THREAD ESCAPE ANALYSIS FOR A MEMORY CONSISTENCY-AWARE COMPILER

BY

CHI-LEUNG WONG

BEng., The Hong Kong University of Science and Technology, 1995

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

The widespread popularity of languages allowing explicitly parallel, multi-threaded programming, e.g. Java and C#, have focused attention on the issue of *memory model* design. The Pensieve Project is building a compiler that will enable both language designers to prototype different memory models, and optimizing compilers to adapt to different memory models. Among the key analyses required to implement this system are *thread escape analysis*, i.e. detecting when a referenced object is accessible by more than one thread, *synchronization* analysis, and *delay set* analysis. This thesis describes the overall Pensieve compiler and presents in detail its thread escape analysis as well as experimental results showing the effectiveness of the compiler when the target code is following the sequentially consistent memory model. On both single-threaded and multi-threaded programs the performance is up to 100% of the performance of the same programs executing under a relaxed memory model.

This work is dedicated to my parents, my wife and my daughter

Acknowledgments

First of all, I would like to give thanks to God for His guidance throughout my graduate career — from Hong Kong to Santa Barbara and from Santa Barbara to Champaign. Throughout these transitions, I have met many good friends and learned a lot from different people, which has enriched my life. I also know my strengths and my weaknesses better through this experience. My graduate studies would not have been completed without His guidance.

All the members of the Pensieve project share credit for the completion of this work: Professor David Padua, Professor Samuel Midkiff, Dr. Jaejin Lee, Dr. Zehra Sura, and Xing Fang.

My advisor, Professor David Padua, provided me with the opportunity to participate in different interesting projects. In the Pensieve project, when my exploration of a fast escape analysis went in the wrong direction, he pointed it out and helped lead my research back in the right direction. Moreover, the quality of this dissertation has been improved tremendously because of his efforts with revisions, even on the weekends.

Many discussions with Professor Samuel Midkiff provided me with insights about the work presented in this dissertation. He also contributed to revisions of this dissertation. I would also like to thank him for being my mentor when I worked as an intern at IBM T. J. Watson Research Center, where I gained my work experience in the industry.

Dr. Zehra Sura plays an important role in the design and implementation of the Pensieve system. I am grateful for her patience in listening to my ideas, even when they were premature. I found the discussions interesting and simulating, which helped a lot in the design of the escape analysis algorithm.

I also want to thank Xing Fang for the implementation of the fence insertion algorithms which are used extensively in this dissertation. His effort in coordinating the use of Purdue machines is highly appreicated.

I would like to thank the other members on my committee, Professor Marc Snir, Professor Sarita Adve, Professor Josep Torrellas, and Professor Martin Rinard. Their comments and suggestions played an important role in the improvement of the work presented in this dissertation.

It has been a memorable experience working with members of the Polaris research group. Many of the former members (George Almasi, Calin Cascaval, Jaejin Lee, Peng Wu, Yuan Lin, Jianxin Xiong, and Jianjing Zhu) helped me adapt to the research group. I also want to thank the current members (Ganesh Bikshandi, Shengnan Cong, María Jesús Garzarán, Jia Guo, Xiaoming Li, Gang Ren, and Nick Rizzolo) and Jun Nakano for their help in giving me comments on my presentations. Special thanks goes to Sheila Clark who helped me with a lot of administrative matters and proofreading of this dissertation.

In the beginning, when I transferred to UIUC, I worked with the Concurrent Systems Architecture Group (CSAG). The CSAG people, including Professor Andrew Chien, Jay Byun, Aaron Coday, Julian Dolby, Greg Koenig, Sudha Krishnamurthy, Scott Pakin, Luis Rivera and Geetanjali Sampemane, helped me get started as a graduate student at UIUC.

I would like to thank Dr. Tin-Fook Ngai for introducing me to the world of compilers when I was studying in Hong Kong. He also provided me with valuable advice in pursuing my graduate career.

In my graduate life at UIUC, I have received lots of support and prayers from brothers and sisters in the Illini Chinese Christian Fellowship and the Champaign Chinese Christian Church. They have made Champaign feel more like a hometown to me.

I also want to thank my parents for providing me a stress-free environment to grow up in. They let me develop my own interests naturally without confining me with their

preferences. They also bought my first computer which was the beginning of my programming life. I also want to thank my grandfather for giving me a book about Calculus at a young age. His intention was to help me learn area calculation, but the book-reading process actually developed my reasoning skills to rigorously handle complicated concepts.

Finally, I would like to thank my wife, King-Shan, for her support and companionship. Working together with her in DCL made my graduate career a two-person battle rather than a one-man job. Toward the end of my graduate career, she took care of our family and our daughter, Dorcas, so that I could focus on my work without worrying about other things. My lovely daughter, Dorcas, has been another source of energy that has helped me complete my research at UIUC.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Shared memory is one of the popular parallel programming paradigms. In this paradigm, the different threads[1] of the program communicate with each other by reading from and writing to shared memory locations. Experience shows that, to improve performance, it is necessary that the memory accesses follow an order of execution that is not the most intuitive one. For this reason, memory models have been developed. They specify the memory system behavior observed by different processors. It is not trivial to define a memory model which is both easy to use and implemented efficiently. In light of this, it was decided to develop the Pensieve compiler system in order to provide a testbed to evaluate memory models by creating "virtual" memory models. Given a program, the Pensieve compiler will some day be able to generate different versions of machine code corresponding to different memory models. An important issue in the system design is performance — both the compilation time and application time should be minimized. In this thesis, we focus on a component of the system — thread escape analysis[2]. We will describe a novel fast escape analysis and the evaluation of the analysis by comparing to relevant analyses.

[1] We use *threads* and *processors* interchangeably in this thesis

[2] Since this thesis focuses on thread escape analysis, we use *thread escape analysis* and *escape analysis* interchangeably

This thesis makes the following contributions:

- it describes the Pensieve compilation system;
- it describes a fast escape analysis usable in JIT time that operates in the presence of dynamic class loading;
- it presents a quantitative comparison with two other efficient escape analyses; and
- it reports data comparing the performance of the relaxed memory model and a sequentially consistent memory model enforced by our compiler using our fast escape analysis.

In this chapter, we will introduce memory models in Section 1.1 and describe how to enforce memory models in Section 1.2. In Section 1.3, we describe the structure of this thesis.

1.1 Memory Models

A memory model specifies the memory system behavior. It can be specified for programming languages as well as hardware.

Definition 1.1.1 *The* **memory model of a programming language** *specifies the memory behavior for programs written in that language independently of the hardware where the program is to execute.*

Definition 1.1.2 *A* **hardware memory model** *specifies the memory behavior of the hardware seen by the machine code.*

Memory models are important because they define the allowable set of outcomes of a parallel program and, as a result, allow programmers to reason about their programs.

Until recently, memory models were of concern only to expert systems programmers, and computer architects. With the advent of languages like Java and C#, more programmers write multi-threaded programs usually targeted at internet, database, and GUI applications, which often require multi-threaded programming. Because of this, memory models have become an issue for a large part of the programmer community and for language and compiler designers. The trade-offs between ease-of-use and performance have become increasingly important.

The issue of memory models can be illustrated by a busy-wait synchronization example shown in Figure 1.1(a). Both `x` and `a` are shared variables accessible by two concurrent threads. Thread 1 does some computation and stores the result in `a`. It uses `x` to inform Thread 2 that a *new* value of `a` is ready to be read. Thread 2 waits for the data by executing a while loop. In the loop it reads `x` and waits until the value becomes non-zero. It will then read the value from `a`. Both the values of `x` and `a` are *eventually* propagated from Thread 1 to Thread 2 . However, for performance reasons the compiler or hardware may reorder the two memory operations done by Thread 1 such that the update of `x` propagates to Thread 2 *before* the update of `a`. If this happens, when `T1` is executed, it could read the updated value of `x` (i.e. 1), prematurely quiting the loop. When `T2` is executed, an *old* value (i.e. 0) of `a` could be read. Therefore, the intention of the program is not achieved.

1.1.1 Sequential Consistency

A well-known memory model is sequential consistency (SC), defined by Lamport as follows[Lam79]:

Definition 1.1.3 (Sequential Consistency [Lam79]) *A multiprocessor system is* **sequentially consistent** *if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

Both `x` and `a` are zero initially.

```
// Thread 1                          // Thread 2
      ⋮                                 ⋮
S1:  a = 1;                       T1:  while (x==0) wait;
S2:  x = 1;                       T2:  print a;
```

(a) Busy-wait synchronization

```
      ⋮                                 ⋮
U1:  a = 1;                       V1:  while (x==0) wait;
U2:  fence                        V2:  fence
U3:  x = 1;                       V3:  print a;
```

(b) Fence instruction insertion to avoid reordering.

Figure 1.1: Fence instruction example.

SC is often considered to be the simplest and most intuitive memory consistency model [Hil98].

Consider again the example shown in Figure 1.1(a). If the system is assumed to follow SC, the result of the execution has to be the same as if the operations `S1`, `S2`, `T1` and `T2` were executed in some sequential order, and the operations of each individual processor appear in the sequence in the order specified by its program — `S1` must appear before `S2` and `T1` must appear before `T2`. With these constraints, we have the following conclusions:

1. By the requirements of SC, `T2` must appear after `T1`, so the suffix of the execution sequence must be $\ldots$ `T1`, `T2`.

2. To exit the loop at `T1`, the value of `x` must not be 0.

3. Since the value of `x` was initially 0, `S2` must have been executed before exiting the loop, so the suffix of the execution sequence must be $\ldots$ `S2`, `T1`, `T2`.

4. By the requirement of SC, `S1` must appear before `S2`, so the suffix of the execution sequence must be `S1` ... `S2`, `T1`, `T2`.

5. Since `S1` appears before `T2` in the sequence, the value of `a` *must not* be 0.

We can see SC precludes some reorderings of memory access, making of outcomes of the program more intuitive to the programmers. However, this comes at a cost in program performance. Therefore, other memory models have been proposed to relax the constraints on the memory access order while keeping the models reasonable for the programmers.

Before describing in more detail the different memory models, some definitions are given below. They are from [GLL+90, SD87]. Following [GLL+90], we assume that local requirements like uniprocessor control and data dependence orders are enforced.

Definition 1.1.4 (Performing a memory access w.r.t a processor [GLL+90]) *A* LOAD *by processor P_i is considered performed with respect to another P_k at a point in time when subsequently issued* STORE *to the same address by P_k cannot affect the value returned by the* LOAD. *A* STORE *by P_i is considered performed with respect to P_k at a point in time when a subsequently issued* LOAD *to the same address by P_k returns the value defined by this* STORE*(or a subsequent* STORE *to the same location).*

Using Definition 1.1.4, we can define the notion of *performing globally.*

Definition 1.1.5 (Performing a memory access globally [GLL+90]) *A* STORE *is globally performed when it is performed with respect to all processors. A* LOAD *is globally performed if it is performed with respect to all processors* ***and*** *the* STORE *which is the source of the returned value has also been globally performed.*

Scheurich and Dubois[SD87] described a sufficient condition for SC and Gharachorloo et al[GLL+90] presented it in a slightly difference way as follows.

Condition 1.1.1 (Sufficient Conditions for SC [GLL+90]) *The following two conditions are sufficient to guarantee SC:*

1. *Before a* LOAD *is allowed to perform with respect to any other processor, all previous* LOAD *accesses from the same processor must be globally performed and all previous* STORE *accesses from the same processor must be globally performed, and*

2. *Before a* STORE *is allowed to perform with respect to any other processor, all previous* LOAD *accesses from the same processor must be globally performed and all previous* STORE *accesses from the same processor must be globally performed.*

Condition 1.1.1 imposes constraints to the hardware so that some performance improving optimizations cannot be applied. In addition, it constrains compiler optimizations that may reorder memory accesses such as redundant load elimination and loop invariant motion. This will be discussed in more detail in Section 1.1.3.

1.1.2 Relaxed Consistency Models

Most multiprocessor systems implement consistency models, such as weak ordering and release consistency [AG96], which impose fewer constraints than SC on the order of shared memory accesses. Where clear, we will refer to these more relaxed models by the acronym RC. RC models allow more instruction reordering, increasing the potential for instruction level parallelism and as a result can potentially deliver better performance. Synchronization primitives, such as *fences*, are used in these systems to force an order on memory operations that is more constrained than that implied by the default consistency model.

Weak consistency is one kind of relaxed consistency models. It distinguishes memory accesses into *synchronization memory accesses* and *non-synchronization memory accesses*. The synchronization accesses are operations like lock and unlock operations which

are used to control ordering among processors. The non-synchronization accesses are regular LOAD and STORE operations.

The conditions to ensure weak consistency presented in [GLL+90] are defined as follows.

Condition 1.1.2 (Conditions for Weak Consistency [GLL+90]) *The following three conditions are sufficient to guarantee weak consistency:*

1. *before an ordinary* LOAD *or* STORE *access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed with respect to all processors, and*
2. *before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary* LOAD *or* STORE *accesses must be performed with respect to all processors, and*
3. *synchronization accesses are sequentially consistent with respect to one another.*

By comparing Condition 1.1.2 and 1.1.1, we can see the weak consistency model allows more reorderings of memory accesses than SC. A LOAD access, for example, does not need to wait for other LOAD or STORE accesses to be performed globally. When the program shown in Figure 1.1(a) is executed assuming weak consistency, the STORE of `x` can be performed before the STORE of `a` is performed globally. Therefore, it is possible that the update of `x` propagate to Thread 2 first before the update of `a`. As described before, this violates the intention of the programmer.

To prevent undesired reorderings, memory models provide mechanisms to delay performing memory accesses. For hardware, `fences` instructions are used to enforce the ordering.

Definition 1.1.6 (Fence) *A* **fence** *instruction is a control instruction that delays execution of memory accesses. It divides the stream of memory accesses into two sets S_b and S_a where:*

- *S_b is the set of memory accesses* **before** *the* **fence** *instruction; and*
- *S_a is the set of memory accesses* **after** *the* **fence** *instruction.*

The **fence** *instruction delays the execution of memory accesses in S_a until memory accesses in S_b are performed*[3]*.*

Figure 1.1(b) shows a correct implementation of the busy-wait construct. We assume that the fence ensures that the memory accesses before the fence are all performed globally before any of the memory accesses after the fence are carried out. In the program, the fence in Thread 1 ensures that by the time `U3` is executed, the STORE of `a` has been performed globally, so its value is available at Thread 2. This ensures that the update of `a` arrives at Thread 2 before that of `x`. The fence executed in Thread 2 ensures that all the LOAD of `x` must be performed globally before executing `V3`. By the time `x` receives the value of 1, the update of `a` has been arrived, so the LOAD of `a` must be 1.

1.1.3 Impact on Compiler Optimizations

In Section 1.1.1 and Section 1.1.2, when we talk about enforcing reordering, we focus on the hardware aspect. In this section, we discuss how the requirements of memory models (e.g. Condition 1.1.1) impact compiler optimizations.

Compiler optimizations may change the memory access pattern of programs. Examples of such optimizations are dead code elimination, common subexpression elimination, and redundant load elimination. In these optimizations, changes in the program cause

[3]Depending on the semantics of a specific fence instruction, the meaning of "performed" may mean "performed to all processors" or "performed globally".

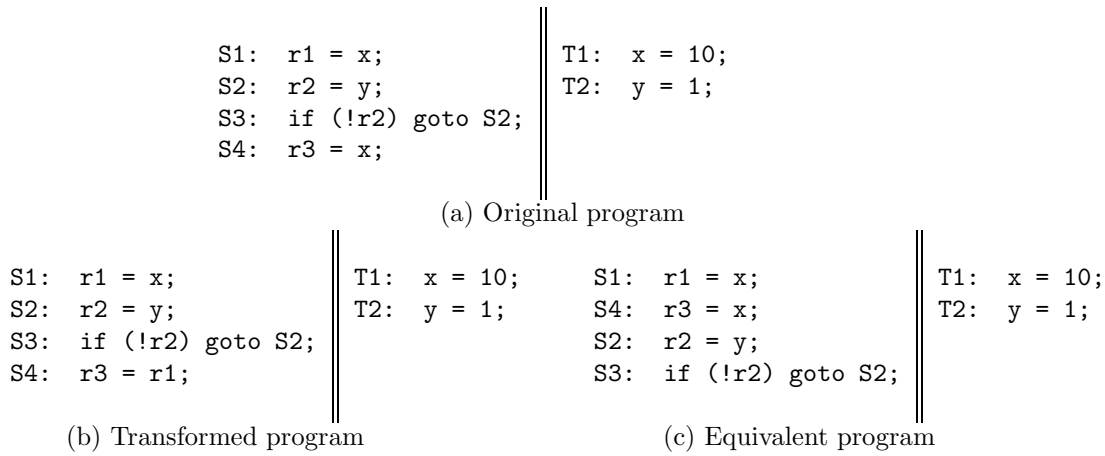


Figure 1.2: Impact on Redundant Load Elimination.

reorderings of memory accesses or in the case of elimination of memory accesses, behaviors equivalent to a reordering. If the reorderings are prohibited by the memory models, the corresponding program transformation should not be done.

A redundant load elimination example has been shown in Figure 1.2. The original program is shown in Figure 1.2(a). Assume that `x` and `y` are memory locations while `r1`, `r2` and `r3` are registers. Redundant load elimination replaces the last load of `x` by reusing the value of first load stored in `r1` and the transformed program is shown in Figure 1.2(b). The transformed program is equivalent to the program shown in Figure 1.2(c). Comparing the original program and the equivalent program, we can see the transformation has essentially reordered `S2` and `S4`. The last load of `x` is performed before the load of `y`. If the memory model requires the last load of `x` be performed after the load of `y`, the transformation has violated that requirement. For example, if the original program assumes SC is enforced, the loop is used to wait for the availability of the value of `x`. The transformed program is essentially skipping the loop, violating the intention of the programmer.

1.2 Enforcing Memory Models

In this section, we describe the theoretical foundation of the Pensieve system. As described in the previous section, memory models imply orderings of memory accesses. However, as shown in [SS88], since what really matters is the final outcome of the program, not all the orderings need to be enforced. All that matters is that the final outcome of the program be consistent with the orderings required by the model. That is, the orderings must "appear to be followed" but they do not have to be enforced. To generate efficient and correct code, a compiler must determine which memory accesses may not be reordered and enforce only those orderings.

Theorems formulated in [SS88] focused on enforcing the sequential consistency (SC) model. The idea is

1. formulate memory model requirements as program statement orderings (described in Section 1.2.1);
2. identify the program statement ordering that must be enforced (described in Section 1.2.2); and
3. use of machine primitives to enforce the orders identified (described in Chapter 2).

All variables equals zero initially.

```
// executed by Thread 1
S1: X = 1;
S2: Y = 2;
```

```
// executed by Thread 2
T1: y = Y;
T2: x = X;
T3: print x, y;
```

Figure 1.3: An example illustrating program statement ordering required by memory models

1.2.1 Representing Memory Model Requirements

To find out program statement orderings required by a memory model, we make use of the conditions imposed by the memory model on the orderings between memory accesses. For SC, we use Condition 1.1.1, while for weak consistency we use Condition 1.1.2. Given a pair of statements `S1` and `S2`, if the condition requires that `S2` appears to be performed only after `S1` is performed, then, `S1` $\rightarrow$ `S2` is an order required by the memory model. Consider a multi-threaded program shown in Figure 1.3. Condition 1.1.1 requires that before any memory access allowed to be performed, all previous memory accesses must be globally performed. Because of that, the following orderings are required: `S1` $\rightarrow$ `S2`, `T1` $\rightarrow$ `T2`, `T1` $\rightarrow$ `T3` and `T2` $\rightarrow$ `T3`, and all transitive orderings. These orderings are the set of program statement orders if sequential consistency is enforced.

The above examples show the program orderings for SC. For weak consistency, fewer orderings are required. Consider the example shown in Figure 1.4,

```
// executed by Thread 1
S0: acquire
S1: x = X;
S2: Y = x;
S3: release
```

```
// executed by Thread 2
T1: X = 1;
T2: release
```

Figure 1.4: An example illustrating apparent relaxed program statement ordering

To enforce weak consistency, Condition 1.1.2 is applied:

- Since before any memory accesses is allowed to be performed, all previous synchronization accesses must be performed, we have the orderings: `S0` $\rightarrow$ `S1` and `S0` $\rightarrow$ `S2`.
- Since before any synchronization access is allowed to perform, all previous memory accesses must be performed, we have the orderings:`S1` $\rightarrow$ `S3`, `S2` $\rightarrow$ `S3`, `T1` $\rightarrow$ `T2`.

- Since synchronization accesses are SC with respect to one another, we have the ordering `S0` $\rightarrow$ `S3`.
- Transitive orders induced by the above orders.

The model, however, does not requires the ordering `S1` $\rightarrow$ `S2`. Note that the order is required locally by data dependence. If we change `S2` to "`Y = 0`", then there is no data dependence between `S1` and `S2` and the order `S1` $\rightarrow$ `S2` needs not be enforced at all.

After finding the program statement orderings, we are ready to find out the program statement orderings that need to be enforced.

1.2.2 Determining Orders to Enforce — Delays

To determine the orders to enforce, a *delay graph* is constructed first.

Definition 1.2.1 *A* **delay graph** *is a graph* $G = (V, P \cup C)$ *where:*

- V *is the set of nodes. It represents the set of simple statements in a shared-memory parallel program*[4]*.*
- P *is the set of program statement ordering required by the memory model (called program edges in [SS88]).*
- $C = \{(S_1, S_2) | S_1$ *and* S_2 *have conflicting memory accesses for some* $S_1, S_2 \in V\}$*. Two memory accesses are conflicting if they address the same memory location and at least one of them is a write (called conflict edges in [SS88]).*

It is shown in [SS88] that the orderings that must be enforced are $(S_1, S_2) \in P$ such that (S_1, S_2) occurs on a minimal mixed cycle.

[4]Assume complicated statements like `X=A+B` have been broken down into statements `t1=A`, `t2=B`, `t3=t1+t2`, `X=t3`, so that each statement contains at most one memory access

Definition 1.2.2 *A minimal mixed cycle is:*

- *minimal — it is not possible to form another cycle using a subset of the nodes on the cycle; and*
- *mixed — the cycle consists of edges from both P and C.*

The orderings that must be enforced are called *delays*. Note that in [SS88] there is following assumption

> A delay between two storage accesses u and v forces access u to complete before access v begins.

Here the meaning of "complete" depends on the memory model. If the delay is to enforce SC, the meaning of "complete" is *globally performed* as required by Condition 1.1.1. If the delay is to enforce weak consistency from ordinary LOAD to a synchronization access, the meaning of "complete" is *performed with respect to all processors* (not *globally performed*) as required by Condition 1.1.2. Both [SS88] and this thesis assume that the hardware provides primitives such as fences powerful enough to enforce these completion orderings.

Consider the example of Figure 1.3. Suppose SC is to be enforced. The delay graph is shown in Figure 1.5.

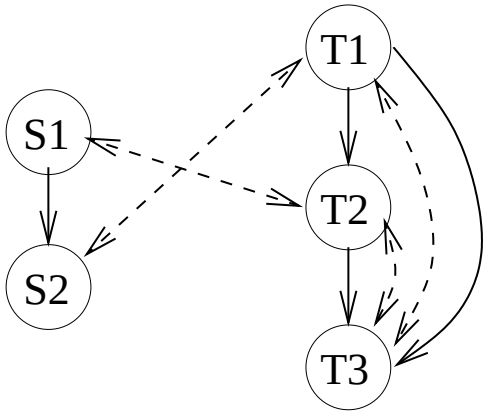


Figure 1.5: Delay Graph of the program shown in Figure 1.3

There are four delays in the graph:

1. T1 → T3 due to the cycle T1, T3, T1

2. T2 → T3 due to the cycle T2, T3, T2

3. S1 → S2 and T1 → T2 due to the cycle S1, S2, T1, T2, S1

We can see the delays correctly capture the fact that S1 and S2 cannot be reordered. Moreover, the first two orders are conventional dependences.

Consider the program shown in Figure 1.6. Again, suppose SC is to be enforced. In this example, the statements S1 and S2 can be reordered without violating SC. In

All variables equals zero initially.

```
// executed by Thread 1
S1: X = 1;
S2: Y = 2;
```

```
// executed by Thread 2
T1: x = X;
T2: y = Y;
T3: print x, y;
```

Figure 1.6: An example illustrating program statement ordering is respect in the presence of reordering

this case, the output 0, 2 resulting from the reordering is also SC conforming because the execution order T1, S1, S2, T2, T3 respects the SC ordering requirement generating output 0, 2.

Consider the example shown in Figure 1.6. The delay graph is shown in Figure 1.7.

There are two delays in the graph:

1. T1 → T3 due to the cycle T1, T3, T1

2. T2 → T3 due to the cycle T2, T3, T2

We can see in this graph we don't have the delay S1 → S2 because there is no minimal mixed cycle containing S1 → S2. The delays correctly capture the fact that S1 and S2 can be reordered.

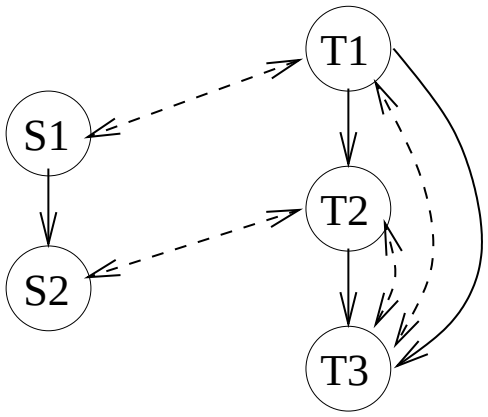


Figure 1.7: Delay Graph of the program shown in Figure 1.6

As described above, a delay $V_1 \rightarrow V_2$ is a program edge in a minimal mixed cycle. There are two possible cases:

- All nodes of the minimal mixed cycle are from the same thread. In this case the only valid minimal mixed cycle is of the form (V_1, V_2, V_1). This can be proved by contradiction. Assume a minimal mixed cycle of size $n > 2$: $(V_1, V_2, \ldots, V_n, V_1)$. Without loss of generality, we can select a conflict edge $V_i \rightarrow V_{i+1}$ where $1 \leq i \leq n$. Since V_i and V_{i+1} are from the same thread, there are two cases:

  - $V_i \rightarrow V_{i+1}$ is a program edge. Since $V_i \rightarrow V_{i+1}$ is a conflict edge, V_i and V_{i+1} have a conflicting access, so $V_{i+1} \rightarrow V_i$ is also a conflict edge. Hence V_i, V_{i+1}, V_i is a cycle containing both conflict and program edges.
  - $V_{i+1} \rightarrow V_i$ is a program edge, so V_i, V_{i+1}, V_i is a cycle containing both conflict and program edges.

  Combining both cases, we see $V_1, V_2, \ldots, V_i, V_{i+1}, \ldots, V_n, V_1$ is not minimal, contradicting that the assumption that $V_1, \ldots, V_n, V_1$ is minimal. Hence the original assumption is wrong and we conclude that a minimal mixed cycle where all nodes are from the same thread contains two nodes and only two nodes.

  Since V_1, V_2, V_1 is a minimal mixed cycle and $V_1 \rightarrow V_2$ is a program edge, $V_2 \rightarrow V_1$ is a conflict edge. Therefore, V_1 and V_2 have conflicting access and the delay $V_1 \rightarrow V_2$

corresponding to a dependence of the program. This is enforced automatically by a correct compiler and architecture.

- The minimal mixed cycle contains nodes from multiple threads.

It is clear that is the second kind of delays that a memory-model aware compiler is needed to enforce.

1.2.3 Conservatively Approximating of Delays By Considering Shared Accesses Only

We can see from the previous section that the delay set analysis is an interthread analysis. In the absence of the thread structure of the program, we can still approximate the delay information conservatively.

As described in Section 1.2.2, delays corresponding to a minimal mixed cycle within a thread is a dependence which is enforced by a correct compiler and architecture. Therefore, in this section, we focus on delays corresponding to a minimal mixed cycle across different threads.

The approximation can be illustrated by Figure 1.8. If `T1` $\rightarrow$ `T2` is a delay, it is within

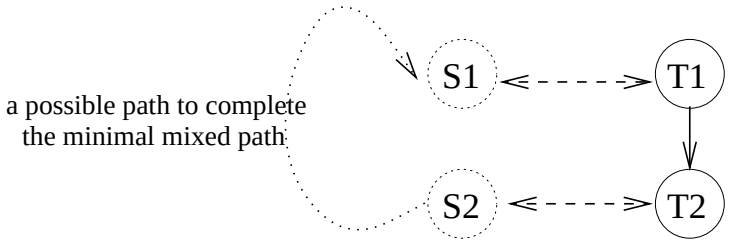


Figure 1.8: Conservative Approximation of Delays

a minimal mixed cycle containing:

- the program edge `T1` $\rightarrow$ `T2`;
- a conflict edge `S1` $\rightarrow$ `T1`;

- a conflict edge `T2` $\rightarrow$ `S2`; and
- the dotted path containing other conflict edges and program edges.

Since `S1` $\rightarrow$ `T1` and `T2` $\rightarrow$ `S2` are conflict edges, `S1` has a conflicting access with `T1` and `S2` has a conflicting access with `T2`. Because `S1`, `S2` are in different threads than that of `T1` and `T2`, `T1` and `T2` access memory locations that are shared with other threads. Therefore, two statements *are not* connected by a delay if either one of them accesses only memory location local to its thread. On the other hand, it is sufficient (though conservative) to assume a delay for each pair of statements T_1 and T_2 if $T_1 \rightarrow T_2$ is a program edge and both access memory location shared with other threads.

Determining whether a statement accesses memory location shared with another thread can be done using a technique called escape analysis, described in detail in Section 3.

1.3 Structure of Thesis

In Chapter 2, we describe the Pensieve system design. In Chapter 3, we describe in detail the escape analysis proposed in this thesis and also other escape analysis algorithms that we compare to the analysis described in this thesis. In Chapter 4, experimental results are presented to evaluate the escape analysis quantitatively. This thesis concludes in Chapter 5.

Chapter 2

Pensieve Compiler System Design

Given the challenges, why consider using any memory model other than one that is relaxed? The design of memory consistency models for both hardware and software is a difficult task [AG96]. It is particularly difficult for a programming language because the target audience is much wider than the target audience for a machine language, making usability a more important criteria. Adding to this problem is the fact that the programming language community has little experience designing programming language consistency models, and therefore each new attempt is very much a voyage into uncharted territory. The difficulty of reaching a consensus on an ideal memory model is exacerbated by the fact that the quality of a model depends both on its ease of use (i.e. how hard it is to write correct programs using the model) and the performance the model can deliver to programs. Therefore, it is desirable to be able to make apples-to-apples comparisons of different memory models, executing on different hardware, so that the trade-offs involved can be quantified. Our Pensieve Compiler System is a tool designed to facilitate this job.

In the rest of this Chapter we describe the goal of the Pensieve Compiler System in Section 2.1. In Section 2.2, we describe the overall organization of the compiler system.

2.1 Goal of the Pensieve Compiler System

The ultimate goal of the Pensieve project is to allow the memory model of the programming language to be implemented by the compiler atop the memory model of the machine. Accomplishing this ultimate goal makes the Pensieve compiler a powerful tool for prototyping different memory models and consistency models and measuring their relative performance on a common, consistent optimization base. A detailed discussion of it is beyond the scope of this thesis. In the near term, our goal is to show that on a significant number of programs we achieve acceptable performance on SC programs relative to those executing using the default relaxed memory model. It is the system for this near term goal that we report on in this thesis. In this thesis we focus on implementing SC on top of two platforms supporting more relaxed memory models — the Intel platform and the PowerPC platform.

2.2 Overall Organization

The Pensieve Compiler System is an extension of the Jikes RVM infrastructure [AAB+00, BCF+99].

Delay identification described in Section 1.2 is the fundamental analysis that needs to be done to ensure program transformations preserve subset correctness. The delays are the ordering constraints to be enforced both by the compiler and the hardware. Figure 2.1 shows the overview of the Pensieve system. It shows three phases:

1. In the analysis phase, a set of delays is computed.

2. We have examined each optimization in the original Jikes [AAB+00, BCF+99] compiler, and augmented it to be aware of delay information in our Pensieve-Jikes system. In the modified code optimization phase, the set of delays identified by the analysis phase is consulted to check whether the optimizations would violate the

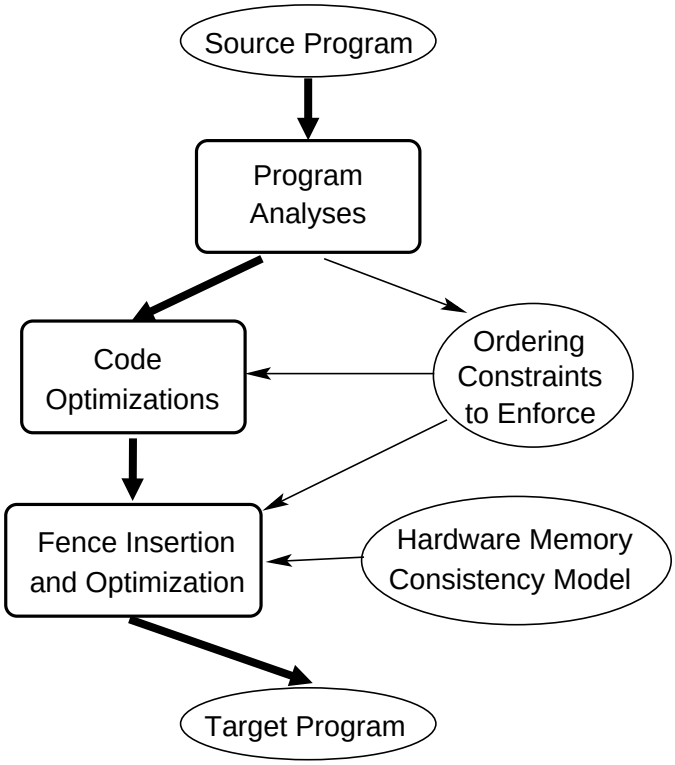


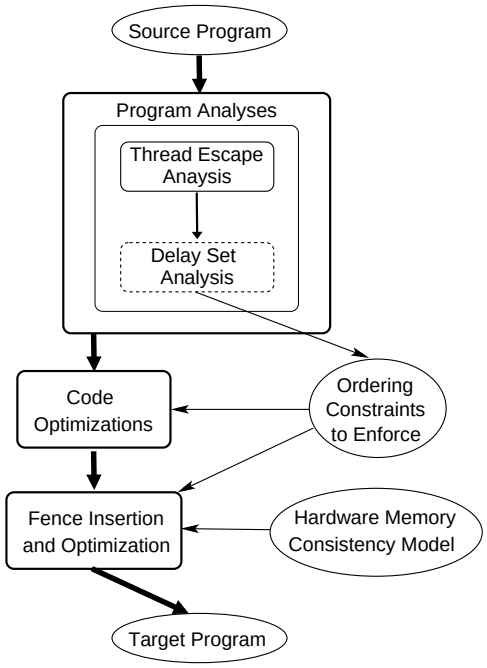
Figure 2.1: Overview of the Pensieve system

delay. If an optimization transformation violate a delay, that transformation would not be applied.

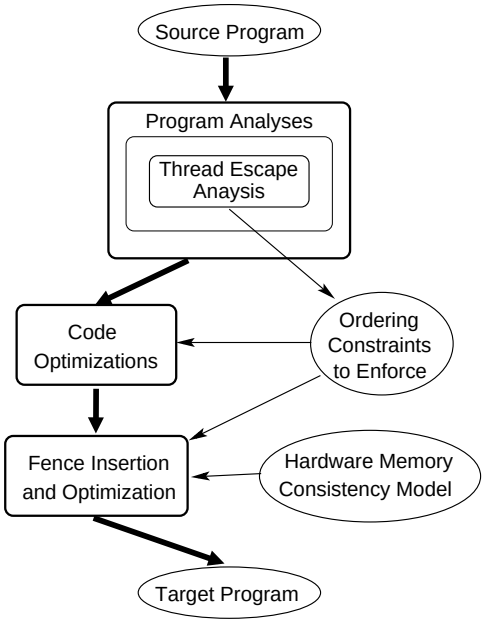
3. In the fence insertion and optimization phase, fences are inserted into the program to make sure the delays are enforced by the hardware. Rather than inserting one fence for each delay, we aim at inserting the fences efficiently. This phase looks for opportunities to synchronize multiple delays with a single fence instruction. The details of this phase are described in [FLM03a, FLM03b].

As described in Section 1.2, precise delay information can be computed by a delay set analysis strategy while (conservative) approximate delay information can be computed by using escape analysis exclusively. Both approaches have been considered and Figure 2.2 gives a graphical overview of the two settings of the Pensieve system. In this thesis we *do not* use delay set analysis. Instead, we use escape analysis to compute the delay information. We can consider the use of delay set analysis to compute the delay

information as an extension to our system. The two settings correspond to the two ways



(a) The Pensieve System setting computing full delay information



(b) The Pensieve System setting computing conservative approximate delay information

Figure 2.2: Two Settings of the Pensieve system(assuming SC)

of computing delay information:

- **Full delay information computation** is shown in Figure 2.2(a), the goal of the program analyses is to find out the ordering constraints using delay set analysis. The thread escape analyses is used to prepare information needed by delay set analysis. The detail of delay set analysis is described and evaluated in Zehra Sura's PhD thesis [Sur04].

- **Conservative approximate delay information computation** is shown in Figure 2.2(b). The only analysis of interest is thread escape analysis which computes approximate delay information (described in Section 1.2.3).

In this thesis, we only deal with the second approach which makes use of escape analysis to compute the delay information.

Chapter 3

Thread Escape Analysis

This chapter focuses on thread escape analysis. In Section 3.1, we first state the problem of thread escape analysis. As our escape analysis is for the Java programming language, we describe some features of Java considered by our analysis in Section 3.2. In Section 3.3, a new escape analysis algorithm, which we call *Connectivity Analysis*, will be presented. In Section 3.4, two uses of escape analysis will be described. In Section 3.5, we briefly describe the next two fastest known escape analysis algorithms and their adaptations to our system:

- Bogda's Escape Analysis [BH99];
- Ruf's Escape Analysis [Ruf00]

Since Bogda's and Ruf's escape analyses were developed for synchronization removal, they were modified so they could be used in the Pensieve system for enforcing SC. These adaptations will be described in Section 3.5. In Section 3.6, the algorithms will be compared qualitatively. After that, we discuss some issues influencing the cost and precision of the analyses:

- Cost and precision trade-offs due to cloning (Section 3.7).

- Our technique to reduce the analysis overhead by caching the intermediate representation (IR) used by the compiler (Section 3.8).
- Analysis issues in a dynamic compilation setting (Section 3.9).

In Section 3.10, we describe an incremental connectivity analysis which is faster and as precise as the non-incremental one. Finally, in Section 3.11, other escape analysis algorithms will be described.

3.1 Problem Statement

Thread escape analysis aims at identifying objects which *may* be accessed by two or more threads.

Figure 3.1 shows a program code accessing two kinds of objects. The object created in Thread 1 and referenced by `esc` is accessed in Thread 2 via the field `this.data`. We say that the referenced object is thread-escaping. In contrast, the object created in Thread 1 referenced by `local` is not accessible from threads other than Thread 1. We say that it is thread-local. Throughout the discussion, we say that a *variable escapes* when the variable may reference an escaping object.

```
1  // main() executed by Thread 1
2  void main(String args) {
3    MyThread t = new MyThread();
4    Data esc = new Data();
5    Data local = new Data();
6    t.data = esc;
7    t.start();
8  }
```

```
1  class MyThread extends Thread {
2    Data data;
3    // run() executed by Thread 2
4    public void run() {
5      Object o = this.data;
6      // work on o
7    }
8  }
```

Figure 3.1: Escaping Object vs Non-escaping object

Besides identifying objects that may be referenced by two or more threads, thread escape analysis also identifies the statements where escaping objects are accessed. To

avoid interthread analysis, many escape analysis algorithms (including Bogda's escape analysis) conservatively assume that an object that may be referenced via a static field, or via a thread object is escaping. In our technique, we refine these assumptions so an object is assumed to be escaping if it may be referenced via a static field or a thread object *and* the object is accessed by more than one thread.

3.2 Escape Analysis for the Java Programming Language

In this thesis we focus on escape analyses for the Java programming language. Our analysis algorithm takes advantages of the type safety feature of Java.

In Section 3.2.1 and Section 3.2.2, we describe two features in Java that can cause objects to be escaping. In Section 3.2.1 we describe static fields while in Section 3.2.2 we describe thread objects. In Section 3.2.3, we list the statements that must be processed by our escape analysis algorithm.

3.2.1 Static Fields

Definition 3.2.1 (Static field) *A field is a* **static field** *if it is declared with a* `static` *keyword in a class declaration.*

A static field can be considered as a global variable because it can be accessed from any point of a program. Figure 3.2 shows an example of accessing a static field `Global.o`. It is declared inside class `Global`. It is accessed inside `foo` by specifying the class name `Global` and the field name `o` using the dot operator.

```
1  class Global {
2    static Object o;
3  }
4  class Main {
5    public void foo() {
6      Global.o = new Object();
7    }
8  }
```

Figure 3.2: Static field

3.2.2 Thread Creation and Thread Objects

In Java, there are two ways to create threads. Figure 3.3 shows how threads are created and started. In both cases, the `start` method is used to start a new thread of execution and the execution starts from the `run` method. Moreover, within `run`, `this` references the same object as `me` in `main`. We say the referenced object is a thread object in this thesis. We can see thread objects are shared between the thread creators and the created threads.

```
class MyThread extends Thread {
  ...
  public MyThread() {
  }
  public void run() {
    this.data = ...;
  }
}
static void main(String[] args) {
  MyThread me = new MyThread();
  me.start();
}
```

(a) by subclassing `java.lang.Thread`

```
class MyThread implements Runnable {
  ...
  public MyThread() {
  }
  public void run() {
    this.data = ...;
  }
}
static void main(String[] args) {
  MyThread me = new MyThread();
  Thread t = new Thread(me);
  t.start();
}
```

(b) by implementing `java.lang.Runnable`

Figure 3.3: Thread Creation

3.2.3 Statements Processed by Escape Analysis

Figure 3.4 shows statements that are processed by escape analyses. The analysis focuses on statements that access reference variables.

$x = y$	Assignment statement
$x[\,] = y$ $y = x[\,]$	Array access statement
$x.f = y$ $y = x.f$	Field access statement
$y =$ `getstatic` f `putstatic` f y	Static field access statement
a_0=$a_1.n(a_2, \ldots, a_k)$	Method call statement
y.`start()`	Thread start call statement
`return` x	Return statement
`throw` x	Exception throw statement

Figure 3.4: Statements processed by our escape analysis algorithm.

3.3 Connectivity Analysis

In this section, a new escape analysis algorithm is proposed. In Section 3.3.1, the goal of the algorithm design is described. In Section 3.3.2, we will introduce the *reachable set* which is a key concept of the algorithm. After that, we describe the three versions of the connectivity analysis:

- A simplified version of connectivity analysis algorithm which conservatively handles program with `Runnable` objects (Section 3.3.3).
- The full version of connectivity analysis which produces more precise result for program with `Runnable` objects (Section 3.3.4).
- The extended version of connectivity analysis which is an extension of the full algorithm which gives even more precise result especially for single threaded programs (Section 3.3.5).

Finally, we describe some properties of connectivity analysis in Section 3.3.6.

3.3.1 Goal of Algorithm Design

In the Pensieve System, the escape analysis algorithms used in this study were implemented as a module within a dynamic compilation system described in more detail in Section 4.2.2. As discussed in Section 2.1, one of the goals of the Pensieve system is to execute programs with acceptable performance, so we need to avoid performance degradation. Because of the dynamic compilation strategy, the time to perform escape analysis is part of the overall execution time. Therefore, to minimize the overall execution time, we cannot use an expensive analysis algorithm where effectiveness is achieved at great cost. In this project, we balanced performance of the analysis algorithm and its accuracy. While we are not aiming at having an escape analysis that is precise for all program points, the analysis should be precise enough for frequently executed methods so that fences are not unnecessarily inserted within those methods. In light of this, we choose to design the simplest possible algorithm in order to minimize the cost of the analysis. As we later show in Chapter 4, this suffices for many programs. Like earlier algorithms, our algorithm analyzes objects ignoring subscripts. This is acceptable for most Java codes.

Also, since the union-find data structure is very efficiently used in both escape analysis and pointer analysis, we chose to use this data structure to represent the analysis information. In the following section, we describe the *reachable set* which is used to compute escape information and can be represented efficiently using the union-find data structure.

3.3.2 Reachable Set

In this section, we describe the notion of reachable set which is crucial to the analysis. Before defining reachable the set, we present several auxiliary definitions.

Definition 3.3.1 (Objects) *We define the set* **Objects** *as the set of objects instantiated in the program.*

Definition 3.3.2 (References) *We say that a local variable v* **references** *an object o where $o \in Objects$, if v may contain the address of o at some point of an execution of the program.*

Definition 3.3.3 (Directly Reaching Relation for Objects) *We say that an object o_1 is* **directly reaching** *an object o_2 iff one the following situations is true:*

Case 1: $o_1 = o_2$

Case 2: $o_1.f$ *contains the address of* o_2 *for some field* f *of* o_1

Definition 3.3.4 (Reaching Relation for Objects) *We define the* **reaching relation** *as the transitive closure of the* **directly reaching relation**. *That is, we say an object o_1 is* **reaching** *an object o_2 iff there exists objects $o'_0, o'_1 \ldots o'_k$ for some $k > 0$ such that:*

- $o'_0 = o_0$;
- $o'_k = o_1$;
- o'_i *is* **directly reaching** *to* o'_{i+1} *for* $0 \leq i \leq k-1$

Definition 3.3.5 (Reachable set of an object o, $ReachableSet(o)$) *An object $o' \in Objects$ belongs to the* **reachable set** *of an object o, $ReachableSet(o)$, if o is reaching o'.*

Definition 3.3.6 (Reachable set of a local variable y, $ReachableSet(y)$) *An object $o \in Objects$ belongs to the* **reachable set** *of a local variable y, $ReachableSet(y)$, if y*

references an object o' and o' is reaching o. That is,

$$ReachableSet(y) = \bigcup_{y \text{ references } o} ReachableSet(o)$$

We can find out the escape information using the following strategy:

- Introduce an artificial variable `ESCAPE`;
- Assume `ESCAPE` points to v if v is:
  - assigned a value from a static field;
  - assigned to a static field; or
  - used to start a thread. That is, there is a call v`.start()` in the method
- We (conservatively) assume that a variable v is escaping if $ReachableSet(v) \cap ReachableSet($`ESCAPE`$)$ is not empty.

Like other escape analysis algorithms, only reference variables are considered when performing the analysis. For example, the analysis does not process the statement "`x=y.f`" if `x` is of `int` type.

3.3.3 The Simplified Version of Connectivity Analysis

In this section we describe the simplified version of the connectivity analysis algorithm. We describe the representation of the connectivity relation in Section 3.3.3.2. Then, in Section 3.3.3.3, we present an outline of the strategy to compute the connectivity relation. We will describe in more detail how connectivity is computed in Section 3.3.3.4, Section 3.3.3.5, and Section 3.3.3.6.

3.3.3.1 Computing the Reachability Set

To compute precise reachability set information, we need to keep track of field references. For example, the three statements "`c.f2 = b`", "`c.f3 = y`" and "`x.f1 = a`" should imply that:

- *ReachableSet*(`c.f2`) = *ReachableSet*(`b`);
- *ReachableSet*(`c.f3`) = *ReachableSet*(`y`); and
- *ReachableSet*(`x.f1`) = *ReachableSet*(`a`).

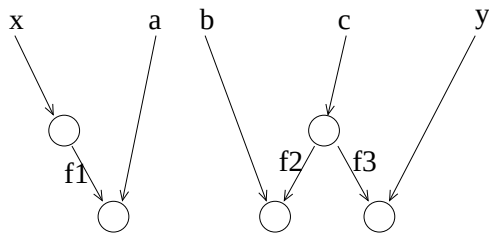


Figure 3.5: Exact vs Approximate Reachability Set Information

The information is pictorially shown in Figure 3.5. As `c.f2` and `c.f3` do not reach any common objects, we conclude that *ReachableSet*(`b`) ∩ *ReachableSet*(`y`) is empty.

However, as shown in Chapter 4, keeping track of this information incurs analysis time because the analysis data structure is more complicated and it takes more time to compute on the data structure. Consider the program in Figure 3.6. To avoid merging the reachable sets of `t.left` and `t.right` in `f1` due to method invocations to the same method `f0`, the analysis data structure of `f0` is copied before used in the analysis of `f1`. When processing the method calls, the analysis data structure of `f0` is copied twice — one copy is used for the method call "`f0(t.left)`" and the other for "`f0(t.right)`". We call this copying operations *cloning*. More detailed discussion of cloning will be given in Section 3.3.3.4. Suppose there are $k+1$ methods `f0`, `f1`, ...`f`k. There are $4k+4$ statements while there are $\sum_{i=0}^{k+1} 2^i = 2^{k+2} - 1$ nodes for `f`k in the analysis data structure

to keep track of precise field reference. This shows that the analysis data structure can be of exponential size of the number of methods.

```
1  ...
2  void f2(t) {
3    f1(t.left);
4    f1(t.right);
5  }
6  void f1(t) {
7    f0(t.left);
8    f0(t.right);
9  }
10 void f0(t) {
11   t.left = new TreeNode();
12   t.right = new TreeNode();
13 }
```

Figure 3.6: A Program Showing the Possibly of Having Exponential Sized Analysis Data Structure

Because of this, we compute the (conservative) approximation, $ReachableSet_{\text{app}}$, instead by *not* recording the field reference. The statement "`c.f2 = b`" implies that $ReachableSet(\texttt{c}) \supseteq ReachableSet(\texttt{b})$. Since we do not record the field reference in the analysis, the analysis conservatively assumes $ReachableSet_{\text{app}}(\texttt{c}) = ReachableSet_{\text{app}}(\texttt{b})$. Similarly, the analysis conservatively assumes that "`c.f3 = y`" implies that $ReachableSet_{\text{app}}(\texttt{c}) = ReachableSet_{\text{app}}(\texttt{y})$. Because of this, we *cannot* conclude that $ReachableSet_{\text{app}}(\texttt{b}) \cap ReachableSet_{\text{app}}(\texttt{y})$ is empty. However, we *can* conclude that $ReachableSet_{\text{app}}(\texttt{x}) \cap ReachableSet_{\text{app}}(\texttt{b})$ is empty. We can see the approximation introduces imprecisions to the analysis result but our experimental result in Chapter 4 shows that the imprecisions do not introduce performance losses for many benchmark programs when SC is enforced. On the other hand, with this approximation, the analysis time is reduced significantly for many programs. Consider the program in Figure 3.6 again. There is just one node for `f`k in the approximate analysis since there is no difference between `t.left` and `t.right`. The node represents all objects reachable from `t`. Another property of the approximate reachable set is that given two such sets A

and B, they are either disjoint ($A \cap B = \phi$) or they are the same ($A = B$). With this property, the approximate reachablet set can be implemented efficiently. A more detailed discussion is given in Section 3.3.3.2.

The idea of the simplified version of analysis is to compute for all variables v, $ReachableSet_{\text{app}}(v)$ which is a superset of $ReachableSet(v)$. Therefore, for any variables v_1, v_2, we have

$$\begin{aligned} & ReachableSet(v_1) \quad \cap \quad ReachableSet(v_2) \neq \phi \\ \Rightarrow\ & ReachableSet_{\text{app}}(v_1) \quad \cap \quad ReachableSet_{\text{app}}(v_2) \neq \phi \end{aligned}$$

However the converse it not true. In the example, we have $ReachableSet_{\text{app}}(\texttt{b}) \cap ReachableSet_{\text{app}}(\texttt{y}) \neq \phi$ but $ReachableSet(\texttt{b}) \cap ReachableSet(\texttt{y}) = \phi$. Moreover, the following negative information is useful:

$$\begin{aligned} & ReachableSet_{\text{app}}(v_1) \quad \cap \quad ReachableSet_{\text{app}}(v_2) = \phi \\ \Rightarrow\ & ReachableSet(v_1) \quad \cap \quad ReachableSet(v_2) = \phi \end{aligned}$$

Therefore, if the approximate reachable set of a variable v and the approximate reachable set of `ESCAPE` have empty intersection, the precise reachable set of a variable v and the precise reachable set have empty intersection as well, which implies the variable v must not be escaping. We say two variables v_1, v_2 are *connected* if $ReachableSet_{\text{app}}(v_1) \cap ReachableSet_{\text{app}}(v_2) \neq \phi$.

3.3.3.2 Representing the Approximate Reachable Set

We can see from the previous section that computing the approximate reachable set requires two basic operations:

- Find the approximate reachable set of v, given a local variable v;

- Merge two approximate reachable set together to form a larger approximate reachable set

These two operations can be implemented efficiently using the union-find data structure [CLR90]. We call this data structure the *connectivity set* which supports the following operations:

- FIND(x) — find the connectivity set for x. Using FIND, we can check whether the approximate reachable sets of two local variables x, y have a non-empty intersection by checking whether FIND(x) equals FIND(y).
- UNION(S_1, S_2) — perform the union of two approximate reachable sets S_1 and S_2. That is, after it is executed, we have that a new value S such that FIND(x)=S for any $x \in S_1 \cup S_2$. Using UNION, we can merge the connectivity sets of variables x and y together by performing UNION(FIND(x), FIND(y)).

In the following discussion, when we perform the union operation on the connectivity sets of two local variables, we say we connect the two local variables. Also, we say two variables are connected if their connectivity sets have non-empty intersection. Using the FIND operation, we can check whether a variable v is escaping by checking whether v is connected to `ESCAPE`.

3.3.3.3 The Algorithm to Compute the Reachability Set by Connecting Variables

The connectivity analysis is a two-phase analysis. Each phase computes a connectivity relation for each method m. The computed relation records the connectivity of formal parameters ($\boldsymbol{Param_m}$), exception value ($Exception_m$), the return value ($Return_m$) and the artificial `ESCAPE` variable.

1. The **bottom-up phase** computes the effect of methods. It computes how the methods connect the formal parameters. This will be described in Section 3.3.3.4.

2. The **top-down phase** computes the context of methods. It combines the connectivity relation of actual arguments in different call sites. This will be described in Section 3.3.3.5.

Note that both the top-down and bottom-up phases do not save the connectivity information of local variables. Given a method, we can construct the connectivity information of local variables whenever needed by using the analysis result of the bottom-up phase and top-down phase. This will be covered in Section 3.3.3.6.

3.3.3.4 Bottom-up Phase

The bottom-up phase is performed by visiting the strongly connected component (SCC) graph induced by the call graph[1] in reverse topological order.

For each method in the SCC, the connectivity information is computed by processing all of its statements, one by one, in any order according to the rules shown in Figure 3.7. Before discussing the rules we present the operations used to define them:

- $ResolvedMethods(a, n)$ returns the set of methods that a call may refer to.
- $InSameSCC(m_1, m_2)$ checks whether the two methods m_1 and m_2 are in the same SCC of the call graph.
- $\textsc{Union}(\langle r_1, \ldots, r_n \rangle, \langle s_1, \ldots, s_n \rangle))$ performs $\textsc{Union}(r_i, s_i)$ for $1 \leq i \leq n$.
- $clone(\langle r_1, \ldots, r_n \rangle)$ creates a copy of the data structure representing the object sets $r_1, \ldots, r_n$ and their connectivity. That is, $clone(\langle r_1, \ldots, r_n \rangle)$ returns the connectivity sets $\langle r'_1, \ldots, r'_n \rangle$ such that for $1 \leq i \leq n$ the following two conditions hold:

[1]The call graph is constructed by our implementation of Rapid Type Analysis[Bac98]

1.	$x = y$	UNION(FIND(x), FIND(y))
2.	$x[\,] = y$ $y = x[\,]$	UNION(FIND(x), FIND(y))
3.	$x.f = y$ $y = x.f$	UNION(FIND(x), FIND(y))
4.	y = `getstatic` y `putstatic` f y y.`start()`	UNION(FIND(y), FIND(`ESCAPE`))

5.	a_0=$a_1.n(a_2, \ldots, a_k)$	$sc = \langle$FIND(a_1), ..., FIND(a_k), FIND(a_0), FIND($Exception_m$)$\rangle$
		`foreach` $f(p_1, \ldots, p_k) \in ResolvedMethods(a_1, n)$ `where` $p_1, \ldots p_k$ `are the formal paramters`
		$fc = \langle$FIND(p_1), ..., FIND(p_k),
		FIND($Return_f$), FIND($Exception_f$)$\rangle$
		`if` $InSameSCC(m, f)$ `then`
		UNION(sc, fc)
		`else`
		UNION($sc, clone(fc)$)
		`endif`
		`end foreach`

6.	`return` x	UNION(FIND(x), FIND($Return_m$))
7.	`throw` x	UNION(FIND(x), FIND($Exception_m$))

Figure 3.7: Rules for analyzing a method m

- If r_i = FIND(`ESCAPE`), then r'_i = FIND(`ESCAPE`). That is, we do not clone the connectivity set for the artificial variable `ESCAPE`.
- If FIND(r_i) = FIND(r_j) for some $1 \leq j \leq n$, then FIND(r'_i) = FIND(r'_j). That is the connectivity relation of r_i and r_j is copied to the returned data structure.

The simplified version of connectivity analysis handles array accesses and field accesses identically. For array accesses, we only have connectivity sets for whole arrays, i.e. the connectivity set of an element is that of the whole array. Similarly, for field accesses, we only have connectivity sets for whole objects, i.e. the connectivity set of a field is that of the whole object.

We illustrate the rules of simplified version of connectivity analysis (shown in Figure 3.7) as follows:

1. For each instruction $x = y$, we do UNION(FIND(x), FIND(y)). Performing this operation makes $ReachableSet_{\text{app}}(x) = ReachableSet_{\text{app}}(y)$.

2. For each instruction $x[\,] = y$ or $y = x[\,]$, we do UNION(FIND(x), FIND(y)). This conservatively reflect that objects reachable from x are also reachable from y and vice versa. This is conservative because the array object pointed by x is actually *not* reachable from y, so the approximate reachable set of y is a superset of the precise one. We do not distinguish the array elements from the array because this can save the analysis time without causing slowdowns of programs as shown in Chapter 4.

3. For each instruction $x.f{=}y$ or $y{=}x.f$, we do UNION(FIND(x), FIND(y)). This is similar to the case of the processing array access instruction. By being field insensitive, the analysis time can be reduced. However, we find that for certain kinds of objects, the analysis should be field sensitive to improve the precision. Detailed discussion is given in Section 3.3.4 when we describe the full version of connectivity analysis.

4. For each instruction $y =$ `getstatic` f or `putstatic` f y , we do UNION(FIND(y), FIND(`ESCAPE`)). This makes $ReachableSet_{\text{app}}(y){=}ReachableSet_{\text{app}}$(`ESCAPE`).

5. For each method call, there may be more than one method that could be invoked. All such possible methods will be processed when the statement containing the invocation is analyzed. When inside the body of a method m, for each method f that an invocation could refer to, we proceed as follows:

Case 1: m and the resolved method f are not in the same SCC in the call graph. Due to the way we traverse the callgraph, the analysis result of f is available when the invocation is analyzed. We simply *clone* the data structure containing the connectivity of the formal parameters, result value and exception value. Using the cloned data structure, we perform a UNION operation for the corresponding

formal and actual arguments, result value, exception value one. The cloning operation helps improve precision as described in more detail in Section 3.7.

Case 2: Both m and the resolved method f are in the same SCC in the callgraph. If we were to clone the result of the analysis of f as in the previous case, a fix-point computation would be needed to analyze m and f. To avoid this fix-point computation, we follow [Ruf00] and do not clone the analysis result of f. We could have done the fix-point computation at a much lower cost than is needed for [Ruf00], but we did not do it because we find the current strategy sufficient for our test cases. Unlike [Ruf00], doing the fixpoint computation is much cheaper in our analysis because we are mostly field-insensitive, making the cost of unification much cheaper. In general, not cloning the method summary hinders the precision because information specific to m could be propagated to f. A detailed discussion of the imprecision due to not cloning method summaries is presented in Section 3.7.

6. For each return statement `return` x, we do UNION(FIND(x), FIND($Return_m$)). This makes $ReachableSet_{\text{app}}(x)$=$ReachableSet_{\text{app}}(Return_m)$.

7. For each exception throwing statement `throw` x, we do UNION(FIND(x), FIND($Exception_m$)). This makes $ReachableSet_{\text{app}}(x) = ReachableSet_{\text{app}}(Exception_m)$.

At the end of the analysis, we save the connectivity information for arguments, return value and exception value only to save the space needed. As mentioned above, the connectivity information of the other variables is recomputed when needed.

Example of bottom-up phase

Consider the program in Figure 3.8. As the bottom-up analysis phase follows reverse topological order when traversing the SCC graph, it visits `buildTree` before `work`. With-

```
1  static TreeNode buildTree(int level) {
2    TreeNode t = new TreeNode();
3    Data td = new Data();
4    t.data = td;
5    if (level != 0) {
6      TreeNode tl = buildTree(level - 1);
7      TreeNode tr = buildTree(level - 1);
8      t.left = tl
9      t.right = tr;
10   }
11   return t;
12 }
13 static TreeNode gt;
14 static void work() {
15   TreeNode t = buildTree(10);
16   gt = t; // putstatic
17   TreeNode gt1 = gt.left;
18   Data gd = gt1.data;
19   TreeNode lt = buildTree(10);
20   TreeNode lt1 = lt.left;
21   Data ld = lt1.data;
22 }
```

Figure 3.8: Bottom-up Phase Example

out going into the detail of how the analysis handles each instruction, we highlight the end result for each method. Note that `buildTree` is recursive with respect to itself, so when the method calls at lines 6 and 7 are analyzed, its analysis result is used without cloning because it is in the same SCC as its caller. The analysis simply concludes that all variables `t`, `td`, `tl`, `tr` are in the same connectivity set. Note that `ESCAPE` is not connected to them, so the bottom-up phase says that executing `buildTree` does not make the variables escape. At the end of analysis of `buildTree`, to reduce the memory requirements, all the information is dropped except the result value and the `ESCAPE` value. It records that the return value does not connect to `ESCAPE`.

When `work` is visited, the analysis of `buildTree` has been done already. The cloned result of `buildTree` is used to analyze method calls at lines 15 and 19. Using the analysis result, the analysis find that `t` and `lt` are not escaping. When line 16 is analyzed, the `putstatic` instruction makes `t` escaping (i.e. `t` and `ESCAPE` are connected). Since the analysis result of `buildTree` has been cloned, connecting `t` and `ESCAPE` does not make the result value of `buildTree` connected to `ESCAPE`. Thus, `lt` is *not* connected to `ESCAPE` due to the analysis of line 16. As the analysis continues, `t`, `gt1`, `gd` and `ESCAPE` will be connected. On the other hand, `lt`, `lt1` and `ld` are connected but none of them are connected to `ESCAPE`, so they are not escaping.

3.3.3.5 Topdown Phase

The top-down phase is performed by visiting the SCC graph induced by the call graph in topological order. In this phase the *context connectivity* information is computed. That is, it determines how the callers connect the formal parameters of a method. For each method m in the SCC, the context connectivity information of m is used as the initial connectivity information to do the analysis. This information was computed when the callers of m is processed. There are two exceptions — the main method of the program and `run` methods for `Thread` classes:

- For the main method, it may not have callers because it is the entry point of the program. We simply assume that the formal parameters of main method are not escaping (i.e. no formal parameters are connected to `ESCAPE`) and use that as the context connectivity information.

- For the `run` method, it is the entry point of threads. We assume `this` is connected to `ESCAPE`.

  Consider the example shown in Figure 3.9, after executing lines 5 and 6, two threads, say $T1$ and $T2$ start execution at `MyThread.run()`[2]. For $T1$, the `this` variable in

```
// Executed by the main thread
void main(String[] args) {
  MyThread t1 = new MyThread();
  MyThread t2 = new MyThread();
  t1.start();  // Starts thread T1
  t2.start();  // Starts thread T2
}
class MyThread {
  Data data;
  MyThread() {
  }
  // Executed by threads T1 and T2
  public void run() {
  }
}
```

Figure 3.9: An Running example illustrating analysis of `run` method

`run` has the same value as `t1` in `main` which is executed by the main thread. For $T2$, the `this` variable in `run` has the same value as `t2` in `main` executed by the main thread. In the simplified connectivity analysis, the algorithm is conservative when analyzing the `run` method because of the sharing behavior of `this` in `run` method. As described above, the analysis constructs the context connectivity information

[2]There is another way to create and start threads by writing code in classes implementing the `Runnable` interface, described in Section 3.2.2. For simplicity we describe the analysis of `run()` for `Thread` classes but in our implementation we also handle `run()` of `Runnable` classes in a similar manner.

for `run` assuming `this` is connected to `ESCAPE`. This is too conservative for some programs. The full version of the analysis described in Section 3.3.4 can handle this pattern more precisely.

Using this information, the analysis performs the intraprocedural analysis as described in Section 3.3.3.4[3]. After the intraprocedural analysis, we are ready to propagate the context connectivity information further to methods invoked by m. It visits each method invocation instructions of m following the rule shown in Figure 3.10

a_0=$a_1.n(a_2, \ldots, a_k)$	$sc = \langle \textsc{Find}(a_1), \ldots, \textsc{Find}(a_k), \textsc{Find}(a_0), \textsc{Find}(Exception_m) \rangle$
	`foreach` $f(p_1 \ldots, p_k) \in ResolvedMethods(a_1, n)$
	$fc = \langle \textsc{Find}(p_1), \ldots, \textsc{Find}(p_k),$
	$\textsc{Find}(Return_f), \textsc{Find}(Exception_f) \rangle$
	`if` $InSameSCC(m, f)$ `then`
	$\textsc{Union}(sc, fc)$
	`else`
	$\textsc{Union}(clone(sc), fc)$
	`endif`
	`end foreach`

Figure 3.10: Rule for analyzing call instruction of a method m

Note that in Figure 3.10, instead of cloning the callee information, the caller information is being cloned. This is to avoid information of one callee propagating to another callee.

Example of top down phase

The topdown phase determining how formal parameters are connected by the actual parameters in the caller. The difference between top-down and bottom-up connectivity information is illustrated by the program shown in Figure 3.11.

[3]When handling method calls, we perform cloning for recursive as well as non-recursive methods

```
void main(String args) {
  Data x, y, z;
  y.f = z;
  f(x, y, z);
}
// top-down: a, b, c connected
// bottom-up: a, b connected
f(Data a, Data b, Data c) {
  a.f = b;
}
```

Figure 3.11: An example showing the difference between top-down and bottom-up phases

After the bottom-up phase, the bottom-up connectivity information of `f` records that `a` and `b` are connected and that `a` and `b` are not connected to `c`. This summarizes what `f` does to its formal parameters. The top-down phase computes `main`, the caller of `f`, affects the connectivity of the formal parameters of `f`. Just before propagating the context connectivity information from `main` to `f`, the working connectivity information summarizes that `x`, `y` and `z` are connected. This information is propagated to `f` as its context connectivity information: `a`, `b` and `c` are connected.

3.3.3.6 Reconstruction

After the top-down and bottom-up phases, for each method m, we have the context connectivity information of m and the connectivity information for callees of m. Using these pieces of information, we can reconstruct the connectivity information of local variables of m. This reconstruction of connectivity information is cheap because it does not perform interprocedural analyses. All it does is using the context connectivity information as the initial working connectivity information to perform intraprocedural analysis described in Section 3.3.3.4.

3.3.4 Full Version of Connectivity Analysis

As we see from our discussion in Section 3.3.3, the simplified version of connectivity analysis is efficient but it may be too conservative when analyzing `run` method of thread classes. In this section, we describe the full version of connectivity analysis which is more precise when compared with the simplified one. Although the full version is more precise, it is similar to the simplified one, and includes two phases. The first phase is the bottom-up phase (described in Section 3.3.4.3) and the second phase is the top-down phase (described in Section 3.3.4.4).

3.3.4.1 A Motivating Example

```
// Executed by the main thread
void main(String[] args) {
  Input shared = new Input();
  MyThread t1 = new MyThread(shared);
  MyThread t2 = new MyThread(shared);
  t1.start();  // Starts thread T1
  t2.start();  // Starts thread T2
}
class MyThread {
  Data data;
  Input shared;
  int sum;
  MyThread(Input shared) {
    this.shared = shared;
    data = new Data();
  }
  // Executed by threads T1 and T2
  public void run() {
    int steps = shared.steps;
    this.sum = 0;
    for (int i = 0 ; i < steps; i++){
      // work on MyThread.data
      ...
    }
  }
}
```

Figure 3.12: An Motivating example for the full version of analysis

An motivating example is shown in Figure 3.12. In this program, for $T1$, we have the following observations:

- The `t1` variable in `main` and `this` in `run` point to the same object. However, the `t1` variable in `main` is only used to start the thread, so it is safe to assume `this.sum` is accessed by $T1$ only.
- `t1.shared` and `t2.shared` point to the same object and it is accessed by both $T1$ and $T2$.
- `t1.data` is accessed by $T1$ only.

The full version of connectivity analysis aims at identifying `this.data` and `this.sum` as unshared accesses. In order to do this, the analysis must be able to distinguish fields of the `Runnable` object, so that it can determine that `this.data` and `this.shared` point to different objects.

3.3.4.2 Computing the Reachability Set

In the simplified version of connectivity analysis, we merge the connectivity sets of local variables v_1 and v_2 whenever it is found that $ReachableSet(v_1) \cap ReachableSet(v_2)$ is not empty. Therefore, the simplified version of the analysis is field-insensitive as we do not distinguish different fields of an object. As illustrated by the program in Figure 3.12, this may be too imprecise. In this section, we present a field sensitive algorithm for two kinds of objects: `Runnable` objects and objects having `Runnable` fields. For example, when the analysis processes a statement "`r.data = d`" where `r` is of `Runnable` type, it does not merge the connectivity set of `r` and `d`. Instead, it records that the connectivity set of `r` points to the connectivity set of `d`.

3.3.4.3 Bottom-up Phase

The bottom-up phase is similar to the one described in Section 3.3.3.4. We present the rules in Figure 3.18 and highlight the differences in this section. Before discussing the rules, we present the operations used in the algorithm:

- $DeclaringClass(f)$ returns the declaring class of the field f.
- $IsRunnableClass(c)$ check whether the class c implements the `java.lang.Runnable` interface.
- $HasRunnableField(c)$ check whether the class c has a field implementing the `java.lang.Runnable` interface.
- $FieldAccess(s, f)$ returns the connectivity set that is referenced by the connectivity set of s with field access f. If no such set is there, a new connectivity set is created and the analysis records that the connectivity set of s references the newly created connectivity set with field access f.
- The UNIFY operation is implemented using UNION and FIND operations as shown in Figure 3.15.
- UNIFY$(\langle r_1, \ldots, r_n \rangle, \langle s_1, \ldots, s_n \rangle))$ performs UNIFY(r_i, s_i) for $1 \leq i \leq n$.

There are two differences between the rules in Figure 3.18 and those in Figure 3.7. These two differences aim at achieving the same goal — to make the analysis field sensitive for runnable objects or objects containing runnable fields:

1. **Handling of field accesses**

   Unlike the simplified version of connectivity analysis, for field accesses, if the field is declared in a runnable class or the declaring class has a runnable field, the analysis will not merge the container and containee connectivity sets together. Consider the field access `x.f = y`: if `x` is of runnable type, $FieldAccess(\text{FIND}(\texttt{x}), \texttt{f})$ is unified

with FIND(y) instead of unifying FIND(x) and FIND(y). Here we are trading off between cost and precision. We are willing to be field sensitive in these cases because the cost is low and it captures a common pattern of Java programs. It is natural for Java programs to allocate data objects which are usually accessed within the running thread only. This situation is illustrated with the example code of Figure 3.13. In the example, `data` is accessed intensively inside `run()` only,

```
class MyThread implements Runnable {
  Data data;
  Input input;
  MyThread(Input i) {
    input = i;
  }
  public void run() {
    // work on data
  }
}
```

Figure 3.13: Importance of being field sensitive for fields of Runnable objects

while `input` refers to an escaping object. If the analysis is not field sensitive, for the method `run()`, `this`, `this.data` and `data.input` will be merged together, making all the variables marked as escaping.

Moreover, since runnable objects are assumed to escape often, distinguishing runnable fields from non-runnable fields saves the analysis from marking too many objects as escaping. This is illustrated by example 3.14. In the example, `myWorker` references the thread that accesses a `Data` object. If the analysis is field insensitive, the connectivity set of, say `x.data` has to be merged with `x.myWorker`, forcing `x.data` to be escaping, although in reality, `x.data` is *not* accessed outside the execution of the thread referenced by `myWorker`.

2. **Use of the UNIFY operation rather than the UNION operation.**

   Since we want to be field-sensitive for some objects, we cannot use UNION operations to merge results. Consider the example shown in Figure 3.16. As we do not

```
class Data {
  Runnable myWorker;
  OtherData data;
}
```

Figure 3.14: Importance of being field sensitive when a class has both Runnable and non-Runnable fields

```
UNIFY(xs, ys)
begin
 if FIND(xs) = FIND(ESCAPE) and FIND(ys) ≠ FIND(ESCAPE) then
  UNIFY(ys, ESCAPE)
 end if
 if FIND(ys) = FIND(ESCAPE) and FIND(xs) ≠ FIND(ESCAPE) then
  UNIFY(xs, ESCAPE)
 end if
 UNION(xs, ys)
 foreach f ∈ SetOfFieldAccesses(xs) ∪ SetOfFieldAccesses(ys)
  xfs = FieldAccess(xs, f)
  yfs = FieldAccess(ys, f)
  if xfs and yfs not unified before then
   UNIFY(xfs, yfs)
  end if
 end foreach
end
```

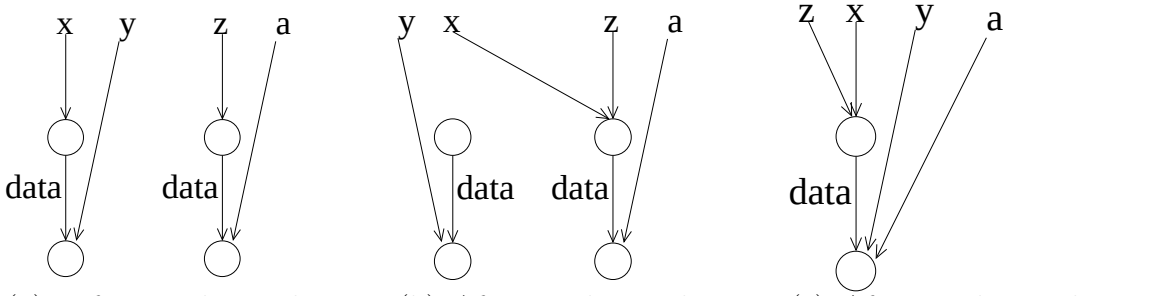
Figure 3.15: Implementing UNIFY operation using FIND and UNION.

restrict the order of analyzing statements, let us consider the order of analyzing lines 7, 9, and then 8. Figure 3.17 shows the result before and after analyzing line 8: as shown in the figure, if UNION operation is used, the analysis cannot correctly identify that `a` and `y` reference the same connectivity set. If a UNIFY operation is used instead, the unification is performed correctly and it computes the information that `a` and `y` reference the same connectivity set.

When method calls are analyzed, instead of using the UNION operations to incorporate callee results to the caller, UNIFY operations are used.

```
1  class MyRunnable implements Runnable{
2    Data data;
3  }
4
5  public void main(String[] args) {
6    MyRunnable x = new MyRunnable();
7    Data y = x.data;
8    MyRunnable z = x;
9    Data a = z.data;
10 }
```

Figure 3.16: Importance of being field sensitive when a class has both Runnable and non-Runnable fields



(a) Before analyzing line 8 (b) After analyzing line 8 using the UNION operation (c) After analyzing line 8 using the UNIFY operation

Figure 3.17: Before and after analyzing line 8

$x = y$	UNIFY(FIND(x), FIND(y))
$x[\,] = y$ $y = x[\,]$	UNIFY(FIND(x), FIND(y))

$x.f = y$
$y = x.f$
	$c = DeclaringClass(f)$
	`if` $IsRunnableClass(c)$ `or` $HasRunnableField(c)$ `then`
	UNIFY($FieldAccess$(FIND(x), f), FIND(y))
	`else`
	UNIFY(FIND(x), FIND(y))
	`endif`

$y =$ `getstatic` y `putstatic` f y y`.start()`	UNIFY(FIND(y), FIND(`ESCAPE`))

$a_0{=}a_1.n(a_2, \ldots, a_k)$	$sc = \langle$FIND$(a_1), \ldots,$ FIND$(a_k),$ FIND$(a_0),$ FIND$(Exception_m)\rangle$
	`foreach` $f(p_1, \ldots, p_k) \in ResolvedMethods(a_1, n)$
	$fc = \langle$FIND$(p_1), \ldots,$ FIND$(p_k),$
	FIND$(Return_f),$ FIND$(Exception_f)\rangle$
	`if` $InSameSCC(m, f)$ `then`
	UNIFY(sc, fc)
	`else`
	UNIFY$(sc, clone(fc))$
	`endif`
	`end foreach`

`return` x	UNIFY(FIND(x), FIND($Return_m$))
`throw` x	UNIFY(FIND(x), FIND($Exception_m$))

Figure 3.18: Rules for analyzing a method m

3.3.4.4 Topdown Phase

The topdown phase of the full version of connectivity analysis is also similar to that of the simplified version. Again, for each method m in the SCC, the context connectivity information of m is used as the initial connectivity information to do the analysis. The information was computed when the callers of m were processed, except for the main method and the `run` methods for `Thread` classes:

- **Context Connectivity of the main method**. The way to determine the context connectivity information of the main method is the same as that of the simplified version.
- **Context Connectivity of `run()` methods for `Thread` classes**. In the full version of connectivity analysis, we do not simply assume `this` of `run()` to be escaping as we did in simplified connectivity analysis. Instead, we construct a more precise context connectivity information where each field of the thread object is checked if no multiple threads read or write to them. This is done by checking the following conditions:
  - The created thread object does not connect to the formal parameters of the thread creating method. In the example, we check whether:
    - `t1` is connected to `args`; and
    - `t2` is connected to `args`.

    This ensures the thread object is not method escaping with respect to the thread creating method.
  - The created thread object O, once started (i.e. O`.start()` is executed), is only used in a `join` call (i.e. O`.join()`). This ensures that the thread object is not accessed inside the method calling the constructor(s) of the thread objects and also not passed to another created thread.

If the check is true, the analysis constructs the context connectivity information for `run()` using the connectivity information of the thread constructors. The algorithm is shown in Figure 3.19.

$runContext = \bot$
`foreach constructor` c `=<init>(`$p_1, \ldots, p_n$`) of` C
  $\langle cs_{p_1}, \ldots, cs_{p_n}, \bot, cs_{Exception_c} \rangle = clone(ConnectivityInfo(c))$
  `for` i `= 2 to` n
    UNIFY(p_i, FIND(ESCAPE))
  `end for`
  UNIFY($runContext, \langle cs_{p_1} \rangle$)
`end foreach`
$ContextConnectivityInfo(C.\texttt{run}()) = runContext$

Figure 3.19: Computing the context connectivity information of `run()` for thread class C using connectivity information of constructors of C.

Conceptually, we use the connectivity information of constructors of the thread class as the context connectivity information of the `run` method. To avoid the analysis of the `run` method changing the analysis results of the connectivity information of the constructors, clonings are performed. The analysis assumes the objects passed to the constructors are potentially shared among different thread instances, so it unifies the argument connectivity sets with the escaping connectivity set.

Similar to the simplified connectivity analysis, the context connectivity information is used to perform the intraprocedural analysis described in Section 3.3.4.3. After the intraprocedural analysis, we propagate the context connectivity information further to methods invoked by m. The analysis visits each method invocation instructions of m following the rule shown in Figure 3.20.

Consider again the example shown in Figure 3.9. Because of the way threads are created and started, thread shared data and thread local data would usually be connected in the previous algorithm. In the example shown in Figure 3.9, in `run()`, `this.shared`

$a_0{=}a_1.n(a_2,\ldots,a_k)$	$sc = \langle \textsc{Find}(a_1),\ldots,\textsc{Find}(a_k),\textsc{Find}(a_0),\textsc{Find}(Exception_m)\rangle$
	`foreach` $f(p_1\ldots,p_k) \in ResolvedMethods(a_1,n)$
	$fc = \langle \textsc{Find}(p_1),\ldots,\textsc{Find}(p_k),$
	$\textsc{Find}(Return_f),\textsc{Find}(Exception_f)\rangle$
	`if` $InSameSCC(m,f)$ `then`
	$\textsc{Unify}(sc,fc)$
	`else`
	$\textsc{Unify}(clone(sc),fc)$
	`endif`
	`end foreach`

Figure 3.20: Rule for analyzing call instruction of a method m

and `this.data` are connected because both of them are connected to `this`. Moreover, `this` in `run()` should be considered escaping because:

- `this` in `run()` and `t1` in `main()` may reference the same object; and
- `run()` and `main()` are executed by different threads.

When the program shown in Figure 3.9 is analyzed, the connectivity information of the constructor of `MyThread` is used. The connectivity information is $\langle c_1, c_2, \bot, \bot\rangle$ where:

- $\textsc{Find}(\texttt{this}) = c_1$ and $\textsc{Find}(\texttt{shared}) = c_2$; and
- $FieldAccess(c_1, \texttt{MyThread.shared}) = c_2$ and $FieldAccess(c_1, \texttt{MyThread.data}) = c_3$.

As `shared` is the argument of the constructor, the analysis unifies c_2 with $\textsc{Find}(\texttt{ESCAPE})$, making $FieldAccess(c_1, \texttt{MyThread.shared})$ equals $\textsc{Find}(\texttt{ESCAPE})$. At the end the context connectivity information for `run()` is $\langle c_1\rangle$ Since there may be more than one constructors available for the thread class in general , the analysis merge all of them together and use the merged information.

3.3.5 Extended Version of Connectivity Analysis — keeping track of thread allocation sites

The full version of connectivity analysis can be strengthen in a way similar to what is done in [Ruf00]. An extra phase is done before the bottom-up phase. Identical to Phase 1 in [Ruf00], it computes for each method m, $InvokingThreads(m)$, the set of thread allocation sites in the program of which thread are spawned to invoke the method directly or indirectly.

We need to annotate each connectivity set c's attributes:

- $c.accessed$ — record whether a load/store operation has been done for the objects represented by c.
- $c.accessThreads$ — record the set of thread allocation sites that represents the threads that perform load/store on the objects represented by c.

Figure 3.21 shows the new rules for the analysis extension. It is analogous to those presented in [Ruf00]. Similar to [Ruf00], when $z = \textsc{Unify}(x, y)$ is performed where either x or y are connected to `ESCAPE` and the other has the *accessed* attribute being *true*, the thread allocation sites associated with the current method are added to $z.accessThreads$.

Using this extended analysis, for a store instruction like `x.f = y`, a fence can be avoided if $\textsc{Find}(\texttt{x}).accessThreads$ is empty, or it contains a single thread allocation site which is executed at most once even if `x` is connected to `ESCAPE`.

3.3.6 Some Properties of Connectivity Analysis

In this section we discuss some properties of connectivity analysis.

With a trade-off between analysis cost and precision, escape analyses can be categorized as:

$x[\,] = y$ $y = x[\,]$	FIND(x).$accessed = true$ `if` FIND(x) = FIND(`ESCAPE`) FIND(x).$accessThreads \cup = InvokingThreads(m)$ `end if` UNIFY(FIND(x), FIND(y))

$x.f = y$
$y = x.f$
	FIND(x).$accessed = true$
	`if` FIND(x) = FIND(`ESCAPE`)
	  FIND(x).$accessThreads \cup = InvokingThreads(m)$
	`end if`
	$c = DeclaringClass(f)$
	`if` $IsRunnableClass(c)$ `or` $HasRunnableField(c)$ `then`
	  UNIFY($FieldAccess$(FIND(x), f), FIND(y))
	`else`
	  UNIFY(FIND(x), FIND(y))
	`endif`

Figure 3.21: Rules for analyzing a method m

- Field sensitive *vs.* field insensitive — do we distinguish different fields of objects? This can be illustrated by an example shown in Figure 3.22. A field sensitive

```
void main(String args) {
  MyThread t = new MyThread();
  Data esc = new Data();
  Data local = new Data();
  t.data = esc;
  Object o = new Object();
  o.f = local;
  o.g = esc;
}
```

Figure 3.22: An Example Illustrating Field Sensitivity of Escape Analysis

escape analysis says that `o.f` is not escaping while `o.g` is. A field insensitive escape analysis can only say that a field of `o` is escaping. Therefore, it cannot distinguish whether `o.f` or `o.g` is the escaping object, and it has to conservatively assumes that both `o.f` and `o.g` are escaping.

- Flow sensitive *vs.* flow insensitive — are we interested in the escaping property for different program points? This can be illustrated by an example shown in Figure 3.23. A flow sensitive escape analysis can tell that `esc` is not escaping

```
1  void main(String args) {
2    MyThread t = new MyThread();
3    Data esc = new Data();
4    Data local = new Data();
5    t.data = esc;
6  }
```

Figure 3.23: An Example Illustrating Flow Sensitivity of Escape Analysis

  before the statement at line 5. A flow insensitive one, however, does not keep track of program point information. Therefore, it can only say that `esc` is escaping at some program point.

Using this terminology, we can say that:

- all the connectivity analyses are flow-insensitive;
- a simplified version of connectivity escape analysis is field-insensitive; and
- a full version and the extended version of connectivity escape analysis is field-insensitive. for most objects but field-sensitive for runnable objects and objects having runnable fields

3.4 Uses of Thread Escape Analysis

The result of thread escape analysis can be used in at least two areas — fence insertion and synchronization removal.

3.4.1 Fence Insertion

In the Pensieve system, thread escape analysis is used to reduce the number of fences inserted to the program. Thread escape analysis computes *may* escape information. If an object O is not marked as escaping by the analysis, O *must not* be accessible by another thread. Because of this, loads and stores to fields of a not marked object can never be observed by another thread. So, no orderings need to be enforced for such memory accesses. As described in Section 2.2, in the Pensieve system setting computing conservative approximate delay information, we conservatively assume that each pair of references to variables reaching escaping objects are in a minimal mixed cycle. This assumption ignores the overall parallel structure of the application and the order of access to variables during execution, but despite its simplicity our analysis produces good results as discussed in the next section.

3.4.2 Synchronization Removal

Another use of thread escape analysis is to remove redundant synchronization. Again, we are using the negative result of escape analysis. For any object O not marked as escaping, O *cannot* be accessible by another thread. Therefore, lock and/or unlock operations done to O are done in the sequence lock, ..., unlock,...,lock,...,unlock and they are done by a single thread only. Because of this, the lock and unlock operations are unnecessary as they will always be executed sequentially by a single thread. Note that to implement the semantics of the `monitorenter` and `monitorexit` bytecode, the compiler should still need to put fences in place of the lock and unlock operations.

3.5 Adapting Bogda's and Ruf's Escape Analyses

In this thesis we will compare our connectivity analysis with two efficient escape analysis algorithms. They are Bogda's analysis [BH99] and Ruf's analysis [Ruf00]. In this section, we outline their algorithms and describe how to adapt them for fence insertion.

3.5.1 Similarity of the Two Analyses

Both analyses were originally developed to remove synchronization by performing method specialization. Figure 3.24 shows an example illustrating how specialization is done to remove synchronization. In the example, the synchronization block in `g` is removed by

```
void main(String[] args) {
  // assume ESC escaping
  // loc not escaping
  f(ESC);
  f(loc);
}
void f(Object o) {
  g(o);
}
void g(Object o) {
  synchronized (o) {
  }
}
```

Before synchronization removal

```
void main(String[] args) {
  // assume ESC escaping
  // loc not escaping
  f(ESC);
  fdash(loc);
}
void f(Object o) {
  g(o);
}
void fdash(Object o) {
  gdash(o);
}
void g(Object o) {
  synchronized (o) {
  }
}
void gdash(Object o) {
  // synchronized block removed
}
```

After synchronization removal

Figure 3.24: Removing synchronizations using method specialization

introducing new methods `fdash` and `gdash` that are nearly identical to `f` and `g` except that the argument is assumed to be non-escaping. In the specialized method if the

synchronization block performed on objects assumed not escaping, it is removed[4]. For those objects which are found to be non-escaping, the specialized method is called instead. In the example, `fdash` is invoked on the non-escaping object. Eventually `gdash` is called on the non-escaping object without performing synchronization.

We can see performing specialization causes code expansion. While this may be fine for synchronization removal, as the number of synchronizations in a program is small, this is not feasible for the Pensieve system, which potentially has a synchronization for each load and store operation.

Because of the difference in the application of escape analysis we need to adapt both analyses for fence insertion without method specialization. Since specialization is not done only one version of code is generated for every method. The generated code is called at all call sites, so it has to be conservative enough that it can be used for all calling contexts.

3.5.2 Outline of Bogda's Analysis

Bogda's analysis is a two phase escape analysis:

1. The first phase determines objects that are stack-escaping. An object is called stack-escaping if its reference value can be stored into a field of another object. For example, an instruction `x.f = y` causes `y` to be stack-escaping because its reference value is stored into `x.f`. The rules for analyzing a method in phase 1 is shown in Figure 3.25. The idea is quite simple:

    - Objects referenced by a reference variable are assumed to be stack escaping (s-escaping) if the value of the variable:
        - is stored into or received from an array or a field of an object;

[4]For correct Java semantics, a fence is needed just before entering the synchronized block and another fence is needed just after the synchronized block. Therefore, even if lock/unlock operations are removed, the fences should *not* be removed

- is stored into or received from a static field;
- is used to start a thread; or
- is thrown as an exception.

- For assignment statements, "$x = y$", the analysis assumes both variables reference to the same set of objects by performing a UNION operation. If either x or y was assumed to be s-escaping, the combined set of objects are assumed to be s-escaping.
- For return statements "`return` x", the analysis handles the statement as if an artificial return variable is being assigned from x.
- For method calls "$a_0 = a_1.n(a_2, \ldots, a_k)$", the stack-escape information is imported from the callee. Since the return value of a call may be the value passed as an argument, the analysis check this case. If the return value is the same as argument a_i, the analysis considers the method call as if an assignment statement "$a_0 = a_i$" is also done.

2. The second phase uses the result of phase 1 to determine objects that are *field-escaping*. An object is field-escaping if its reference can be stored into a field of a *stack-escaping* object. For example, two instructions `x.f = y` and `y.g = z` cause `z` to be field-escaping because its value is stored into `y.f`, where `y` is stack-escaping. Note that `y` is not field-escaping if we only consider these two instructions. This phase, in additional to keeping track of alias set of variables, also considers the alias set of fields of objects. To reduce the analysis cost, only one level of field reference is considered, so objects reachable by two or more field references are assumed to be escaping. The rules for second phase are shown in Figure 3.27 and Figure 3.28:

   - Objects referenced by a reference variable are assumed to be field escaping (f-escape) if the value of the variable:
     - is stored into or received from a static field;
     - is used to start a thread; or

$x = y$	Let this be $Unify_{\text{Phase 1}}(x, y)$ $xs = \text{FIND}(x)$ $ys = \text{FIND}(y)$ UNION(xs, ys) $xys = \text{FIND}(x)$ $xys.sescape = xs.sescape$ `or` $ys.sescape$
$x[\,] = y$ $y = x[\,]$ $x.f = y$ $y = x.f$	$\text{FIND}(y).seacape =$ `true`
$y =$ `getstatic` y `putstatic` f y y`.start()` `throw` y	$\text{FIND}(y).seacape =$ `true`

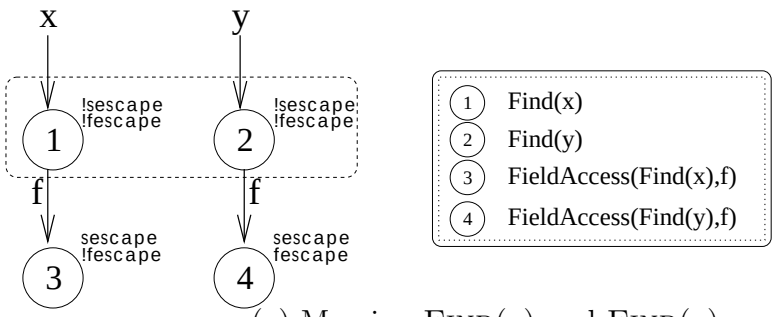
$a_0{=}a_1.n(a_2, \ldots, a_k)$	`foreach` $f(p_1, \ldots, p_k) \in ResolvedMethods(a_1, n)$
	$pa = \langle Return_f, p_1, \ldots, p_k \rangle$
	`for` i `= 0 to` k
	$\text{FIND}(a_i).sescape = \text{FIND}(a_i).sescape$ `or` $\text{FIND}(pa_i).sescape$
	`end for`
	`for` i `= 1 to` k
	`if` $\text{FIND}(p_i) = \text{FIND}(Return_f)$
	$Unify_{\text{Phase 1}}(a_i, a_0)$
	`end if`
	`end for`
	`end foreach`

`return` x	$Unify_{\text{Phase 1}}(x, Return_m)$

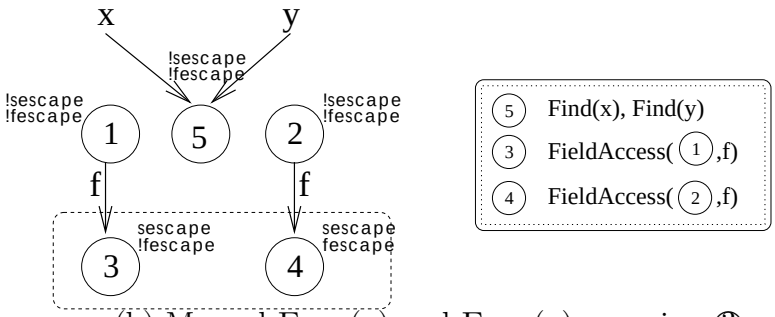
Figure 3.25: Rules for analyzing a method m to compute s-escape information

– is thrown as an exception.

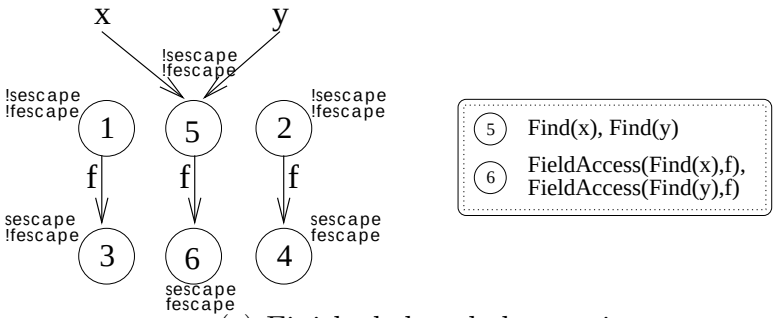
- For assignment statement, "$x{=}y$", the analysis first merges the f-escape information in a way similar to phase 1. This ensures the merged alias set inherits the alias f-escaping set if either one of the original alias set is f-escaping. In this phase, the analysis keeps track of one level of field references, so when merging two alias sets together, we may need to recursively merge alias sets referenced by the original alias sets. This can be illustrated by Figure 3.26. In Figure 3.26(a), two alias sets are merged. ①. Following the rule, the f-escape information is being merged first. The result is shown in Figure 3.26(b). Now



(a) Merging FIND(x) and FIND(y)



(b) Merged FIND(x) and FIND(y), merging ③ and ④.



(c) Finished the whole merging process

Figure 3.26: Merging two alias sets in phase 2

that the stack-escaping properties of ① and ② are both false, meaning that both ① and ② may point to another alias set mark as not f-escaping. This is indeed the case for ①. Therefore, the analysis continues the merging process for ③ and ④. Since the f-escape property of ④ is true, the merged alias set inherits this property, making the merged alias set f-escaping. Since ③ and ④ are stack-escaping, the analysis does not continue the merging process further.

- For field access statements, "$x.f = y$" or $y = x.f$, there are two cases:
  - If the alias set for x is stack-escaping, the analysis simply marks the alias set for y f-escaping, following the definition of f-escaping.
  - If the alias set for x is not stack-escaping, then the alias set for $x.f$ (i.e $FieldAccess(\textsc{Find}(x), f)$) may or may not be f-escaping, depending on the f-escaping property of the alias set of y. Therefore, the analysis merges $FieldAccess(\textsc{Find}(x), f)$ with $\textsc{Find}(y)$.
- Array access statements are processed as if statements accessing an artificial field $\$ELT$.
- For return statements "`return` x", the analysis handles the statement as if an artificial return variable is being assigned from x.
- For method calls "$a_0 = a_1.n(a_2, \ldots, a_k)$", there are four loops shown in Figure 3.28. The first loop imports f-escape information from the callee. The second loop merges alias sets of return value and arguments if the alias information of the callee says the method called may return the argument passed. The third loop merges the alias sets of $a_i.f$ and a_j, if the alias information of the callee says the method call may make $a_i.f$ and a_j point to the same object. It is handled as if processing the statement $a_i.f = a_j$. The fourth loop handles the case that the alias information says the method called may make $a_i.f$ and $a_j.h$ pointing to the same object. There are two cases:

- Both the alias sets of a_i and a_j are not stack-escaping. Since the analysis keeps track of one level of field reference, it merges the alias set of $a_i.f$ and $a_j.h$ following the information from the callee.
- One of the alias sets, say a_j, is stack-escaping. This means the alias set of a_j is pointed to by another alias set of field reference. Since the analysis keeps track of one level of field reference, it does *not* keep track of alias information of $a_j.h$ because $a_j.h$ is reachable by *two* levels of field references. It simply assumes alias sets of $a_j.h$ are f-escaping. Since the alias information of callee says $a_i.f$ and $a_j.h$ may point to the same object, the analysis marks the alias set of $a_i.f$ f-escaping.

Bogda's analysis marks objects to be escaping if they are field-escaping. We can see Bogda's analysis is an 1-level escaping analysis — objects reachable by another object via more than one field references are assumed to be escaping. Unlike the connectivity analysis, the Bogda's analysis performs fixpoint iteration for recursive functions.

3.5.3 Adapting Bogda's Analysis

The analysis was formulated as constraints in [BH99]. The authors did not describe how the analysis is performed, but only say the constraint problem converges to a fixpoint. Because of this, we choose to implement the analysis as efficiently as possible: we traverse the call graph in a way similar to our connectivity analysis and we use the union-find data structure to implement the analysis.

As described in Section 3.5.1, synchronization is removed by method specialization. Bogda does this by finding objects not reachable from any formal parameter. Such objects found to be not escaping are candidates for optimization and specialization by the optimizer. Because of this technique of finding candidate objects, the context of a method is not relevant. In the Pensieve system, we are interested in the context of a

$x = y$	This is handled by $Unify_{\text{Phase 2}}(\text{FIND}(x), \text{FIND}(y))$ where $Unify_{\text{Phase 2}}(xs, ys)$ is defined as follows:
	
	UNION(*xs, ys*)
	$xys = \text{FIND}(xs)$
	$xys.fescape = xs.fescape$ `or` $ys.fescape$
	`if` $xs.sescape$ =`false and` $ys.sescape$ =`false then`
	`for each field` f `of` xs `and` ys
	$Unify_{\texttt{Phase 2}}(FieldAccess(xs, f), FieldAccess(ys, f))$
	`end for`
	`end if`

$x.f = y$
$y = x.f$
	This is handled by $UnifyFieldAccess(x, f, y)$ where $UnifyFieldAccess(x, f, y)$ is defined as follows:
	$xs = \text{FIND}(x)$
	$ys = \text{FIND}(y)$
	`if` $xs.sescape =$ `false then`
	$Unify_{\texttt{Phase 2}}(FieldAccess(xs, f), ys)$
	`else`
	$ys.fescape =$ `false`
	`end if`

$x[\,] = y$ $y = x[\,]$	$UnifyFieldAccess(x, \$ELT, y)$
y = `getstatic` f `putstatic` f y y.`start()` `throw` y	$\text{FIND}(y).feacape =$ `true`
a_0=$a_1.n(a_2, \ldots, a_k)$	See Figure 3.28
`return` x	$Unify_{\texttt{Phase 2}}(x, Return_m)$

Figure 3.27: Rules for analyzing a method m to compute f-escape information

a_0=$a_1.n(a_2, \ldots, a_k)$	foreach $f(p_1, \ldots, p_k) \in ResolvedMethods(a_1, n)$
	$pa = \langle Return_f, p_1, \ldots, p_k \rangle$
	for i = 0 to k
	FIND(a_i).$fescape$ = FIND(a_i).$fescape$ or FIND(pa_i).$fescape$
	if FIND(a_i).$sescape$ =false then
	foreach field f of pa_i
	if $FieldAccess$(FIND(pa_i), f).$fescape$ =true then
	$FieldAccess$(FIND(a_i), f).$fescape$ =true
	end if
	end foreach
	end if
	end for
	for i = 1 to k
	if FIND(p_i) = FIND($Return_f$)
	$Unify_{\text{Phase 2}}(a_i, a_0)$
	end if
	end for
	for $i, j \in [0, k]$
	foreach f such that $FieldAccess(pa_i, f)$ = FIND(pa_j)
	$UnifyFieldAccess(a_i, f, a_j)$
	end foreach
	end for
	for $i, j \in [0, k]$
	foreach f, h such that
	$FieldAccess(pa_i, f) = FieldAccess(pa_j, h)$
	if FIND(a_i).$sescape$ = false
	and FIND(a_j).$sescape$ = false then
	$Unify_{\text{Phase 2}}(FieldAccess(a_i, f), FieldAccess(a_j, h))$
	else if FIND(a_i).$sescape$ = false
	$FieldAccess(a_i, f).fescape$ =true
	end if
	end foreach
	end for
	end foreach

Figure 3.28: The Rule for analyzing call statement of a method m to compute f-escape information

method when inserting fences because the same code is executed no matter where the method is invoked.

In order to compute the context escape information, we adapt Bogda's analysis to propagate context information as we do for connectivity analysis. This is done by traversing the SCC graph of the call graph in topological order and propagate information from callers to callees. To do this, two additional phases are needed. The first propagates stack-escape information and the second propagates field-escaping information.

After performing these phases, we can perform the reconstruction phase to recover escape information for local variables in way similar to what is described in Section 3.3.3.6.

3.5.4 Outline of Ruf Analysis

Like connectivity analysis, Ruf's analysis does not need a fixpoint computation for recursive functions. It is a three phase escape analysis:

1. The first phase computes for each method the set of thread allocation sites in the program of which thread are spawned to invoke the method directly or indirectly. The analysis also records whether a thread allocation site is executed multiple times or not.

2. The second phase computes

   (a) for each method the escape information of its formal parameters, return value and exception value. The escape information is represented as an annotated alias set. The alias set represents objects at runtime. Each set keeps track of:

   - whether the set is synchronized on;
   - whether the set is escaping; and
   - the thread allocation sites representing the threads that synchronize on the objects represented by this alias set.

(b) for each static field (and objects reaching from that field), the thread allocation sites representing the threads that synchronize on the field.

The rules are shown in Figure 3.31. They make use of the UNIFY operation shown in Figure 3.15. When two alias sets are unified, all alias sets reachable are unified as well. The idea of the analysis is to perform unification operations according to the semantics of the statements:

- For statements, "$x = y$", the analysis unifies the alias sets of x and y.
- For field access statements, "$x.f = y$" and "$y = x.f$", the analysis unifies the alias sets of $x.f$ and y.
- For array access statements, the analysis considers the statement as if statements access the artificial field $\$ELT$.
- For static field access statements, "$y =$ `getstatic` f" and "`putstatic` f y", the analysis unifies the alias set of y and the alias set for the static field. The *isGlobal* attribute of the alias set for static fields is assumed to be `true`, i.e. FIND(f).*isGlobal* = `true`. Therefore, after the unification operation, all alias sets reachable from the alias set of y have the *isGlobal* begin `true`.
- For return statements, the analysis handles the statements as if assignment statements to an artificial return variable.
- For throw statements, the analysis handles the statements as an assignment statements to an artificial exception variable if the exception may be caught outside the method being analyzed. If the analysis can identify exception handlers within the same method corresponding to the exception being thrown, it will perform unification accordingly. This is illustrated by the example shown in Figure 3.29. Since the exception `te` is caught at line 4, the alias sets of `te` and `ce` are being unified. This is not shown in Figure 3.31 for brevity.
- For "`monitorEnter` y" and "`monitorExit` y statements, the analysis marks the alias set of as y being synchronized. If the alias set of y has been marked

```
1  try {
2    throw te;
3  } catch (Exception ce) {
4    ...
5  }
```

Figure 3.29: An example illustrating processing of `throw` and `catch` statements

as global ($isGlobal$ = `true`), then it has been found to be shared with other threads. This implies that a shared object represented by the alias set is synchronized. Therefore, the analysis adds to the set of threads that synchronized on the alias set of y, FIND(y).$syncThreads$,the threads that invokes the method being analyzed ($InvokingThreads(m)$). A symmetric case is possible — an object synchronized somewhere is marked as escaping. An example is shown in Figure 3.30. When line 3 (`monitorEnter o`) and line 4 (`monitorExit o`) are processed, it is not yet known if the variable `o` will point to an escaping object, so the analysis only marks the alias set of `o` being synchronized, i.e. FIND(`o`).$synchronized$ = `true`. Later when line 5 is processed, FIND(`o`) is unified with FIND(`ESC`). This makes the alias set of `o` escaping. The analysis recognize this fact and will add to to the set of threads that synchronized on the alias set of `o`, FIND(`o`).$syncThreads$,the threads that invokes `foo`, (`T`).

- As with the connectivity analysis, for method calls, "a_0=$a_1.n(a_2, \ldots, a_k)$", there are two cases:

```
1  static Object ESC;
2  // invoked by thread T
3  void foo(Object o) {
4    synchronized (o) {
5    }
6    ESC = o;
7  }
```

Figure 3.30: An synchronized object referenced by `o` published to a static field `ESC`

(a) If the resolved method is in the same SCC as the method being analyzed, the alias summary of the resolved method is used without cloning. For each actual argument a, its alias set FIND(a) is unified with the alias set of the corresponding formal parameter p (i.e UNIFY(FIND(a), FIND(p)) is performed). Similarly, unification is performed for return and exception values. This corresponds to the UNIFY operations shown in the rules.

(b) If the resolved method is not in the same SCC as the method being analyzed, the alias summary of the resolved method is cloned and the UNIFY operation is performed as the previous case.

$x = y$	UNIFY(FIND(x), FIND(y))
$x.f = y$ $y = x.f$	This is handled by $UnifyFieldAccess(x, f, y)$ where $UnifyFieldAccess(x, f, y)$ is defined as follows: UNIFY($FieldAccess$(FIND(x), f), FIND(y))
$x[\,] = y$ $y = x[\,]$	$UnifyFieldAccess(x, \$ELT, y)$
`monitorEnter` y `monitorExit` y	FIND(y).$synchronized = true$ `if` FIND(y).$isGlobal = true$ FIND(y).$syncThreads \cup = InvokingThreads(m)$ `end if`
$y =$ `getstatic` f `putstatic` f y	UNIFY(y, FIND(f))

$a_0 = a_1.n(a_2, \ldots, a_k)$	$sc = \langle$FIND(a_1), …, FIND(a_k), FIND(a_0), FIND($Exception_m$)$\rangle$
	`foreach` $f(p_1 \ldots, p_k) \in ResolvedMethods(a_1, n)$
	$fc = \langle$FIND(p_1), …, FIND(p_k),
	FIND($Return_f$), FIND($Exception_f$)$\rangle$
	`if` $InSameSCC(m, f)$ `then`
	UNIFY(sc, fc)
	`else`
	UNIFY($sc, clone(fc)$)
	`endif`
	`end foreach`

`return` x	UNIFY(x, $Return_m$)
`throw` x	UNIFY(x, $Exception_m$)

Figure 3.31: Rules for analyzing a method in Ruf's analysis

3. Remove the synchronization by specialization. This is done in a top-down order with respect to the SCC call graph similar to the top-down phase of the connectivity analysis. When a method is visited, the context alias information is used as the initial analysis result. The analysis then computes alias information for local variables following the rules described in Figure 3.31 with the exception of method call handling. In this phase, whether or not the called method is in the same SCC as the method being analyzed, the summary information is cloned before performing the UNIFY operation. After computing the information for local variables there are two possible things to do for code generation:

   - If the method has `monitorEnter` o and `monitorExit` o statements and the alias set for o does not have multiple threads in the alias set's $syncThreads$ attribute. This implies the objects pointed to by o do not have multiple threads synchronizing on, so the synchronization operations can be replaced by fence insertions.
   - If the method has method calls, the analysis uses the alias information of actual arguments, return values and exception values to compute the context alias information for the method. If the computed context alias information is different from the alias information for the method computed in the bottom-up phase, a specialized version of the method is invoked instead. For each method, the analysis maintains context alias information associated to the method, each context corresponds to a specialized version of method. This is illustrated by a program shown Figure 3.32 The information for `bar` computed in the bottom-phase says that:
     - FIND(`o`).$isGlobal$ = `false`;
     - FIND(`o`).$synchronized$ = `true`; and
     - FIND(`o`).$syncThreads$ = {}.

     In the topdown phase, when `foo` is processed, two calls to `bar` are encountered. The local variables' alias information is reconstructed. Using the reconstructed

```
static Object ESC;
// invoked by thread T1, T2
static void foo() {
  bar(new Object o);
  bar(ESC);
}
static void bar(Object o) {
  synchronized(o) {
  }
}
```

Figure 3.32: Removing synchronization in the top-down pass

information, the context alias information for the calls are computed. For the first call, the information is the same as that computed in the bottom-up phase, so the call invokes a non-specialized version of `bar`. Since the synchronization is performed for `o` where FIND(`o`).*syncThreads* = {}, the synchronization operations are removed in the non-specialized version of `bar`. For the second call, the context information says that:

- FIND(`o`).*isGlobal* = `true`;
- FIND(`o`).*synchronized* = `true`; and
- FIND(`o`).$syncThreads = \{T1, T2\}$.

Since the information is different from that computed in the bottom-up phase, a specialized version of `bar` is invoked. In this specialized version of `bar`, the synchronization is performed for `o` where FIND(`o`).$syncThreads = \{T1, T2\}$, so the synchronization operations cannot be removed.

3.5.5 Adapting Ruf's Analysis

We can see Ruf's analysis can be considered as an extension of alias analysis style escape analysis by keeping track of threads synchronizing on objects. In this study, we adapt the analyses directly by keeping track of threads *accessing* objects rather than *synchronizing* on them. We have two implementations in our study:

1. The first implementation does *not* keep track of the threads that access objects. An object is consider escaping if the object is reachable from static fields or thread objects.

2. The second implementation *does* keep track of the threads that access objects. An object is consider escaping if it is escaping in the first implementation's sense and if it is being accessed by multiple threads.

To avoid performing specialization, we modify phase 3 of the analysis. Instead of performing specialization when propagating escaping information from callers to callees, we only propagate the information from callers to callees. Given a method, we merge all of the escape information from its callers. This is the context escape information for the method. Note that this propagating process involves fixpoint computations inside SCCs. For example, for a program having method `f` calling `g` which calls `f` recursively, the context information propagation is done by performing an iterative process for `f` and `g` until no context information is changed.

Again, after performing all phases, we can reconstruct escape information for local variables by using the context escape information computed in modified phase 3 and escape information computed in phase 2.

3.6 Qualitative Comparison between the Analyses

In this section, we attempt to study the difference between the analyses in a qualitative way. We will compare the differences in their lattice, precision and complexity. A quantitative discussion is presented in Chapter 4.

3.6.1 Precision

In this section, we study the precision of the analyses. We are interested in identifying cases where one analysis is more precise than others.

3.6.1.1 Cases Where Connectivity Analysis is More Precise

An example program that demonstrates the precision of connectivity analysis is shown in Figure 3.33. Connectivity analysis can identify `this.data` is only accessed within the thread that executes `run()` and claims that `this.data` is not escaping. Ruf's and Bogda's analysis are conservative for runnable objects — all objects reachable from runnable objects are considered escaping, so they consider `this.data` to be a shared access[5].

```
void main(String[] args) {
  new MyThread().start(); // creates thread T1
  new MyThread().start(); // creates thread T2
}
class MyThread {
  Object data;
  // executed by threads T1, T2
  public void run() {
    this.data = ...;
    // is this.data a shared access?
  }
}
```

Figure 3.33: A program where connectivity analysis is more precise

3.6.1.2 Cases Where Bogda's Analysis is More Precise

An example program that demonstrates the precision of Bogda's analysis is shown in Figure 3.34. Bogda's analysis is more precise than other two in this case because it com-

[5]For adapted Ruf's analysis, keeping track of set of threads accessing `this.data` does not help proving `this.data` not escaping. As `run` is executed by multiple threads ($T1$ and $T2$), Ruf's analysis claims that `this.data` is written by multiple threads.

putes the escape information for recursive methods by performing fix point computation. It can correctly identify that `foo` does not cause its argument `in` to escape. In fact, `in` is not even accessed in the body of `foo`. Because of this, it correctly claims that `o` is not escaping and therefore, `o.data` is not a shared access. For connectivity and Ruf's analyses, fixpoint computation is avoided by not cloning the information of `foo` when analyzing `foo` itself. The analyses unify `in` and `ESC`, making `in` escaping. This causes `o` to be marked as escaping. Because `bar` is invoked by multiple threads, connectivity analysis, extended connectivity analysis and the adapted Ruf's escape analysis claim that `o.data` is a shared access.

```
// invoked by threads T1, T2
void bar() {
  Object o = new Object();
  foo(o);
  o.data = ...;
  // is o.data a shared access?
}
static Object ESC;
// invoked by threads T1, T2
void foo(Object in) {
  foo(ESC);
}
```

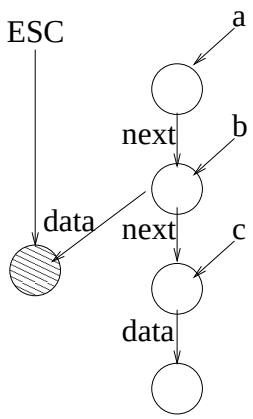
Figure 3.34: A program where Bogda's analysis is more precise

3.6.1.3 Cases Where Ruf's Analysis is More Precise

An example program that demonstrates the precision of Ruf's analysis is shown in Figure 3.35. Ruf's analysis is more precise in this case because it keeps a more precise alias information. As we can see from Figure 3.35(b), Ruf's analysis can correctly identify that `c` is pointing to a non-escaping object, so `c.data` is not a shared access. For connectivity analysis, since `c` and `ESC` are connected, they point to the same connectivity set, so `c` is considered escaping. Therefore, connectivity analysis concludes that `c.data` is a shared access. Bogda's analysis assumes objects reachable by more than one field's references

```
class ListNode {
  ListNode next;
  Data data;
}
static Data ESC;
// invoked by threads T1, T2
void foo() {
 ListNode a = new ListNode();
 ListNode b = new ListNode();
 ListNode c = new ListNode();
 a.next = b;
 b.next = c;
 b.data = ESC;
 c.data = ...;
 // is c.data a shared access?
}
```

(a) The program



(b) The lattice when the program is analyzed by Ruf's analysis

Figure 3.35: A program where Ruf's analysis is more precise

are escaping, so `c` is pointing to an escaping object, as it is pointed by `a.next.next` (two field references). Hence, Bogda's analysis concludes that `c.data` is a shared access.

3.6.1.4 All Cases

In previous sections, we have seen cases where one analysis is more precise than the rest. Figure 3.36 shows three sets corresponding to objects found to be escaping by the three escape analyses. For the discussion in this section, for simplicity, let us assume that Ruf's analysis is a simplified version which marks objects if they are reachable from static variables or runnable objects, and the analysis does not consider whether the object is accessed by multiple threads. Similarly, the connectivity analysis is the base analysis not including the extension given in Section 3.3.5.

The cases described in previous sections are due to objects in the shaded regions. For example, connectivity analysis is more precise in the program described in Section 3.6.1.1 because there are objects in the top shaded region — objects which are Bogda-escaping and Ruf-escaping but not connectivity-escaping. In summary:

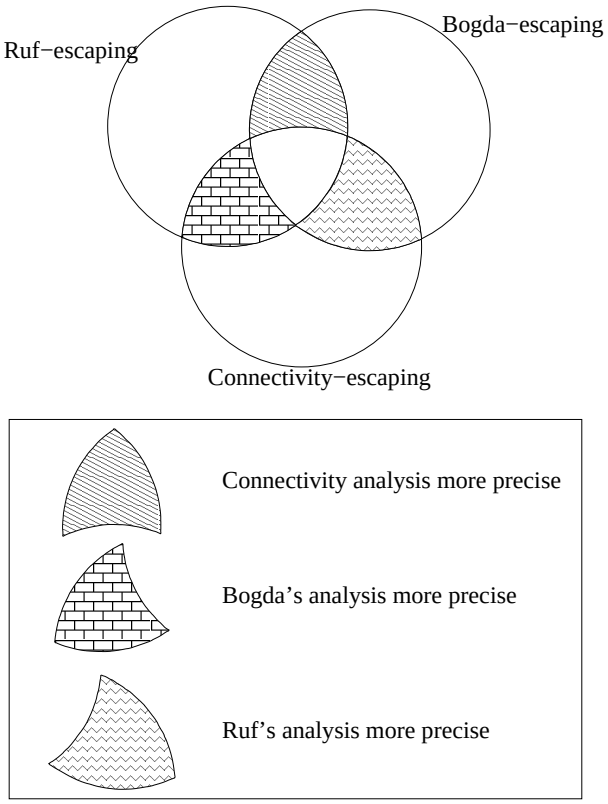


Figure 3.36: An example illustrating different kinds of objects w.r.t different escape analyses

- Connectivity analysis can precisely identify objects reachable by a thread but that are exclusively accessed by the thread.

- Bogda's analysis can precisely identify non-escaping objects passed to recursive methods. For example, in the program described in Section 3.6.1.2, it can identify that `o` is not escaping while connectivity and Ruf's analyses cannot.

- Ruf's analysis can precisely identify objects that are "deeply inside" some data structure (i.e. reachable by multiple field references `x.f.g`) and the data structure has some escape objects (i.e. connected to an escaping object).

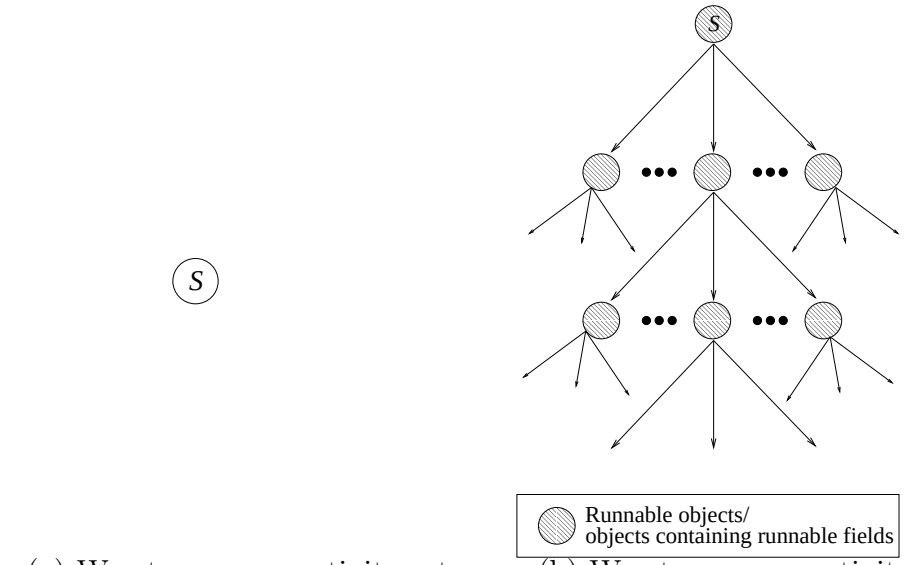
We can see the properties of these distinctive objects are orthogonal. For example, an object can fulfill the first two properties but not the third, then the object will be found to be non-connectivity-escaping and non-Bogda-escaping analysis but is Ruf-escaping. In Chapter 4, we will quantitatively study objects with different escaping properties w.r.t. different escape analyses.

3.6.2 Lattice

In this section, we study the lattices of the analyses, focusing on how complex they can be. This is related to the cost of the analysis as more complicated lattices takes more time to merge. Given a connectivity set/alias set S, we can see how large the lattice structure can be. Figure 3.37 shows the lattices for different algorithms:

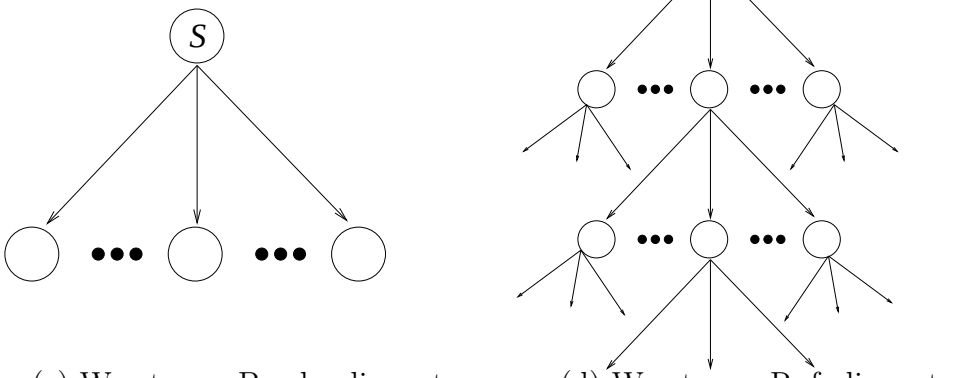
The worst case connectivity lattices can be divided into two cases:

- Without runnable type objects, a connectivity set does not point to another connectivity set via a field reference. This scenario is shown in Figure 3.37(a). This is the common case in a program as most of the time a program does not work on runnable type objects.



(a) Worst case connectivity sets representing objects of non-runnable types

(b) Worst case connectivity sets representing objects having runnable types



(c) Worst case Bogda alias sets

(d) Worst case Ruf alias sets

Figure 3.37: Merging two alias sets in phase 2

- With runnable type objects, the worst case is shown in Figure 3.37(b) where the connectivity sets represents some object of runnable type or containing runnable type fields. We do not expect this case to happen in practice. Even in a program with many recursive calls, it is unusual for the calls to be working on data structures with many of related runnable objects.

The worst case Bogda's lattices is shown in Figure 3.37(c). Since the analysis is an 1-level limited analysis, it restricts the depth of lattice to one level. An object is conservatively assumed to be escaping if it is pointed by an object represented by one of the bottom alias sets.

The worst case Ruf's lattices is shown in Figure 3.37(d). This case happens for program with recursive calls. An example program is shown in Figure 3.38. In the

```
static TreeNode f1() {
  t = new TreeNode();
  t.left = null;
  t.right = null;
  return t;
}
static TreeNode f2() {
  t = new TreeNode();
  t.left = f1();
  t.right = f1();
  return t;
}
static TreeNode f3() {
  t = new TreeNode();
  t.left = f2();
  t.right = f2();
  return t;
}
...
static TreeNode f100() {
  t = new TreeNode();
  t.left = f99();
  t.right = f99();
  return t;
}
```

Figure 3.38: A program causing big lattice when performing Ruf's analysis

$a(f)$	the number of formal parameters declared in method f
V	the maximal number of local variable seen in the programs
A	the maximal number of formal parameters seen in the program, i.e. $\max_{m \in Methods} a(m)$
$Methods$	the set of methods in the program
M	number of methods in the program, i.e. $\|Methods\|$
F	the maximal number of fields among all class
C	number of static fields

Table 3.1: Notations used in time and space complexity analyses

program, the lattice size of `f1` is $2^0 - 1$, `f2` is $2^2 - 1$, `f3` is $2^3 - 1$, ..., `f100` is $2^{100} - 1$, so the size of lattice for the topmost recursive method is of exponential size in the number of methods. This pattern happens in recursive programs like parsers.

3.6.3 Space Complexity

In this section, we discuss the worst case space complexity of the different algorithms. Table 3.1 shows the notations used in both time and space complexity analyses. In all analyses, the cost can be divided into three parts:

- space saved for each method;
- space saved for each global data (for static fields and if needed for thread objects); and
- space for reconstruction.

The space complexity analysis of **Ruf's analysis**:

- For each method, the alias sets of arguments, and the exception and return value are saved. From the rules of Ruf's analysis, we see that at most two alias sets are created for each statement not performing method call. For each method call, the worst case scenario happens when the method called is not recursive, which

causes information cloning. Therefore we have the following equation for $N(m)$, the number of alias sets created when analyzing m:

$$N(m) = 2 \times nc(m) + \sum_{s \in CallStmts(m)} \max_{m' \in CalledMethods(s)} N(m')$$

where $nc(m)$ is the number of statement not performing method call for m, $CalledMethods(s)$ is the set of methods called by s. Therefore, $N(m) = O(N_{nc}N_c^M)$. The complexity is $O(\text{number of alias sets} + \text{number of outgoing edges}) = O(N_{nc}N_c^M + N_{nc}N_c^M F) = O(N_{nc}N_c^M F)$.

- Ruf's analysis saves information for static fields and thread objects. As we see from the rule, an alias set is created for each field or for the runnable type, so the cost is $O(C + 1 + (C + 1)F) = O(CF)$.
- In reconstruction, for each local variable, alias sets are constructed. For each local variable, an alias set is created, which contribute to a cost of $O(V)$. Therefore, the cost is $O(V(1 + F) + N_{nc}N_c^M F)$.

Combining, the cost is $O(M * N_{nc}N_c^M F + CF + VF + N_{nc}N_c^M F) = O(F(M * N_{nc}N_c^M + C + V))$. The space complexity analysis of **Connectivity analysis**:

- For sequential programs, there are no runnable objects, so no connectivity set points to another connectivity set:
  - For each method, the connectivity sets of arguments, return and exception values are saved. Since no connectivity set points to another connectivity set, the space complexity for a method f is $O(a(f) + 2)$
  - A connectivity set is reserved for `ESCAPE` which takes $O(1)$
  - In reconstruction, for each local variable, one connectivity set is constructed. For each local variable, space cost is $O(1)$

  Therefore, the space complexity is $O(M \times (A + 2) + 1 + V) = O(MA + V)$.

- For multi-threaded programs, the worst case is the same as that of Ruf's analysis. The cost is, therefore, $O(F(M * N_{nc}N_c^M + C + V))$.

The space complexity analysis of **Bogda's analysis**:

- For each method, the alias sets of arguments and the return value are saved. Since the analysis is an 1-limiting analysis, the space complexity for a method f is $O((a(f) + 1) \times (1 + F))$ where F is the number of fields in a class having maximal number of fields.
- Bogda analysis does not save information for global data.
- In reconstruction, for each local variable, alias sets are constructed. For each local variable, space cost is $O(1 + F)$.

Therefore, the space complexity is $O(M \times (A+1) \times (1+F) + V \times (1+F)) = O((MA+V) \times F))$ where M is the number of methods, A is the maximal number of formal parameters seen in the program.

3.6.4 Time Complexity

In this section, we discuss the worst case time complexities of different algorithms. Table 3.2 shows the notations used in time complexity analysis.

In all analyses, the cost can be divided into three parts:

- a Bottom-up phase computing method information;
- a Top-down phase computing context information; and
- a Reconstruction information using the above two pieces of information.

So,

$\alpha(n)$	the inverse Ackermann's function
$I(m)$	the cost of interprocedural analysis for method m
$P(m)$	the cost of propagating context information from m
$c(s)$	the time complexity of a statement s
Z	the total number of alias/connectivity sets in the analysis
Stmts	the set of all statements in the program
$Stmts(m)$	the set of statements in method m
$CallStmts(m)$	the set of call statements in method m
ReconstructMethods	the set of methods performed reconstruction to obtain information for each local variable
ReconstMethodStmts	$\cup_{m \in ReconstructMethods} Stmts(m)$
N_{nc}	the maximal number of statement not performing method call among all methods, i.e. $\max \lvert Stmts(m) - CallStmts(m) \rvert$
N_c	the maximal number of call statement among all methods, i.e.$\max_{m \in Methods} \lvert CallStmts(m) \rvert$

Table 3.2: Notations used in time complexity analysis

$$\begin{aligned} \text{Time Complexity} &= \text{Complexity of bottom-up phase} + \text{Complexity of top-down phase} \\ &+ \text{Complexity of reconstruction} \end{aligned}$$

The time complexity of **Connectivity analysis**:

In both the top-down and bottom-up phase, each method is visited once to invoke the intraprocedural analysis. In the top-down phase, context information propagation is done by visiting call statements one more times following the rule shown in Figure 3.10. For reconstruction, an intraprocedural analysis is invoked to compute information for methods that needs the information. Therefore,

$$\begin{aligned} \text{Time Complexity} &= \sum_{m \in Methods} I(m) + \sum_{m \in Methods} (I(m) + P(m)) + \sum_{m \in ReconstructMethods} I(m) \\ &= 2 \sum_{m \in Methods} I(m) + \sum_{m \in Methods} P(m) + \sum_{m \in ReconstructMethods} I(m) \end{aligned}$$

The interprocedural analysis cost can be divided into a cost for processing call statements and a cost for processing statements not performing method call. Also, the cost of propagating context information is the same as processing all call statements in the method, so we have

$$I(m) = \sum_{s \in Stmts(m)} c(s)$$

and

$$P(m) = \sum_{s \in CallStmts(m)} c(s)$$

so

$$\begin{aligned} \text{Time Complexity} &= 2 \sum_{m \in Methods} \sum_{s \in Stmts(m)} c(s) + \sum_{m \in Methods} \sum_{s \in CallStmts(m)} c(s) \\ &\quad + \sum_{m \in ReconstructMethods} \sum_{s \in Stmts(m)} c(s) \end{aligned}$$

As we can see from the rule, call statements are more expensive to analyze than other statements. Let $e(m)$ be the most expensive call statement to analyze for method m, then the time complexity can be bounded as

$$\begin{aligned} \text{Time Complexity} &\leq 2 \sum_{m \in Methods} \sum_{s \in Stmts(m)} c(e(m)) + \sum_{m \in Methods} \sum_{s \in CallStmts(m)} c(e(m)) \\ &\quad + \sum_{m \in ReconstructMethods} \sum_{s \in Stmts(m)} c(e(m)) \\ &= \sum_{m \in Methods} (2|Stmts(m)| + |CallStmts(m)|)c(e(m)) \\ &\quad + \sum_{m \in ReconstructMethods} |Stmts(m)|c(e(m)) \\ &= (2|Stmts| + |CallStmts| + |ReconstMethodStmts|)c(S) \\ &= O((|Stmts| + |ReconstMethodStmts|)c(S)) \end{aligned}$$

where S is the most expensive call instruction to analyze in the program. The value of $c(S)$ includes two components — cloning and unification of arguments connectivity sets. Since both are traversing connectivity sets reachable from the arguments, their complexities are the same. The complexity can be bounded by the product of:

- the max number of formal parameter seen in the program A;
- the max number of methods resolved, bounded by M; and
- the number of nodes reachable from a given connectivity set.

There are two cases for the bound:

- For single threaded programs, there is no runnable objects, so each connectivity set does not point to another connectivity set. Therefore, $c(S) = O((A \times M \times 1)\alpha(Z)) = O(AM\alpha(Z))$.
- For multithreaded programs, in an unlikely worst case, there can be an exponential number of connectivity sets reachable from a given connectivity set, so $c(S) = O((A \times M \times N_{nc}N_c^M)\alpha(Z)) = O(AMN_{nc}N_c^M\alpha(Z))$.

Therefore, the time complexity is $O((|Stmts|+|ReconstMethodStmts|)AM\alpha(Z))$ for single threaded program and $O((|Stmts|+|ReconstMethodStmts|)AMN_{nc}N_c^M\alpha(Z))$ for program with runnable objects.

The time complexity of **Bogda's analysis**:
In both top-down and bottom-up phase, two subphases are performed — one for computing s-escape information and then other for computing f-escape information. As we see from the rules in Figure 3.27 and Figure 3.28, every rule for f-escaping computation is more expensive than the corresponding rule for s-escaping, so the time complexity of Bogda's analysis is dominated by the time complexity of f-escape information computation. Thus, we just need to compute the bound of f-escape information computation.

Like connectivity analysis, call statements are more expensive to analyze than other statements, so using a notation similar to the case of connectivity analysis, we have

$$\begin{aligned}\text{Time Complexity} &= (d+2) \times \sum_{m \in Methods} I(m) + (d+2) \times \sum_{m \in Methods} (I(m) + P(m)) \\ &\quad + \sum_{m \in ReconstructMethods} I(m) \\ &= O(d \sum_{m \in Methods} I(m) + d \sum_{m \in Methods} P(m) + \sum_{m \in ReconstructMethods} I(m))\end{aligned}$$

where d is the maximal number of back edges on any acyclic path in a call graph. The factor d is due to Bogda's analysis being an iterative analysis. Following calculation similar to that for connectivity analysis, we have

$$\text{Time complexity} = O((d|Stmts| + |ReconstMethodStmts|)c(S))$$

where S is the most expensive call instruction to analyze in the program. The value of $c(S)$ can be bounded by considering unifying a call which has A actual arguments, M resolvable methods and each alias set is pointing to F alias sets. Therefore, $c(S)$ is the product of M and the sum of the following four complexities, each corresponds to one of the loops in Figure 3.28:

1. Cost of importing of escape information, C_1, this is the case where escape information for alias sets reachable from the arguments is imported, so

$$C_1 = O((A + A \times F)\alpha(Z)) = O(AF\alpha(Z))$$

2. Cost of unifying with return value, C_2

   This is the case where every argument needs to unify with the return value, so

$$C_2 = O(A \times (1 + F)\alpha(Z)) = O(AF\alpha(Z))$$

3. Cost of unifying cases where $a_i.f = a_j$ for some $i, j \in [0, A]$ and some field f, C_3

   This is the case where for every $i, j \in [0, A]$, the alias set of $a_i.f$ is unified with that of a_j for some field f. Since $a_i.f$ is s-escaping already, the unification is done after performing the union operation for alias sets of $a_i.f$ and a_j. It does not attempt to unify fields of a_j, so

$$C_3 = O(A \times A \times F\alpha(Z)) = O(A^2 F\alpha(Z))$$

4. Cost of unifying cases where $a_i.f = a_j.h$ for some $i, j \in [0, A]$ and some field f, h, C_4

   This is the case where for every $i, j \in [0, A]$, the alias set of $a_i.f$ is unified with that of $a_j.h$ for some field f, h. Since both $a_i.f$ and $a_j.h$ are s-escaping already, the unification is done after performing the union operation for alias sets of $a_i.f$ and $a_j.h$, so

$$C_4 = O(A \times A \times F \times F\alpha(Z)) = O(A^2 F^2 \alpha(Z))$$

Combining all cases, we see $c(S) = O(A^2F^2\alpha(Z))$, so the time complexity is $O((d|Stmts| + |ReconstMethodStmts|)A^2F^2\alpha(Z))$.

The time complexity of **Ruf's analysis**:
In bottom-up phase, each method is visited once and in each method, each statement is visited once. In the top-down phase, each SCC is visited once and methods inside an SCC are visited multiple times until a fixpoint is achieved. Following similar notations described above, we have

$$\begin{aligned}
\text{Time Complexity} &= \sum_{m \in Methods} I(m) + (d+2) \times \sum_{m \in Methods} (I(m) + P(m)) \\
&\quad + \sum_{m \in ReconstructMethods} I(m) \\
&= O(d \sum_{m \in Methods} I(m) + d \sum_{m \in Methods} P(m) + \sum_{m \in ReconstructMethods} I(m))
\end{aligned}$$

where d is the maximal number of back edges on any acyclic path in a call graph. The factor d results from the topdown phase being done iteratively inside an SCC. In the worse case, the whole program is in a single SCC. Following a similar calculation as in the connectivity analysis, we have

$$\text{Time complexity} = O((d|Stmts| + |ReconstMethodStmts|)c(S))$$

where S is the most expensive call instruction to analyze in the program. The complexity of $c(S)$ is obtained in the way similar to the multithreaded case of connectivity analysis

$$c(S) = O(AMN_{nc}N_c^M\alpha(Z))$$

Therefore, the complexity is $O((d|Stmts| + |ReconstMethodStmts|)AMN_{nc}N_c^M\alpha(Z))$.

3.7 Issues of Method Summaries Cloning

By not cloning the method summary of a recursive method, connectivity and Ruf's analyses can avoid performing an iterative fixpoint computation for methods inside the same SCC. While this can reduce the analysis time, it reduces precision, and see in the benchmark programs we evaluated. In this section, we describe how the imprecision can arise by not cloning method summaries. In Section 3.7.1, we discuss what would happen if we do not perform cloning for non-recursive method calls. In Section 3.7.2 and Section 3.7.3, we discuss how imprecision is incurred for recursive method calls in the a single calling context and multiple calling context cases respectively.

3.7.1 Not Cloning Non-recursive Method Calls

In both connectivity analysis and Ruf's analysis, two phases are needed because cloning is performed when incorporating non-recursive method call summaries. The issue is

illustrated in the example shown in Figure 3.39 The analyses first visit methods in

```
1  static Object ESC;
2
3  f() {
4    Object o = new Object();
5    Object p = new Object();
6    g(o, ESC);
7    g(ESC, p);
8  }
9
10 g(Object x, Object y) {
11 }
```

Figure 3.39: An Example Illustrating the issues of not cloning non-recursive method calls

bottom-up order. Therefore, when `g` is analyzed, `f` is not yet analyzed. In this example, nothing is done for `g` as it is empty. When `f` is analyzed, the method calls at line 6 and line 7 are processed. Since cloning is applied for the summary of `g`, any operations performed in analysis of `f` will *not* change the analysis summary of `g`, so the information that `x` can receive an escaping actual argument is not saved in the summary of `g`. Because of this, a top-down phase is needed to propagate the calling context information from callers to callee. In this example, the top-down phase will propagate the information that `x` is escaping and `y` is escaping from `f` to `g`.

If cloning is not performed when incorporating the summary of `g` to `f`, we need not perform two phases (bottom-up then top-down) — only one phase is needed. Regardless of the order of visiting methods, when `g` is analyzed, nothing is done for its summary. When `f` is analyzed, without cloning of summary of `g`, `ESCAPE` and `y` are placed in the same connectivity set or alias set, and `ESCAPE` and `x` are placed in the same connectivity set or alias set. Therefore, after analyzing `f` and `g` once, the summary of `g` should have recorded that `x` and `y` receive escaping actual arguments. However, the saving of one analysis comes with a cost in analysis precision. Since cloning is not applied when incorporating summary of `g`, after line 6 and line 7 of `f` are analyzed, we have `o`, `ESCAPE`

and x sharing the same alias set or connectivity set, so o is marked as escaping. Similarly, p is marked as escaping. Nevertheless, it is obvious that o and p are not escaping. In fact, if cloning is applied when incorporating the summary of g, and a two phase analysis is done, o and p will not mark as escaping because:

- in the bottom-up phase, the summary of g is cloned, so the analysis of line 6 does not mark y as escaping. Hence p will not be marked as escaping. Similarly, o will not be marked as escaping.
- in the top-down phase, as shown in Figure 3.10, the caller's (f's) information is cloned before unifying with the callee's (g's) information. Therefore, the processing of line 6 does not change the information of o and the processing of line 7 does not change the information of p.

Therefore, this example shows that cloning method summaries, while increases the analysis time, can improve the precision of the analysis.

3.7.2 Imprecision due to a Single Context

Figure 3.40 shows an example to illustrate the imprecision due to a single context. Since f and g are in the same SCC, their summaries are not cloned when the program is analyzed. Therefore, when line 3 is analyzed, the connectivity set or alias set of a and of x are merged. Later, when line 4 is analyzed, the connectivity set or alias set of ESC and of x are merged. Thus, we have a, ESC and x sharing the same connectivity set, and so a to be marked as escaping. We see from the example, that the variable x is not even accessed inside g but this recursive call causes a escaping. This imprecision can propagate *outside* the SCC. In the example, h is outside the SCC containing f and g. The call at line 13 makes y escaping even if cloning is done for g when analyzing h.

```
1  static Object ESC;
2
3  f() {
4    g(a);
5    g(ESC);
6  }
7
8  g(Object x) {
9    f();
10 }
11
12 h() {
13   g(y);
14 }
```

Figure 3.40: An Example Illustrating the imprecision of not cloning method summaries due to a single context

3.7.3 Imprecision due to Multiple Contexts

A formal parameter inside a recursive method can be marked as escaping even if *no* caller passes an escaping actual argument to that method. Consider the example shown in Figure 3.41. We can see the formal parameter `y` does not receive an escaping argument

```
1  static Object ESC;
2
3  f() {
4    g(a, a);
5    g(ESC, b);
6  }
7
8  g(Object x, Object y) {
9    f();
10 }
```

Figure 3.41: An Example Illustrating the imprecision of not cloning method summaries due to multiple contexts

from any calling context. However, with the calls at line 4 and line 5, `y` is marked as escaping:

1. line 4 causes connectivity set or alias sets of `a`, `x` and `y` merged

2. line 5 causes connectivity set or alias sets of `ESC` and `x`; `b` and `y` merged

Combining the two steps, `ESC`, `a`, `b`, `x` and `y` all share the same alias set or connectivity set. Hence, `y` is escaping.

3.8 Reducing the analysis overhead — IR Caching

In the Jikes RVM, before analyzing a method, the IR for the method must be generated first. The original implementation of Jikes RVM requires IR regeneration even if a method has been analyzed before. Because of this, we find that Bogda's analysis runs slower even if it performs the same number of union/find operations when compared with connectivity analysis and Ruf's analysis. This is because Bogda's analysis is an iterative analysis that converges on a fixpoint and a method can be revisited many times. To have a fair raw time comparison, we augmented the Jikes system to perform IR caching, significantly reducing the overhead of IR regeneration.

3.9 Issues in a Dynamic System Setting

In previous sections, to simplify the discussion, we described the escape analyses algorithm assuming that all methods are available when the analyses are performed. That is, the analyses we presented are *whole program analyses*. However, this assumption is not true in a dynamic compilation system as the program may need to be be considered *partial* all the time as new classes may be loaded as the program is running. At any point in the program execution, the methods compiled may be based on partial information about the program. Later when new classes are loaded, some assumptions about the program may be violated, and the previously generated code may no longer be valid. The system should invalidate these methods, causing them to be compiled later if they

are called. In this section, we discuss our speculative approach which is used to handle the cases when complete information is not available.

Because of dynamic loading, it is not always possible to know the code associated with each method invocation. This can be illustrated by the example in Figure 3.42 Depending on the value of `n`, at line 2, either `C1.f()` or `C2.f()` is called at line 4. When

```
1  void foo(String n) {
2    Class c = Class.forName(n);
3    A a = (A)c.newInstance();
4    a.f();
5  }
6  abstract class A {
7    abstract public void f();
8  }
9  class C1 extends A {
10   public void f() { }
11 }
12 class C2 extends A {
13   public void f() { }
14 }
```

Figure 3.42: An Example Illustrating Incomplete program at runtime

`foo` is compiled, it is not known which class is being loaded at line 2 as line 2 has not been executed yet, so it is not known what method is invoked at line 4.

In the Pensieve system, we use an incremental strategy to determine target methods of a method invocation. When `foo` is analyzed, without knowing any possible concrete types of `a`, the analysis just assumes it is calling nothing. Later when line 3 is executed, an instance of class `A` is created by the `newInstance` call. This causes the constructor of, say `C1` (not shown in the example for brevity) to be compiled. After the compilation of the constructor, escape analysis assumes that objects of class `C1` could be instantiated, so the concrete type of `a` can be `C1` and `C1.f()` may be invoked at line 4. Therefore, the information about the program has been changed and the whole program is re-analyzed. This may or may not change the escape information of `foo`. Because of this change, we need to invalidate methods if the previously compiled code used the escape information

which is now too optimistic after the change. Figure 3.43 shows the need of method

```
void foo(String n) {
  bar(new Data());
  Class c = Class.forName(n);
  A a = (A)c.newInstance();
  a.f();
}
void bar(Data d) {
  data = d.data;
}
abstract class A {
  abstract public void f();
}
class C1 extends A {
  public void f() {
    ...
    // esc is escaping
    bar(esc);
  }
}
```

Figure 3.43: An Example Illustrating the Need of Method Invalidation

invalidation. When `bar` is compiled, class `C1` is not yet loaded. Later, when `bar` is compiled, it is known that the formal parameter `d` of `bar` references a non-escaping object, so no fences are needed for the load `d.data`. Later, when the analysis learns that `C1` is loaded, the whole program is re-analyzed and it is found that the formal parameter of `bar` can reference an escaping object if `bar` is called from line 17. Hence, the code for `bar` is invalidated. Later when `bar` is invoked, the compiler will use the correct information (saying that `d` could be escaping) and a fence is inserted before the load of `d.data`.

In some situations, the method being invalidated is actually on the activation stack. This can be illustrated by an example in Figure 3.44. Again, when `foo` is compiled, the analysis does not know `C1.f()` is called. It assumes `a` is not escaping so fences are not needed for the load `a.data`. Later, when `C1` is being loaded during `foo`'s execution, the whole program is re-analyzed and found that `a` can escape because of the call `a.f()`. Therefore, an invalidation should be done for `foo` because a fence is needed for the load

```
void foo(String n) {
  Class c = Class.forName(n);
  A a = (A)c.newInstance();
  a.f();
  d = a.data;
}
abstract class A {
  int data;
  abstract public void f();
}
class C1 extends A {
  public void f() { // this escapes in f }
}
```

Figure 3.44: An Example Illustrating the possibility of invalidating method on stack

`a.data`. However, `foo` is still on the activation stack, so invalidation does not change the code being executed. We have found that connectivity analysis does not encounter such situations with SPECjvm98 and Java Grande benchmarks. In general, techniques like on-stack-replacement[FQ03] are needed to address this problem.

3.10 Incremental Connectivity Analysis

Because of dynamic class loading, the whole program is analyzed whenever the call graph changes. Since the change is usually small, a full-fledged analysis is not usually necessary. In the Pensieve system, an incremental strategy is implemented for connectivity analysis. The idea is to reuse previously computed information when possible. Figure 3.45 shows an example where an incremental strategy can reduce the analysis cost. In both incremental and non-incremental strategies, the program is executed in the following way:

1. when when `main` is executed, class `A1` is loaded and `A1.g()` and `A1.bar` are included to the call graph.

```
class Main {
  static public void main(String[] args) {
    Class c = Class.forName(``A1'');
    A a = (A)c.newInstance();
    a.g();
    bar();
  }
  static void bar() {}
}
class A1 extends A {
  void g() {
    Class c = Class.forName(``A2'');
    A b = (A).c.newInstance();
    b.h();
  }
}
class A2 extends A {
  void h() {
    // this escapes in h()
  }
}
```

Figure 3.45: An Motivating example for incremental analysis

2. Later when `A1.g` is executed, class `A2` is loaded and `A2.h()` is included in the call graph.

In the absence of the incremental strategy, the whole program is re-analyzed, when call graph is changed. Therefore, `main`, `g` and `bar` will be analyzed when `A1` is loaded. Later, when `A2` is loaded, `main`, `g`, `bar` and `h` will all be re-analyzed, i.e., all methods are re-analyzed.

In the our incremental strategy, when `A1` is loaded, in the bottom up phase:

- `g` is analyzed because it is newly included in the call graph;
- `main` is analyzed because it has a new callee `g`; and
- `bar` is *not* analyzed because do not have any new direct or indirect callee.

and in the top down phase:

- `main` is analyzed because it has a newly included callee `g`;
- `bar` is *not* analyzed because the context escape information for `bar` is not changed; and
- `g` is analyzed because it is newly included in the call graph. changed.

Later, when `A2` is loaded, in the bottom up phase:

- `h` is analyzed because it is newly included in the call graph;
- `g` is analyzed because it has a new callee `h`;
- `main` is *not* analyzed because the escape information for `g` does not change; and
- `bar` is *not* analyzed because do not have any new direct or indirect callee.

and in the top down phase:

- `main` is *not* analyzed because its context escape information is not changed and escape information for `g` and `bar` is the same;
- `bar` is *not* analyzed for the reason same as that of `main`;
- `g` is analyzed because it has a new callee `h`; and
- `h` is analyzed because it is newly included in the call graph. changed.

We can see from the example that the incremental strategy can skip analyzing methods which are not affected by the newly loaded method. For example, `bar` can be skipped during re-analysis resulting from both class loadings while `main` is skipped during the re-analysis resulting from the second class loading.

Note that this incremental strategy is *not* implemented for Bogda's and Ruf's analyses. When we perform comparisons among the escape analyses, we do not enable the incremental strategy of connectivity analysis.

3.11 Previous Works

In addition to [BH99, Ruf00], much has been done for escape analysis for multi-threaded object oriented languages developed [CGS+99, WR99, Bla99, SR01, RMR01, VR01, GS00]. Some of them [CGS+99, WR99, SR01, RMR01, VR01] focus on analysis precision while other [Bla99] focus on reducing analysis cost. As for application, many [CGS+99, WR99, Bla99, RMR01] are used for synchronization removal while other [Bla99, GS00, RMR01, VR01] for allocating objects on stack.

We chose to compare our analysis with [BH99, Ruf00] because they are efficient and they are similar in nature. These can be consider unification-based analyses. On the other hand, [CGS+99, WR99, SR01, RMR01, VR01] are graph-based analyses. While graph-based analyses are more precise, they are more expensive when compared with unification-based analyses. Blanchet's analysis[Bla99] is a simplified alias analysis style escape analysis using integers as the lattice. It is not clear whether [Bla99] is more efficient than [BH99, Ruf00] and connectivity analysis. Finally, [GS00] is a linear algorithm in term of time and space complexity. However, it is focusing on stack allocation and does not compute enough information to perform fence insertion.

Chapter 4

Experimental Results

In this section we present the results of executing benchmark programs compiled with our Pensieve compiler using the escape analyses described in Section 3. In Section 4.1, we describe the evaluation criteria. Then, in Section 4.2, we describe the environment where the experiments were conducted. In Section 4.3, we present the benchmarks used to evaluate the system. Performance numbers will be presented in Section 4.4 and Section 4.5

4.1 Evaluation Criteria

In this study, the effectiveness of escape analyses is evaluated. This includes analysis time and analysis precision:

- **Analysis time** can be evaluated directly by measuring the time spent in performing the analysis. A disadvantage of using raw analysis time is that it is a function of the implementation. Because of this, we would like to use some implementation independent figure of merit to evaluate the analysis time. This quantity should reflect the algorithmic cost but be independent of implementation choices (e.g. using a `java.lang.HashMap` vs using a `java.lang.TreeMap`). From the earlier

discussions, we can see that the basic operations, used by all analysis algorithms are FIND and UNION, so the number of FIND and UNION are ideal implementation independent quantities to compare analysis complexity. Therefore, in Section 4.4, both the raw analysis time and the number of FIND and UNION are presented.

- **Analysis precision** can be evaluated in many ways as it depends on the usage of the analysis. In this study, the precision is evaluated in the following ways:

  1. Number of created objects marked as escaping by different escape analyses.

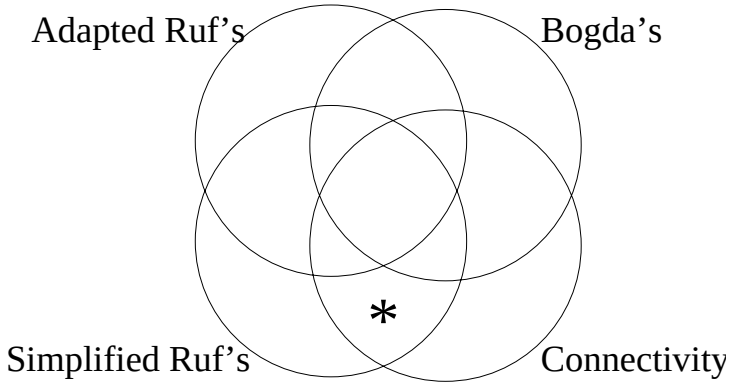


Figure 4.1: Classifying Objects Created

     Figure 4.1 shows a Venn diagram of the sets of objects marked as escaping. The region marked by "*" represents the set of objects marked as escaping by both simplified Ruf's analysis and connectivity analysis but *not* escaping by neither adapted Ruf's analysis nor Bogda's analysis. In Section 4.5.1, we discuss our findings in the number of objects falling into different regions. We report both the dynamic count — number of objects created at runtime, and static count — number of object creation points in the source program.

  2. Fence insertion driven by escape analysis. The number of fences inserted reflects the effectiveness of escape analysis in helping reducing the number of unnecessary fences. In Section 4.5.2, we include both the dynamic number of fences executed and the static number of fences inserted in the code.

3. Synchronization removal driven by escape analysis. The number of synchronization operations removed reflects the effectiveness of escape analysis in helping reducing amount of unnecessary synchronization. In this study, instead of removing synchronizations explicitly as done in [BH99, Ruf00], we put a runtime check in the lock/unlock runtime library call. If the object is marked as non-escaping, the lock and unlock operation can be skipped. In Section 4.5.3, we include both the static number of object allocation sites marked as creating thread local objects and the dynamic number of synchronization executed.

4.2 Experiment Settings

In this section, we describe the machine and software settings for the experiments. We describe the machine settings in Section 4.2.1 and the software settings in Section 4.2.2.

4.2.1 Target Architectures

The experiments are performed on two platforms — the Intel platform and the PowerPC platform:

- The Intel platform is a Dell PowerEdge 6600 SMP with 4 Intel 1.5Ghz Xeon processors with 1MB cache each, and 6G system memory.
- The PowerPC platform is an IBM SP 9076-550 with 8 375Mhz processors with 8GB system memory.

4.2.2 Software Settings

Our compiler system is implemented on top of the Jikes Research Virtual Machine [AAB+00, AFG+00a, BCF+99] version 2.3.1. It is a virtual machine written mostly in Java. We use the FastAdaptiveSemiSpace configuration (internal assertions removed, with adaptive infrastructure and a semispace copying garbage collector) with no fences inserted within the virtual machine code. For the experiments reported below, we do not use the adaptive compilation system. This is to avoid the nondeterministic behavior due to the adaptive compiler system's decisions in performing optimizations. We force the system to use the optimizing compiler and code invalidation is done when the compiler finds that previously generated code is too optimistic. To evaluate the overhead of our system, we compare the performance of programs assuming a SC programming language memory model to the performance of programs assuming the default Jikes RVM programming language memory model. Under this default memory model, the compiler performs memory access optimizations such as redundant load elimination, dead store elimination, and loop invariant code motion, without being constrained by inter-thread effects. When compiling for the SC model , we assume there is a delay edge between every pair of shared data accesses found from the algorithm described in Chapter 3.

There are two major settings of the compiler:

1. Baseline setting(**base**) — default setting without inserting fences nor constraining compiler optimizations. This is the original Jikes compiler without modifications.

2. Escape analysis settings — inserting fences and constraining compiler optimizations using escape analysis results. There are five sub-settings:

   (a) Bogda's Escape Analysis (**bogda**) says that objects are escaping if they can be reached by the global state or can be reached by another object via two or more heap references

(b) Connectivity Analysis (**connect2**) says objects are escaping if they are connected to the global state.

(c) Extended Connectivity Analysis has two criteria depending on the use of escape analysis:

- If the analysis is used for fence insertion(**connect3**), objects are escaping if they are connected to the global state and are loaded and/or stored by multiple threads.
- If the analysis is used for synchronization removal(**connect4**), objects are escaping if they are connected to the global state and are synchronized by multiple threads.

(d) Simplified Ruf's Escape Analysis (**ruf3**) says objects are escaping if they are marked reachable by the global states.

(e) Adapted Ruf's Escape Analysis has two criteria depending on the use of escape analysis:

- If the analysis is used for fence insertion (**ruf5**), objects are escaping if they are reachable by the global state and are loaded and/or stored by multiple threads.
- If the analysis is used for synchronization removal (**ruf4**), objects are escaping if they are reachable by the global state and are synchronized by multiple threads.

4.3 The Benchmarks

We evaluate our system using two sets of benchmarks — the SPECjvm98 benchmark suite and the Java Grande benchmark suite. SPECjvm98 is a general purpose computing suite from the Standard Performance Evaluation Corporation (SPEC). The Java Grande Forum Multi-threaded Benchmarks Suite is a multi-threaded scientific computation benchmark from the Edinburgh Parallel Computing Centre (EPCC). Table 4.1

Program	#methods	Lines of Src	Description
_201_compress	58	11273	Modified Lempel-Ziv method (LZW)
_202_jess	450	20925	The Java Expert ShellSystem
_209_db	56	11374	Memory resident database
_213_javac	778	-	Java compiler from the JDK 1.0.2
_222_mpegaudio	190	-	An application decompresses audio files
_227_mtrt	184	14146	A multithreaded implementations of raytracer
_228_jack	294	-	A Java parser generator

Table 4.1: SPECjvm98 Benchmarks Suite information

Program	# methods	Lines of Src	Description
moldyn	39	11414	Molecular Dynamics simulation
montecarlo	109	13678	Monte Carlo simulation
raytracer	73	11981	3D Ray Tracer

Table 4.2: Java Grande Multi-threaded Benchmarks Suite information

briefly describes the SPECjvm98 benchmarks suite and Table 4.2 briefly describes the Java Grande Forum Multi-threaded Benchmarks suite. For some of the benchmark programs, the source file is not available, and so the source line counts are not available for them.

4.4 Evaluating Analysis Time

In this section, we present the analysis time results. Both the analysis time in seconds and the number of union and find operations are reported. In Section 4.4.1, we presents some data to show the relationship between raw analysis time and number of union and find operations. The analysis time is shown in Table 4.3 and its graphical plot is shown in Figure 4.4. Note that in the graph, we also include the geometric means of all benchmark programs' analysis time with respect to different escape analyses algorithm. The number of union and find operations are shown in Table 4.4 and its graphical plot is shown in

Figure 4.5. Similar to the case of the analysis time graph, we include the geometric means of all benchmark programs' number of union and find operations.

4.4.1 Raw Analysis Time vs Number of Union and Find Operations

In this section, we relate the raw analysis time and the number of union and find operations. Figure 4.2 and Figure 4.3 shows the regression graphs for Intel and PowerPC platforms respectively. For each platform, we show the regression graph for individual analysis algorithms as well as that for combined data for all analysis algorithms. We can see in all cases, the values of R^2 are very close to 1 where R is the correlation coefficient. This shows that in all cases there is a linear relationship between the raw analysis time and the number of union and find operations.

4.4.2 Observations

We can see from the data that the connectivity analysis and the extended connectivity analyses are the fastest analyses. This is reflected in both the raw analysis time and the number of union-find operations on both platforms:

- For the Intel platform, it takes less than 42 seconds to perform the full connectivity analysis and 47 seconds to perform the extended connectivity analysis, while it takes at most 316 seconds to perform Bogda's analysis, 934 seconds to perform simplified Ruf's analysis and 991 seconds to perform adapted Ruf's analysis. Similarly, for the number of union-find operations, the connectivity analyses require less than 75 million union-find operations to analyze programs while Bogda's analysis requires more than 402 million union-find operations to analyze `_202_jess`, and Ruf's analyses require more than 1800 million union-find operations to analyze `_213_javac`.

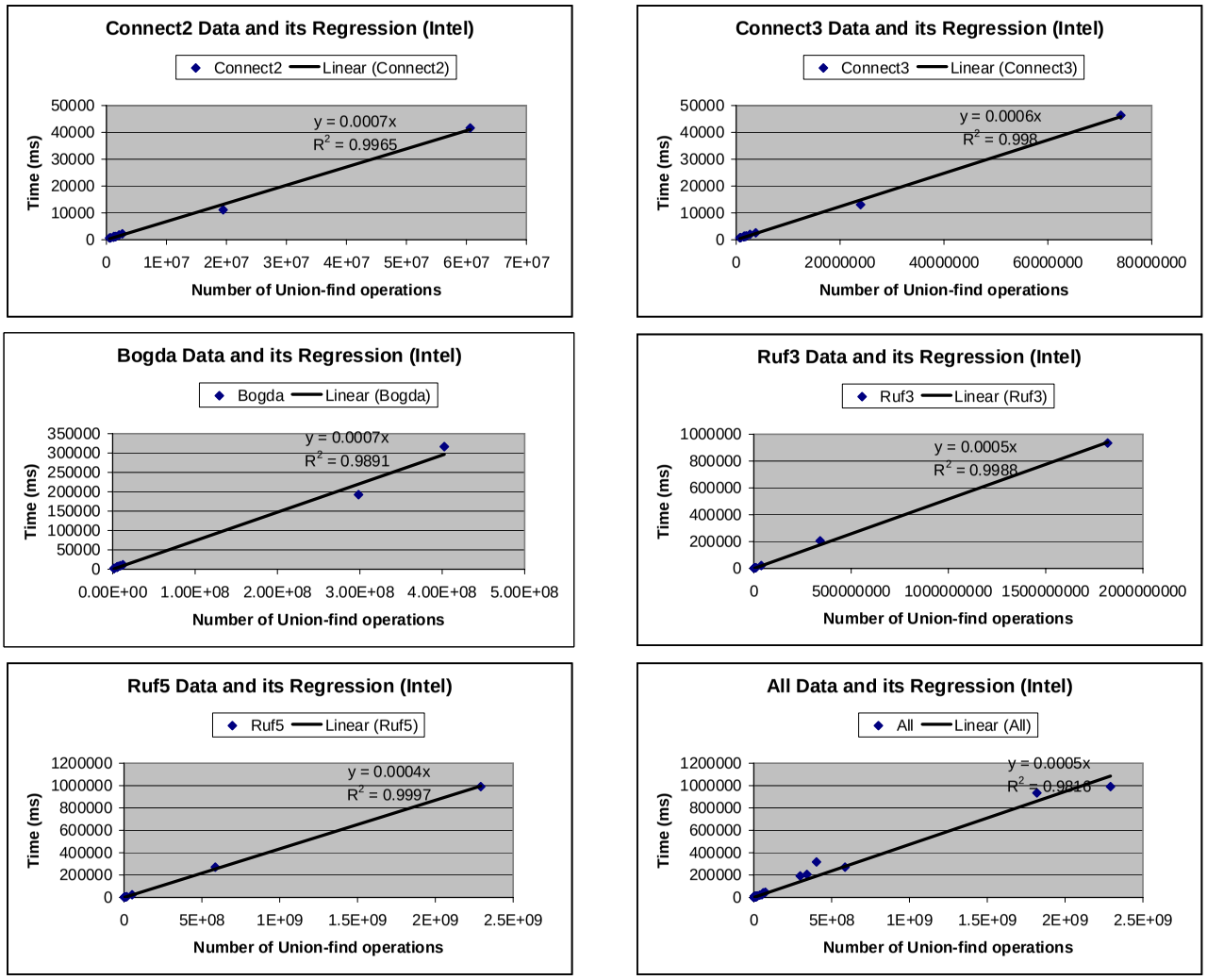


Figure 4.2: Analysis Time Regression Graphs for Intel Platform

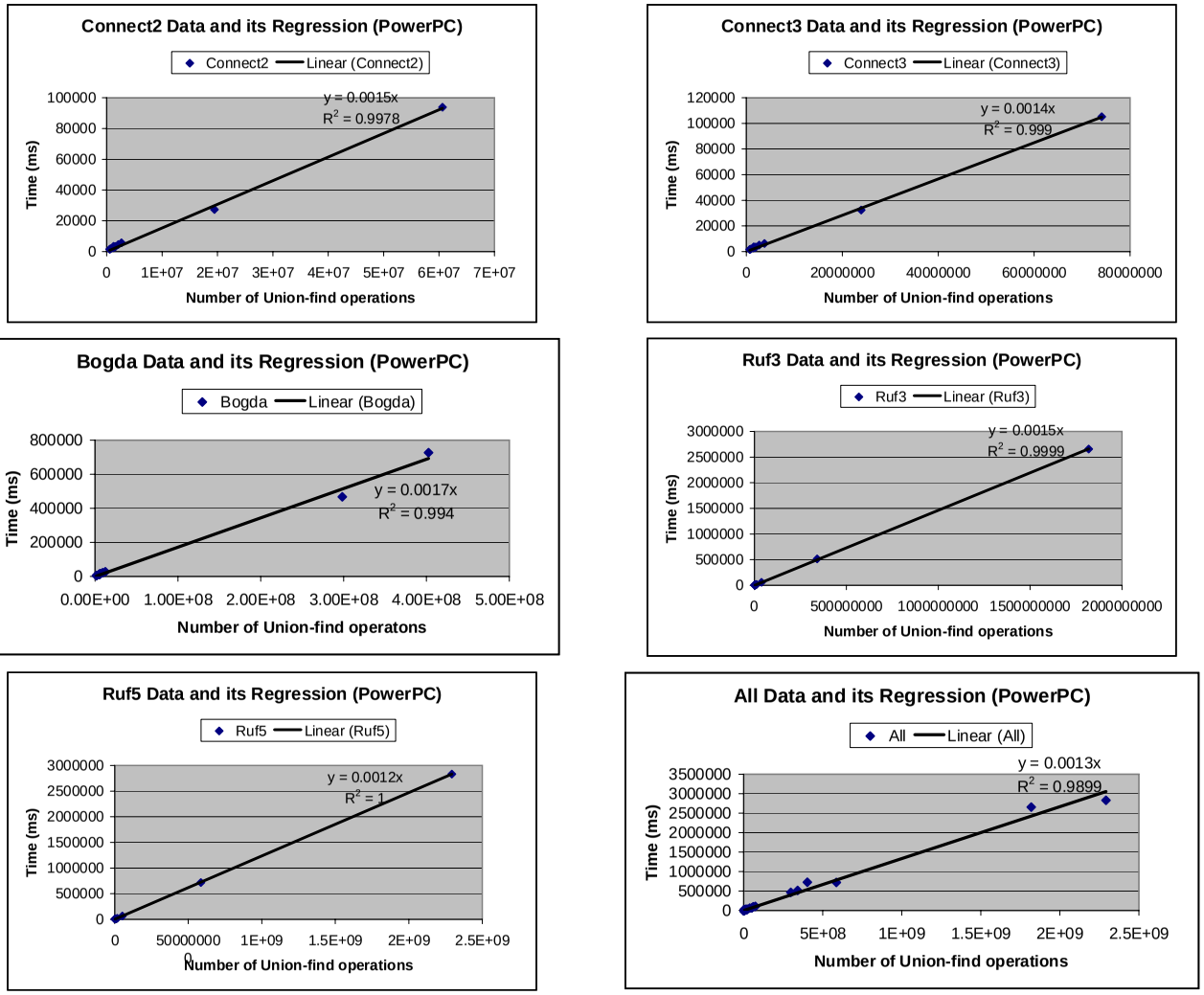


Figure 4.3: Analysis Time Regression Graphs for PowerPC Platform

- For the PowerPC platform, It takes less than 94 seconds to perform the full connectivity analysis and 106 seconds to perform the extended connectivity analysis, while it takes at most 726 seconds to perform Bogda's analysis, 2656 seconds to perform simplified Ruf's analysis and 2831 seconds to perform adapted Ruf's analysis. Similarly, for the number of union-find operations, the connectivity analyses require less than 75 million union-find operations to analyze programs while Bogda's analysis requires more than 402 million union-find operations to analyze `_202_jess`, and Ruf's analyses require more than 1800 million union-find operations to analyze `_213_javac`.

Moreover, we see that the variation of analysis cost (in terms of time and number of union-find operations) of connectivity analyses is smaller than that of Bogda's and Ruf's analyses.

4.4.3 Interpretations

As expected, the analysis time for connectivity analyses is smaller because:

- Unlike Bogda's analysis, it does not need to do fixpoint computations for recursive methods. We can see this from the large variation in Bogda's analysis time for *jess* and *javac* and the much smaller variation in connectivity analysis time for these two programs. This is because these two programs are highly recursive.

- Unlike Ruf's analyses, it does not need to merge complicated analysis data structures. Again, we can see this from the variation in analysis times for *jess* and *javac*. In these two programs, the analysis data structures are large because the programs build recursive tree structures and Ruf's analyses need to model them in analysis time. Connectivity analyses, on the contrary, only model the tree structures using a smaller number of connectivity sets.

- For programs other than *jess* and *javac*, connectivity analyses is also cheaper than Bogda's and Ruf's analyses because the structure of connectivity sets is simpler than the alias sets used by Ruf's and Bogda's analyses.

Also, the analysis cost for **connect3** is larger than that of **connect2** while the analysis cost for **ruf5** is larger than that of **ruf3** as expected. This is because **connect3** and **ruf5** need to maintain extra data structures (the set of thread accessing the connectivity/alias sets). Note also that from the data, we see that the number of union-find operation counts has a high correlation with the analysis time, so both quantities are consistent with each other.

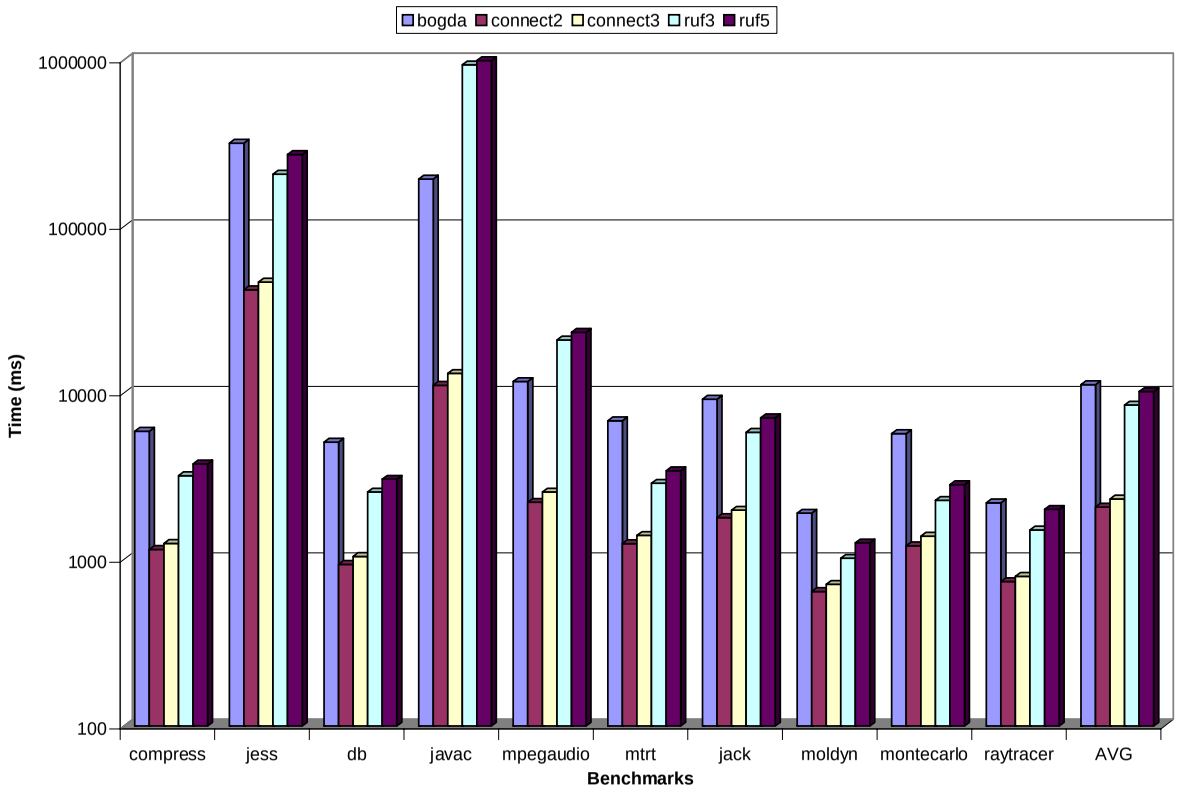
Benchmark	**bogda**	**connect2**	**connect3**	**ruf3**	**ruf5**
_201_compress	5890	1149	1248	3189	3750
_202_jess	316305	41664	46329	206198	270723
_209_db	5072	934	1044	2542	3037
_213_javac	192596	11123	13104	934818	991159
_222_mpegaudio	11710	2205	2541	20822	23158
_227_mtrt	6816	1245	1395	2871	3410
_228_jack	9176	1785	1978	5803	7104
moldyn	1897	642	709	1017	1259
montecarlo	5704	1209	1380	2266	2817
raytracer	2189	737	793	1504	2002

(a) Intel Platform

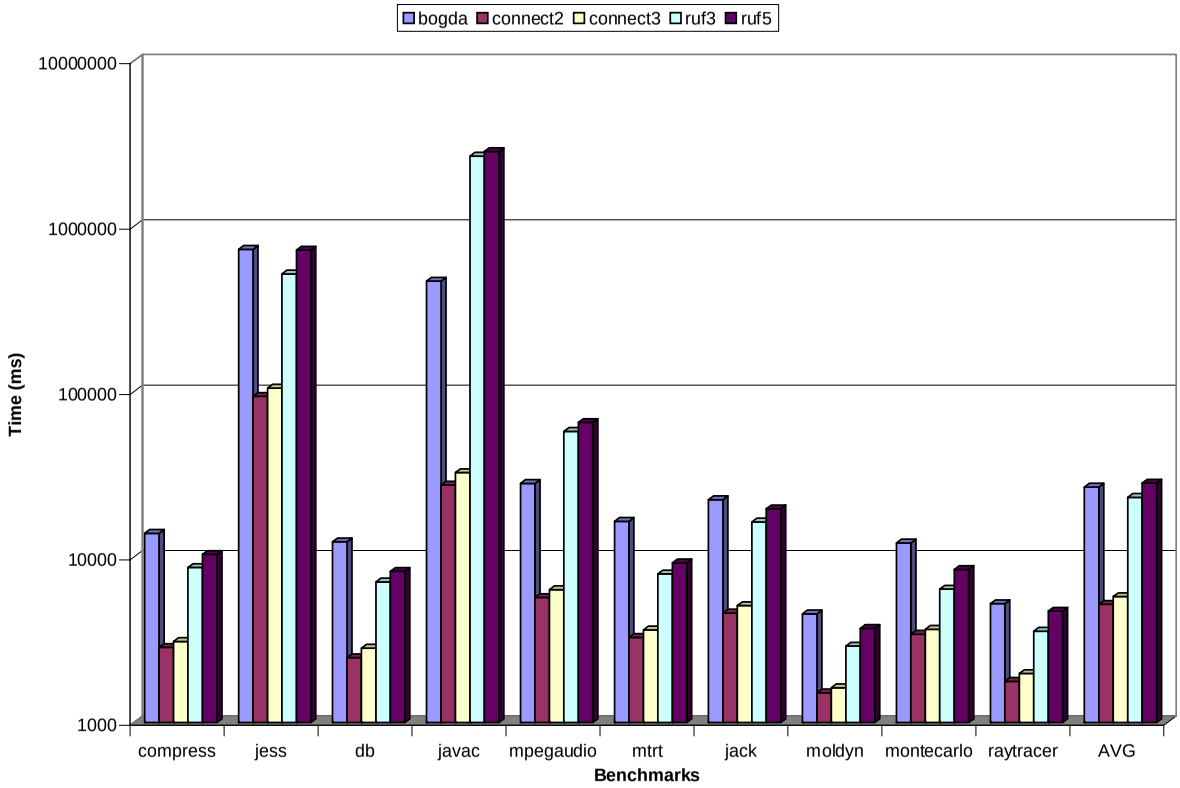
Benchmark	**bogda**	**connect2**	**connect3**	**ruf3**	**ruf5**
_201_compress	13934	2847	3090	8657	10385
_202_jess	726847	93800	105207	515244	718604
_209_db	12377	2466	2816	7079	8212
_213_javac	466940	27294	32446	2656319	2831840
_222_mpegaudio	27898	5706	6363	57611	65231
_227_mtrt	16464	3273	3629	7913	9239
_228_jack	22262	4597	5092	16316	19592
moldyn	4539	1512	1624	2899	3709
montecarlo	12188	3419	3666	6407	8417
raytracer	5232	1775	1980	3565	4713

(b) PowerPC Platform

Table 4.3: Analysis time comparison of escape analyses in ms

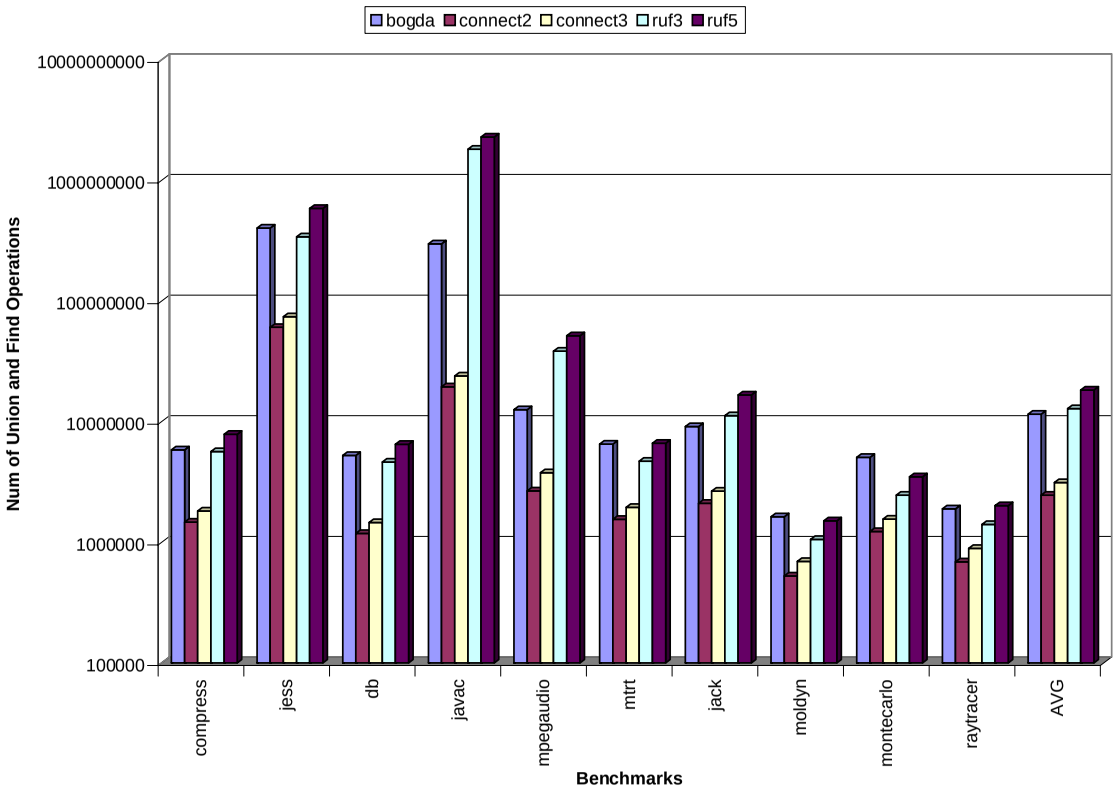


(a) Intel Platform

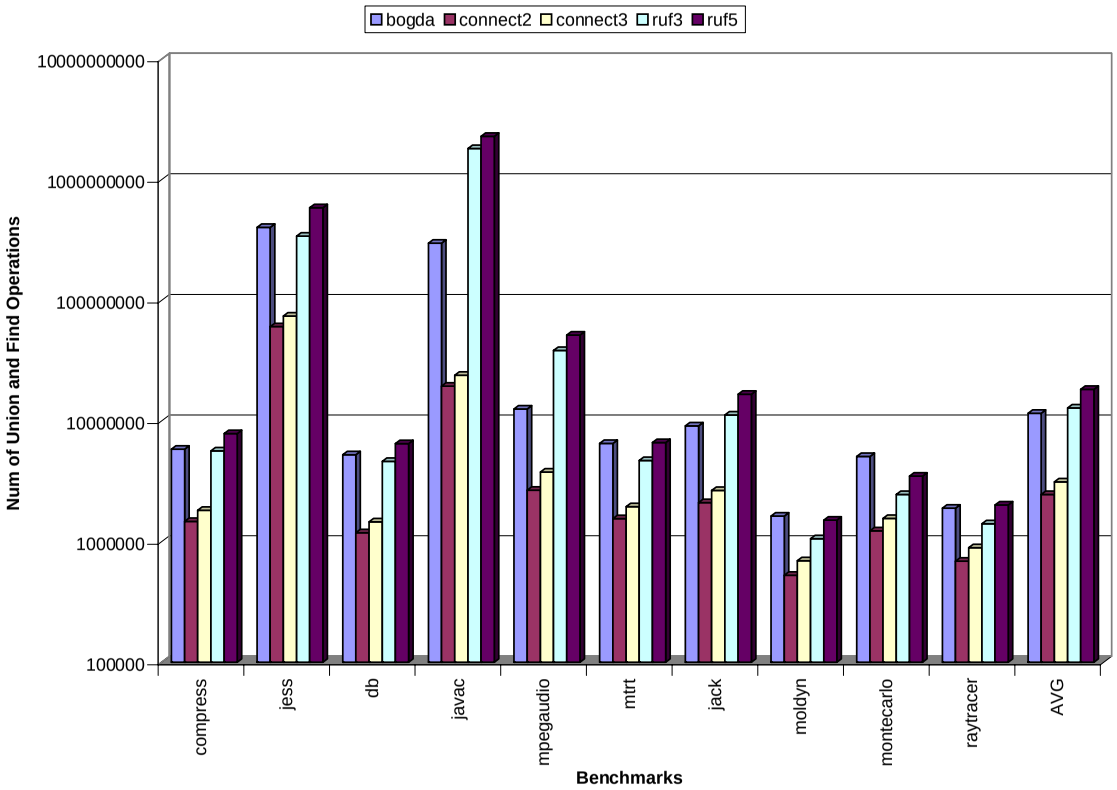


(b) PowerPC Platform

Figure 4.4: Analysis Time Graph



(a) Intel Platform



(b) PowerPC Platform

Figure 4.5: Union-Find Count Graph

Benchmark	bogda	connect2	connect3	ruf3	ruf5
_201_compress	5840614	1472308	1818152	5651462	7868717
_202_jess	402579343	60636462	74076066	340784808	586013018
_209_db	5259505	1183538	1460413	4631151	6506659
_213_javac	298407665	19457373	23950918	1819024377	2291711654
_222_mpegaudio	12553305	2665286	3776638	38411127	51628566
_227_mtrt	6524657	1549956	1946474	4708326	6630883
_228_jack	9102907	2104451	2661841	11226287	16640929
moldyn	1628818	526080	694086	1058056	1508353
montecarlo	5056461	1231069	1555353	2464042	3490701
raytracer	1893013	688356	892159	1409270	2013174

(a) Intel Platform

Benchmark	bogda	connect2	connect3	ruf3	ruf5
_201_compress	5840463	1479384	1818152	5651409	7868618
_202_jess	402579024	61040205	74076133	340765843	586077538
_209_db	5259307	1188496	1460419	4631552	6506478
_213_javac	298407121	20135740	23950926	1819047444	2292376018
_222_mpegaudio	12553212	2693776	3776632	38411072	51628504
_227_mtrt	6524556	1555768	1946474	4708246	6630797
_228_jack	9102253	2114067	2661833	11238773	16654691
moldyn	1628649	532701	694086	1057983	1508278
montecarlo	5087972	1240398	1555367	2462971	3491627
raytracer	1892844	696499	892171	1409236	2014399

(b) PowerPC Platform

Table 4.4: Union-find count comparison of escape analyses

4.4.4 Incremental Analysis Time

For connectivity analyses, we collect the time taken and the number of unions and finds performed for the incremental analysis described in Section 3.10. The data are shown in Table 4.5 and Table 4.6 respectively. We see that both platforms have similar behavior. The extended connectivity analysis is slightly more expensive than the full one. For the Intel platform, the incremental strategy speeds up the full analysis from 31% to 99% of the non-incremental analysis time and the extended analysis from 25% to 99%. For the PowerPC platform, the incremental strategy speeds up the full analysis from 23% to 92% of the non-incremental analysis time and the extended analysis from 21% to 87%. We can also see from Table 4.6, that the reduction in number of union-find operations has similar trend.

4.5 Evaluating Analysis Precision

In this section, we present the analysis precision results. In Section 4.5.1, we report data keeping track of the number of objects marked as escaping with respect to different escape analyses. In Section 4.5.2, we report the number of fences inserted statically and executed dynamically when enforcing SC. Finally, in Section 4.5.3, we report the number when the escape analyses are used for synchronization removal.

4.5.1 Number of Object Created Marked as Escaping

For the same object in a program, different escape analyses may mark it differently due to the precision of the analyses. For example, an object located in a binary tree may be marked as escaping by Bogda's escape analysis but not by Connectivity and Ruf's analyses. A qualitative discussion on this issue has been presented in Section 3.6.1. In this section, we present some quantitative data to evaluate the analyses. Table 4.9 shows the data when programs are executed on the Intel platform while Table 4.10 shows the

Benchmark	connect2			connect3		
	orig	inc	speedup	orig	inc	speedup
_201_compress	1149	649	1.77042	1248	712	1.75281
_202_jess	41664	23486	1.77399	46329	26193	1.76876
_209_db	934	685	1.3635	1044	758	1.37731
_213_javac	11123	6455	1.72316	13104	7743	1.69237
_222_mpegaudio	2205	1197	1.84211	2541	1231	2.06418
_227_mtrt	1245	882	1.41156	1395	998	1.3978
_228_jack	1785	1318	1.35432	1978	1446	1.36791
moldyn	642	482	1.33195	709	532	1.33271
montecarlo	1209	837	1.44444	1380	948	1.4557
raytracer	737	580	1.27069	793	644	1.23137

(a) Intel Platform

Benchmark	connect2			connect3		
	orig	inc	speedup	orig	inc	speedup
_201_compress	2847	1693	1.68163	3090	1955	1.58056
_202_jess	93800	54468	1.72211	105207	60390	1.74213
_209_db	2466	1770	1.39322	2816	1944	1.44856
_213_javac	27294	16367	1.66762	32446	19456	1.66766
_222_mpegaudio	5706	2994	1.90581	6363	3270	1.94587
_227_mtrt	3273	2386	1.37175	3629	2740	1.32445
_228_jack	4597	3505	1.31155	5092	3900	1.30564
moldyn	1512	1172	1.2901	1624	1327	1.22381
montecarlo	3419	2297	1.48846	3666	2469	1.48481
raytracer	1775	1399	1.26876	1980	1603	1.23518

(b) PowerPC Platform

Table 4.5: Incremental Connectivity Analyses time in ms

Benchmark	connect2	connect3
_201_compress	656861	812199
_202_jess	28694898	35281097
_209_db	737101	910078
_213_javac	7905266	9814283
_222_mpegaudio	929371	1315681
_227_mtrt	945285	1187056
_228_jack	1313672	1668082
moldyn	348649	462522
montecarlo	643456	817748
raytracer	457786	595562

(a) Intel Platform

Benchmark	connect2	connect3
_201_compress	656805	812133
_202_jess	28694739	35280568
_209_db	737030	909988
_213_javac	7905000	9813952
_222_mpegaudio	929334	1315632
_227_mtrt	945189	1186933
_228_jack	1313251	1667569
moldyn	348549	462398
montecarlo	643522	817846
raytracer	457707	595456

(b) PowerPC Platform

Table 4.6: Union-find count of Incremental Connectivity Analyses

Abbreviation	Meaning
b	Bogda escaping (**bogda**)
c2	Full Connectivity escaping (**connect2**)
c3	Extended Connectivity escaping (**connect3**)
r3	Simplified Ruf escaping (**ruf3**)
r5	Adapted Ruf escaping (**ruf5**)

Table 4.7: Abbreviations used in Table 4.8

data when programs are executed on the PowerPC platform. We expect the static and dynamic counts for single threaded programs should be identical for both platforms. However, as we see from the tables, there are three different counts when comparing both platforms:

- The dynamic counts in column 8 and 26 of *javac*. To investigate the difference, we identify the locations where these objects are allocated and then insert statements manually to record the number of times these objects are allocated. We found that the numbers are different for different runs (even for the same platform). A similar non-deterministic behavior has been reported by the Jikes RVM Researchers mailing list[IBM03]. The reason suggested there was that the benchmark makes use of `java.util.Hashtable`. Since hashcodes for the same object may vary in different runs, the behavior in the use of `java.util.Hashtable` may be different.

- The static and dynamic counts at column 26 of *mpegaudio*. We think this is due to some minor bugs in the counting mechanism provided by the Jikes RVM. We found that the counter has been inserted but the system fail to report the final values of them. We verified that if the final values are reported correctly by including the missed values, both platforms should have the same static and dynamic values.

Table 4.7 shows some abbreviations used in classifying objects. Using these abbreviations, objects are classified into 32 types as shown in Table 4.8. For example, if an object is marked as not escaping by Bogda's, escaping by both full connectivity and extended

Type	Meaning	Type	Meaning
0	!b+!c2+!c3+!r3+!r5	16	b+!c2+!c3+!r3+!r5
1	!b+!c2+!c3+!r3+r5	17	b+!c2+!c3+!r3+r5
2	!b+!c2+!c3+r3+!r5	18	b+!c2+!c3+r3+!r5
3	!b+!c2+!c3+r3+r5	19	b+!c2+!c3+r3+r5
4	!b+!c2+c3+!r3+!r5	20	b+!c2+c3+!r3+!r5
5	!b+!c2+c3+!r3+r5	21	b+!c2+c3+!r3+r5
6	!b+!c2+c3+r3+!r5	22	b+!c2+c3+r3+!r5
7	!b+!c2+c3+r3+r5	23	b+!c2+c3+r3+r5
8	!b+c2+!c3+!r3+!r5	24	b+c2+!c3+!r3+!r5
9	!b+c2+!c3+!r3+r5	25	b+c2+!c3+!r3+r5
10	!b+c2+!c3+r3+!r5	26	b+c2+!c3+r3+!r5
11	!b+c2+!c3+r3+r5	27	b+c2+!c3+r3+r5
12	!b+c2+c3+!r3+!r5	28	b+c2+c3+!r3+!r5
13	!b+c2+c3+!r3+r5	29	b+c2+c3+!r3+r5
14	!b+c2+c3+r3+!r5	30	b+c2+c3+r3+!r5
15	!b+c2+c3+r3+r5	31	b+c2+c3+r3+r5

Table 4.8: Object types used in Table 4.9 and Table 4.10 to classify objects

connectivity analyses, but not by either of Ruf's analyses, the fact can be represented as !b+c2+c3+!r3+!r5 and the object is classified as type 12. By consulting Table 4.9 and Table 4.10 we find that there are indeed objects of type 12 created in *mtrt*.

4.5.1.1 Static Counts

The *static count* for a given type of object records the number of object creation sites that create that kind of object. For example, both Table 4.9 and Table 4.10 tell us that the static count of type 10 objects for *jess* is 5, meaning that there are five instructions in *jess* that create objects of type 10.

Using the static counts shown in Table 4.9 and Table 4.10, we can infer some relationships between the analyses (with respect to the benchmark programs). For example, we can infer, as expected, that adapted Ruf analysis is more precise than the simplified Ruf analysis. This can be done by showing r5 $\Rightarrow$ r3 (if adapted Ruf analysis marks an

BENCHMARK	0	3	8	10	12	15	16
_201_compress	1/0	-	24/145	1/125	-	-	-
_202_jess	12/2105	-	68/14544	5/5	-	-	-
_209_db	3/405	-	34/1900	6/30	-	-	-
_213_javac	3/94220	-	101/1635715	27/201440	-	-	-
_222_mpegaudio	9/100	-	25/30	-	-	-	-
_227_mtrt	46/27724302	-	-	-	21/665	2/20	65/1720993
_228_jack	10/4165	-	129/1908440	2/3749860	-	-	2/85
moldyn	6/6	-	-	-	22/5	-	-
montecarlo	4/60001	-	-	-	49/485207	1/1	-
raytracer	10/37094616	3/12	-	-	23/8	-	-

BENCHMARK	18	19	24	26	28	30	31
_201_compress	-	-	14/2125	21/521	-	-	-
_202_jess	-	-	-	262/39494340	-	-	-
_209_db	-	-	-	24/769337	-	-	-
_213_javac	-	-	1/1520	537/16752321	-	-	-
_222_mpegaudio	-	-	-	1028/6544	-	-	-
_227_mtrt	-	-	-	-	-	-	29/2850096
_228_jack	-	-	4/4590	203/1507735	-	-	-
moldyn	-	10/32830	-	-	-	1/1	7/10
montecarlo	-	2/4	-	-	24/60001	3/4	7/180006
raytracer	5/273	29/19139639	-	-	-	1/1	7/8913003

Table 4.9: Classify objects created for Intel platform. First numbers are the static counts while the second numbers are the dynamic counts

BENCHMARK	0	3	8	10	12	15	16
_201_compress	1/0	-	24/145	1/125	-	-	-
_202_jess	12/2105	-	68/14544	5/5	-	-	-
_209_db	3/405	-	34/1900	6/30	-	-	-
_213_javac	3/94220	-	101/1635739	27/201440	-	-	-
_222_mpegaudio	9/100	-	25/30	-	-	-	-
_227_mtrt	46/27341003	-	-	-	21/665	2/20	65/1716024
_228_jack	10/4165	-	129/1908440	2/3749860	-	-	2/85
moldyn	6/6	-	-	-	22/5	-	-
montecarlo	4/60000	-	-	-	49/485149	1/1	-
raytracer	10/35838886	3/11	-	-	23/6	-	-

BENCHMARK	18	19	24	26	28	30	31
_201_compress	-	-	14/2125	21/521	-	-	-
_202_jess	-	-	-	262/39494340	-	-	-
_209_db	-	-	-	24/769337	-	-	-
_213_javac	-	-	1/1520	537/16752338	-	-	-
_222_mpegaudio	-	-	-	1032/6548	-	-	-
_227_mtrt	-	-	-	-	-	-	29/2846502
_228_jack	-	-	4/4590	203/1507735	-	-	-
moldyn	-	10/23600	-	-	-	1/1	7/10
montecarlo	-	2/4	-	-	24/59998	3/4	7/179998
raytracer	5/273	29/18872583	-	-	-	1/1	7/8852836

Table 4.10: Classify objects created for PowerPC platform. First numbers are the static counts while the second numbers are the dynamic counts

object as escaping, the simplified Ruf analysis will mark it as escaping as well), which can be shown by the absence of objects classified as both r5 and !r3. This can be shown by the absence of type 1, 5, 9, 13, 17, 21, 25 and 29 objects. Checking the static counts, we see that there do not exist columns for type 1, 5, 9, 13, 17, 21, 25 and 29 objects, so the adapted Ruf analysis is more precise. Similar arguments can be made to show that the extended connectivity analysis is indeed more precise than the full connectivity analysis. Also, the presence of columns 3, 12 and 16 suggests that none of the analyses are absolutely more precise than others. For example, the presence of column 3 infers that there exist cases (in *raytracer*) where adapted Ruf's analysis marks an object as escaping but Bogda's and connectivity analyses do not.

4.5.1.2 Dynamic Counts

The *dynamic counts* for a given type of object records the number of objects created that are classified as that type. For example, both Table 4.9 and Table 4.10 tell us that the dynamic count of type 10 objects for *db* is 30, meaning that *db* creates 30 objects of type 10.

While static counts give some insight about the precision of the analyses, the counts may not reflect the programs' runtime behavior. For example, in the case of *raytracer*, there are 10 object creation sites marked as creating type 0 objects and there are 23 object creation sites marked as creating type 12 objects. However, at runtime, there are more than 35000000 type 0 objects created while there are less than 10 type 12 objects created.

From the dynamic counts, we have the following observations:

- *compress*. Most created objects are of type 24. This means that most objects are marked as escaping by Bogda's and full connectivity analysis but are marked as not escaping by the extended connectivity and the two Ruf's analyses.

- *jess*, *db*, *javac* and *mpegaudio*. Most created objects are of type 26. This means that most objects are marked as escaping by Bogda's, full connectivity and simplified Ruf's analysis but are marked as not escaping by extended connectivity and adapted Ruf's analyses.

- *mtrt* and *raytracer*. Most created objects are of type 0. This means that most objects are not marked as escaping by all the analyses.

- *jack*. Most created objects are of type 10. This means that most objects are marked as escaping by full connectivity and simplified Ruf's analyses but are marked as not escaping by Bogda, extended connectivity and adapted Ruf's analyses.

- *moldyn*. Most created objects are of type 19. This means that most objects are marked as escaping by Bogda's and both Ruf's analyses but are marked as not escaping by both connectivity analyses.

- *montecarlo*. Most created objects are of type 12. This means that most objects are marked as escaping by both connectivity analyses but are marked as not escaping by Bogda's and both Ruf's analyses.

We can see the extended connectivity is very precise for most benchmark programs except *montecarlo*. Also, the adapted Ruf's analysis is very precise for most benchmark programs except *moldyn*. Bogda's analysis is very precise for *mtrt*, *raytracer*, *jack* and *montecarlo*. The full connectivity is precise for *mtrt*, *raytracer* and *moldyn*. Finally, the simplified Ruf's analysis is precise for *compress*, *mtrt*, *raytracer* and *montecarlo*.

Note that the dynamic counts reflect the number of object *created*. It does not reflect the access pattern of objects. It is possible an analysis marked just a few object as escaping but those objects are accessed very frequently, lowering the performance of fence inserted application. To evaluate the performance impact of escape analyses, data are reported in Section 4.5.2.

4.5.2 Fences Inserted to enforce SC using Thread Escape Analysis

In this section, we focus on comparing the differences of escape analyses with respect to fence insertion to enforce SC. In Section 4.5.2.1 we present static fence counts when using different escape analyses. After that we present dynamic fence counts data in Section 4.5.2.2 and finally we present the application execution times and slowdowns in Section 4.5.2.3. Since the interpretations of fence counts and timing data are similar, we will first present the data and then do the interpretation after Section 4.5.2.3.

4.5.2.1 Static Fence Counts

Table 4.11 shows the static counts of fences. It records the number of fences *inserted* in the programs. As we can see the numbers for both platforms are similar. However, they are not exactly the same because the fence insertion algorithms are different for different platforms. They take advantage of different architectural characteristics. Despite that, a more precise escape analysis causes fewer fences to be inserted as confirmed by the data.

4.5.2.2 Dynamic Fence Counts

Table 4.12 shows the dynamic fence counts. It records the number of fences *executed* when the programs are running. We can see the static fence counts are related to the number of fences executed at runtime. For example, consider *javac*where the static fence counts for full connectivity analysis is much greater than that for extended connectivity analysis and similar behavior is observed for dynamic counts for *javac*. However, the quantitative differences of static fence counts are not necessarily the same as that of dynamic fence counts. For example, consider *compress*where the static fence count for Bogda's analysis is less than twice that for the extended connectivity analysis but the dynamic counts for Bogda's analysis is more than 50 times of that of extended connectivity analysis.

BENCHMARK	S(connect2)	S(connect3)	S(bogda)	S(ruf3)	S(ruf5)
_201_compress	1506	679	1262	974	679
_202_jess	4922	689	4293	4222	689
_209_db	1719	590	990	1057	590
_213_javac	13069	1166	10841	11115	1170
_222_mpegaudio	8254	798	8096	8105	803
_227_mtrt	1173	1173	2450	997	997
_228_jack	6570	675	4146	4650	675
moldyn	968	968	1457	1461	1461
montecarlo	1677	1677	981	790	722
raytracer	1036	1036	1172	1182	1158

(a) Intel Platform

BENCHMARK	S(connect2)	S(connect3)	S(bogda)	S(ruf3)	S(ruf5)
_201_compress	1373	556	1129	845	556
_202_jess	5104	1171	4486	4403	1173
_209_db	1725	618	1004	1071	618
_213_javac	13173	1826	10951	11219	1838
_222_mpegaudio	8258	913	8099	8108	918
_227_mtrt	1265	1265	2460	1089	1089
_228_jack	6719	974	4299	4801	976
moldyn	978	978	1461	1465	1465
montecarlo	1690	1690	996	805	737
raytracer	1058	1058	1176	1186	1162

(b) PowerPC Platform

Table 4.11: Static fence counts

BENCHMARK	D(connect2)	D(connect3)	D(bogda)	D(ruf3)	D(ruf5)
_201_compress	1.51735e+10	265723709	1.51735e+10	2.34777e+09	265723723
_202_jess	1808372170	42060856	1808180546	1808153231	42060856
_209_db	1705012963	812457	1168632291	1182428572	812457
_213_javac	1352309030	69664708	973522787	1026356623	69655402
_222_mpegaudio	1.36863e+10	781257377	1.36402e+10	1.38202e+10	781266165
_227_mtrt	19984436	19972459	1810604198	19976208	19974240
_228_jack	451078489	20766545	386632536	393931292	20766495
moldyn	1980726762	872997309	1.98864e+10	1.97922e+10	1.94185e+10
montecarlo	2.80338e+09	2.82196e+09	532396601	533682231	536327969
raytracer	932888205	949044196	2.83907e+10	2.82577e+10	2.87009e+10

(a) Intel Platform

BENCHMARK	D(connect2)	D(connect3)	D(bogda)	D(ruf3)	D(ruf5)
_201_compress	1.51735e+10	265728618	1.51735e+10	2.34778e+09	265728608
_202_jess	1808372044	235550824	1808183710	1808153237	235550820
_209_db	1705012933	15339247	1168638661	1182431502	15339247
_213_javac	1382801274	367816016	1037187195	1056942764	367820644
_222_mpegaudio	1.36863e+10	1067565473	1.36402e+10	1.38202e+10	1067574261
_227_mtrt	210643599	210318334	1738557774	209899191	210126781
_228_jack	451195990	58469549	388982065	394051543	58470039
moldyn	438220258	426972206	8.76589e+09	8.81127e+09	8.7397e+09
montecarlo	2.47588e+09	2.41452e+09	678088188	681266938	680966770
raytracer	1764423727	1786246544	1.56e+10	1.55852e+10	1.56259e+10

(b) PowerPC Platform

Table 4.12: Dynamic fence counts

4.5.2.3 Application Execution Times and Slowdowns

Table 4.13 shows the execution time of the benchmarks using different escape analyses. This includes the baseline case where no fences are inserted due to enforcement of SC using escape analysis. Using the data in Table 4.13, the slowdowns data is shown in Table 4.14 and is graphically plotted in Figure 4.6.

We can see from Table 4.14 that the slowdown behavior of both the Intel and PowerPC platforms are similar.

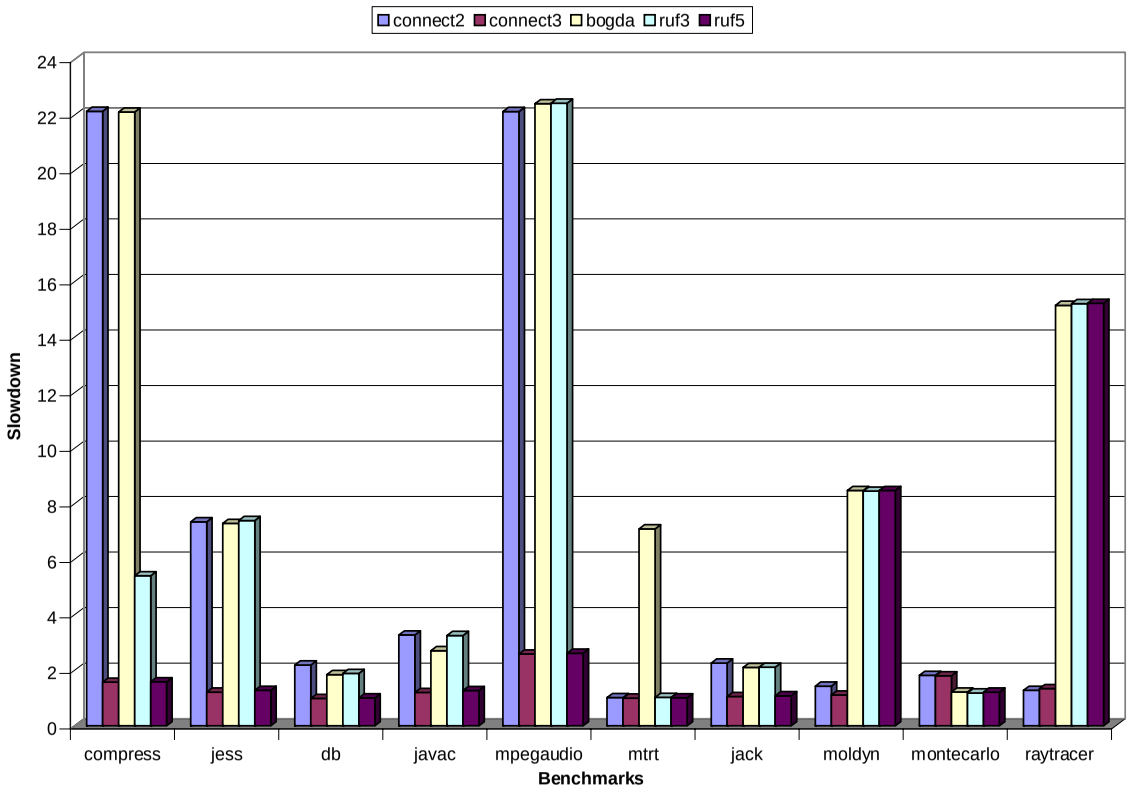
Benchmark	**base**	**connect2**	**connect3**	**bogda**	**ruf3**	**ruf5**
_201_compress	11.0315	244.28175	17.34725	243.8885	59.64125	17.5355
_202_jess	4.67325	34.34325	5.676	34.07575	34.56125	5.9705
_209_db	25.926	56.832	25.56375	47.8855	48.99875	26.117
_213_javac	10.907	35.6875	13.124	29.53675	35.49025	13.8465
_222_mpegaudio	9.91925	219.51725	25.696	222.246	222.4535	25.947
_227_mtrt	3.6785	3.7495	3.66875	26.09775	3.7745	3.713
_228_jack	6.33	14.318	6.68725	13.30475	13.3595	6.8465
moldyn	72.759	103.958	80.817	616.887	615.129	616.896
montecarlo	71.898	130.81	129.737	87.655	85.2	87.277
raytracer	55.533	71.085	74.548	841.51	844.803	844.997

(a) Intel Platform

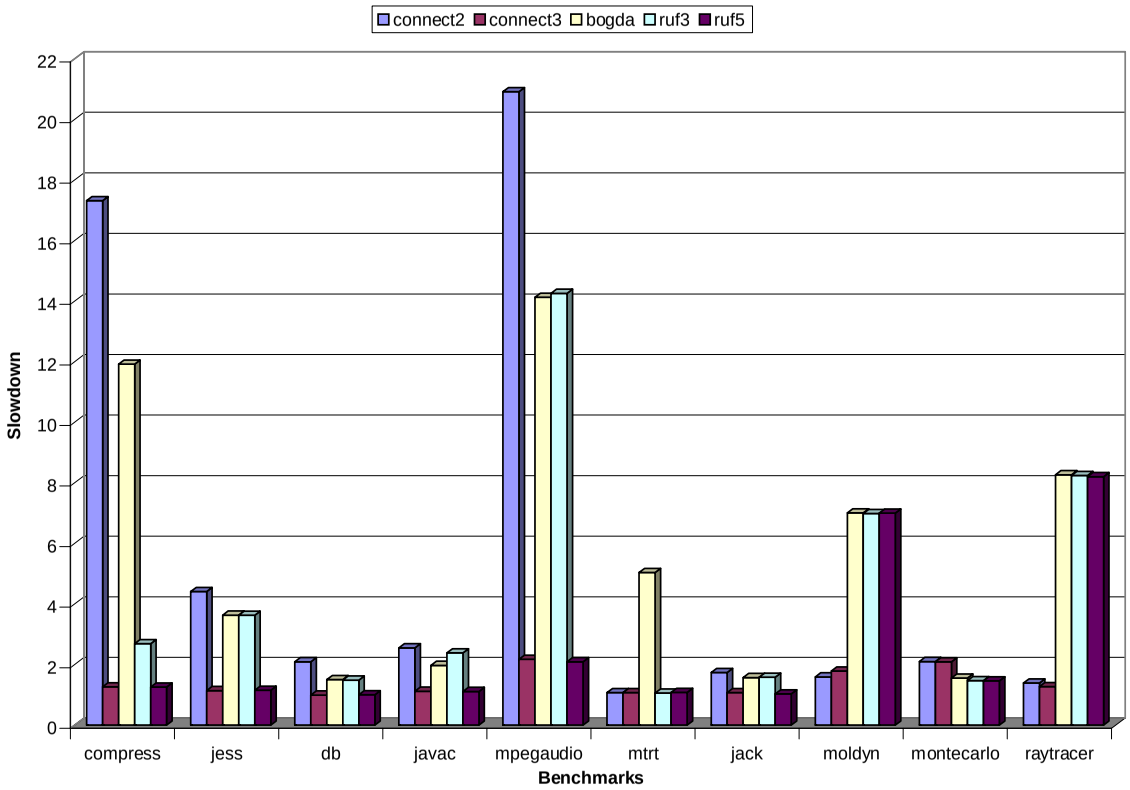
Benchmark	**base**	**connect2**	**connect3**	**bogda**	**ruf3**	**ruf5**
_201_compress	20.96225	363.0475	26.5075	249.7985	56.44675	26.35575
_202_jess	12.166	53.67075	13.877	44.203	44.17225	14.01225
_209_db	35.4985	74.36925	35.31075	53.48975	52.81775	35.596
_213_javac	19.88425	50.74875	22.2815	39.2605	47.4545	22.13275
_222_mpegaudio	15.9815	334.38075	34.796	225.7545	227.913	33.46975
_227_mtrt	5.555	5.961	5.97125	27.99075	5.89975	6.00375
_228_jack	14.4135	25.0865	15.48175	22.64775	22.7945	14.88375
moldyn	67.689	107.809	120.98	474.45	473.025	473.996
montecarlo	104.318	219.325	218.322	161.971	153.435	152.477
raytracer	156.448	217.858	199.082	1293.485	1290.055	1283.624

(b) PowerPC Platform

Table 4.13: Performance of benchmarks: time in seconds



(a) Intel Platform



(b) PowerPC Platform

Figure 4.6: Slowdown Graph

Benchmark	**connect2**	**connect3**	**bogda**	**ruf3**	**ruf5**
_201_compress	22.14	1.573	22.11	5.406	1.59
_202_jess	7.349	1.215	7.292	7.396	1.278
_209_db	2.192	0.986	1.847	1.89	1.007
_213_javac	3.272	1.203	2.708	3.254	1.27
_222_mpegaudio	22.13	2.591	22.41	22.43	2.616
_227_mtrt	1.019	0.9973	7.095	1.026	1.009
_228_jack	2.262	1.056	2.102	2.111	1.082
moldyn	1.429	1.111	8.478	8.454	8.479
montecarlo	1.819	1.804	1.219	1.185	1.214
raytracer	1.28	1.342	15.15	15.21	15.22

(a) Intel Platform

Benchmark	**connect2**	**connect3**	**bogda**	**ruf3**	**ruf5**
_201_compress	17.32	1.265	11.92	2.693	1.257
_202_jess	4.412	1.141	3.633	3.631	1.152
_209_db	2.095	0.9947	1.507	1.488	1.003
_213_javac	2.552	1.121	1.974	2.387	1.113
_222_mpegaudio	20.92	2.177	14.13	14.26	2.094
_227_mtrt	1.073	1.075	5.039	1.062	1.081
_228_jack	1.74	1.074	1.571	1.581	1.033
moldyn	1.593	1.787	7.009	6.988	7.003
montecarlo	2.102	2.093	1.553	1.471	1.462
raytracer	1.393	1.273	8.268	8.246	8.205

(b) PowerPC Platform

Table 4.14: Performance of benchmarks: slowdowns

4.5.2.4 Interpretation

Starting from Section 4.5.2.5, we are going to interpret the data related to fence insertion. As described in Section 3.6.1, the following properties of the analysis algorithms are important to the difference in the performances of the fence inserted programs:

- Handling of thread creation pattern in Java. This refers to whether the analysis can identify the thread local objects which are reachable from `Runnable` objects but not accessed concurrently. Connectivity analyses (**connect2** and **connect3**) have this property so they have good performance for *moldyn* and *raytracer*.
- Handling of recursive methods. This refers to whether the analysis performs fix-point computation for recursive method calls. Bogda's analysis is more precise than other analyses because of this property.
- Keeping track of precise alias information. This refers to the ability of the analysis to maintain alias information. Ruf's analyses (**ruf3** and **ruf5**) are more precise than other analyses because of this properties.
- Keeping track of threads accessing escaping objects. This refers to whether the analysis can identify objects reachable from the static fields not accessed by multiple threads. This property enable good performance single threaded programs. Only **connect3** and **ruf5** has this property.

4.5.2.5 Extended Connectivity analysis (connect3)

Extended connectivity analysis has good performance for single threaded benchmarks (*compress*, *jess*,*db*,*javac*,*mpegaudio* and *jack*) because the extension can correctly identify that the escaping objects are accessed by a single thread. Note that we have slowdowns for single-threaded benchmarks like *compress* and *mpegaudio*. The reason is that they

access static arrays via `getstatic` or `putstatic` operations and our analyses assume that these accesses are shared accesses, so our implementation conservatively assumes that a delay is needed for each `getstatic` or `putstatic` operation.

For *mtrt*, good performance is observed because the frequently accessed objects do not connect to an escaping object.

For *moldyn*, extended connectivity analysis outperforms Bogda's and Ruf's analyses because it manages to identify a frequently accessed object that is not escaping. Bogda's and Ruf's analyses cannot do that because the object is stored in a `Runnable` object. A moderate slowdown is observed because in a frequently executed method, the program accesses a shared array and fences are inserted there.

As in the case of *moldyn*, extended connectivity analysis outperforms Bogda's and Ruf's analyses for *raytracer* because it can identify frequently accessed object as not shared even though they are reachable from a `Runnable` object. It experiences moderate slowdown because of the following code in the program:

```
1  static Vec voidVec;
2  Vec shade () {
3    ...
4    ... = trace ();
5    ...
6    // work on col
7    ...
8    return col;
9  }
10 Vec trace () {
11   ...
12   return shade (..);
13   ...
14   return voidVec;
15 }
```

The method `shade` is executed moderately frequently, so fences in this method contribute to slowdowns. As we can see `shade` and `trace` are mutually recursive, and their summaries are not cloned when performing analysis. Notice that lines 8 and 12 cause the return values of `shade` and `trace` to be merged. Since the object `voidVec` is static and

is accessed by multiple threads, the return value of `shade` is considered shared too. This causes `col` escaping, which leads to the insertion of fences inside `shade`.

The only benchmark that extended connectivity analysis does not outperform the other analyses on is *montecarlo*. This is because most objects (including frequently accessed objects) in *montecarlo* has a field pointing to a shared `String` object. Since they are connected to a shared object, they are considered escaping and fences are inserted to frequently executed methods.

4.5.2.6 Full Connectivity analysis (connect2)

Full Connectivity analysis behaves similar to extended connectivity analysis for multi-threaded benchmarks (*mtrt*, *moldyn*, *montecarlo* and *raytracer*).

For single threaded benchmarks, it has similar precision as the simplified Ruf's analysis except for *compress*. This is because for many benchmarks like *jess*, *db*, *javac*, *mpegaudio* and *jack*, the frequently accessed objects are found to be escaping by the simplified Ruf's analysis, so the full connectivity analysis does not introduce many extra fences.

Benchmark *compress* shows the difference in precision between simplified Ruf's analysis and the full connectivity analysis. In an frequently executed methods `decompress`, many objects reachable from `this` are accessed. While most of the objects reachable from `this` are found to be non-escaping, the input and output buffers reachable from `this` are found to be escaping for both full connectivity analysis and the simplified Ruf's analysis. Simplified Ruf's analysis, representing alias information most precisely, can distinguish escaping objects from non-escaping ones by representing them by different alias sets. Full connectivity analysis, however, represents them by one connectivity set. Since one of the object are found to be escaping, the whole connectivity set is assumed to be escaping as well. This cause all objects represented by this connectivity set considered escaping, causing. extra fences inserted in hotspot.

4.5.2.7 Bogda's analysis (bogda)

Bogda's analysis causes slowdowns for *moldyn*, *raytracer* and *montecarlo*for the reason described in extended connectivity analysis.

Unlike other analyses, Bogda analysis causes slowdown for *mtrt*. This is because the frequently accessed object is an octree node. Since the node can be reached by more than 1 field references, Bogda's analysis assumes it is escaping even though it is not reachable by static fields nor `Runnable` objects.

It causes a slowdown for *compress*. This is because the following pattern:

```
1  class Decompressor {
2    ...
3    De_Stack de_stack;
4
5    ...
6    class De_Stack {
7      ...
8    }
9  }
```

The frequently accessed object is of type `Decompressor`. This object has a field `de_stack` which is of type `De_Stack`. Note that `De_Stack` is an inner class declared in `Decompressor`, so for each `De_Stack` object O the Java compiler introduce a compiler generated field `this$0` so that O.`this$0.de_stack` $= O$. Because of this cyclic structure, Bogda's analysis assumes all `Decompressor` objects escaping. This include the frequently accessed object, as a result, causes slowdown.

It causes slowdown for *jess* because the frequently called method is invoked with receiver `this._succ[..].node` and the receiver is frequently accessed. We can see the passed object is reachable by more than 1 field references, so Bogda's analysis assumes it is escaping.

It causes slowdown for *db* because in a hot method, it accesses data like `this.index[...].items.elementData[..]` frequently. Bogda's analysis assumes

that `this.index[..]` as escaping, so `this.index[...].items.elementData[..]` is escaping and fences are inserted to hotspot.

It causes *javac* slowdown because for a frequently executed method `ScannerInputStream.read()`, the frequently accessed object (referenced by `this`) is marked as escaping. This is because of the following code pattern:

```
1  // ScannerInputStream is of type InputStream
2  class ScannerInputStream {
3    int read () {
4      ...
5      // super.in is of type InputStream
6      super.in.read ();
7      ...
8      // using fields of this object a lot
9      ...
10   }
11 }
12 // FilterInputStream is of type InputStream
13 class FilterInputStream {
14   int read () {
15     // this.in is of type InputStream
16     return (this.in.read ());
17   }
18 }
```

This benchmark shows the imprecision due to using object type to resolve method invocation. In real execution, the `ScannerInputStream` object is `not` stored inside a `FilterInputStream` object. However, because of the use of type base method resolution, the above two methods look like they are mutually recursive. It looks as if the following call is possible:

1. In `ScannerInputStream.read`, `super.in.read` is called
2. `super.in` could be a `FilterInputStream` object, so `FilterInputStream.read` is called in line 6.
3. In `FilterInputStream.read`, `this.in.read` is called
4. `this.in` could be a `FilterInputStream` object, so `FilterInputStream.read` is called in line 16 again.

5. ...

6. In `FilterInputStream.read`, `this.in.read` is called

7. `this.in` is a `ScannerInputStream` object, so `ScannerInputStream.read` is called in line 16 again.

Because of this imprecision in method resolution, the analysis assumes that the `this` of `FilterInputStream.read` may be O.`in`.`in`.....`in` for some object O and it assumes `this` is escaping because it can be reached by more than 1 field reference.

Benchmark *mpegaudio* experiences slowdown because the frequently called method works on objects which are reachable from a static field,so these objects are assumed to be escaping.

The slowdown of *jack* stems from a reuse of variable to reference both an non-escaping object and an escaping exception object. that non-escaping object is reaching an object accessed in a frequently executed method. Because of that Bogda's analysis conservatively assume that frequently accessed object escaping.

4.5.2.8 Adapted Ruf's analysis (ruf5)

Adapted Ruf's analysis has good performance for *compress*, *jess*,*db*,*javac*,*mpegaudio*, *jack* and *mtrt* for similar reason as extended connectivity analysis.

It causes slowdowns for *moldyn* and *raytracer* because the hotspot methods accesses objects reachable from `Runnable` objects which is assumed to be accessed by multiple thread.

It outperforms extended connectivity analysis for *montecarlo* because it represents alias information precisely, so it can distinguish the shared string from other non-escaping objects. It still causes moderate slowdown because in the hotspot, a shared array is accessed. The shared array is accessed in the following way:

1. A thread T is started by its creator thread P.

2. Thread T creates an array A and performs computation on that array.

3. At the end, thread T publish the array A by making it reachable from a static field.

4. Thread T finishes execution

5. The thread P processes the result of T by accessing the array A.

As we can see, though the shared array A is accessed by both thread T and thread A, there is no concurrent accesses. Adapted Ruf's analysis do not recognize this fact, so it causes fences inserted for code where A is accessed.

4.5.2.9 Simplified Ruf's analysis (ruf3)

Its behavior for *moldyn*, *raytracer*, *montecarlo* and *mtrt* is similar to that of adapted Ruf's analysis.

It causes slowdown for *compress* because the input and output buffers are found to escaping and these buffers are accessed in the hotspot. The slowdown is not as big as full connectivity and Bogda's analysis because its field sensitivity allows it to discover many non-escaping object used in the hotspot. Note that the buffers are not actually reachable from static fields.

Like *compress*, *jess*, *db*, and *javac* experience slowdowns because for hotspot methods, there are some frequently accessed objects found to be escaping. These objects are not really reachable from static fields at runtime. They are marked as escaping because they have been passed to/returned from recursive functions. Some illustrative cases have been shown in Section 3.7. The imprecision in method resolution makes the analysis more imprecise because more methods are considered recursive although they are not.

Benchmarks *mpegaudio* and *jack* experience slowdown for the reason described in Bogda's analysis.

4.5.2.10 Summary

We present the summary of the issues causes bad performance of appplication program with respect to different escape analyses in Table 4.15 where the issues are numbered as follows:

1. Handling of thread creation pattern in Java
2. Handling of recursive methods
3. Keeping track of precise alias information
4. Keeping track of threads accessing escaping objects
5. Precision of method resolution
6. Reuse of variables in generated IR

The bold numbers represents the issues are contributing to good performances while the non-bold numbers represents the issues are contributing to bad performance.

Benchmark	connect2	connect3	bogda	ruf3	ruf5
_201_compress	3, 5	**4**	3	5	**4**
_202_jess	2, 5	**4**	3	2, 5	**4**
_209_db	2, 5	**4**	3	2, 5	**4**
_213_javac	2, 5	**4**	3, 5	2, 5	**4**
_222_mpegaudio	4	**4**	4	4	**4**
_227_mtrt	**3**	**3**	3	**3**	**3**
_228_jack	6	**4**	6	6	**4**
moldyn	**1**	**1**	1	1	1
montecarlo	3	3	**3**	**3**	**3**
raytracer	**1**	**1**	1	1	1

Table 4.15: A summary of issues of difference escape analyses on performance of application programs

4.5.3 Synchronization Removal Driven by Thread Escape Analysis

In this section, we report the effect of escape analyses on synchronization removal. Table 4.16 shows the number of object allocation sites marked as creating thread local objects. Table 4.17 shows the number of thread local objects participated in synchronization (lock and unlock operations). Since the objects are thread local, the lock and unlock operations could have been omitted, but they are inserted due to imprecision of the analysis. We can see the numbers for both platforms are very similar, so we can have our discussion with distinguishing the platforms:

- Full Connectivity analysis. Unlike the case when the analyses is applied for fence insertion, the behavior of full connectivity analysis is quite different from simplified Ruf's analysis. This shows the importance of being field sensitive for synchronization removal.

- Extended Connectivity analysis manages to remove many synchronization operations for single threaded programs because it can identify that the escaping objects are synchronized by a thread only. For multi-thread programs, its behavior is similar to the full connectivity analysis.

- Simplified Ruf's analysis. It can remove many synchronization even being conservative — objects reachable by `Runnable` objects or static fields are escaping. This shows the importance of field sensitivity as described in the full connectivity analysis.

- Adapted Ruf's analysis removes the greatest number of synchronization for most programs except *mtrt*. Comparing simplified Ruf's analysis and Adapted Ruf's analysis for single threaded programs, we see there is a big increase in synchronization removed. This shows that many object reachable from the static fields are being synchronized on.

- Bogda's analysis. For some programs like *compress*, *db*, *mtrt*, it performs better in removing synchronization while for other programs the precision is similar to that of Simplified Ruf's analysis. This suggest that limiting analysis to object reachable by 1 level of field reference is good enough for many programs. Performing the analysis in a fixpoint computation gives extra precision. For example, it has better precise for *mtrt* even if being compared with adapted Ruf's analysis.

As a whole, the data suggest that:

- being field sensitive is important;

- limiting the lattice to 1-level field reference is good enough;

BENCHMARK	S(connect2)	S(connect4)	S(bogda)	S(ruf3)	S(ruf4)
_201_compress	1	61	26	39	61
_202_jess	13	368	93	109	645
_209_db	4	80	53	48	80
_213_javac	3	674	131	114	805
_222_mpegaudio	9	1066	34	36	1073
_227_mtrt	110	110	68	131	144
_228_jack	12	350	141	145	351
moldyn	16	16	28	32	47
montecarlo	7	7	55	79	125
raytracer	48	48	36	38	81

(a) Intel Platform

BENCHMARK	S(connect2)	S(connect4)	S(bogda)	S(ruf3)	S(ruf4)
_201_compress	1	61	26	39	61
_202_jess	13	368	93	109	645
_209_db	4	80	53	48	80
_213_javac	3	674	131	114	805
_222_mpegaudio	9	1066	34	36	1073
_227_mtrt	110	110	68	131	144
_228_jack	12	350	141	145	351
moldyn	16	16	28	32	49
montecarlo	7	7	55	79	129
raytracer	48	48	36	38	81

(b) PowerPC Platform

Table 4.16: Static number of object allocation site marked as local

- detecting object synchronized by a single thread is important; and
- being a iterative fix point analysis is important

BENCHMARK	D(connect2)	D(connect4)	D(bogda)	D(ruf3)	D(ruf4)
_201_compress	0	1788	655	405	1788
_202_jess	1320	24519497	94617	94666	24503865
_209_db	0	240497757	182420	2030	240497858
_213_javac	0	118679796	18761053	18328385	118681866
_222_mpegaudio	0	20282	90	90	20282
_227_mtrt	0	0	3475902	2044	2045
_228_jack	0	61522476	4056159	4062965	61522458
moldyn	0	0	29	29	29
montecarlo	198107102	206572805	178149900	190836750	178988416
raytracer	580	580	40	40	628

(a) Intel Platform

BENCHMARK	D(connect2)	D(connect4)	D(bogda)	D(ruf3)	D(ruf4)
_201_compress	0	1788	655	405	1788
_202_jess	1320	24519486	94617	94666	24503865
_209_db	0	240497979	182420	2030	240497858
_213_javac	0	118681866	18761053	18328385	118681866
_222_mpegaudio	0	20282	90	90	20282
_227_mtrt	0	0	3475902	2044	2045
_228_jack	0	61522499	4056159	4062965	61522458
moldyn	0	0	29	29	29
montecarlo	194461463	197494188	178149900	190836750	178988416
raytracer	580	580	40	40	628

(b) PowerPC Platform

Table 4.17: Dynamic number of synchronization removed

Chapter 5

Conclusion

In this thesis, we have presented the Pensieve Compiler System. The system presented in this thesis focuses enforcing SC on top of the Intel and PowerPC platforms. We also presented the fast thread escape analysis, the connectivity analysis, implemented in the system. From the data shown in Chapter 4, we can see that connectivity analysis is faster than the escape analysis presented in [BH99, Ruf00]. The analysis time of connectivity analysis is quite promising usable for a dynamic compilation system (for Intel platform from 0.6 sec to 41 sec; for PowerPC from 1.5 sec to 94 sec). With the incremental analysis enable, the analysis time can be reduced up to 50%. When the thread escape analysis is used to enforce SC by inserting fences, the application performance is also quite promising for both the Intel and PowerPC platforms. We performed qualitative comparison between connectivity analysis, Bogda's and Ruf's analyses in Section 3.6. In Section 4.5.1 we performed a quantitative comparison between these analyses. To our understanding, this is the first quantitative comparison between different escape analyses.

5.1 Limitation

A limitation of the Pensieve system is the lack of on-stack replacement (OSR) mechanism [HU94][1]. On-stack replacement is a technique to replace a method while the method is running. As described in Section 3.9, method invalidation is needed in general for a dynamic compilation system which performs compilation using information computed by *whole program analyses*. In our experiments, we found that all the methods invalidated due to connectivity analysis are all *not* running when invalidation is performed. However, in case of Ruf's analysis, we do find invalidations performed for methods that are running. In our implementation, the running methods are *not* replaced, so the old and over-optimistic code is executed when the program returns to that method. Therefore, the application performances for Ruf's analyses presented in Chapter 4 are better than they should be. Although the performance data for connectivity and Bogda's analyses are still valid, the system will be more complete if OSR is implemented.

5.2 Open Problem

With the introduction of the connectivity analysis, there are more areas to explore.

5.2.1 Improve Precision of Connectivity Analysis

Since the connectivity analysis is efficient, it is useful to explore how to extend the connectivity analysis to be more precise without significant increase in analysis time. There are at least three ways to extend the connectivity analysis:

1. Perform fixpoint computation for methods inside SCC. When analyzing recursive methods inside an SCC, connectivity does *not* clone the method summary to avoid

[1]The Jikes RVM includes an implementation of OSR [FQ03] which is tailor for handling inlining, so it is not (directly) usable for our purpose.

a fixpoint computation. While this improve the analysis time, it is sacrificing the analysis precision. Unlike Ruf's analysis, the lattice of connectivity analysis is much simpler in practice. Therefore, it may be beneficial to *clone* the method summary to improve the precision of analysis.

2. Explore more criteria to be field sensitive. From the analysis time of Ruf's analysis, we can see that fully field sensitive causes high analysis time. However, we also see that being field sensitive for some important fields (e.g. `Runnable` fields) can improve the analysis precision. Therefore, it is possible to improve the precision of the analysis by finding other "important" fields and be field sensitive for them. However, the choice of such fields should be careful to avoid making the connectivity analysis too costly.

3. Make connectivity analysis adaptive. While being fully field sensitive for all method would be too co-sty (like Ruf's analysis), it would be beneficial to be more field sensitive for frequently executed methods. We expect the number of frequently executed methods should not be very large, so it should not increase the analysis time too much by being more field sensitive for these method.

5.2.2 Another application of connectivity analysis — Object Coallocation

Object co-allocation is known to be beneficial to the performance of programs with heap allocated data [CHL99]. As pointed out by [SGBS02], one of the difficulty of object co-allocation is to decide which objects to be co-allocated together.

One of the interesting property specific to our escape analysis is that when it marks an object to be non-escaping. It not only means the object is not reachable by another thread, but also means that starting from that object it is not possible to reach an object reachable by another thread. Therefore, if we allocate all objects marked as non-escaping together, the garbage collector can perform garbage collection without touching objects

in another thread (hence another processor). We expect this makes the garbage collection more efficient because it only moves objects inside the same processor memory and it does not need cooperation of garbage collector located at another processor.

Bibliography

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[AFG+00a] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming and Systems, Languages, and Applications (OOPSLA) 2000*, Minneapolis, MN, October 2000.

[AFG+00b] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. *Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of The 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–14, May 1990.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[Bac98] David F Bacon. Fast and effective optimization of statically typed object-oriented. Technical report, 1998.

[BCF+99] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the 1999 ACM Java Grande Conference*, pages 129–141, Palo Alto, CA, USA, Jun 1999.

[BH99] Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46. ACM Press, 1999.

[BHJ+03] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, Bill Pugh, P. Sadayappan, Jaime Spacco, and Chau-Wen Tseng. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In *16th Annual Workshop on Languages and Compilers for Parallel Computing*, October 2003.

[BK89] Vasanth Balasundaram and Ken Kennedy. Compile-time detection of race conditions in a parallel program. In *Proceedings of the 3rd international conference on Supercomputing*, pages 175–185. ACM Press, 1989.

[Bla98] Bruno Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37. ACM Press, 1998.

[Bla99] Bruno Blanchet. Escape analysis for object oriented languages: Applications to java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[CADG+93] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in split-c. In *Supercomputing*, pages 262–273, 1993.

[CGHS99] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, September 1999.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for java. In *Proceedings of the ACM SIGPLAN 1999 Conference on Objec-Oritented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, November 1999.

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[CKS90] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, Seattle WA, 1990.

[CKY03] Wei-Yu Chen, Arvind Krishnamurthy, and Katherine Yelick. Polynomial-time algorithms for enforcing sequential consistency in spmd programs with arrays. In *Sixteenth Annual Workshop on Languages and Compilers for Parallel Computing*, October 2003.

[CL97] M. Cierniak and W. Li. Just-in-time optimization for high-performance java programs. *Concurrency: Practice and Experience*, 9(11):1063–73, November 1997.

[CLL+02] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, June 2002.

[CLR90] Cormen, Leiserson, and Rivest. *Introduction to Algorithms*. MIT Press, Cambridge Mass., 1990.

[CS89] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, January 1989.

[DS91] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 36–48. ACM Press, 1991.

[DSB88] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. 21(2):9–21, 1988.

[EGP89] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing '89*, pages 580–588, 1989.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[FLM03a] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory processing. In *2003 ACM International Conference on Supercomputing*, June 2003.

[FLM03b] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. An optimizing and retargetable fence insertion algorithm. Technical Report ECE-HPCLab-033002, High Performance Computing Lab, School of Electrical and Computer Engineering, Purdue University, 2003.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[FQ03] S. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement, 2003.

[GFV99] K. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Annual Int'l Symp. on Computer Architecture (ISCA'99)*, 1999.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification, Second Edition*. Addison-Wesley, 1996.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of The 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–26, May 1990.

[GS93] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 1993.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the 9th International Conference on Compiler Construction*, pages 82–93. Springer-Verlag, 2000.

[Har98] Stephen Hartley. *Concurrent Programming: the Java Programming Language*. Oxford University Press, 1998.

[Hil98] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, August 1998.

[HM91] David P. Helmbold and Charles E. McDowell. Computing reachable states of parallel programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):76–84, 1991.

[HU94] Urs Holzle and David Ungar. A third-generation self implementation: Reconciling responsiveness with performance. In *Proceedings of the 9th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 1994.

[IA3] IA-32 Intel architecture software developer's manual, volume 2 and 3. URL: developer.intel.com/design/pentium4/manuals/index_new.htm.

[IBM03] IBM. Jikes rvm researchers mailing list. In *Archive at* `http://www-124.ibm.com/pipermail/jikesrvm-researchers/2003-May/001747.html` , 2003.

[Jav03] JavaMemoryModel. Java memory model mailing list. In *Archive at* `http://www.cs.umd.edu/ pugh/java/memoryModel/arc hive/`, 2003.

[jgf] The Java Grande Forum Multi-threaded Benchmarks. URL: http://www.epcc.ed.ac.uk/javagrande/threads/contents.html.

[KJCS99] Sanjeev Kumar, Dongming Jiang, Rohit Chandra, and Jaswinder Pal Singh. Evaluating synchronization on shared address space multiprocessors: methodology and performance. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 23–34. ACM Press, 1999.

[KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[KY94] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel SPMD programs. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[KY95] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–204, June 1995.

[KY96] Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38:139–144, 1996.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LAY03] Ben Liblit, Alex Aiken, and Katherine Yelick. Type systems for distributed data sharing. In *Proceedings of the Tenth International Static Analysis Symposium*, 2003.

[Lea] Doug Lea. Java specification request (jsr) 166: Concurrency utilities. In `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html` .

[Lea99a] Doug Lea. *Concurrent Programming in Java.* Addison Wesley, 1999. URL: `http://gee.cs.oswego.edu/dl/cpj`.

[Lea99b] Doug Lea. Javamemorymodel: recap: concurrent reads. December 1999. URL: `www.cs.umd.edu/ pugh/java/memoryModel/archive/0358.html`.

[LMP97] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In Z. Li, P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan, and D. Sehr, editors, *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.

[LP00] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with compilers. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, October 2000.

[LPM99] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, May 1999.

[MC93] John M. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 129–139, 1993.

[Mid95] S. Midkiff. Dependence analysis in parallel loops with i+/-k subscripts s.p. midkiff. In *1995 Workshop on Languages and Compilers for Parallel Computing*, 1995. available as Springer Lecture Notes in Computer Science Vol. N. 1033.

[Mid03] Samuel P. Midkiff. The overhead of sequential consistency in well synchronized programs. Technical Report ECE-HPCLab-033001, High Performance Computing Lab, School of Electrical and Computer Engineering, Purdue University, 2003.

[MP87] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[MP90] S. Midkiff and D. Padua. Issues in the compile-time optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol. II*, pages 105–113, August 1990.

[MP01] J. Manson and W. Pugh. Core semantics of multithreaded java. In *Proceedings of the ACM SIGPLAN 2001 ISCOPE/Java Grande Conferenc e*, pages 29–38, 2001.

[MPC90] Samuel P. Midkiff, David A. Padua, and Ron Cytron. Compiling programs with user parallelism. In *Languages and Compilers for Parallel Computing*, pages 402–422, 1990.

[MR93] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–138. ACM Press, 1993.

[NAC99] Gleb Naumovich, George S. Avruninand, and Lori A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 1999.

[NG92] Robert H. B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume II, Software, pages II:242–246, Boca Raton, Florida, 1992. CRC Press.

[NM90] Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of 1990 International Conference on Parallel Processing*, pages II.93–II.97, University Park PA, 1990.

[NM91] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP, published in ACM SIGPLAN NOTICES*, 26(7):133–144, 1991.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), March 1992.

[PPC] PowerPC microprocessor family: Programming environments manual. URL: www-3.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_970_and_970FX_Microprocessors.

[Pug99] William Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[Rin01] Martin Rinard. Analysis of multithreaded programs. *Lecture Notes in Computer Science*, 2126:1–19, 2001.

[RL98] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, 1 May 1998.

[RM94] J. Ramanujam and A. Mathew. Analysis of event synchronization in parallel programs. In *Languages and Compilers for Parallel Computing*, pages 300–315, 1994.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the ACM SIGPLAN 2001 Conference on Objec-Oritented Programming Systems, Languages, and Applications (OOPSLA)*, pages 43–55, October 2001.

[RPA97] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of The 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 199–210, June 1997.

[Ruf00] Erik Ruf. Effective synchronization removal for java. In *Conference on Programming Languages, Design, and Implementation (PLDI)*, 2000.

[SAR99] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-Reconcile & Fences (CRF): A new memory model for architects and compiler writers. In *Proceedings of The 26th Annual International Symposium on Computer Architecture (ISCA)*, pages 150–161, May 1999.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Sch89] Edmond Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 285–297, Portland, OR, June 1989.

[SD87] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proc. of the 14th Annual Int'l Symp. on Computer Architecture (ISCA '87)*, pages 234–243, 1987.

[SGBS02] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Symposium on Principles of Programming Languages*, pages 295–306, 2002.

[Sin96] Pradeep K. Sinha. *Distributed Operating Systems, Concepts and Design.* IEEE Press, 1996.

[SL94] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Operating Systems Design and Implementation*, pages 101–114, 1994.

[spe] SPEC JVM Client98 Suite. URL: http://www.specbench.org/jvm98/jvm98.

[SR01] Alexandru Sălcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2001.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[Ste90] Guy L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231. ACM Press, 1990.

[Sur04] Zehra N. Sura. *Analyzing Threads for Shared Memory Consistency.* PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[SWF+02] Z. Sura, C.-L. Wong, X. Fang, J. Lee, S.P. Midkiff, and D. Padua. Automatic implementation of programming language consistency models. In *15th Annual Workshop on Languages and Compilers for Parallel Computing*, July 2002.

[Tay83] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM*, 26(5):361–376, 1983.

[vP04] Christoph von Praun. Efficient computation of communicator variables for programs with unstructured parallelism. In *Seventeenth Annual Workshop on Languages and Compilers for Parallel Computing*, September 2004.

[vPG03] 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 (PLDI)*, June 2003.

[VR01] Frederic Vivien and Martin Rinard. Incremental pointer and escape analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Objec-Oritented Programming Systems, Languages, and Applications (OOPSLA)*, pages 187–206, November 1999.

[WSF+02] C.-L. Wong, Z. Sura, X. Fang, S.P. Midkiff, J. Lee, and D. Padua. The Pensieve project: A compiler infrastructure for memory models. *International Symposium on Parallel Architectures, Algorithms, and Networks*, May 2002.

[YSP+98] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Col ella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. URL: `http://www.cs.ucsb.edu/conferences/java98`.

[YT88] M. Young and R. M. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Trans. Softw. Eng.*, 14(10):1499–1511, 1988.

Vita

Research Interest

Program analysis and optimization of object-oriented and/or explicitly parallel programs, dynamic compilation, program optimization under different memory models.

Personal Information

Address	:	1107 West Green Street, #521 Urbana, IL 61801
Phone	:	(217) 244-5979 (O), (217) 332-2578 (H)
Email	:	cwong1@uiuc.edu
Citizenship	:	Hong Kong (US permanent resident)

Education

Jan. 98 - present	University of Illinois at Urbana-Champaign (UIUC) **PhD Candidate in Computer Science** PhD Thesis: Thread Escape Analysis for a Memory Consistency-Aware Compiler Advisor: Prof. David Padua
Sept. 96 - May 97	University of California at Santa Barbara (UCSB) PhD in Computer Science. Transferred to UIUC
Sept. 95 - Aug. 96	Hong Kong University of Science and Technology (HKUST) Master of Philosophy in Computer Science. Transferred to UCSB
Sept. 92 - June 95	Hong Kong University of Science and Technology (HKUST) **BEng in Computer Science** GPA: 10.75/12. First Class Honors Graduation project: Implementation of a parallelizing compiler for Intel Paragon

Teaching Experience

Jan 1997 - Jun 1997	Teaching Assistant, Department of Computer Science, UCSB Course: Programming Languages
Sept 1995 - May 1996	Teaching Assistant, Department of Computer Science, HKUST Course: Design and Analysis of Algorithms

Research Experience

Sep 2000 - present	Research Assistant, Department of Computer Science, UIUC Advisor: Professor David Padua Project: The Pensieve Project
Aug 1998 - Sep 2000	Research Assistant, Department of Computer Science, UIUC Advisor: Professor David Padua Project: Fortran 95 to Java translator
2001 summer	Summer Intern, IBM TJ Watson Research Center, Yorktown Heights, NY Mentor: Sam Midkiff, Manager: Manish Gupta Project: Embedded Java Virtual Machine
Jan 1998 - Aug 1998	Research Assistant, Department of Computer Science, UIUC Advisor: Professor Andrew Chien Project: Illinois Concert C++ compiler
1993 - 1996 summers	Research Assistant, Department of Computer Science, HKUST Projects: Object-oriented program parallelization (1996) Automatic parallelizing compiler for array-based applications(1995) GUI for Chinese input method(1994) Development for Chinese computing environment(1993)

Awards and Honors

1995	First Class Honours from HKUST
1995	HKUST Academic Achievement Medal
1994 - 1995	Dean's List (HKUST)
1993 - 1995	Zheng Ge Ru Foundation Scholarship
1992 - 1993	Joyce M. Kuok Foundation Scholarship

Professional Activities

Reviewer for the Proceedings of the IEEE. Special Issue on Program Generation, Optimization, and Platform Adaptation

Reviewer for the EuroPar Conference and the International Conference on Parallel Architectures and Compilation Techniques (PACT)

Reviewer for the International Conference for High Performance Computing and Communications (SC'02)

Reviewer for the Workshop on Languages and Compilers for Parallel Computing (LCPC'02)

Reviewer for the Workshop on Compilers for Parallel Computers (CPC'03)

Member of ACM

Publications

Zehra Sura, Chi-Leung Wong, Xing Fang, Jaejin Lee, S.P. Midkiff, and David Padua, "Automatic Implementation of Programming Language Consistency Models", 15th Workshop on Languages and Compilers for Parallel Computing (LCPC), July, 2002

Chi-Leung Wong, Zehra Sura, Xing Fang, Jaejin Lee, Samuel Midkiff, David Padua, "The Pensieve Project: A Compiler Infrastructure for Memory Models", Midwest Society for Programming Languages and Systems Workshop (MSPLS), Bloomington, Indiana, April 2002

Chi-Leung Wong, Anthony Bolmarcich, Samuel Midkiff, Peng Wu, "ROMable code Generation for Java". under preparation

Chi-Leung Wong, "Source to Source Translation from Fortran 95 to Java and Evaluation of Translated Programs", under preparation

PhD Thesis

Chi-Leung Wong, "Thread Escape Analysis for a Memory Consistency-Aware Compiler", Department of Computer Science, University of Illinois at Urbana-Champaign, 2005

Invited Paper

Chi-Leung Wong, Zehra Sura, Xing Fang, Jaejin Lee, Samuel Midkiff, David Padua, "The Pensieve Project: Compiling for Sequential Consistency", 6th International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN), Metro Manila, Philippines, May 2002

Presentation

Zehra Sura, Chi-Leung Wong, Xing Fang, Jaejin Lee, Samuel P. Midkiff, David Padua, "A Testbed for the Design of Software Memory Consistency Models", Work In Progress Session, 11th International Conference on Parallel Architectures and Compilation Techniques (PACT Work In Progress Session), Charlottesville, Virgina, September, 2002

Research Projects

An Optimizing Compiler for Languages with Programmable Memory Models

This work will be part of my PhD thesis. Our research focuses on building a Java optimizing compiler for explicitly parallel shared memory programs that hides the underlying relaxed memory consistency model. The compiler presents an intuitive and natural memory consistency model to ease programming and debugging. Moreover, it provides correct compiler optimizations that are not considered by conventional compilers. In addition, the compiler will serve as a testbed to prototype new memory consistency models at the language level, and to measure the effects of different memory models on program performance.

Embedded Java Virtual Machine

This is my IBM summer intern project. Our basic approach is to perform ahead-of-time compilation of classes and to generate ROMable code for those classes. The methods are compiled ahead-of-time rather than on the target platform, so we get the performance of compiled code without memory and time overhead of supporting a full just-in-time compiler on the embedded device. Moreover, since we perform the compilation off-line, we will be able to expand the necessary resources to aggressive optimizations. Since Java is a very dynamic language, machine code generated for Java programs are traditionally self modifying. This is not satisfactory in embedded system because machine code stored in ROM is not modifiable. Therefore, we need to isolate the dynamically changing components and store it in RAM while keeping the static components in ROM.

Fortran 95 to Java translator

I have constructed the whole translator including a Fortran 95 frontend and the Java source code generator. The translator is implemented completely in Java generate the Fortran parser. There are over 60000 lines of code. The translator can compile Fortran 95 programs into equivalent Java programs. The generated program uses the IBM array package and modified Fortran format package by Jocelyn Paine.

Other Previous Projects

Java JIT compiler

It is a class project supervised by Dr Urs Höezle which aimed at building a Java JIT compiler on Sparc platform from scratch. The compiler compiles methods on demand. I was responsible for implementation of method dispatch using virtual function table. I have designed the calling convention and virtual function table is created and used to do method dispatch.I also participated in the design of code generation module.

Parallelizing compiler for Intel Paragon

This is my undergraduate final year project supervised by Dr Tin-Fook Ngai. Our goal was to parallelize loops by appropriate program transformations and synchronization system calls insertion. I and another two teammates modified the GNU C compiler to generate our own intermediate representation (IR). The transformed IR is then fed into the backend of the GNU C compiler to generate Intel Paragon machine code.

Referees

Professor David Padua
Department of Computer Science
University of Illinois, Urbana-Champaign
Siebel Center for Computer Science
201 N. Goodwin Avenue
Urbana, IL 61801-2302
Phone: +1 217 333-4223
Fax: +1 217 333-3501
Email:padua@uiuc.edu

Professor Samuel P. Midkiff
School of Electrical and Computer Engineering
Purdue University
465 Northwestern Ave.
West Lafayette, Indiana 47907-2035
Phone: +1 765 494-3440
Fax: +1 765 494-6440
Email: smidkiff@purdue.edu

Professor Jaejin Lee
School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea
Phone: +82-2-880-1863
Email: jlee@cse.snu.ac.kr