implementation of Parallel Execution Graph **37**
6.1 PegCallGraph 40
6.2 Finding Runtime Objects 42
6.2.1 Finding Out if a Statement Is Executed only Once Inside a Method 44
6.2.2 Finding Out If a Method May Be Called More Than Once . . 44
6.2.3 Finding Allocation Sites and If the Allocation Sites May Represent More Than One Object 48
6.3 Finding Target Methods 50
6.4 Runtime Objects and Alias Resolution for Threads 51
6.5 Parallel Execution Graph in the Context of Soot 51
6.5.1 Nodes 53
6.5.2 Edges 53
6.6 PEG construction 55

6.6.1 Finding Methods that Need to Be Inlined 55
6.6.2 Safety of Inlining 57
6.6.3 Discovering Inlining Order 57
6.6.4 Special Handling for Synchronized Methods 58

7 MHP Implementation and Optimization **59**
7.1 Finding Monitor Objects 59
7.2 Implementation of the Worklist Flow Analysis Algorithm 61
7.3 Optimizations 61
7.3.1 Merging Strongly Connected Components 62
7.3.2 Merging Sequential Nodes 63
7.3.3 Updating the PEG 65

8 Experimental Results **66**
8.1 Benchmarks 66
8.2 Results 68

9 Conclusions and Future work **73**
9.1 Conclusion 73
9.2 Future work 74

Appendices

Bibliography **76**

List of Figures

3.1 Thread states 10
3.2 An example of synchronization in bytecode 13

4.1 An example of a PEG, a simplified version of figure 3 in [NSA99]. . . 16

5.1 An example Java code 29
5.2 Corresponding bytecode of Figure 5.1 30
5.3 Corresponding Jimple code of Figure 5.1 31
5.4 Method relationships in Call Graphs. 33
5.5 Class inheritance hierarchy. 34
5.6 Overview of our MHP analysis. 36

6.1 An example of recursive method invocations 38
6.2 An example of a trimmed PegCallGraph 42
6.3 An example of an allocation site corresponding to multi-objects . . . 43
6.4 A PegCallGraph containing multi-called methods 46
6.5 The algorithm for computing multi-called methods 47
6.6 An example of thread actions needing alias resolution 52
6.7 An example of PegCallGraph 56

7.1 Three different types of locks in Java 60
7.2 An example of Strongly Connected Component 62
7.3 An example of sequential nodes 64

List of Tables

6.1 Summary of methods of interface DirectedGraph 41
6.2 JVM Instruction VS. InvokeExpr in Soot 50

8.1 Experimental results without PEG simplification 67
8.2 Experimental results without PEG simplification 69
8.3 Experimental results after optimization 71
8.4 Experimental results after optimization 72

Chapter 1
Introduction and Contributions

1.1 Motivation

Java is a popular, high-level, object-oriented language [GJSB00] designed to support various architectures. Java provides a rich set of language features, including garbage collection, runtime safety checks, dynamic loading, *etc.* One important characteristic that makes Java unique among most general-purpose programming languages like C and C++ is that it has explicit, built-in support for *concurrent* programming. A programmer can specify an application containing threads of execution, and each thread designates a part of the program that may execute in parallel with other threads. This capability, called *concurrent* or *multi-threaded* programming, gives Java developers powerful capabilities not available in C and C++. (Of course, external multi-threaded libraries for C, C++ are commonly available [NBF96]).

As with most computer languages, Java programs are optimized by various computer transformations during compilation (and in the case of Java, during actual execution as well). Unfortunately, traditional compiler optimization is developed in the context of sequential or single-threaded programs, and it is not trivial to extend a sequential optimization to a concurrent situation.

Attempts have naturally been made in compiler optimization to rise to the challenge of extending the scope of sequential analysis and optimization to concurrent

multi-threaded programs. Specific techniques for handling problems related to compiling multi-threaded languages are being actively researched, *e.g.*, synchronization removal [E.R00], and race detection [CLL+02]. More general techniques, however, that also allow one to compute the impact of concurrency on other compiler analyses or optimizations are still desirable. In concurrent programs, information about which statements could be executed by different threads at the same time can be used for detecting data races, program optimization, debugging, program understanding, and improving the accuracy of data flow analysis. Such a more general approach for Java is provided by Naumovich *et al.*'s *May Happen in Parallel* (MHP) analysis [NSA99]. This analysis only determines which statements may be executed concurrently, but from this information on potential data races and synchronization problems can be derived.

The original MHP algorithm relies on a simplified program structure. All methods need to be *inlined*, and *cloning* is necessary to eliminate polymorphism and aliasing. Unfortunately, while these limitations still allow a variety of small applications to be analyzed, the associated potential exponential growth in code size due to these techniques means they cannot be feasibly applied to more complex programs. Whole program inlining is not possible for non-trivial programs, and moreover excludes many recursive programs. Cloning further expands the program size, and even in the presence of good alias resolution is likely to cause space concerns. Thus although Naumovich *et al.*'s results are encouraging, it is important to also know how well the analysis would work in a more practical compiler setting.

In this thesis, we present our experiences with an implementation of MHP for Java that attempts to address such practical concerns. Our implementation of MHP incorporates several simple analyses as well as modifications to MHP structures in order to reduce many of the practical limitations. We also provide experimental results and show how simple optimizations on the MHP internal data structures can make MHP analysis of even moderate size programs quite feasible.

1.2 Contributions

This thesis aims at providing a practical MHP information analysis. Computing MHP information involves two aspects: building the appropriate internal data structures, and running algorithms based on them. As mentioned earlier, MHP analysis requires the whole program be *inlined* producing one large graph representing the complete program execution. We have developed new analyses and techniques to reduce the size of the graphs, and identify and handle certain common situations that prevent inlining, such as recursive method calls. A further difficulty with performance in MHP analysis is its reliance on knowing exact, runtime object identities. In the original MHP analysis [NSA99], this is handled using *cloning*, or code replication, to eliminated aliases. Finding runtime target objects and methods is very important to get correct and precise information. In our case, we use SPARK [LH03], a fast and precise flow analysis to resolve aliases, and then find runtime targets through the aid of a further custom flow analysis. This is a more feasible technique than cloning, although it will in general imply a tradeoff between precision and efficiency.

Through these improvements, we have been able to run MHP analysis on a variety of "moderately-large" programs.

In summary, the main contributions of this thesis are as follows:

- **Design and Implementation**

  We have implemented MHP analysis in the context of the Soot program analysis framework [VRHS+99]. We have used features and other analyses available in Soot to assist the computation of MHP information. We further develop several small, custom analyses to support practical considerations. Our implementation thus demonstrates that MHP information can be practically computed, and also what aspects of a compiler optimization and analysis infrastructures are required and/or useful for such a task.

- **Optimization**

  Even with special techniques for improving inlining, the data structures used to represent the whole program may be huge. We use two approaches based on

knowing how MHP information is computed to simplify graphs, reducing the size of the internal data structures and improving analysis performance. The specific techniques we use, merging strongly connected components and merging sequential nodes, are fairly straightforward graph simplication techniques, but their application in this context is shown to result in an impressive improvement in the cost of MHP analysis.

- **Experiments**

  Our final implementation is sufficiently practical to analyze non-trivial benchmarks. We provide the first MHP information computation on non-trivial programs. These experiments reveal that benchmarks that are larger in terms of code size do not always consume more analysis time. Thread communication and synchronization complexity is a better indication of MHP cost. Finally, the comparison between the results before and after optimization proves that our optimization greatly improves the performance.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Related work is described in Chapter 2. In Chapter 3, we introduce the Java thread states and constructs related to the synchronization. In Chapter 4, we give a brief description of Gleb Naumovich *et al.*'s MHP analysis [NSA99]. In Chapter 5, we describe our MHP computation structures in the context of Soot. Further details on our implementation and improvement are then developed in Chapter 6 and 7. In Chapter 8, we introduce our benchmarks and describe our experimental results and analysis. Finally, we state our conclusions and future work in Chapter 9.

Chapter 2
Related Work

Our work here is based most directly on the MHP analysis originally designed by Naumovich *et al.* [NSA99]. There are of course other approaches to analyze and represent concurrent programs. Some of them are general purpose, while others have specific purposes. In the latter case, *datarace* detection is perhaps the most common intended application. A *datarace* occurs when two read/write operations access the same memory location (like a variable) without ordering constraints between the two operations, and at least one of them is a write. Dataraces result in indeterminacy in concurrent programs, and are programming errors in most cases. Datarace detection is very important for multi-threaded programs.

Whatever the goal, an appropriate representation of the concurrent program is critical. Traditionally, Control Flow Graphs (CFGs) are used for intermediate representation of sequential programs in compiler analysis [Muc97]. But CFGs have many limitations in representing parallel programs; in particular, they are not able to describe non-sequential control flow.

Program Dependence Graphs (PDGs) [FJW87] only present essential data dependence relationships and control dependence relationships, without unnecessary sequencing in the control flow graph. Because dependences in PDGs connect relevant parts of a program, some optimizations using PDGs require less time to perform than with other program representations. PDGs can be used for general program optimizations where dependency is a concern; for example, detecting medium to

fine-grain parallelism for sequential programs. And they also can be used in other contexts, for example, performing slicing in software development or maintenance. They are however not designed to represent parallel programs, and cannot represent, for example, notification, locks, and the parallel execution of multiple threads. Thus PDGs have limitations in representing real parallel programs and while they are an improvement over CFGs, they are not suitable for analyzing a language such as Java.

Parallel Program Graphs (PPGs) [Sar97, SS98] allow the representation of both sequential programs and parallel programs. They can be used for determining the semantic equivalence of parallel programs, detecting deadlocks, program optimization, and have applications in automatic code generation. PPGs are generalizations of PDGs and CFGs. Similar to control dependence and data dependence edges in PDGs, PPGs contain control edges representing parallel flow of control and synchronization edges representing ordering constraints of execution instances of PPG nodes. In addition, PPGs also contain special "MGOTO" nodes. An *MGOTO* node represents the construction of parallel threads; *i.e.*, the immediate successors of an MGOTO node are in different, parallel threads. Compared to PDGs, with MGOTO nodes, PPGs can be used to fully represent sequential and parallel programs. Unfortunately, although PPGs are more general than PDGs and CFGs, not all parallel constructs can be directly mapped to PPGs; *e.g.*, it is not possible to represent a situation where synchronization conditions depend on runtime data values. In addition, analysis and optimization [Sar97] can only be applied to deterministic parallel programs, and more work needs to be done to handle non-deterministic parallel programs.

Ferrante *et al.* [FGS97] proposed a *Parallel Control Flow Graph* (PFG) for optimizing explicitly parallel programs. They provided dataflow equations for the reaching definitions analysis and used a *copy-in/copy-out* semantics for accessing shared variables in parallel constructs. The *copy-in/copy-out* semantics allow each created thread to copy variables at the beginning and modify its own copy at the end of the thread execution or at a `Wait` statement (in post/wait schema). Thus PFGs cannot represent general parallel constructs (like busy-wait synchronization), nor can they handle programs containing dataraces.

A *Concurrent Control Flow Graph* (CCFG) [Lee99] is an intermediate representation for explicitly parallel programs with structured concurrency control ("cobegin/coend" parallel constructs) and post/wait synchronization. CCFGs are similar to PPGs and PFGs, but differ in that CCFGs contain "conflict" edges in addition to synchronization and control flow edges. A *conflict* edge is a bidirectional edge connecting two *basic blocks* (straight-line sections of code) that may be executed in parallel and each of these two basic blocks contains at most one shared memory location access.

Approaches have also been considered that build on other well-known sequential representations. Static single assignment (SSA) form, for instance, is a more and more popular intermediate representation for sequential programs. Lee *et al.* [LPA97a] used CCFGs as intermediate representations for parallel programs in order to transform the programs to *Concurrent Static Single Assignment* (CSSA) form, which provides advantages of SSA form in a concurrent setting.

Any analysis of Java must consider the fact that Java is an object-oriented languages. Information about the sharing of objects, especially the sharing of objects by threads is important for a compiler of an object-oriented programming language. Escape analysis [Bla99, BU99, WR99, CGS+99, E.R00] can be used to compute this specific information. Christoph von Praun and Thomas R.Gross [vPR03] used Object Use Graphs (OURS) in the context of escape analysis. OUGs are an extension of Heap Shape Graphs (HSGs), in which the nodes represent the runtime objects and edges denotes the reference relations between these objects. Computed during compile-time, HSGs can be used to represent the information of the sharing of runtime objects. But an HSG may lose some precision; for example, an object is regarded as shared when accessed by two threads. This is not always true. If the second thread starts after the first one terminates, then the object is not shared by these two threads. OUGs can detect these cases. The OUG can find the structural, temporal, and lock-based protection of accesses in different periods of an object by using control flow analysis in different threads and information about lock protection, object escape, thread-start and join. Escape analysis approaches, while useful for the intended goal, are not as general as MHP analysis.

For the purpose of data race detection, Savage *et al.* developed *Eraser*, a race checker in the C, C++ environment. Of course, the C/C++ threading model is not identical to the one in Java (though there are strong similarities). Jong-Deok Choi *et al.* [CLL+02] proposed an approach for datarace detection for object-oriented languages, and demonstrated their approach in Java. They use a combination of dynamic checking and static analysis, the latter being itself a combination of interthread control flow analysis and a flow-insensitive inter-procedural points-to analysis. *Interthread Control Flow Graphs* (ICFGs) are used to represent multi-threaded programs and *Interthread Call Graphs* (ICGs) are used as abstractions of the ICFGs for scalability. At each stage of their analysis and implementation they provide optimizations to improve efficiency and precision. The "weaker-than" relation is used during both static analysis and dynamic detection. Informally, if event a and any other event c has a datarace and this implies that events b and c must have a datarace, we say b *is weaker-than* a. Thus if we have the information b *weaker-than* a, during datarace detection, we only need think about b without thinking about a. This can reduce both time and space overhead. Another reduction of overhead results from reporting dataraces only once. A lot of accesses may have a datarace in the same memory location, but they guarantee reporting at least one access rather than all of the accesses.

Flanagan and Freund analyze large Java programs for race conditions by examining user-provided type annotations for code [FF00]. Improvements to accuracy and efficiency of data race detection continue to be addressed; e.g., through dynamic techniques [vG01], and by combining information from multiple analyses [OC03].

A similar concentration of efforts has looked at other concurrency related program analysis and optimization problem, such as synchronization removal [E.R00, BU99]. The aim of MHP analysis is to give general information useful to a wide variety of situations, and not to focus on a specific aspect of concurrency analysis.

Our implementation and optimization techniques largely depend on a combination of well known approaches. Good points-to analysis is one of the more complex and expensive compiler problems, and has been addressed in a variety of settings [EGH94, Ste96,RMR01,BLQ+03]. SPARK [LH03] produces precise points-to information, and this has been quite crucial to our ability to analyze non-trivial programs.

Chapter 3
Java Thread and Concurrency Model

In this chapter, we introduce the Java infrastructure related to concurrency. Java has built-in support for multi-threaded programming and provides the communication of threads in its core. We present the states of a thread in Java and how threads transform between these states. Furthermore, when many threads are started and interacting with an object, some mechanism is needed to ensure the safety of these threads, *i.e.*, prevent the threads from adversely affecting one another. This mechanism is also introduced in this chapter.

3.1 Java Thread States

In Java, concurrency is modelled with *threads*, which are defined using a `Thread` class. Threads in Java follow a fairly standard lifecycle [HC02], and the different states of a thread are shown in Figure 3.1. Creating a Thread object is the only way to **create** a thread. When the thread is created, it does not begin to execute until its `start()` method is called. The `start()` method does low level thread initialization, so the thread can automatically execute its `run()` method. When the operating system assigns a processor to it, the code inside the thread begin to execute, *i.e.*, it is in the **running** state. A thread becomes **dead** once the execution of its `run` method is done or terminated because of an uncaught exception. Threads can also voluntarily give up their CPU time. When the `sleep` method of a running method is called,

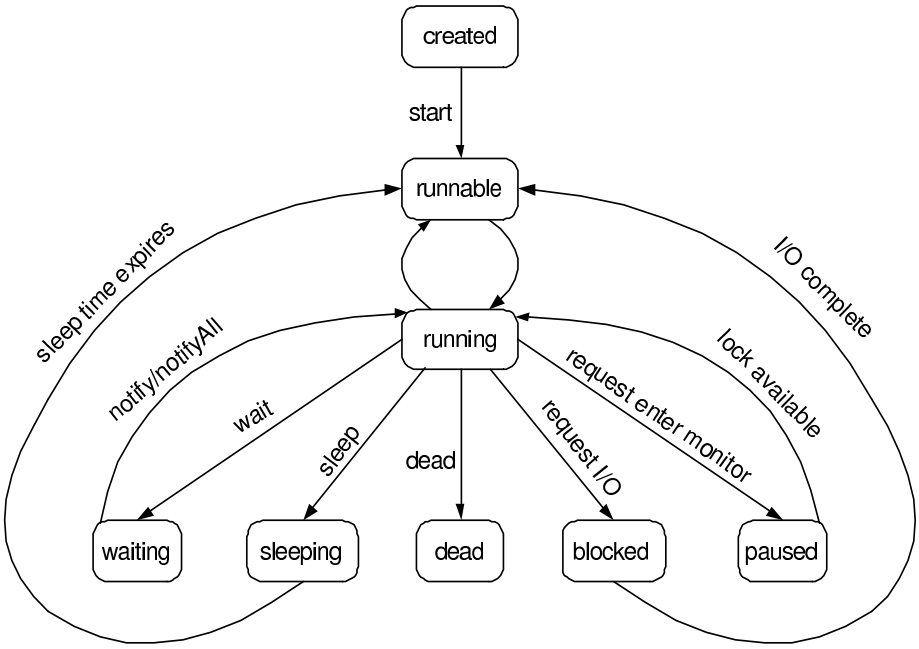


Figure 3.1: Thread states

the thread enters the **sleeping** state, becoming runnable again after the sleeping time expires. Java provides *condition synchronization*: a running thread enters the **waiting** state when it calls the `wait` method of an object. A thread in a waiting state for an object becomes runnable again when the `notify/notifyAll` method of the object is called by another thread. If a thread tries to lock an object which is already locked by another thread, it is temporarily **paused**. It becomes runnable when the thread owning the lock of the desired object releases the lock. A running thread enters a **blocked** state when the thread issues an input/output request, and remains there until the input/output operation is completed.

3.2 Java Lock Model

3.2.1 High-level Constructs for Synchronization

In Java, each object is associated with a lock. But there are no explicit high-level lock and unlock actions in Java programs; instead, two high-level constructs, *synchronized methods* and *synchronized statements*, are used to perform synchronization. A *synchronized method* of an object tries to obtain a lock on the object when it is invoked. The body of the *synchronized method* begins to execute after getting the lock; and it releases the lock upon return. A synchronized statement has the format shown in Figure 3.1. It evaluates the `expression` to obtain an object that is locked for the execution of the block. The curly braces used for scoping in the synchronized statement thus correspond to object lock/unlock actions.

As a difference from other lock models [NBF96], Java locks can always be "recursively locked"–a thread that has acquired, but not released a lock on an object may relock the same object, and must then unlock it as many times as it is locked in order to release it.

3.2.2 Wait and Notification

Although Java does not provide monitors in a true sense [Han99], it does provide a limited form of condition synchronization. The Java language defines `wait`, `notify`, and `notifyAll` operations to facilitate communication between threads, and like locking, these operations are performed relative to an object. A thread must obtain the lock of an object before executing `wait, notify,` and `notifyAll`. Once the lock is released, other threads can acquire the ownership. Informally, the `wait` operation releases the lock, suspends the current thread, and adds the current thread to the `wait set` for the object. Every object has an associated `wait set`, which contains a set of threads that want to lock the object but do not have the ownership. The waiting threads becomes eligible to run again when another thread performs a `notify` or `notifyAll` operation on the same object. A `notify` operation wakes one arbitrary waiting thread in the `wait set` of the object. The thread is removed from the `wait`

`set` and competes for the lock with other threads. The `notifyAll` operation wakes all waiting threads inside the `wait set` and every thread is removed from the `wait set`. These threads compete for the lock with each other, as well as the other threads attempting to execute a synchronized method or statement on the object. Once a thread re-acquires the lock, the `wait` operation is completed. Notice that Java does not in general include fairness guarantees of this.

3.2.3 Bytecode-level Implementation of Synchronization

JVM locks allow the creation of monitors and critical sections. After compilation to bytecode, these high-level abstractions must still be represented as bytecode. The *monitorenter* and *monitorexit* bytecode are thus used as JVM instructions to implement the lock and unlock actions of objects and the *wait/notify* is implemented through method calls. In Figure 3.2, (a) is an example of Java code containing *synchronized statements*, (b) is part of the corresponding bytecode. Note in (b), 3 is a *monitorenter* and both 9 and 15 are *monitorexit* instructions. The code from 13 to 17 implements an implicitly "finally" block, part of the way Java ensures synchronized blocks are properly exited even in the presence of exceptions.

While explicit *monitorenter/monitorexit* instructions are only for synchronized blocks, synchronized methods acquire and release the object lock implicitly with no special bytecode instructions. When a *synchronized method* is called, the JVM acquires the lock first, then executes the body of the method, and finally releases the lock again. The synchronization is released regardless of exceptional method exit.

```
void  Foo(Bar f) {
      synchronized(f){
          wait();
      }
}
```

(a) An example method

```
Method void Foo(Bar)
 0      aload_1                    // push f to stack
 1      dup                        // keep a copy of f in a local variable
 2      astore_1
 3      monitorenter               // lock f
 4      aload_0
 5      invokespecial #2 <Method void wait()>
 8      aload_2
 9      monitorexit                // unlock f
10      goto 18                    // done
13      astore_3                   // exception handle
14      aload_2
15      monitorexit                // make sure we unlock f
16      aload_3
17      athrow                     // rethrow exception
18      return
```

(b) Part of bytecode

Figure 3.2: An example of synchronization in bytecode

Chapter 4
May Happen in Parallel (MHP) Analysis

In this chapter, we describe the basics of the MHP algorithm proposed by Gleb Naumovich *et al.* [NSA99]. We introduce the important assumptions and requirements of this algorithm. MHP analysis relies on a particular internal data structure, the *Parallel Execution Graph* (PEG). We sketch out the major components of the PEG and its structure here. The algorithms used to compute MHP information are also presented in this chapter.

4.1 Limitations and Requirements

MHP analysis is not yet completely general. Input processes and information have to satisfy a number of limiting requirements as follows.

- Knowing an upper bound of started threads

  In general, the number of runtime threads in a program cannot be precisely determined *a priori*. However, MHP analysis requires data structures specific to individual threads, and so an upper bound on the number of thread instances must be known. This requirement represents reduced generality of input. For our benchmarks we have manually unrolled all thread creation loops, and otherwise ensured each thread creation site is easily identified.

- Alias resolution and cloning

  A requirement of the MHP analysis is that alias resolution is done, and code cloning used to eliminate polymorphism and ensure precise variable and method targets are known. For example, if a variable in a program may refer to two different threads, we can create a structure which contains a copy of the body of the first thread and a copy of the second thread. Alias resolution and cloning brings space concerns for moderate to large programs.

- Inlining

  The MHP analysis uses a simplified program structure. It uses an internal structure, a Parallel Execution Graph (PEG) to represent the whole program. Every method must be inlined to build the whole program representation. However, this whole program inlining is not feasible for non-trivial programs. Furthermore, inlining every method cannot be used in programs containing recursive method calls.

4.2 Parallel Execution Graph

A *Control Flow Graph* (CFG) is an abstract data structure used to represent programs in compiler optimization, program analysis, and so on. CFGs consist of nodes and directed edges. Each node in a CFG represents a *basic block*, *i.e.*, a straight-line piece of code that can only be entered at the beginning and exited at the end. Directed edges point to the targets of branches or jumps in the control flow from the current node. CFGs are described in most compiler optimization texts [Muc97].

The *Parallel Execution Graph* or PEG is a superstructure of a normal control flow graph. Special edges and nodes are incorporated to explicitly represent potential thread communication and synchronization. Since thread bounds are known, the actions of each thread are also uniquely represented in the graph. Figure 4.1 gives an example of a PEG for a simple program that launches two threads `t1` and `t2` from a main thread and then attempts to signal them using a global lock and a

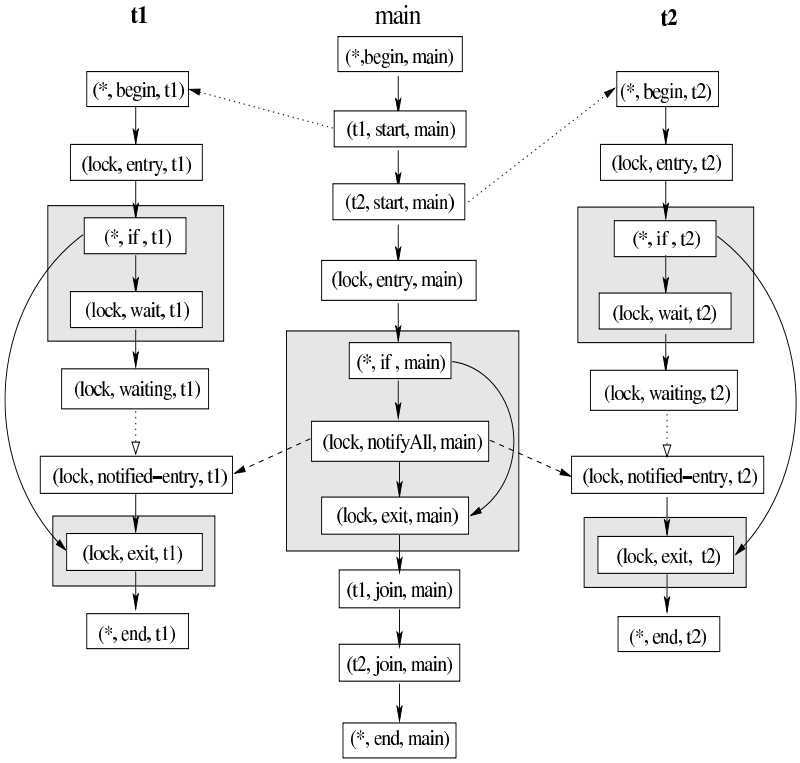


Figure 4.1: An example of a PEG, a simplified version of figure 3 in [NSA99].

wait/notify pattern. In Figure 4.1 nodes in the shaded areas are actually protected by the monitor.

Nodes in PEG's are structured as triples; e.g., for *communication methods* (wait, notify, *etc.*) the triple `(object, name, caller)` is used, where the field `object` represents the object controlling the communication, `name` is the method name, and `caller` is the caller thread name. For nodes that do not represent *communication methods*, a wildcard symbol (*) is used for the object field.

Certain new nodes are added to aid in later analysis. Most simply, `(*, begin,t)` and `(*, end,t)` nodes are inserted to mark the beginning and end of each thread t, and `(lock, entry,t)` and `(lock, exit,t)` nodes indicate `monitorenter` and `monitorexit` operations by t on object `lock`. Condition synchronization is only slightly more complex. A `wait()` method call is broken down into a chain of `wait`,

`waiting` and `notified-entry` nodes, representing the substeps of starting the call to `wait()`, actually sleeping after the lock is released, and having been notified and trying to reacquire the lock, respectively. For example, in Figure 4.1, `(lock, wait, t1)`, `(lock, waiting, t1)`, and `(lock, notified-entry, t1)` represent thread $t1$'s call to `lock.wait()`.

There are four kinds of edges in PEGs: wait edges, local edges, notify edges and start edges:

- Waiting edge

  A *waiting edge* is an edge between waiting nodes and notified-entry nodes. In Figure 4.1, the dotted edges with empty arrowheads from `(lock, waiting, t1)` to `(lock, notified-entry, t1)` and from `(lock, waiting, t2)` to `(lock, notified-entry, t2)` are *waiting edges*. These represent the transition from waiting to being notified.

- Notify edge

  Notify edges are dynamically created during the analysis process. They allow precedence information to flow from the notifier to the waiting thread, representing the inter-thread communications implied by the notify call. Since they are inserted during analysis, this information flow can be more precise than a static approach. Notify edges are only inserted from an `(object, notify/notifyAll, t1)` node to a `(object, notified-entry, t2)` node if the same object is involved, the threads are distinct, and the analysis has computed that these two events may indeed happen in parallel. For example, in Figure 4.1, the dashed edges represent notify edges. There are two notify edges in this figure, an edge from `(lock, notifyAll, main)` to `(lock, notified-entry, t1)`, and another one from `(lock, notifyAll, main)` to `(lock, notified-entry, t2)`. This conservatively represents the nondeterminism that is inherent in Java's notify mechanism: one, none, or both of t1 and t2 will be actually waiting when the main thread performs its notify call. Both of them have chance to be notified and compete for the lock again.

- Start edge

  A *start edge* is created from a call to `Thread.start()` to the first action of the initiated thread. Start edges allow information from before a thread has started to flow to the new thread, and ensures that events prior to a start statement are never in parallel with events after. These edges are shown in Figure 4.1 as dotted lines with solid arrowheads, *i.e.*, the dotted edge from `(t1, start, *)` to `(*, begin, t1)` and the one from `(t2, start, *)` to `(*, begin, t2)` are *start edges*.

- Local edge

  A *local edge* represents normal, intra-thread control flow, not dependent on thread communication. These edges are inherited from the base CFG, and are shown as solid edges in Figure 4.1. All the edges that are not *waiting edges*, *notify edges*, or *start edges* are *local edges*.

4.3 A Worklist Flow Analysis Algorithm

4.3.1 Overview

MHP analysis is performed using a worklist dataflow algorithm. The goal is to find for each PEG node the set of other PEG nodes which may execute concurrently. For each PEG node a set $M(n)$ is initialized to the empty set, and a least fixed-point based flow algorithm propagates set information around the PEG. $M(n)$ then contains the set of nodes which may execute in parallel with n. Although this largely follows the template of a standard dataflow analysis, with special modifications to create notify edges and flow information across and through the various special edges and nodes, the algorithm also includes a "symmetry step" to guarantee that if $m \in M(n)$ then $n \in M(m)$, *i.e.*, if m may be executed in parallel with n, then certainly n may happen in parallel with m. This non-standard component of the analysis ensures information is accurately maintained as the actions of concurrently executing threads

are analyzed. Note that as with most static analyses the computed information is a conservative approximation.

4.3.2 Terminology

Some functions and terms are used to help illustrate this algorithm.

Functions

- LocalPred(n) and LocalSucc(n)

  LocalPred(n) and LocalSucc(n) return the collection containing all immediate local predecessors and successors of n respectively. Local predecessors/successors are the predecessors/successors in the same thread.

- NotifyPred(n) and NotifySucc(n)

  Notify edges go from `notify` or `notifyAll` nodes to the corresponding `notified-entry` nodes. NotifyPred(n) computes the set of all the `notify` predecessors of a `notified-entry` node n. NotifySucc(n) returns the set of all the `notified-entry` successors of a `notify` node or `notifyAll` node n.

- StartPred(n) and StartSucc(n)

  As introduced in section 4.2, the PEG contains start edges that go from `start` nodes to the first node of each started thread. The first node of a thread is always a `begin` node. StartPred(n) returns the set of all the `start` predecessors of a `begin` node of a thread. StartSucc(n) returns the collection of `begin` successor nodes of a `start` node n.

- WaitingPred(n) and WaitingSucc(n)

  WaitingPred(n) returns a `waiting` predecessor of a `notified-entry` node n. WaitingSucc(n) returns the `notified-entry` successor of a `waiting` node n.

- N(t)

  All PEG nodes in the thread t.

Collections

Three collections are associated with each lock object in this algorithm:

- notifyNodes(*obj*)

  notifyNodes(*obj*) contains the collection of all `notify` and `notifyAll` nodes for lock object *obj*. Such nodes have a format of `(obj, notify, caller)` or `(obj, notifyAll, caller)`, where `caller` can be any thread but every node in the same collection has the same *obj* field. In Figure 4.1, the notifiedNodes(*lock*) contains one element, *i.e.*, `(lock, notifyAll, main)`.

- waitingNodes(*obj*)

  waitingNodes(*obj*) contains the collection of all `waiting` nodes of lock object *obj*. For example, in Figure 4.1, waitingNodes(*lock*) contains two elements, *i.e.*, `(lock, waiting, t1)` and `(lock, waiting, t2)`.

- Monitor(*obj*)

  Monitor(*obj*) represents the collection of PEG nodes in the monitor of the lock object *obj*. For example, in Figure 4.1, the nodes insides the shaded area are protected by the monitor. Specifically, Monitor(*lock*) contains the following nodes:

  - `(*, if, t1)`, `(lock, wait, t1)`, and `(lock, exit, t1)` in thread $t1$,
  - `(*, if, t2)`, `(lock, wait, t2)`, and `(lock, exit, t2)` in thread $t2$,
  - `(*, if, main)`, `(lock, notifyAll, main)`, and `(lock, exit, main)` in thread *main*.

Computing Notify Edges

There are four type of edges in a PEG. `Wait` edges, `local` edges, and `start` edges are included as part of building the PEG. But `notify` edges are built during the flow analysis—the flow analysis computes `notify` successors of `notify` nodes. The

equation for computing `notify` successors is below. This simply adds a notify edge if a `notify` node and a matching `waiting` node may happen in parallel.

$$notifySucc(n) = \begin{cases} \{\text{m} \mid \text{m} \in (\text{obj, notified-entry, *}) \wedge \\ \quad \text{WaitingPred(m)} \in \text{M(n)}\}, & \text{if n} \in \textit{notifyNodes}(\text{obj}) \\ \text{undefined}, & \textit{otherwise}. \end{cases} \tag{4.1}$$

4.3.3 Worklist Version of MHP Flow Analysis

Dynamic Computing Equations

This algorithm mostly follows the technology of a standard data flow analysis on a CFG [Muc97]. Here we introduce and describe the equations verbatim from the original paper used to compute information during the flow analysis. The following well-known data flow analysis equation, specialized to each kind of node, is used to define how information is propagated.

$$OUT(\text{n}) = GEN(\text{n}) \cup M(\text{n}) \setminus KILL(\text{n}) \tag{4.2}$$

M(n) contains the nodes that may be executed in parallel with n, while $OUT(n)$ contains the nodes that may be executed in parallel with the successor of n. $GEN(n)$ contains the nodes that may be executed in parallel with the next node but not with n. Equation 4.2 gives the rule to compute the GEN set of node n. If the current node is a `start` node, GEN set only contains the `begin` node of the target thread. If the current node is a `notify` node or `notifyAll` node for Object `obj`, GEN consists of the notify successors, *i.e.*, all the `notified-entry` nodes of Object `obj`. In all the other cases, GEN is empty.

$$GEN(n) = \begin{cases} (\text{*, begin, t}), & \text{if n} \in (\text{t,start,*}) \\ \textit{NotifySucc}(\text{n}), & \text{if } \exists \text{obj} : \text{n} \in \textit{notifyNodes}(\text{obj}) \\ \emptyset, & \textit{otherwise} \end{cases} \tag{4.3}$$

$KILL(n)$ contains the nodes that will definitely not be executed in parallel with the next node, although they may be in parallel with n. The below equation gives the rule for how to compute the $KILL(n)$ set.

$$KILL(n) = \begin{cases} N(t), & \text{if n} \in \text{(t, join, *)} \\ \textit{Monitor(obj)}, & \text{if n} \in \text{(obj, entry, *)} \cup \\ & \quad \text{(obj, notified-entry, *)} \\ \textit{waitingNodes(obj)}, & \text{if (n} \in \text{(obj, notify, *)} \wedge \\ & \quad |\textit{waitingNodes}(\text{obj})| = 1) \vee \\ & \quad \text{(n} \in \text{(obj, notifyAll, *))} \\ \emptyset, & \textit{otherwise} \end{cases} \tag{4.4}$$

If the current node n is a `join` node, the successors of n cannot be executed with any statement of the target thread of n because the finish of the execution of n means the execution of the target thread terminates. Thus the $KILL$ set is all of thread t. If the current node is an `entry` or `notified-entry` node of object `obj`, which means it is outside the monitor of `obj` and trying to enter the monitor, the $KILL$ set consists of all the statements inside the monitor of `obj`. If the current node n is a `notify` node of object `obj` and there is only one `waiting` node for `obj`, the waiting node is "notified" and cannot be in a waiting state any longer. If the current node n represents a `notifyAll` node of object `obj`, all the `waiting` nodes are "notified" and no thread is waiting for `obj`. Thus, in these two cases, the $KILL$ set consists of the `waiting` nodes of object `obj`. In the rest of the cases, the $KILL$ set is empty.

$M(n)$ contains the nodes that may be executed in parallel with n. During the flow analysis, the M set is recomputed and added to the old M set of the current node. The following equation gives the rule for computing the M set of the current node n.

Computing M Sets

$$M(\mathrm{n}) = M(\mathrm{n}) \cup \begin{cases} \bigcup_{p \in StartPred(n)} OUT(p) & \\ \quad \backslash N(thread(n))), & \text{if n} \in (*, \text{begin}, *) \\ ((\bigcup_{p \in NotifyPred(n)} OUT(p)) & \\ \quad \cap OUT(WaitingPred(n))) & \\ \quad \cup GEN_{notifyAll}(n), & \text{if n} \in (*, \text{notified-entry}, *) \\ \bigcup_{p \in LocalPred(n)} OUT(p), & otherwise \end{cases} \tag{4.5}$$

If the current node n is a `begin` node, the nodes that can be executed in parallel with it are the nodes in the OUT set of the `start` predecessors of n. If the current node n is a `notified-entry` node, the computation of the M set is more complicated and needs the following equation.

$$GEN_{notifyAll}(n) = \begin{cases} \emptyset, & \text{if n} \notin (\text{obj, notified-entry}, *) \\ \{\ \mathrm{m} | \mathrm{m} \in (\text{obj, notified-entry}, *) \wedge & \\ \quad WaitingPred(\mathrm{n}) \in M(WaitingPred(\mathrm{m})) \wedge & \\ \quad (\exists\ \mathrm{r} \in N\text{: r} \in (\text{obj, notifyAll}, *) \wedge & \\ \quad \mathrm{r} \in (M(WaitingPred(\mathrm{m})) \cap M\ (WaitingPred(\mathrm{n})))\ \} & \\ & \text{if n} \in (\text{obj, notified-entry}, *) \end{cases} \tag{4.6}$$

If a `notifyAll` statement wakes all the waiting threads, all the corresponding `notified-entry` nodes in these thread may be executed in parallel. In this situation, the $GEN_{notifyAll}(n)$ set contains all the other `notified-entry` nodes. Specifically, a `notified-entry` node m should in the $GEN_{notifyAll}(n)$ if the waiting predecessor of n may happen in parallel with the waiting predecessor of m and there is a `notifyAll` node which may happen in parallel with both `waiting` predecessors of m and n.

To compute the M set of a `notified-entry` node n, we first find the union of the OUT set of the notify predecessors of n. Then the intersection of that union with the OUT set of the waiting predecessor of n is computed. Finally, the nodes in the $GEN_{notifyAll}(n)$ are added to the result. Computing the M set in the rest of the cases is intuitive: the M set consists of the union of the OUT set of all the local

predecessors of the current node.

Worklist Flow Analysis Algorithm

Algorithm 1 The first stage: Initialization

$\forall$ n $\in N$: *KILL(n)* = *GEN(n)* = *M(n)* = *OUT(n)* = $\emptyset$

Initialize the worklist W to include all start nodes in the *main* thread that are reachable from the *begin* node of the *main* thread

$\forall$ n $\in N$:

case

n $\in$ (t, join, *) $\Rightarrow$ $KILL$(n) = N(t)

n $\in$ (obj, entry, *) $\cup$ (obj, notified-entry, *) $\Rightarrow$ $KILL$(n) = $Monitor_{obj}$

n $\in$ (obj, notifyAll, *)$\Rightarrow$ $KILL$(n) = *waitingNodes*(obj)

n $\in$ (obj, notify, *) $\Rightarrow$

if $|waitingNodes(obj)| = 1$ then

$KILL$(n) = *waitingNodes*(obj)

n $\in$ (t, start, *) $\Rightarrow$ GEN(n) = (*, begin, t)

The worklist algorithm is divided into two stages as Algorithm 1 and Algorithm 2 (these algorithms are verbatim from [NSA99]). The first stage as shown in Algorithm 1 is the initialization stage. First of all, the *KILL(n)*, *GEN(n)*, *M(n)*, and *OUT(n)* set of each node is initialized to empty; and the worklist W is initialized to only contain the `start` nodes of the `main` thread. Then the *KILL* sets are computed for all nodes and the *GEN* sets for the `start` nodes.

The second stage shown in Algorithm 2 is the main loop stage of MHP algorithms. The inputs of this stage are graphs representing all threads and the initialized sets for each node. In this stage, $M(n)$ and $OUT(n)$ collections for each node are computed, and this information is propagated to the next nodes. W is the worklist containing nodes to be processed in the algorithm. The execution of this stage begins with the first node n of the worklist W, and n is deleted from W as in line 1 and 2. Lines 3 and 4 backup the current value of the M and OUT set of the current node to M_{old}

and OUT_{old} respectively. There are four kind of edges in PEGs and only *notify* edges are computed dynamically. Lines 6 to 8 find the *NotifySucc()* of the current node if the current node is a *notifyNode* and also computes *notify edges*. After building the *notify edges*, the graphs representing all the threads are connected to become the real PEG. Lines 9 to 10 add any new `NotifySucc(n)` entries to the worklist W to ensure MHP information is propagated properly. Line 11 computes *GEN(n)* set for *notifyAll* nodes. Line 12 computes *M()* set for the current node. If the current node n is a *notifyNode*, the *GEN(n)* set is computed as line 13 and 14. Line 15 computes the *OUT(n)* set using equation 4.2. As previously mentioned, the difference of this algorithm from a standard forward flow analysis lies in that it has a special *symmetry step*. Lines 16 to 19 detail the *symmetry step*. The symmetry step is based on the observation that if a node x is in the *M()* set of a node y, then y must be in the *M()* set of node x. Lines 20 and 21 add the current nodes' successors to the worklist W when *OUT(n)* is different from the backup old value of *OUT()* of current node. The output of this stage is *M(n)* for each node in the PEG, which contains all the PEG nodes that may be executed in parallel with n.

4.4 Discussion

The limitations discussed in Section 4.1, particularly the inlining and cloning requirement mean that this MHP analysis is expensive and not feasible for moderate to large programs. Thus, a refined and more practical analysis is needed for computing MHP information. Some approaches to do this are described in the Chapter 6 and 7.

Algorithm 2 The second stage: Main loop

We evaluate the following statements repeatedly until $W = \emptyset$

// n is the current node:

1. $n = head(W)$

// n is removed from the worklist:

2. $W = tail(W)$

// M_{old}, OUT_{old}, and $NotifySucc_{old}$ are th copies of the M, OUT,
// and *NotifySucc$_{old}$ sets for this node, computed to determine*
// new nodes inserted in these sets on this iteration

3. $M_{old} = M(n)$
4. $OUT_{old} = OUT(n)$
5. $NotifySucc_{old} = NotifySucc(n)$

//computing the new set of notify successors for **notify** and **notifyAll** nodes

6. if $\exists o : n \in notifyNodes$(obj)
7. $\forall m \in M(n) \bigcap waitingNodes$(obj):
   //create a new notify edge from node n to the waiting
   //successor of node m
8. $NotifySucc(n) = NotifySucc(n) \cup WaitingSucc(m)$

//if new notify edges were added from this node

9. if $NotifySucc_{old}(n) \neq NotifySucc(n)$ then
10. $W = W \bigcup NotifySucc(n)$
11. Compute the set $GEN_{notifyAll}(n)$ as in equation 2.6
12. Compute the set (M(n)) as in equation 2.5

//the only nodes for which the *GEN* set has to be recomputed are notify
//and notifyAll nodes; their *GEN* sets are their notify successors:

13. if $\exists o : n \in notifyNodes$(obj) then
14. $GEN(n) = NotifySucc(n)$
15. Compute the set $OUT(n)$ as in equation 2.2

//do the symmetry step for all new nodes in $M(n)$:

16. if $M_{old} \neq M(n)$ then
17. $\forall\ m \in (M(n) \setminus M_{old}(n))$:
18. $M(m) = M(m) \cup n$
    //add m to the worklist because the change in $M(m)$ may lead to a
    //change in OUT(m)
19. $W = W \cup$ m

//if new nodes has been added to the OUT set of n, add all n's successors
//to the worklist

20. if $OUT_{old} \neq OUT(n)$
21. $W = W \cup (LocalSucc(n) \cup StartSucc(n)$

Chapter 5
MHP computing in the Context of Soot

Our implementation is based on Soot [VRHS+99], a free compiler infrastructure written in Java. One of the goals of this thesis is to extend Soot to allow multi-threaded Java program analysis. In this chapter, we introduce the Soot framework and the main components we used to simplify our effort. At the end of this chapter, we present how our MHP analysis is integrated into the Soot framework.

5.1 Soot Framework

The Soot framework was originally designed to provide a common infrastructure for analyzing and transforming Java bytecode. After years of development, it allows users to analyze, transform, optimize, and annotate Java bytecode. Currently, it has been extended to include decompilation and visualization.

Soot reads in a Java bytecode class file and converts it to different intermediate representations, according to a user's instruction and the needs of any analyses to be run.

Soot provides five intermediate representations (IR):

- **Baf**:

  A stack-based, streamlined representation of bytecode which is simpler to manipulate than bytecode itself.

- **Jimple**:

  A stackless, typed, "3-address" intermediate representation suitable for optimization. "3-address" code has been traditionally used in the process of compiler analyses and optimizations [VSD86].

- **Shimple**:

  A Static Single Assignment (SSA) [CFR+91] variation of Jimple.

- **Grimp**:

  An aggregated version of Jimple suitable for decompilation and code inspection. In this model, series of expressions are aggregated into more complicated expressions. It is closer to Java source code and much easier to read than **Baf** and **Jimple**.

- **Dava**: A structured representation used for decompiling Java [Mie03].

For each intermediate representation, Soot provides a corresponding processing phase and associated Application Program Interface (API). Baf, Jimple, and Grimp are unstructured representations designed to allow analyses and optimizations of Java bytecode at different levels. Shimple is an variation of Jimple provided for users needing the Static Single Assignment (SSA) form. Users can use Dava for Java decompilation. The Grimp IR can be converted into the Dava IR which, when printed to a text file is recompilable Java source code.

5.2 MHP Computing in the Context of Soot

5.2.1 Jimple

The main internal program representation in Soot is Jimple. Jimple is a typed, "3-address" code representation of bytecode. There are a number of advantages to use Jimple for our MHP implementation. Primarily, Jimple provides control flow graph construction and various control flow analyses that we can use for our MHP analysis.

```
    public  class A{
      private void act(){
1           synchronized(buffer)
2           {
3               a = b + c;
4               buffer.write();
5               buffer.notifyAll();
6           }
7     }
      ...
8   }
```

Figure 5.1: An example Java code

Jimple is also used by most Soot users, and so this simplifies interaction with other analyses that may wish to consume MHP information in the Soot framework.

Java source programs are compiled to bytecode (.class files), then transformed to Jimple IR (.jimple files). Figure 5.1 shows a segment of Java source code of method `act()` in class A. Figure 5.2 shows the corresponding bytecode of Figure 5.1. In Figure 5.2 instruction 0 pushes "this" to the stack while instruction 1 loads parameter `buffer`. Instructions 8 to 36 execute the synchronized block while 39 to 43 handle exceptions. Specifically, instructions 9 to 17 implement line 3, `a = b + c`, in Figure 5.1 while instruction 35 implements the unlock operation on `buffer`. Notice instruction 41 makes sure that `buffer` is unlocked in the case of any exceptional method exits.

In bytecode, full object/method synchronizations are also available, as are other low-level operations, such as *monitorenter/montorexit* (to enter/exit a synchronized block), four kinds of invocations (virtual, static, special, interface). Normally, a method invocation for an instance method is decided by the runtime type of the object, and these cases are implemented using the **invokevirtual** instruction (*e.g.*, lines 24 and 31 in Figure 5.2. Most of the method invocations in Java fall in the category of the "virtual invoke". The instruction **invokestatic** is used for invoking a static method. And the instruction **invokespecial** must be used when an instance initialization method is invoked, when a method in the superclass is invoked, or when a **private** method is invoked. The instruction `invokeinterface` is used when a

```
Method void act()
   0 aload_0                                          // push this on the stack
   1 getfield #5<Field Buffer buffer>                 // get parameter buffer
   4 dup
   5 astore_1                                         // save a copy for later unlock
   6 monitorenter                                     // lock buffer
   7 aload_0
   8 aload_0
   9 getfield #3 <Field int b>                        //get field b
  12 aload_0
  13 getfield #4 <Field int c>                        // get field c
  16 iadd                                             // b + c
  17 putfield #2 <Field int a>                        // put the result of b + c to a
  20 aload_0
  21 getfield #5 <Field Buffer buffer>                // get field buffer
  24 invokevirtual #6 <Method void write()>
  27 aload_0
  28 getfield #5 <Field Buffer buffer>                // get field buffer
  31 invokevirtual #7 <Method void notifyAll()>
  34 aload_1
  35 monitorexit                                      // unlock buffer
  36 goto 44
  39 astore_2                                         // exception handler
  40 aload_1
  41 monitorexit                                      // make sure we unlock buffer
  42 aload_2
  43 athrow                                           // rethrow exception
  44 return

  Exception table:
  from   to  target   type
    7    36     39    any
   39    42     39    any
```

Figure 5.2: Corresponding bytecode of Figure 5.1

```
        private void act()
        {
1           A  r0;
2           Buffer r1, $r3, $r4, $r5;
            {
3               A r0;
4               Buffer r1, $r3, $r4, $r5;
5               java.lang.Throwable r2, $r6;
6               int $i0;  $i1, $i2;

7               r0  := @this: A;
8               $r3 = r0.<A: Buffer buffer>;                    // load buffer
9               r1 = $r3;
10              entermonitor $r3;                               // lock buffer

            label0:
11              $i0 = r0.<A: int b>;                            // load b
12              $i1 = r0.<A: int c>;                            // load c
13              $i2 = $i0 + $i1;                                // b + c
14              r0.<A: int a> = $i2;
15              $r4 = r0.<A: Buffer buffer >;
16              virtualinvoke $r4. <Buffer: void write()>();
17              $r5 = r0.<A: Buffer buffer>;
18              virtualinvoke $r5.<java.lang.Object:void notifyAll()>();
19              exitmonitor   r1;                               // unlock buffer

            label1:
20              goto label5;

            label:2
21              $r6:= @caughtexception;                         // exception handler

            label3:
22              r2 = $r6;
23              exitmonitor  r1;                                // make sure unlock buffer

            label4:
24              throw r2;                                       // rethrow exception

            label5:
25              return;

26          catch java.lang.Throwable from label0 to label1 with label2;
27          catch java.lang.Throwable from label3 to label4 with label2;
        }
```

Figure 5.3: Corresponding Jimple code of Figure 5.1

method which is implemented by an interface is called.

Figure 5.3 shows the corresponding Jimple code of Figure 5.1 and Figure 5.2. Jimple does not have stack operations; instead, it uses some temporary variables to store operation results. For example line 11 to 14 in Figure 5.3 implement `a = b + c` and some temporary variables `$i0, $i1` and `$i2` are used in this implementation instead of stack operations. Notice that in Jimple, as in Java source code, variables are explicit in operations, whereas in bytecode operands are implicit stack locations. Jimple also provides different method invocation and monitor enter/exit structures.

5.2.2 Intra-procedural Analysis

Soot provides many useful analyses, both intra- and inter- procedural analyses. In this section, we introduce the intra-procedural analyses used in MHP computation.

- **Control Flow Graphs (CFGs)**

  Control Flow Graphs can have varied forms in Soot. `BlockGraph` is a traditional representation [Muc97] of control flow. To facilitate program analysis and optimization, the `UnitGraph` is provided in Soot, in which nodes represent statements in a program and edges indicate control dependence of the nodes. We use UnitGraphs for our intra-procedural analysis.

- **Flow analysis**

  For data flow analyses, Soot has two built-in intra-procedural analysis schemata: ForwardFlowAnalysis and BackwardFlowAnalysis. These provide standard data flow analysis frameworks suitable for most conventional compiler analyses. MHP analysis, however, is technically neither a forward flow analysis nor a backward flow analysis. Mostly it is a forward analysis with a special *symmetry step*, which we showed in Section 4.3.3. We implement our MHP analysis based on the ForwardFlowAnalysis framework, modified to incorporate the symmetry step.

Figure 5.4: Method relationships in Call Graphs.

5.2.3 Inter-procedural Analysis

Here, we introduce the inter-procedural analyses used in our MHP analysis.

- **Call Graphs**

  Consisting of nodes and directed edges, a *call graph* contains information about the possible targets of virtual method calls. For a single-threaded program, the call graph must include all the methods that can be reached from the **main** method. For a multi-threaded program, the call graph must include all the methods that can be reached from **main** method and the **start** or **run** methods of all instantiated threads or runnable objects passed to instantiated threads.

  Nodes in a call graph denote methods, and edges in a call graph represent possible calling relationships between the caller method and the callee method. For example, if we have a statement calling "B()" in method "A". Figure 5.4 expresses the calling relationship of the statements. Figure 5.4 (a) represents the case when there is only one call to B in method A while (b) denotes a situation in which A calls B twice. In Java programs, call graphs are important for whole-program analyses because method bodies tend to be small, and method calls very frequently [DDHV03].

- **Class hierarchy analysis (CHA)**

  The Soot framework also provides *Class hierarchy analysis*. *Class hierarchy analysis* [DGC95] conservatively estimates the run-time targets of method calls by using the class-subclass relationships in the type hierarchy. First, a representation of the class inheritance hierarchy is built. The nodes in a class inheritance

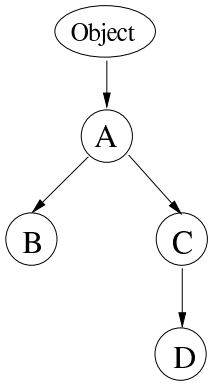


Figure 5.5: Class inheritance hierarchy.

hierarchy represent classes while edges denote the immediate superclass-subclass relationship; there is a directed edge A → B if B inherits from A. Assume we defined four classes A, B, C, and D, and the class inheritance hierarchy is as shown in Figure 5.5. Because every class in Java is a child of `Object`, the root is necessarily `Object`. From Figure 5.5, we can see class B and C are subclasses of A, and D is a subclass of C. The potential runtime types of a class or an interface are computed over this representation. For a method call on an object r of declared type t, the runtime type of the receiver r can be t or any subclass of t. For an interface type t, the runtime types of the receiver r can be any class c implementing t or a class implementing any subinterface of t, or any subclass of c.

- **Points-to analysis**

  In a C-like programming languages, many compiler analyses need to make accurate conclusions about the effects of writing to a variable and the possible read location too. Thus, identifying points-to relations is important to many compiler analyses. In the case of Java, points-to analysis has been extended to compute the set of objects a given class reference may assume at runtime. This identifies variable aliasing, and in an object-oriented language like Java, method targets too, by identifying potential types of the receiver object of a

method call. This can help significantly in reducing the size of the CHA graph.

Spark [LH03] is a flexible and modular framework providing points-to analysis for Java programs based on Soot. Spark offers precise points-to information, a more precise call graph, as well as good time/space performance. For our analysis we used SPARK rather than CHA to retrieve call graph information, in order to take advantage of the greater accuracy.

5.3 MHP Analysis in Soot

Figure 5.6 shows how our MHP analysis is integrated with Soot. Java class files are first input into the Soot framework, producing Jimple files. The Call Graph, as well as CHA and Spark analysis information are computed based on Jimple. Then the MHP analysis module computes the may happen in parallel information for each PEG node based on the Jimple files, Call Graph, CHA, and points-to analysis information from Spark.

The shaded area represents MHP analysis. Our MHP implementation is composed of three mainly phases. The first phase is a PEG Builder which uses Jimple and takes input from CallGraphs, CHA, and SPARK. We get a PEG after the PEG builder phase. Then a PEG Simplifier works on the PEG to get a smaller PEG by aggregating some nodes into one node. The PEG Simplifier is an important component of how we improve the performance of the MHP computation. Note that many of our simplifications are based on the observation (made in [NSA99]) that code not containing synchronization does not need to be explicitly modelled. The final phase of our MHP analysis is an MHP analyzer which runs the worklist algorithm based on the simplified PEG. The details of these processes are discussed in Chapter 6 and 7 .

MHP information can subsequently be used for further program analyses and optimization offered by Soot, and/or the analysis results provided by Soot can be exported through class file attributes for consumption by a virtual machine or another process.

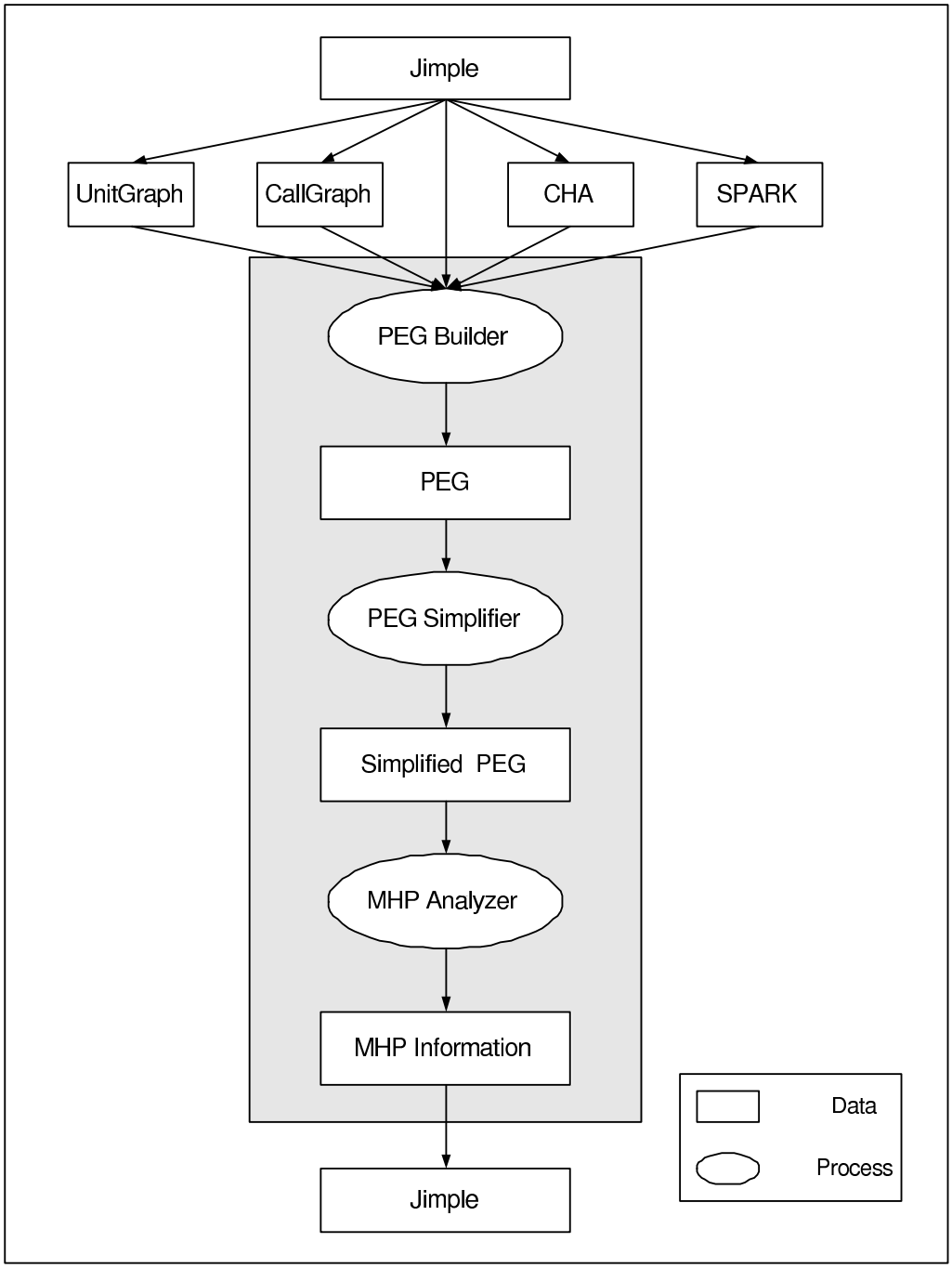


Figure 5.6: Overview of our MHP analysis.

Chapter 6
Design and Implementation of Parallel Execution Graph

In this chapter, we demonstrate our design and implementation for building Parallel Execution Graphs based on Soot. Conceptually, building PEGs from CFGs is straightforward. In practice, non-obvious information needs to be computed to make correct decisions and to ensure practicality, and this motivates a series of necessary optimizations and analyses.

In order to keep the data size manageable, a realistic implementation must also incorporate techniques to limit the size of the resulting data structures. An obvious way of restricting data size is to focus attention on application code only. This restricts the size of the call graphs and thus the PEG. Java includes a very large standard class library, and so even for a very small program a complete call graph tends to be quite large. However, in many cases the application itself is of main interest, and so if external actions are assumed safe enough, greater efficiency can be derived by excluding library and startup information. To facilitate the MHP analysis, we therefore define a `PegCallGraph` to be a call graph restricted to methods inside application classes, *i.e.*, user defined classes. Like a CallGraph of Soot, a PegCallGraph is a DirectedGraph, the edges of a PegCallGraph go from nodes to their successors.

Another difference in our PEG construction from the original MHP analysis is

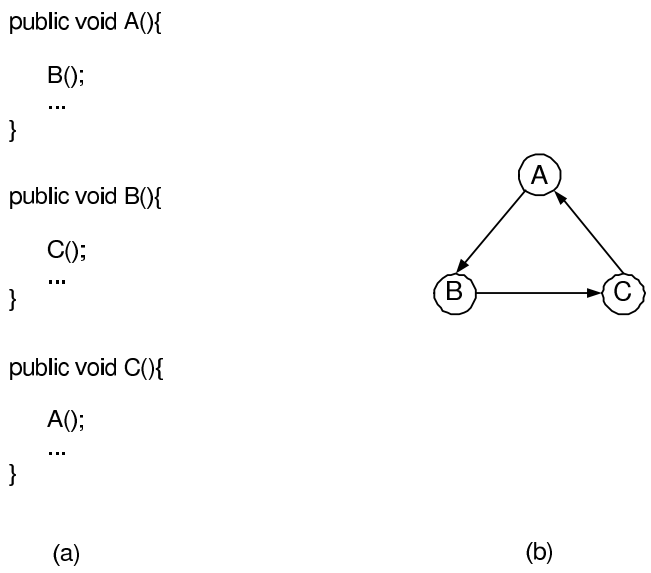


Figure 6.1: An example of recursive method invocations

related to inlining. The MHP analysis presented in Chapter 4 inlines every method, except *communication methods* into the control flow graphs for the threads. We do not use this strategy. First of all, this strategy has a drawback in that it easily fails when the program contain recursive method calls. In figure 6.1, (a) is a chunk of program code and (b) is the part of the call graph representing (a). Note in (b) methods A, B, and C form a cycle. Inlining every method as proposed in Chapter 2 requires inlining B into A, C into B, A into C, B into A, C into B, A into C, ..., . The inlining process is endless and the program will keep running until the memory run out. Unfortunately, although recursion is not present in every program, it is used in many non-trivial programs including, as we discuss in Chapter 8, one of the SPEC benchmarks.

The second reason that we do not inline every method lies in the lacks of necessity to always do so for MHP analysis. This requires a definition of what methods are *interesting* to MHP analysis. Here, by *interesting statements* we refer to statements related to modelling execution of threads and synchronization of Java programs:

Definition 6.0.1 *A statement is* interesting *if it is*

1) a monitorenter or monitorexit bytecode operations (including entry/exit of synchronized methods)
2) a call to wait, notify, notifyAll of an object, or start() and join() methods of a thread.

A method is interesting *if it either contains an interesting statement, or any callee is* interesting.

Usually, only a small part of a program consists of *interesting statements*. Whether we extend and inline those methods that do not contain an *interesting statements* does not effect the result of MHP analysis. In addition, the smaller the PEGs, the faster the MHP computation is expected to be.

Determining whether a method should be inlined thus requires identifying *interesting methods*. It is easy to identify *communication methods* and *synchronized methods* from the signature of the method, *i.e.*, the name, return type, declaration, and parameters of the method. The problem is to figure out which methods directly or indirectly through other method calls contain *interesting statements*. Our solution to this is described in Section 6.6.1

MHP analysis requires knowledge of precise runtime object identities. For example, we need to know which actual runtime thread is started in call statement `t.start()` if t is a (subclass of) `Thread`. The CFG representing the `run()` method for each runtime thread must be built for the MHP analysis. Similarly, if we do not know which thread corresponds to a method call to `join()`, we do not know which nodes should be "killed" in computing the `kill` set as introduced in Section 4.3.3. MHP analysis also relies on knowing the value of the *Object* field in PEG triples for determining lock ownership and monitor-based information flow. The MHP analysis introduced in Chapter 4 uses alias sets and cloning techniques to resolve object and method polymorphism, and to ensure runtime objects are identified. In this thesis, we try to find the runtime object when building PEG nodes to reduce the size of PEG. In cases when we cannot find the target statically; we use cloning techniques.

Because of the above differences from the original design of MHP illustrated in Chapter 4 and the practical usage concerns, we designed a different PEG format and

implementation steps for building the PEG. In the rest of this chapter, we present how we solved the problems for a practical MHP analysis based on Soot, as well as our PEG design.

6.1 PegCallGraph

A PegCallGraph is a special call graph that only contains methods in application classes. Java has a large class library. Even for a very small Java program, the JVM may load and initialize many classes for the running of the program. Thus we may get a large call graph even for a very tiny program; *e.g.*, consider the following HelloWorld Java program:

```
public class HelloWorld{
   public static void main{String[] args){
       System.out.println("Hello World!");
   }
}
```

The call graph of the "HelloWorld" program built by Soot contains 5075 different methods and most of them are methods of library classes. Methods inside application classes only count for a very small part of the call graph.

To build PegCallGraph, firstly, we use the Soot framework to generate a CallGraph. Then a filter will work to only keep the methods and associated edges in the application classes. And the native methods will be excluded from PegCallGraphs. Thus, in the cases that some library code calls back into the application code, or where the library code performs synchronization operations on objects created in the application code, the PegCallGraph should contain the methods in the called back application code (if Soot handles them).

The PegCallGraph implements the interface `DirectedGraph` provided by Soot. The summary of methods of the `DirectedGraph` is shown in Table 6.1, and represents basic graph query and traversal functionality. In addition to implementing all the methods in Table 6.1, the PegCallGraph provides another two public methods: `getTrimSuccsOf(Object o)` which returns a list of the **unique** successors of the

Methods	Return Type	Description
getHeads()	java.util.List	Returns the collection of the entry points for this graph.
getTails()	java.util.List	Returns the collection of the exit nodes of this graph.
getPredsOf(java.lang.Object o)	java.util.List	Returns the collection of the predecessors of the given node *o*.
getSuccsOf(java.lang.Object o)	java.util.List	Returns the collection of the successors of the given node *o*.
iterator()	java.util.Iterator	Returns an iterator of the nodes in this graph.
size()	int	Returns the number of the nodes in this graph.

Table 6.1: Summary of methods of interface DirectedGraph

given node *o* and `getClinitMethods()` which returns a list of `clinit` methods. When the initialization method of a class or interface is static and has no arguments, the Java compiler creates a special method with the name "<clinit>" in bytecode [LY99]. The `getClinitMethods()` returns the list containing all `clinit` methods.

A special method to return unique successors is necessary since Soot's call graph does not guarantee this property. In Figure 6.2, (a) is an example of a Java program in which method `foo()` contains two method invocations to method `bar()`, and (b) is the corresponding call graph. Notice in (b) there are two edges from method `foo` to `bar`. In (c) of Figure 6.2, we only keep one edge from `foo` to `bar`. If there is more than one edge from a method *a* to method *b*, the list returned by method `getTrimSuccsOf(Object o)` only contains one *b*. The applications of method `getTrimSuccsOf(Object o)` and `getClinitMethods()` will be introduced

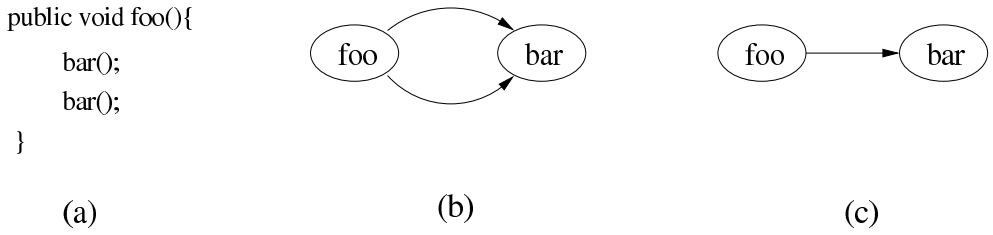


Figure 6.2: An example of a trimmed PegCallGraph

in Section 6.2.

6.2 Finding Runtime Objects

Nodes in PEGs have the format `(object, name, caller)` where `Object` is the object owning the method at runtime. In Soot and by using SPARK, it is possible to find the potential textual allocation sites corresponding to a given object reference. Allocation sites are locations in the code, and thus one can easily determine a set of potential types of an object reference, and this is sufficient for many analyses (including call graph refinement).

For MHP analysis, however, decisions as to whether synchronization has occurred requires knowing that an object involved in a *monitorexit* is the same *runtime* object involved in a previously examined *monitorenter*. In addition, SPARK computes **may-**alias information, and so even the same singleton allocation site sets for the respective objects are not sufficient for this conclusion, since one allocation site in a loop may spawn more than one runtime object. For example, in the method foo in Figure 6.3, the allocation site new A() will be executed 10 times, that means it corresponds to 10 objects with type A. A form of inter-procedural value numbering analysis is thus required. Again for simplicity of implementation and as well as asymptotic complexity concerns we have elected for a custom analysis, composed of an intra-procedural analysis and a flow-insensitive inter-procedural step.

Given a Jimple statement of method invocation, we can find the target allocation sites by the aid of SPARK. As discussed, even a singular allocation site for an object

```
private void foo(){
    int i = 0;
    for (i<0; i<10; i++){
        Object a = new A();
    }
}
```

Figure 6.3: An example of an allocation site corresponding to multi-objects

reference does not mean that the runtime object has been found. However, an obvious guarantee that two or more synchronization operations are operating on the same value can be provided if the computed sets of allocation sites are both the same singletons, and the allocation site is **only ever executed once**.

To figure out if an allocation site can be executed at most once, we need intra-procedural analysis and inter-procedural analysis. By the aid of the control-flow analysis framework provided by Soot, we can intra-procedurally find out which allocation sites may be executed more than once. If an allocation site is inside a method which may be called multiple times, this allocation site may be executed more than once, too; thus we also need the information of which methods may be executed more than once. This is more complicated and need both intra-procedural and inter-procedural analyses.

Inter-procedurally, we can find which methods may be invoked more than once using the PegCallGraph. If a method is inside an intra-procedural control flow cycle, it may be called many times. To find out if a particular method call will be executed more than once. We use the intra-flow analysis provided by Soot. We find which method calls are inside cycles, and then propagate this information throughout the PegCallGraph. The results of these two analyses are merged and propagated together to figure out if only one object is spawned at a specific allocation site. In summary, finding out all the allocation sites and which of them can be executed more than once needs the following:

- Finding out if a method may be called more than once
- Finding out if an statement is executed only once inside a method
- Finding out which allocation sites correspond to only one object

The implementation details are as follows.

6.2.1 Finding Out if a Statement Is Executed only Once Inside a Method

Intra-procedurally, a statement is surely executed at most once if it is not included in any control flow cycles. This information is computed for each allocation site of every method in the PegCallGraph. This is done by using the intra-procedural flow analysis framework provided by Soot. With the aid of control flow graph, Soot provides an intra-procedural flow analysis framework. The users can implement data flow and/or control flow analyses based on the framework. Usually, a `FlowSet` is used to store the variables or other information when the control flow graph is traversed. The users can have different ways to implement data flow analyses; but usually the users should define the flow equations, operations at merging points, and the initial value of the `FlowSet`. We compute this information for every allocation site of all the methods in the `PegCallGraph` to make sure that we compute this information for every user allocation site.

6.2.2 Finding Out If a Method May Be Called More Than Once

The steps used to find out if a method may be called more than once are as follows.

1. Finding out if a method call may be execute more than once intra-procedurally

   If a method invocation statement is inside a loop, this statement may be executed more than once, thus the body of the method may be executed more than once, too. Here we use the intra-procedural flow analysis framework of Soot to find the methods inside control flow cycles at the same time as we look for the allocation sites that may be executed once intra-procedurally.

2. Propagating the information computed in Step 1 throughout the PegCallGraph

   If we find a methods that may be called more than once intra-procedurally, all the methods that can be reached from this method may be called more than once. We propagate the information computed in Step 1 throughout the PegCallGraph using a simple depth-first algorithm.

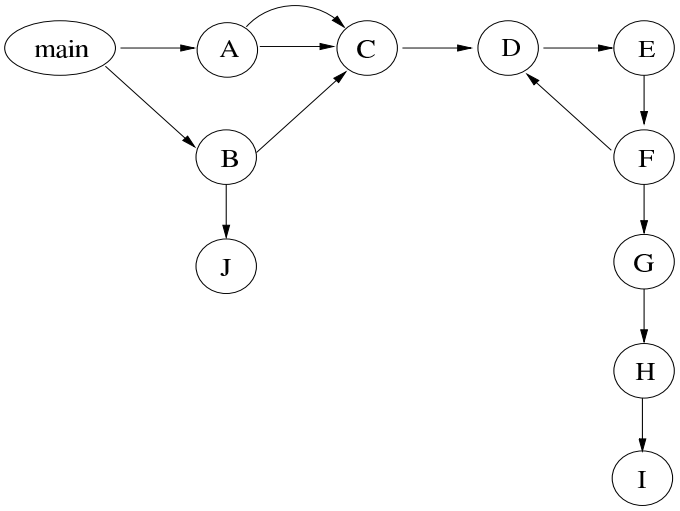
3. Finding out if a method is called more than once inter-procedurally

   To find out which methods are called more than once inter-procedurally, let us look at some examples first. In Figure 6.4, (a) is a tiny PegCallGraph where main, A, B, C, D, E, F, G, H, I and J represent methods. The nodes D, E, and F may be executed more than once because they form a cycle. The nodes G, H, and I may be executed more than once because any call to F may go to G, H, and I. There are two edges that go from A to C and one edge that goes from B to C, thus C may be called multiple times. Thus methods C, D, E, F, G, H, and I may be executed more than once.

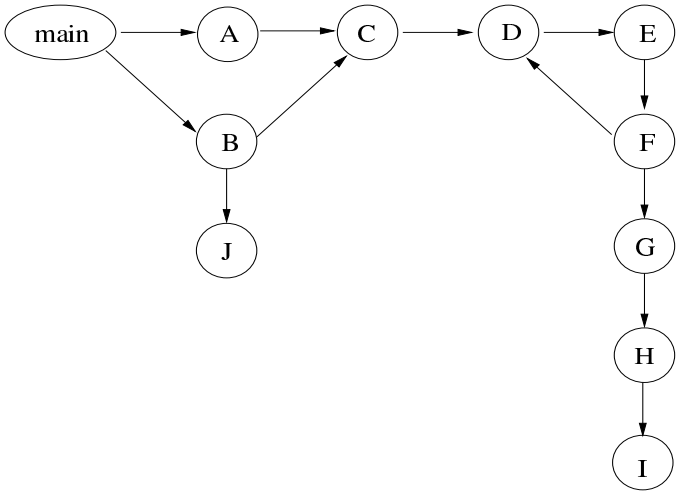
   Our approach to figure out which methods are called more than once inter-procedurally is made up of two stages. The first stage is a breadth-first search to find out which nodes in the PegCallGraph are visited more than once, and so the methods represented by these nodes may be called more than once.

   In the second stage, we use a modified depth-first search on the PegCallGraph to detect whether a node is potentially reachable more than once from `main`. Before the execution of this stage, the PegCallGraph is trimmed to be like (b) of Figure 6.4 *i.e.*, if there is more than one edge from a node a to b, we only keep one of them.

   Figure 6.5 shows the algorithm used to find which methods are called more than once inter-procedurally. In this algorithm, the methods that may be called more than once are stored in the collection multiCalledMethods. The **Search** procedure works as follows. Lines 1-2 paint all vertices white. Lines 3-5 check each head in PegCallGraph G and, when a white vertex is found, visit it using **Visit**. In each call **Visit(v)**, if v is gray, line 7 paints it black, and if collection



(a)



(b)

Figure 6.4: A PegCallGraph containing multi-called methods

```
        Search(G)
 1      for each vertex u of G
 2          mark u WHITE
 3      for each head v of G
 4          if v is WHITE
 5              visit(v)


        Visit(v)
 6      if v is GRAY
 7          mark v BLACK
 8          if multiCalledMethods does not contains v and v is not a clinit method
 9              add v to multiCalledMethods
10      else
11          mark v GRAY
12      for each successor s of v do
13          if s is not BLACK
14              visit(s)
```

Figure 6.5: The algorithm for computing multi-called methods

multiCalledMethods does not contain it and it is not a `clinit` method, v may be a method called more than once and is put into multiCalledMethods. The reason that we have a special handling for `clinit` methods here lies in that in Java bytecode, the `clinit` method may be called in different statement, however, the `clinit` method can be executed only once according to Java semantics. If v is not gray, it must be white, and is painted gray. Lines 12-14 examine each successor s of v, if the successor is not black, the traversal continues using **Visit(s)**.

6.2.3 Finding Allocation Sites and If the Allocation Sites May Represent More Than One Object

Methods that can be called more than once conservatively imply each statement in them can be executed more than once, regardless of internal control flow. Our algorithm actually computes both intra and inter-procedural information together, performing intra-procedural analysis as the inter-procedural analysis proceeds, and only if required. This allows the conclusions of each analysis to be merged and propagated together. Procedure 3 shows the process of finding all the allocation sites and which allocation sites may correspond to more than one object. The output of this algorithm is two collections: `allocNodeSet` containing all the allocation sites and `multiObjAllocNodes` containing the allocation sites corresponding to more than one object.

Before the execution of this algorithm, we compute which methods may be called more than once and store this information in the collection `multiCalledMethods`. This algorithm checks each method in the `PegCallGraph` to see if the `multiCalledMethods` contains it. If the current method may be called more than once, every allocation site in this method may correspond to more than one object. Line 3 checks each statement of the current method by scanning each node (represented as a `unit` of the `UnitGraph`). If the current unit is an allocation site it is added to the allocation site collection and to a collection of call sites that may be called multiple times as in lines 5 and 6. If the `multiCalledMethods` does not contain the current method, the

Procedure 3

```
1: for all method m in the PegCallGraph do
2:   if multiCalledMethods contains m then
3:     for all unit in the UnitGraph do
4:       if unit is an allocation site then
5:         add this allocation site to allocNodeSet
6:         add this allocation site to multiObjAllocNode
7:       end if
8:     end for
9:   else
10:    find all the multiAllocSites inside method m
11:    for all unit in the UnitGraph do
12:      if unit is an allocation site then
13:        add this allocation site to allocNodeSet
14:        if s contains unit then
15:          add this allocation site to multiObjAllocNode
16:        end if
17:      end if
18:    end for
19:  end if
20: end for
```

JVM Instruction	InvokeExpr in Soot
invokevirtual	VirtualInvokeExpr
invokespecial	SpecialInvokeExpr
invokeinterface	InterfaceInvokeExpr
invokestatic	StaticInvokeExpr

Table 6.2: JVM Instruction VS. InvokeExpr in Soot

procedure of intra-procedurally finding allocation sites corresponding to more than one object is called and this information is stored in the collection `multiAllocSites`. Every allocation site is put into the collection `allocNodeSet` as in line 13. If the current `unit` is an allocation site and found in the collection `multiAllocSites`, the corresponding AllocNode is put into `multiObjAllocNode` as in lines 14 and 15.

6.3 Finding Target Methods

Because inlining is used in computing MHP information, finding target methods is very important. Finding target methods is not as complicated as finding runtime objects because we only need to find the runtime **type**, not the specific object of references.

Java Virtual Machine provides various instructions for method invocations. As introduced in Section 5.2, there are four kinds of JVM instructions for method invocations, *i.e.*, invokevirtual, invokestatic, invokespecial, invokeinterface. Soot creates an expression implementing an interface `InvokeExpr` for each method invocation. Table 6.2 shows the JVM instructions used in invoking a method and their corresponding representations in Soot. In table 6.2 the first column specifies the instructions for method invocations and the second column shows the interfaces (which are subclass of InvokeExpr) used to represent the first column.

To find target runtime methods, we need to analyze the various **InvokeExpr** in the second column of Table 6.2. If a statement contains a `StaticInvokeExpr`, it is

easy to find the target method because it is in the class defining the called method. But in the rest of the cases, we need to make some effort to find the target method.

The `CallGraph` provided by Soot contains conservative information for method targets. With the aid of SPARK, we can get a refined `CallGraph` that has more precise method target information. It is important to note that the `CallGraph` includes all kinds of methods, including, for example, native methods. Our analysis does not handle native methods, and so these are excluded.

6.4 Runtime Objects and Alias Resolution for Threads

We are trying to statically find the runtime objects and target method in our MHP analysis; unfortunately, sometimes we do not know this information until runtime. Figure 6.6 is an example of part of a Java program. We do not know which thread is started in statement `thread1.start()` in method `bar` of class `Foo` until runtime. Similarly, at the statement `thread1.join()` we do not know which exact thread will die until runtime. With the aid of SPARK and CHA, however, we can find the possible threads that `thread1` points to, and use this to form a conservative solution. In the example program of Figure 6.6, both an instance of `Thread1` and an instance of `Thread2` may be started in statement `thread1.start()`. We create two CFGs representing the run() method of `Thread1` and `Thread2`. There are two *start edges* built for them. For the `join` statement, it does not kill any statement because we do not know which thread it should kill.

6.5 Parallel Execution Graph in the Context of Soot

Our PEG implements the interface `DirectedGraph` provided by Soot. The contents of the PEG are as follows:

```
public class Thread1 extends Thread{
        public void run(){
            ...
        }
}
public class Thread2 extends Thread{
        public void run(){
          ...
        }
}

public class Foo{
    int argument = 0;
    Thread thread1  = new Thread1();
    Thread thread2   = new Thread2();

    public int getArgument(){
            return argument;
    }

    public void setArgument(int arg){
            argument =   arg;
    }
    public void bar () {
            if  (argument > 0 )
                thread1 = thread2;
            thread1.start();
            thread1.join();
    }
}
public class Main{
     public static void main(String[] args){
        Foo foo = new foo();
}
```

Figure 6.6: An example of thread actions needing alias resolution

6.5.1 Nodes

Building nodes is essential for generating PEGs. We use a JPegStmt to represent the nodes in our PEGs. Because the nodes in PEGs are used to represent *interesting statements* specifically and other statements more generally in programs, they are designed to be subclasses of JPegStmt. JPegStmt has the following subclasses with the obvious functionality corresponding to the node name.

StartStmt	NotifyStmt
JoinStmt	NotifyAllStmt
WaitStmt	WaitingStmt
NotifiedEntryStmt	BeginStmt
MonitorEntryStmt	MonitorExitStmt
OtherStmt	

All the subclasses except `OtherStmt` are used to denote one kind of *interesting statement*, and `OtherStmt`s denote all the rest. The format of our PEG nodes is `(Object, Name, Caller, Unit, UnitGraph, SootMethod)` where `Object`, `name`, and `caller` are the same as in Chapter 4, `UnitGraph` is a reference to the UnitGraph (CFG) of the current method, `Unit` is a reference to the original Unit in the UnitGraph, and `SootMethod` is a reference of current method. For those statements that are not method calls, we label `object` with "*".

The process of building PEG nodes begins with calling Soot to build `UnitGraphs`. Every node or Unit of the `UnitGraph` represents a Jimple statement and is then transformed to a PEG node. To complete PEG node construction, we have specified how to find the runtime `object`. Note a statement containing an invocation of the `wait` method needs special handling; it is transformed into three PEG nodes, as per the original PEG definition, `WaitStmt`, `WaitingStmt`, and `NotifiedEntryStmt`.

6.5.2 Edges

There are four kinds of edges in PEGs. `Notify edges` are built dynamically during the data flow analysis. Here we introduce how to build the other three kinds of edges.

`UnitGraph` does not have an explicit `Edge` field, instead, it use maps to store the predecessor lists and successor lists for each node and method `getPredsOf()` and `getSuccsOf()` to locate these lists. Following the same design, we use various maps to store nodes and their predecessor and successor lists.

There are in fact several kinds of `UnitGraphs`, including `BriefUnitGraphs`, `CompleteUnitGraphs`. The `CompleteUnitGraph` contains all the statement and edges accounting for control flow between them while the `BriefUnitGraph` does not include the control flow edges associated with exceptions. Here we use the `CompleteUnitGraph` because we need to consider all the control flow edges.

- Local edges

  Local edges are the same edges as the edges in the CFG for each thread, and thus we can take advantage of the edges in UnitGraph. If there is an edge from node a to b in a UnitGraph and the PEG nodes m, n are generated from a, b respectively, there must be a `local edge` from m to n.

- Start edges

  If a Jimple statement contains a method call to `start()`, the target method is checked to see if the object containing it is a thread. If so, a *start edge* is built from the PEG node corresponding to this Jimple statement to the first node, *i.e.*, the `begin` node of the thread.

- Waiting edges

  A Jimple statement containing a method invocation to `wait` is transformed into three PEG nodes, `WaitStmt`, `WaitingStmt`, and `NotifiedEntryStmt`. To build the *waiting edge*, we put the `NotifiedEntryStmt` to the successor list of the `WaitingStmt` and the `WaitingStmt` to the predecessor list of the `NotifiedEntryStmt`.

6.6 PEG construction

Constructing the PEG can involve a lot of duplicated effort, as the same method is inlined in various places. Our strategy is to build small PEGs, one for each method a thread may invoke, and then combine these small PEGs into a PEG for the whole program. This of course doesn't change the final PEG size, and other techniques are necessary for that. Methods without *interesting statements* are good candidates for pruning, and so our PEG construction first proceeds with a simple, fast inter-procedural analysis to identify and compact such methods, followed by a standard inlining operation.

Roughly, building PEGs involves the following implementation.

- **Finding Methods that Need To Be Inlined**
- **Safety of Inlining**
- **Discovering Inlining Order**
- **Implementing Inlining**

6.6.1 Finding Methods that Need to Be Inlined

Clearly methods that will never execute any interesting statements are of little interest to the MHP analysis: any MHP information true on entry to such a method is true at exit, and at all points in between. Since thread communication code is typically a small part of any significant program, restricting the PEG to useful parts of the program is very effective.

Unfortunately, knowing whether a method is interesting is recursively dependent on the status of all callee methods. A precise, flow-sensitive inter-procedural analysis would be most effective, but is of course both complex and expensive. We have elected for a more pragmatic flow-insensitive approach, implemented in two stages.

The body of each method in the PegCallGraph is first scanned to see if contains an interesting statement. If so the method node in the PegCallGraph is marked

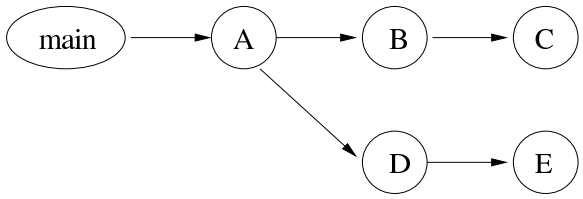


Figure 6.7: An example of PegCallGraph

interesting. Once all methods are examined, marks are propagated in the reverse direction of call graph edges, and logically OR'd at each merge point using a depth first search of the PegCallGraph. The result is a conservative overapproximation of interesting methods. During actual PEG construction uninteresting methods are represented by single node placeholders, greatly reducing PEG size.

Scan each method in the PegCallGraph to find out if it needs to be inlined

The body of each method in the PegCallGraph is examined to see if contains *interesting statements*. If so, it needs to be inlined. But, this is not enough to identify all methods that need inlining. In Figure 6.7, main, A, B, C, D, and E are methods in a PegCallGraph and main is the main entry point of a program. Assume A, C, and E are found to need inlining at this step. Because C is called by B and E is called by D, B and D need to be inlined, too. Thus, a second propagation step is also necessary.

Propagation inside Call Graph

At this stage, the information found in the last step is propagated inside the PegCallGraph. We use a depth-first search to scan the PegCallGraph: if any successor of a method needs to be inlined, this method must be inlined too. So after this step, we can determine that methods B and D in Figure 6.7 also need to be inlined.

6.6.2 Safety of Inlining

As mentioned, recursive method calls will result in the failure of inlining. Thus we should identify such methods. Here, we use the `PegCallGraph` to find recursive method calls. Recursive method calls form loops in a call graph, and so we simply need to find cycles inside the `PegCallGraph`.

One step to find all cycles in a graph is to find strongly connected components [CLR90]. Informally, a strongly connected component (SCC) of a graph is a maximal subgraph in which there is a path from one vertex to every other vertex along the edges of the graph. Since the `PegCallGraph` is a directed graph, we use the well-known depth-first search based algorithm in [CLR90] to find SCCs.

Our MHP analysis does not quit immediately after finding SCCs in the `PegCallGraph`. Recursive calls may be irrelevant to MHP analysis if they do not contain *interesting statements*. Thus, if a SCC does not contain *communication methods* or *synchronized methods*, it is ignored because it will not effect the inlining.

6.6.3 Discovering Inlining Order

The order in which methods are inlined is also important. We describe inlining order in terms of priorities; a method with a higher priority should be inlined earlier than a method with a lower priority. In a PegCallGraph, along a directed edge, the method at the head has higher priority than the method at the tail. For example, assume all the methods except `main` should be inlined in Figure 6.7. The leaves, methods C, and E have highest priorities and should be inlined first. B and D have higher priorities than A. A has the lowest priority for inlining in this PegCallGraph.

We use a list, specifically, with the format {*inlinee, place, inliner*} where `inlinee` is the reference to the PEG will be inlined, `place` is the method invocation statement, and `inliner` is the container PEG to store the inlining information. This information is then sorted and processed according to the inlining order.

6.6.4 Special Handling for Synchronized Methods

Java provides two high-level constructs for locking: *synchronized methods* and *synchronized blocks*. In the low-level implementation, *monitorenter* and *monitorexit* bytecodes are used by the Java Virtual Machine for entering and exiting a *synchronized block*. However, the JVM does not explicitly add *monitorenter* and *monitorexit* instructions for *synchronized methods*; instead, the synchronization of *synchronized methods* is executed during runtime by checking a special flag `ACC_SYNCHRONIZED` associated with the method definition. If it is set, the current thread will request a lock on the invoking object before executing the method, and then release the lock after the method execution is done. Here, we use a simplification for handling for this by manually adding explicit *monitorenter* and *monitorexit* instructions at the beginning and end of each *synchronized method*.

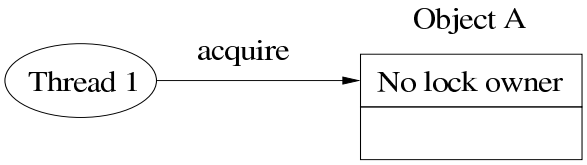
Chapter 7
MHP Implementation and Optimization

The basic MHP algorithm is described in Chapter 4. In this chapter we describe various implementation and optimization issues that must be addressed in a practical setting.

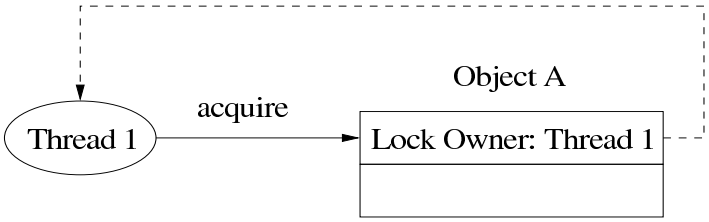
7.1 Finding Monitor Objects

If a Java program uses locks, we need the information of which PEG nodes are protected in monitors to implement the MHP algorithm. In Java, there is a lock associated with every object. Usually, Java programs have three different type of locks, *single entrant locks*, *reentrant locks*, and *enclosed locks*. The *single entrant locks* are the simplest among these three; in this situation a thread tries to acquire a lock which is not owned by any thread. The Java runtime system also allows a thread to try to acquire multiple locks. If a thread tries to acquire a lock held by itself, it is called a *reentrant lock*. If a thread tries to acquire another lock l while it has held a lock m, lock l is called *enclosed lock* [ACSE99].

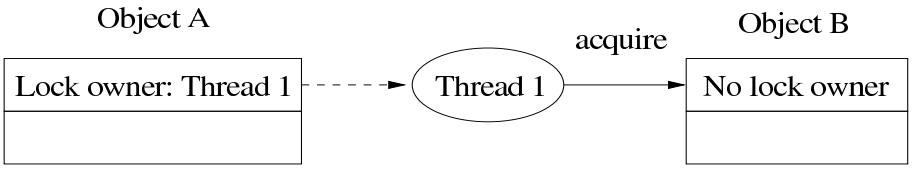
To model these three types of locks, we design a class `MonitorDepth`. The class `MonitorDepth` has two member variables, `String objName` and `int depth`. Lock actions are always based on a specific object. Member variable `objName` represents the object which is being locked. The `depth` field represents the level of the recursive locking and can be 1, 2, or more. For each lock object, an instance of `MonitorDepth`



(a) Single Entrant Lock



(b) Re–entrant Lock



(c) Enclosed Lock

Figure 7.1: Three different types of locks in Java

is created and a standard flow-sensitive analysis is used to propagate this information through the PEG, incrementing the count for the object specified at each *monitorenter* operation and decrementing counts at *monitorexit*'s. Unbounded recursive locking, as well as general merge points with unmatched locking depths for corresponding objects (not possible with Java programs) and are not handled, so this is guaranteed to reach a fixed point.

With lock depth information the MHP analysis can make sound judgements as to whether a PEG node is truly in a monitor or not.

7.2 Implementation of the Worklist Flow Analysis Algorithm

The main difference of our implementation from the main loop of the algorithm presented in Section 4.3.3 is that we need to handle a simplified PEG, where a single PEG node may correspond to more than one CFG node. Thus when mapping information back to the CFG we must consider the possibility that a PEG node represents a list of CFG nodes.

A further difference is due to our conservative treatment of object identity. When we compute `KILL` sets for `JoinStmts`, if the number of target threads is more than one, the `KILL` set of the node is empty.

7.3 Optimizations

We can proceed to use the MHP algorithms to compute MHP information once the PEG is built. However, even with the inlining strategy introduced in Chapter 6, we may still have a large PEG. Further optimization techniques can still be useful to simplify the PEG before running the MHP algorithms. Since the MHP analysis is a fixed-point flow analysis manipulating sets of PEG nodes, the size of the PEG may effect the execution time and space consumption. We hope that reducing the size of the PEG could make the MHP analysis cost less time and save some space.

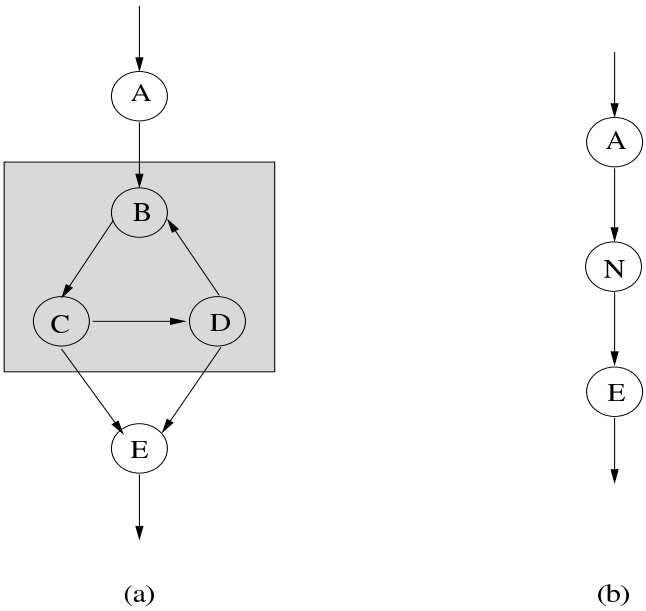


Figure 7.2: An example of Strongly Connected Component

Our approach of inlining was based on the observation that the worklist algorithm is affected only by *interesting statements*, and usually most statements of a Java program are not related to *interesting statements*. This gives us another opportunity for simplifying PEGs to get smaller graphs. We applied two straightforward graph reductions as optimizations: merging lists, and collapsing strongly connected components. Data on the effects of these optimization are given in Chapter 8.

7.3.1 Merging Strongly Connected Components

A *strongly connected component* (SCC) of a graph g is a subgraph sg of g in which each node is reachable from any other node of sg along edges in sg. Compacting a strongly connected component into one node is based on an observation: suppose a strongly connected component S inside a PEG does not contain *interesting statements*. If a statement A can be concurrently executed with a statement B inside S, it should also be possible for A to be concurrently executed with all the other nodes inside S. For example, in (a) of Figure 7.2, suppose A, B, C, D, and E are statements not

containing *interesting statements* in a program. Notice the nodes in shaded area, *i.e.*, B, C, and D form an SCC, thus the MHP information of node B, C, and D may be propagated to any other node of the SCC. The MHP information of node B, C, and D are the same because they do not containing *interesting statements* and nothing will be added or deleted from their $M()$ set after the propagation inside the SCC is done. Since the $M()$ sets of every node of an SCC are necessarily the same, we can merge the nodes inside this SCC and create a single, new node to represent the entire SCC.

We use the well-known, depth-first-search based algorithm specified in [CLRS01] to find SCCs. After finding the SCCs, we check if they contains *interesting statements*. If not, we create a `List` which contains all the nodes in the SCC, then create a new node in the position of the SCC. This new node is a reference to the list containing the nodes of the SCC. For example, in (a) of Figure 7.2, the SCC containing nodes B, C, and D is transformed to a new node N.

7.3.2 Merging Sequential Nodes

A sequence of nodes with no *interesting statements*, and no branching in or out except at the beginning and end respectively necessarily has the same MHP information at each node in the sequence; whatever is true upon entry is true at exit and at all points between.

Tarjan proposed an efficient FIND-UNION algorithm [Tar75] to collapse nodes. This algorithm merges two successive nodes and uses one of them to represent the pair. Whenever a set of nodes are combined, one of the combined nodes is used as the unique representative of the whole set.

Here we use a different approach to collapse nodes. First we locate all maximal chains of sequential nodes. Then the maximal chains are merged and replaced by a new node.

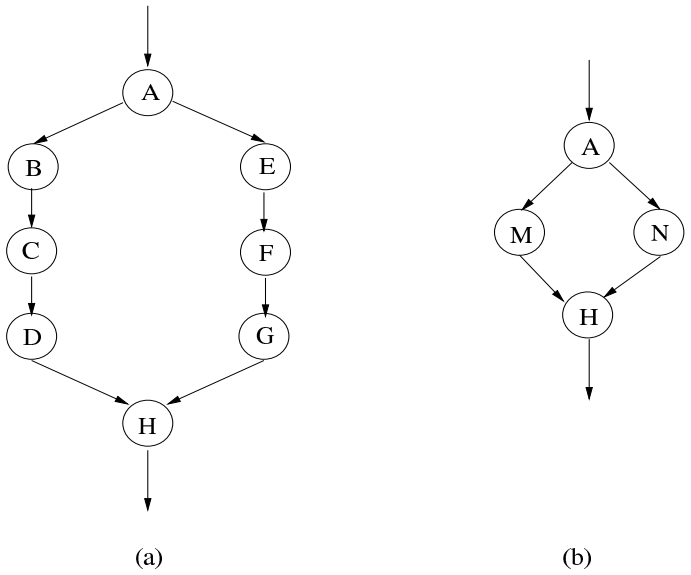


Figure 7.3: An example of sequential nodes

Locate the maximal number of sequential nodes that can be merged

Using an arbitrary order to visit nodes will not work well here because we need to find all the sequential nodes. A linear order according to a depth-first search based topological sort [CLRS01] is built for accessing nodes. In a directed graph, an edge from a descendant to an ancestor in a depth-first tree is call a *back edge* [CLRS01]. Back edges form cycles in a directed graph. Note that if a PEG is cyclic, no linear order is possible, and in such cases we ignore the back edge when building the topological order.

As we visit each node according to the topological order, each node is examined to check if it has only one predecessor, only one successor, and does not contain *interesting statements*. If so, it is added to a list and we proceed to visit the next node. If not, we terminate the current list as a maximal chain of sequential nodes, create a new list, and proceed to visit the next node. This process continues until all the nodes have been visited.

As an example, consider the graph in Figure 7.3. On the left, suppose B, C, D,

E, F, G do not contain *interesting statements*. Our topological sort will produce an ordering ABCDEFGH (or AEFGBCDH), and so we will find a maximal chains of sequential nodes, ⟨B, C, D⟩ and ⟨E, F, G⟩.

Merging the sequential nodes

After locating maximal chains, each is collapsed and replaced by a new node that represents the entire sequence. In (a) of Figure 7.3, chain B, C, and D is replaced by the newly created node M while chain E, F, and G is replaced by N as shown in (b).

7.3.3 Updating the PEG

Whenever nodes are merged, the PEG should be updated. The nodes that are merged are removed from the PEG, and the new nodes representing merged nodes are added to the PEG. Of course it is also necessary to preserve control flow in this process, and so all the edges to and from the merged SCCs or sequential nodes are replaced with the edges to and from the new nodes representing the merged nodes. In (b) of Figure 7.2, for instance, the nodes and edges inside the SCC are gone, and the newly created node N, is added to the PEG and it has the original predecessors and successors of the SCC (B, C, and D). In (b) of Figure 7.3, the nodes B, C, D, E, F, G and the edges connecting them are removed while the new nodes M and N are inserted and have the original predecessors and successors of the chain B, C, and D, and the chain E, F, and G respectively.

Chapter 8
Experimental Results

Here we describe experimental results for our implementation, comparing performance with and without our various optimizations and improvements.

8.1 Benchmarks

We collected our benchmarks from several sources. Most of the benchmarks are multi-threaded benchmarks from the Java Grande Benchmark Suite [Sui]: ForkJoin, Sync and Barrier represent low level benchmarks that test synchronization, Series, LuFact, Sor, Crypt and SparseMat test specific "kernel" operations, and MonteCarlo, RayTracer and MolDyn are larger, more complete applications. From the SPECJvm98 suite we included mtrt, the only multi-threaded benchmark in that benchmark set. In order to fit our input requirements, we modified most of these benchmarks by manually unrolling all the loops containing method calls to *communication methods*. All tests were run on a Pentium 4 1.8GHz, with the Sun HotSpot VM (Linux, version 1.4.1) using a 1500M heap.

For comparative purposes we have also attempted to collect some of the same benchmarks used in Naumovich *et al.*'s paper. However, most of the code we have been able to acquire is in the form of incomplete program fragments that require a driving main program to analyze in our system. Fine-grained comparisons are thus not likely to be meaningful. We therefore include AuBanking and PeBanking, programs

Programs	Threads	Nodes	Edges	$\|M()\|$			Pairs	LOC
				Ave	Min	Max		
ForkJoin	4	308	331	64	15	173	6105	544
Sync	5	656	712	118	28	459	28944	642
Barrier	5	561	716	175	53	339	34651	754
Crypt	5	1025	1061	672	193	772	297220	1107
MonteCarlo	3	405	433	104	35	182	11340	3569
RayTracer	3	660	724	125	43	318	25188	2252
Series	3	315	342	109	46	130	9660	826
LuFact	3	465	510	202	112	224	32032	1480
Sor	3	622	673	289	182	363	66430	730
SparseMat	3	305	329	81	30	120	6180	726
MolDyn	3	2173	2295	1093	917	1866	1088392	1346
Cyclic	5	162	201	69	30	124	4580	81
mtrt	4	188	211	43	14	108	2819	3812
AuBanking	3	170	203	31	18	92	4114	301
PeBanking	3	154	270	63	47	137	4414	442

Table 8.1: Experimental results without PEG simplification

based on the examples AutomatedBanking and PessimBankAccount from Doug Lea's book [Lea97]. We have focussed on these two examples since in [NSA99] Naumovich *et al.*'s version of these benchmarks had the largest PEG sizes and also had the largest MHP analysis times (by an order of magnitude) of all their benchmarks. We also include Cyclic, a benchmark from the CyclicBarrier example in the second edition of Lea's book [Lea99] which was also analyzed by Naumovich *et al.*. In each case we added an appropriate main method, modifying them to be complete applications.

8.2 Results

Tables 8.1 and 8.2 presents the experimental results for the benchmarks without most of our optimizations in effect. In Table 8.1 the first column gives the names of the benchmarks, the second column gives the number of threads (including the main thread), and the next two columns give the number of nodes and edges in the PEGs representing each program respectively.

In the fifth, sixth, and seventh columns, we specify the average, minimal, and maximal number of nodes in the computed $M()$ set for each node, *i.e.*, how many nodes were determined may be executed in parallel with each node. This gives some notion of analysis accuracy, at least in the absence of measuring a consuming analysis. The eighth column gives the total number of node pairs found in the entire PEG—as well as the PEG itself, this represents the total space requirements of the analysis. The last column gives the size of the benchmarks using the number of lines of code.

Tables 8.1 shows that larger programs does not always have bigger PEG. For instance, MolDyn (1346 lines of code) is smaller than mtrt (3812 lines of code), but has a bigger PEG; specifically, MolDyn 2173 nodes and 2295 edges while mtrt has 188 nodes and 211 edges. The reason lies in that we only consider the *interesting statements* and *interesting methods*. Graph size for smaller programs look reasonable, though the larger MolDyn benchmark suggests we may encounter scaling issues.

Table 8.2 measures time for the various stages of the analysis. PEG time is the time to build the PEG, MHP is the subsequent analysis time, and SPARK time is the total cost of points-to analysis. Total time is greater than the sum of the these stages; the remainder represents time required to load and initialize and shutdown the Soot environment.

The data and timing in Tables 8.1 and 8.2 already represent application of many of the previously discussed simplification and implementation techniques (excessive data sizes prevented computation of totally unoptimized data), we only exclude the PEG node merging techniques of Chapter 7. Note that mtrt contains recursive method calls and method inlining for it would normally fail; however, using the techniques of Section 6.6 we determined that the recursive calls do not involve *interesting*

Programs	PEG(s)	MHP(s)	SPARK(s)	Total(s)
ForkJoin	0.18	4.46	67.2	88.5
Sync	0.40	51.51	68.2	136.8
Barrier	0.34	72.72	68.7	160.4
Crypt	0.52	6812.68	67.2	6917.74
MonteCarlo	0.28	14.15	68.0	102.3
RayTracer	0.37	57.58	67.5	143.42
Series	0.24	8.84	67.8	93.3
LuFact	0.23	87.86	68.8	163.08
Sor	0.29	259.26	68.0	347.9
SparseMat	0.21	3.98	67.2	88.1
MolDyn	1.86	44313.44	69.2	44553.9
Cyclic	0.14	1.13	67.8	86.2
mtrt	0.33	1.53	139.7	232.8
AuBanking	0.17	1.14	66.5	86.4
PeBanking	0.14	1.17	66.4	85.3

Table 8.2: Experimental results without PEG simplification

statements, and so we are still able to get results.

For most benchmarks the time to build the PEG is small, and in all but one case well under a second. MHP analysis time clearly dominates PEG construction time. This is unsurprising given the $O(n^3)$ time complexity of MHP analysis, but was considerably less evident in the data presented in [NSA99], where the majority of benchmarks were very small (mostly < 100 PEG nodes) and so PEG time generally appeared to dominate. For larger programs the cubic behavior of MHP becomes more evident: MolDyn, the largest benchmark in terms of PEG nodes we examined takes less than 2 seconds to build the PEG, but over 12 hours to analyze. MolDyn is not an especially large program (1346 lines of code) and so these running times are clearly still excessive for even moderate programs, and further steps are necessary to reduce PEG size, and thus MHP analysis time.

Table 8.3 shows similar experimental results when the PEG is optimized using the techniques of Chapter 7. The second and third columns give the PEG size reductions supplied by the two techniques of merging SCCs and merging sequential nodes respectively; the resulting graph size is given in the fourth and fifth columns. The remaining columns present the relative size of the PEGs; specifically, the sixth column shows the percent of (the number of nodes in optimized PEG)/(the number of nodes in original PEG), and the last column shows the percent of (the number of edges in optimized PEG)/(the number of edges in original PEG). In every case our PEG optimizations were able to reduce the graph, and in some cases quite dramatically: MolDyn is reduced from 2173 nodes to 144. In smaller programs sequential node contractions are most effective, but in the bigger programs the volume of modular, synchronization independent sections of code sometimes made SCC merging quite valuable.

Table 8.4 gives the timing data when the optimization techniques presented in Chapter 7 are used. The second and third columns give the time in seconds taken to perform the PEG simplifications and run MHP analysis on the smaller PEG. The fourth and fifth columns show the total running time including SPARK and Soot overhead. The remaining columns give the relative speedup (old-time/new-time) ratio achieved by the optimized version versus the base approach, for both total

Programs	Sim.Scc	Sim.Seq.	Nodes	Edges	Relative size of Nodes(%)	Relative size of Edges(%)
ForkJoin	0	199	109	132	35.4	39.9
Sync	2	389	255	307	38.9	43.1
Barrier	12	287	262	411	46.7	57.4
Crypt	662	240	121	149	11.8	14.1
MonteCarlo	26	247	132	158	32.6	36.5
RayTracer	18	431	211	267	32.0	36.9
Series	26	180	109	134	34.6	39.2
LuFact	166	194	105	130	22.6	25.5
Sor	298	223	101	124	16.2	18.4
SparseMat	55	165	85	104	27.9	31.6
MolDyn	1482	547	144	174	6.6	7.6
Cyclic	0	51	111	150	68.6	74.6
mtrt	3	107	78	95	41.5	45.0
AuBanking	2	71	97	126	57.1	62.1
PeBanking	0	66	88	204	57.1	75.6

Table 8.3: Experimental results after optimization

running time, and the time just to construct and simplify the PEG and run the MHP analysis. Speedups in MHP+PEG construction range from 40% to over 13,000%. Again, MolDyn speedups were most significant, as running time drops from half a day to just over 1 second. As a general rule, larger benchmarks have more nodes, and hence more opportunities for PEG compaction, which is quite encouraging for analysis of reasonable size programs. The benchmarks with minimal or no total speedup, Cyclic, mtrt, AuBanking, and PeBanking all spend minimal time in MHP analysis—even our base MHP and PEG times account for less than 2 seconds, no more than 5% of total time.

Programs	Sim.(s)	MHP(s)	Total time(s)	Total Speedup	PEG+MHP
ForkJoin	0.02	0.41	84.4	1.05	4.76
Sync	0.07	8.81	94.1	1.45	5.95
Barrier	0.06	21.21	108.8	1.47	3.71
Crypt	0.10	0.93	105.1	65.82	4395.80
MonteCarlo	0.03	0.53	88.7	1.15	17.13
RayTracer	0.07	6.66	92.5	1.55	8.48
Series	0.03	0.64	85.0	1.09	9.98
LuFact	0.04	0.53	87.9	1.91	110.06
Sor	0.04	0.39	89.0	3.91	360.37
SparseMat	0.02	0.09	84.5	1.04	12.65
MolDyn	0.18	1.18	90.0	495.04	13763.80
Cyclic	0.02	0.74	85.8	1.00	1.40
mtrt	0.02	0.10	231.8	1.00	3.73
AuBanking	0.02	0.53	85.8	1.01	1.75
PeBanking	0.02	0.62	84.7	1.01	1.68

Table 8.4: Experimental results after optimization

SCC and sequential merging have clear benefits, with a fairly minimal cost—even for MolDyn simplification takes less than 1/5s. Merging in combination with an already efficient initial PEG construction allows reasonable size programs to be analyzed. Interestingly, after optimization efforts, the Barrier benchmark is the most expensive to analyze. With optimization overall analysis cost is related more closely to number and density of communication operations than input program size.

Chapter 9
Conclusions and Future work

In this thesis, we presented a practical MHP analysis for concurrent Java programs. There are of course a number of extensions and improvements still required to achieve an industrial strength solution.

9.1 Conclusion

We have presented a more realistic implementation of MHP analysis for Java. Focusing on the practical concerns, we designed and implemented a refined approach to build PEG for MHP analysis; our design makes use of a variety of existing and small custom analyses in order to build a feasible implementation that can analyze programs of a reasonable size, bypassing many of the previous input restrictions.

Moreover, we provided both design and implementation data for optimizations intended to improve the performance of our MHP analysis. Our base PEG construction already excludes large amounts of code by considering only code that may be relevant to MHP data, the *interesting statements*. We further compact PEGs by collapsing subsequences of uninteresting code forming *strongly connected components* or sequential chains obtaining smaller PEGs. Because the MHP data-flow algorithms run faster on the simplified PEGs than the original PEGs, the performance of our MHP computation is greatly improved.

We have presented experimental results from such an implementation, and have thus shown how excessive MHP analysis time can be efficiently handled through simple input compaction techniques. Our optimizations work for all the benchmarks, in some cases achieving speedups in MHP analysis time of several orders of magnitude. We also include techniques that allow us to handle benchmarks excluded from the original MHP presentation, *e.g.*, benchmarks that contain (uninteresting) recursive method calls.

9.2 Future work

Our work has clear extensions in a number of ways, including analysis and potential implementation improvements. Certainly accuracy of the resulting information deserves examination. Naumovich *et al.* compare MHP information to precise reachability analyses in order to verify the resulting analysis data, but the complexity of reachability analysis means such a technique is not feasible for larger programs. We have exhaustively examined (small) test cases, and spot-checked larger results to ensure we have a correct implementation, but a more thorough and deterministic approach is desirable.

Given the success of our simple techniques, further PEG compaction or reduction approaches seem worth exploring. For example, by considering the flow of MHP information through other identifiable PEG substructures, such as "hyperblocks" of nodes—collections of connected nodes with only one entry point, though possibly more than one exit. Movement to a PEG design that does not require inlining at all is of course most desirable.

We also aim to expand the range of acceptable input programs. Programs with an unbounded number of threads, use of timed synchronization constructs, and so on could be handled, and this would allow more programs to be analyzed with less manual intervention.

Some client analyses for the MHP computation would be interesting, and could also serve as indicates of the quality of MHP information. Because the MHP analysis

provides information of which statements may be executed in parallel with a given statement, by checking the MHP information for a statement involving a variable access, we can find information on which variable accesses may happen in parallel. Thus static datarace detection becomes straightforward and would also serve as an indicator of the quality or accuracy of MHP information. Of course, our effort in this thesis is directed at achieving a reasonably efficient MHP implementation. Client analyses, however useful are left as future work.

Bibliography

[ACSE99] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessasry synchronization from Java programs. In *Proceedings of the Sixth International Static Analyses Symposium*, pages 19–38. Springer-Verlag, September 1999.

[Ben] SPEC JVM98 Benchmarks. http://wwww.spec.org/jvm98.

[Bla99] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the ACM SIGPLAN 1999 Conference on Object-Oriented Programming, Systems, Languages, and Application*, pages 20–34, November 1999.

[BLQ+03] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114. ACM Press, 2003.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[BU99] J. Bogda and U.Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the ACM SIGPLAN 1999 Conference on Object-Oriented Programming, Systems, Languages, and Application*, pages 35–46, November 1999.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Language and Systems (TOPLAS)*, volume 13, pages 451–490. ACM Press, October 1991.

[CGS+99] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the ACM SIGPLAN 1999 Conference on Object-Oriented Programming, Systems, Languages, and Application*, pages 1–19, November 1999.

[Che00] Zhiqun Chen. *Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 2000.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sirdharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, Berlin, Germany, June 2002.

[CLR90] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 23.5. MIT Press, Cambridge, Massachusetts, 1990.

[CLRS01] Thoman H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN 2003*

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03), pages 149–168. ACM Press, 2003.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecure Notes in Computer Science*, pages 77–101, Åarhus, Denmark, 7-11 August 1995. Springer.

[DJ88] D.Callahan and J.Subhlok. Static analysis of low-level synchronization. In *Proceedings of the SIGPLAN/SIGOPS workshop on Parallel and Distributed debugging*, pages 100–111, 1988.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[EM98] E.Duesterwald and M.L.Soffa. Fast interprocedural class analysis. In *Proceedings of the ACM SIGSOFT Fourth Workshop on Software Testing, Analysis, and Verification*, pages 36–48, Victoria, B.C., October 1998.

[E.R00] E.Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming language design and implementation*, pages 208–218, June 2000.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232. ACM Press, 2000.

[FF01] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 90–96. ACM Press, 2001.

[FGS97] Jeanne Ferrante, Dirk Grunwald, and Harini Srinivasan. Compile-time analysis and optimization of explicitly parallel programs. In *Journal of Parallel algorithms and applications*, volume 12, pages 21–56, 1997.

[FJW87] Jeanne Ferrante, Karl J.Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. In *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[Han99] Per Brinch Hansen. Java's insecure parallelism. In *ACM SIGPLAN Notices*, volume 34, pages 38–45. ACM Press, April 1999.

[HC02] Cay S. Horstmann and Gary Cornell. *Core Java 2*, volume 2. Sun Microsystems Press, 2002.

[Lea] Doug Lea. *Concurrent Programming in Java. Design principles and patterns, online supplement*. http://gee.cs.oswego.edu/dl/cpj/index.html.

[Lea97] Doug Lea. *Concurrent Programming in Java Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, 1997.

[Lea99] Doug Lea. *Concurrent Programming in Java Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, second edition, 1999.

[Lee99] Jaejin Lee. *Compilation techniques for explicitly parallel programs*. PhD thesis, University of Illinois at Unbana-Champaign, 1999.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[LPA97a] Jaejin Lee, Samuel P.Midkiff, and David A.Padua. Concurrent static single assignment form and constant propagation for explicitly parallel

program. In *Proceedings of The 10th International Workshop on Languages and Compilers for Parallel Computing, number 1366 in Lecture Notes in Computer Science*, pages 114–130, Springer, August, 1997.

[LPA97b] Jaejin Lee, Samuel P.Midkiff, and David A.Padua. How to make a correct multiprocess program execute correctly on a multiprocessor. In *IEEE Trans. on Computers*, pages 46(7):779–782, July 1997.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[Mie03] Jerome Miecznikowski. New algorithms for a Java decompiler and their implementation in Soot. Master's thesis, McGill University, Montreal, Canada, 2003.

[MLP01] S. Midkiff, J. Lee, and D. Padua. A compiler for multiple memory models. In *9th Workshop Compilers for Parallel Computers (CPC'01)*, Edinburgh, Scotland, UK, June 2001.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly, 1996.

[NSA99] Gleb Naumovich, George S.Avrumin, and Lori A.Clarke. An efficient algorithm for computing MHP information for concurrent Java program. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 24, pages 338–354, Toulous, France, November 1999.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on*

Principles and practice of parallel programming, pages 167–178. ACM Press, 2003.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 43–55. ACM Press, 2001.

[Sar97] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing, LNCS Springer-Verlag*, pages 94–113, Minneapolis, MN, August 1997.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallee-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the conference on Object-Oriented Programming, System, Languages, and Applications*, pages 264–280, 2000.

[SS98] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *Proceedings of ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Montreal, Quebec, Canada, 1998.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM Press, 1996.

[Sui] Java Grande Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[vG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82. ACM Press, 2001.

[vPR03] Christoph von Praun and Thomas R.Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming language design and implementation*, pages 115–128, San Diego, California, USA, June 2003.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[VSD86] Alfred V.aho, Ravi Sethi, and Jeffrey D.Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[WR99] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Object-Oriented Programming, Systems, Languages, and Application*, pages 187–206, November 1999.