Dynamic Optimization through the use of Automatic Runtime Specialization

by

John Whaley

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science

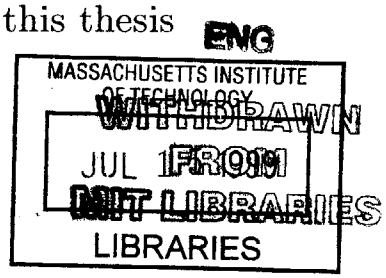
and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999
[June 1999]



Author ....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by .....
Martin Rinard
Assistant Professor
Thesis Supervisor

Accepted by ..
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Dynamic Optimization through the use of Automatic Runtime Specialization

by

John Whaley

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Profile-driven optimizations and dynamic optimization through specialization have taken optimizations to a new level. By using actual run-time data, optimizers can generate code that is specially tuned for the task at hand. However, most existing compilers that perform these optimizations require separate test runs to gather profile information, and/or user annotations in the code. In this thesis, I describe run-time optimizations that a dynamic compiler can perform automatically — without user annotations — by utilizing real-time performance data. I describe the implementation of the dynamic optimizations in the framework of a Java Virtual Machine and give performance results.

Thesis Supervisor: Martin Rinard
Title: Assistant Professor

Acknowledgments

First and foremost, I would like to thank all of the past and present members of the Jalapeño team at IBM T.J. Watson Research: Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Hummel, Derek Lieber, Vassily Litvinov, Mark Mergen, Ton Ngo, Igor Pechtchanski, Jim Russell, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Steve Smith, V.C. Sreedhar, and Harini Srinivasan. Each one of them contributed in some way to this thesis. I am eternally grateful to them for allowing me the opportunity to work on my 'dream' project. I could have never done it without them.

I'd especially like to thank Vivek Sarkar, my IBM thesis advisor, for inciting my interest in compilers and compiler research. I'd also like to thank him for the many hours he spent helping me understand compilers and his insights on conferences and writing compiler research papers.

I'd also like to thank Martin Rinard, my MIT thesis advisor, for all of his help and advice, and his numerous insights into compiler research. His enthusiasm for the subject is absolutely contagious; he even made the most dreaded part of research — writing papers — enjoyable. He was also an excellent role model, and I only regret that I didn't get more of a chance to work with him.

I'd like to thank my coworkers at IBM for their permission to describe their work in my thesis. The sections on the compiler backend (sections 3.5.1 through 3.5.6) are based very heavily on work and writings by others at IBM. Some of the graphs and figures in this thesis are also due to coworkers at IBM, and have been used in prior publications [20, 37, 8, 30]. Finally, sections 4 through 6 describe ongoing work by myself and others at IBM, and will be the subject of future publications.

I'd also like to thank Jong-Deok Choi for acting as my 'senpai' at IBM, for sitting through all of my attempted explanations and for teaching me so much about compilers and writing.

I'd like to thank all of my fraternity brothers at Theta Xi, Delta Chapter for being

so supportive as I rushed to finish this thesis and in all my other endeavours.

Finally, I'd like to thank my wonderful girlfriend, Shiho Iwanaga, for pushing me to get started on writing my thesis. (I procrastinated anyway, but at least I'm acknowledging her for trying...)

Contents

List of Figures

Chapter 1

Introduction

wataran to	Had I ever thought
omoi ya kakeshi	to be crossing these waters?
azumaji ni	Of the Chrysanthemum River
ari to bakari wa	I had merely heard men say,
kikugawa no mizu	"It is on the eastland road."

— *Abutsu, Izayoi nikki (Journal of the Sixteenth-Night Moon), thirteenth century.*

Traditional optimizers optimize code without regard to the actual run time characteristics of the program. Many optimizations can benefit from the additional knowledge available from run time data. However, most optimizations in static compilers that benefit from run time information make simple rough estimates as to the nature of that data. For example, restructuring of loops, elimination of induction variables, and loop invariant code motion all assume that a loop will most likely execute a large number of times. Loop unrolling and some register allocation algorithms make a simplifying assumption that a loop will execute a given number of times. All of these optimizations could benefit from actual runtime performance data.

Having run-time data available also makes new optimizations available to static compilation. For example, using trace scheduling with profile information, a static compiler can minimize the dynamic instruction count along critical paths [71, 72].

It can specialize procedures for common argument values. Using a calling context tree with actual branch percentages, it can optimize branches for the typical case and maximize cache locality by putting related code together [117, 87, 48]. It can utilize the actual execution frequencies to reorder if-elseif constructs to put the most common cases at the top, and compile switch statements into a Huffman tree to minimize average lookup time [63]. For hash tables, it can come up with a near-perfect hash function that is suited to the actual data that the program will be using. Static compiler writers have realized the benefit of using actual performance data, and profile-directed optimization has already been incorporated into some modern commercial compilers [88, 131, 89].

However, profile-directed static compilation has its faults. Profile-directed static compilers require a number of test runs to collect profile information to 'prime' the compilation. If the profile of the program during the test runs is different than from an actual execution, then the compiler has been 'primed' with the wrong data, and so it may make choices that can actually reduce performance on an actual execution. Finding 'typical' inputs for a program is very difficult for some applications. If the 'priming' misses a few common cases, those cases may have sufficiently poor performance as to negate the gains from the other cases. Even if 'priming' manages to cover all typical cases, the test runs are often on low-optimized and heavily instrumented code. The profile from this code may be drastically different from the profile of the final, optimized application.

The correctness of profile data aside, there is still a fundamental problem with profile-directed static compilation. Most programs go through a number of stages, or modes, where they do very different things. For example, even simple programs go through a 'loading/initialization stage' followed by some sort of 'computation stage.' Some interactive applications may have hundreds of different modes, each with different characteristics. Ideally, the compiler should be able to have different versions of the code, each suited to a particular stage. Today's static compilers that rely on post-mortem profile information cannot distinguish between stages — everything, from initialization to shutdown, is lumped together, indistinguishable.

Even if the profile information were time-sensitive, the static compiler would need some sort of runtime support to handle the 'switching' between modes.

Static compilers also have a problem adapting to different architectures. Due to the rapid advancement of hardware technology, new microprocessors are released very quickly. A program that is optimized assuming a particular processor may have less-than-ideal performance on the other processors in its line. Even when running on a system that has the same processor, the performance may be wildly different. External factors such as cache size, cache flush policies, motherboard type, bus speed, amount of free memory, and hard disk speed can all affect performance and can cause the optimizer to make different decisions that do not match a system with different specifications.

Performing the profile-directed optimizations at run time can solve many of these problems. The information available at actual run time will most likely be a more accurate model of the current execution. If the profile-directed optimizations are run continuously, they can 'track' the changing modes of the program. The optimizer can also be tuned to the specific system that it is running on. Performing optimizations at run-time also opens the door to new optimizations that are intractable or impossible to perform statically. For example, using partial evaluation or other run-time code generation mechanisms, the run time optimizer can create specialized code that is generated with the knowledge that certain invariants are true. Code compiled with these invariants can be many times more efficient than 'general' code. These invariants are not known at compile time, only at run time, so a static compiler cannot generate such code.

However, performing profile-directed optimizations at run time has its own share of problems. First and foremost is the additional runtime overhead due to the collection and management of the profile data, and the extra time and resources spent analyzing that data and performing the optimizations, as well as the extra overhead for the runtime to support such features. Because the analysis and optimization occurs at run time rather than compile time, the time spent performing them must be factored into the equation — if the amount of time spent performing the optimizations

is greater than the amount of time gained by performing the optimizations, then the strategy was a loser — it would have been better to not have attempted the runtime optimizations at all. This makes it very difficult to justify using expensive optimizations. And it is very difficult to anticipate how much benefit an optimization will bring without actually going ahead and performing the optimization.

There is another reason why performing profile-directed optimizations at run time can have worse results than performing them at compile time. At run time, the optimizations do not know what will happen in the future. They can only make decisions based on events that have already occurred. This is fine if the past events are representative of the future, but this might not always be the case. A program whose profile is very random would do very poorly with a run time optimizer — the optimizer would be operating with the wrong information. A profile-directed static optimizer that operates on post-mortem profile data has the entire profile at its fingertips, and therefore can optimize based on the 'future'.

There are currently two types of run time optimizers being experimented with. The first type is one which utilizes code profile information — information about what code the program executes. This 'code profile-directed optimization' strategy is sometimes used in dynamic systems, where the code for the program may not be available a priori and therefore static analysis is not possible. The optimizer uses information about 'hot spots' in the program to decide where to concentrate its efforts.

The second type is one which utilizes information about run time data values. This 'data-directed optimization' strategy is used in some dynamic compilation systems. The dynamic compiler can generate code which is specialized to a particular set of data values. As such, it can sometimes generate much faster code, code which, if executed enough times, can more than make up for the time spent in the dynamic compiler.

This thesis describes a much more aggressive approach than any previously attempted dynamic optimization technique. The current code profile-directed optimization strategies are very rudimentary — they simply use the profile information to decide what to spend time compiling and optimizing. We propose a much more

sophisticated use of the information, to optimize 'hot' traces and reorder code blocks in memory. The current data-directed optimizations are all programmer-specified through annotations or special constructs. This is an unsatisfactory solution. Programmers can only make rough guesses as to when and where to apply these optimizations, and presumably an optimizer could potentially identify some locations that a programmer might overlook or which might be too cumbersome for him to specify. We propose a scheme to automatically identify and exploit opportunities for data-directed optimizations.

Simplifying assumptions Successful dynamic compilation is a difficult problem in general. Full coverage of all of the issues involved in dynamic compilation is, unfortunately, beyond the scope of a masters thesis. Therefore, in this thesis we made a few simplifying assumptions so that we could focus on a few key issues.

The first assumption that we made was that run time information can be collected with zero overhead. This thesis is not focused on the optimization of the collection of run time information (although, in some of our presented algorithms and techniques, the efficiency of the information gathering is taken into consideration.) We believe this is a reasonable assumption to make for a few reasons. First, many techniques exist for minimizing the cost of gathering profiling information; even very fine-grained exact basic block level profile information can be gathered with as low as 10% overhead [19, 13, 14]. Second, overhead can be brought down significantly if additional hardware profiling support were added. If/when dynamic compilation systems become more mainstream, we may begin to see more hardware profiling support.

The second assumption that we made was that there is no a priori static program analysis allowed. Allowing offline static program analysis opens up a plethora of new issues, including what analysis and optimization should be performed at offline compile time versus run time. This assumption was mostly made out of necessity — our implementation is in the context of a Java Virtual Machine that supports the dynamic loading of as-of-yet unseen classes, and therefore we could not assume any static analysis information would be available. This fact necessitates a wholely dynamic ap-

proach, and allows us to focus on dynamic analysis, compilation, and optimization techniques, rather than get caught up in evaluating static analysis techniques.

The third assumption is that our target system is one which is designed for long-running applications, such as server applications. This allows us to focus on the heretofore unexplored area of performing more heavyweight dynamic optimizations, without being overly concerned about the dynamic compilation overhead.

Organization of our approach Because we are investigating more heavyweight dynamic compilation strategies in this thesis, we use compilation stages that are similar to those found in a static compiler. Namely, we first convert the bytecode into an intermediate representation (IR), perform optimizations on that IR, and use a backend to generate machine code. However, because this is still a dynamic compiler, compilation efficiency is still a concern. Therefore, much of the functionality of the compilation process is compressed into the single pass conversion process described in chapter 3, section 3.3, called BC2IR. Because it is the largest step and because it incorporates most of the interesting dynamic compilation methods, this thesis focuses on BC2IR, and only gives a cursory overview of the remainder of the dynamic compilation process. More information about the rest of the dynamic compiler and of the rest of the system can be found in other publications [20, 37, 8, 30].

Why Java? We implemented the dynamic compiler in a Java Virtual Machine for a number of reasons:

- Java is popular. The popularity of Java in both academia and industry makes it easy to find interesting, typical, and diverse applications. Different aspects of the language are fairly well understood, which means that we can avoid explaining idiosyncrasies and concentrate on the relevant issues. It also makes the work more relevant to a wider audience.

- Java is portable. Because Java is portable, the work that we are doing is not tied to a particular machine architecture, which, again, makes the work more relevant to a wider audience.

- Java is easy. This allowed us to implement things quickly and spend time on the important issues, rather than wasting time on implementation.

- Java is an inherently dynamic language. Because the code contained in dynamically loaded class files is in a machine-independent bytecode format that cannot be efficiently executed directly on typical microprocessors, some form of run time code generation is necessary. The fact that the code may not be available a priori for static analysis reinforces the need for effective dynamic compilation techniques.

Organization of this document This thesis is organized as follows. Chapter 2 provides some background to dynamic compilation by outlining the existing work in the area. Chapter 3 describes the dynamic compiler infrastructure used in the system. Chapter 4 describes the weighted calling context tree data structure. Chapter 5 describes the criteria by which we choose when, what, and how to invoke the optimizing compiler. Chapter 6 describes how the dynamic optimizations are performed.[1] Chapter 7 gives some preliminary experimental results. Chapter 8 outlines related work, and chapter 9 concludes.

[1]Chapters 4 through 6 are still ongoing work and will be covered in more detail in future publications.

Chapter 2

Background

kogite yuku	Gazing from the boat
fune nite mireba	as it goes rowing along,
ashihiki no	we see that the hills,
yama sae yuku o	the very hills, are moving.
matsu wa shirazu ya	Don't the pine trees know it?

— Ki no Tsurayuki, Tosa nikki (A Tosa Journal), tenth century.

The first serious look at run-time code generation (RTCG) for performance improvement was by Calton Pu and Henry Massalin in the Synthesis kernel [124, 125, 122, 106, 107, 105, 123, 104, 108]. The Synthesis kernel used RTCG to optimize frequently used kernel routines — queues, buffers, context switchers, interrupt handlers, and system call dispatchers — for specific situations. It only used a few simple optimizations — constant folding/propagation and procedure inlining. The RTCG was limited in that it was mostly 'fill-in-the-blanks' code generation, and it was programmer-directed, not automatic. They kept the RTCG lightweight and limited to a few circumstances. RTCG was manually applied to a few specific routines, and there was no pattern or framework for using RTCG. Despite the simplicity of his optimizations, they were able to get impressive performance improvements for some applications. Their results encouraged others to look more deeply into the area of RTCG.

2.1 Partial evaluation

Pu and Massalin drew heavily from the work on partial evaluation. Partial evaluation is a program transformation technique for specializing programs with respect to parts of their input. Partial evaluation is traditionally a source-to-source transformation. It was first developed in the sixties for use in LISP [103], and later spread to or was independently developed in the areas of artificial intelligence applications [15], compiling and compiler generation [15, 45, 44, 52, 76, 90, 92, 94], string and pattern matching [42, 57, 126, 80], computer graphics [10, 110], numerical computation [16], circuit simulation [11], and hard real-time systems [115]. There was a major step forward in partial evaluation in the eighties with the MIX partial evaluator [91]. The MIX partial evaluator was different than earlier efforts because it was self-applicable — one could apply partial evaluation to the partial evaluator itself, specializing the partial evaluator to the input of the partial evaluator. In order to make partial evaluators self-applicable, they had to be drastically simplified, and as a result new paradigms of partial evaluation were developed [46].

2.2 Programmer-directed dynamic compilers

More recently, there has been great interest in programmer-directed dynamic code generation systems. Tempo is a partial evaluator that can perform both compile-time and run-time specialization based on programmer hints and simple program analysis [43, 114]. 'C is an extension of ANSI C that allows the programmer to compose arbitrary fragments of dynamically generated code [118]. 'C includes two dynamic compilers — the first is DCG [66], which is relatively heavyweight, and the second is VCODE [65], which is much faster and more lightweight. DyC is a dynamic-compilation system that uses programmer annotations to specify the variables and code on which dynamic compilation should take place [74]. It uses a binding-time analysis to compute the set of run-time constants at each point in the dynamic region.

2.3 Profile-driven optimization

Profile-driven optimization is a relatively new field. Some static compilers utilize profile information from prior 'test runs' to perform better optimizations, for example, trace scheduling [79], improving cache locality [87, 117, 48, 128, 127], or traditional optimizations [35, 34, 36, 23]. Profile driven optimizations have shown up in recent commercial products [89, 88, 131]. There has been work on using profile information in dynamic compilers for Scheme [18, 17, 19], Self [61], Cecil [26] and ML [96, 101]. More recently, the upcoming HotSpot dynamic compiler for Java claims to perform profile-driven optimizations [84].

2.4 Dynamic code specialization

The SELF Compiler [28, 133, 27] was the first system to incorporate automatic code specialization. SELF is a completely dynamically-typed language, and therefore advanced specialization techniques are required into order to get adequate performance. The largest overhead in SELF came from dynamic dispatch, and therefore the work was focused on converting dynamic calls to static calls [31, 32], or using type feedback with polymorphic inline caches to reduce the overhead [82]. There was also some later derivative work in the same area [59, 73, 62, 75].

More recently, Burger and Dybvig describe a profile-driven dynamic recompilation system for Scheme [19]. They describe using profile information to reorder basic blocks to improve branch prediction and instruction cache locality.

2.5 Automatic dynamic specialization with respect to data

Automatic dynamic specialization on data values is an as-of-yet-unexplored area of computer science. Many dynamic compilation projects state that automatic dynamic optimization is their eventual goal, but that it is beyond the current state of the

art. The only results available in this area are very preliminary. Audrey and Wolfe describe an analysis to automatically identify so-called 'glacial variables', which are variables that only change infrequently and are therefore good candidates for specialization [12]. However, their analysis is static, not dynamic, and it doesn't take into account dynamic execution frequency or profile information. Execution frequency is estimated simply by loop nesting level. A proposal for the Dynamo project contains a paragraph on their plans to use profile information to help identify candidate variables for specialization [97]. There is no mention of how they plan to use the profile information, however.

Chapter 3

Dynamic Compiler

yama toyomu	The echo of axes
ono no hibiki o	reverberating in the hills
tazunureba	proves to be none other
iwai no tsue no	than the sound of men felling trees
oto ni zo arikeru	to make the festive wands.

— *Sei Shōnagon, Makura no sōshi (Pillow Book), tenth century.*

A dynamic compiler has very different design criteria as compared to a static compiler. Because compilation occurs at run time in a dynamic compiler, it is imperative that resource usage is minimized. A dynamic compiler should also be able to effectively utilize the extra information about how the program is actually running — for example, path profile information, run time data values, etc. Finally, if the system is to support loading code dynamically, the compiler should be able to deal with incomplete knowledge of the program.

Traditional compiler wisdom states that a compiler writer should split the functionality of the compiler into as many separate, small, simple, independent pieces as possible. For example, many optimization passes, as traditionally implemented, output suboptimal code (such as dead code or code that uses extra registers) with the expectation that a later compiler pass will "clean up" its output [7]. Also, many optimization passes simply invalidate auxiliary information and recompute it, rather

than incrementally update the information. This simplifies the overall design of system and of each individual piece — compiler writers can concentrate on making each compiler pass correct, and not have to worry about complicated interactions between the passes.

However, with this modularity comes inefficiency. Because each piece of a traditional compiler must be able to operate independently, it is typically less efficient than if multiple pieces were combined. For example, calculating reaching definitions and performing sparse conditional constant propagation are traditionally separate compilation passes. However, they can easily be combined, saving the overhead of set up and traversing the IR multiple times. Furthermore, it is often possible to do a better job in optimizing if two optimization passes are combined compared to as if they were invoked separately, regardless of their order or the number of times that they are invoked [41].

In a dynamic compiler, however, efficiency of the utmost concern. For this reason, many dynamic compilers forego the traditional idea of compilation stages or an intermediate representation altogether [98, 96, 99, 65]. They use simple table driven compilation or "fill in the blanks" compilation. However, code quality is sacrificed, of course, and therefore using an intermediate representation and a staged compiler, despite the longer compile times, might pay off in the long run for some applications.

For this reason, our system employs two distinct compilers. The first is a quick "baseline" compiler, whose job is to generate decent code quickly. The online measurements system keeps track of the particularly hot spots of the program by instrumenting the baseline compiled code or through sampling. The controller then selectively invokes the "heavyweight" dynamic compiler on selected pieces of code with a certain plan. The controller continues to monitor the performance of the system and selectively (re-)invokes the "heavyweight" dynamic compiler with different plans as the information becomes more accurate or as the program profile changes.

This thesis focuses almost entirely on the "heavyweight" dynamic compiler and the controller mechanism, because of the more interesting dynamic optimization opportunities that it can exploit and the fact that performing heavy optimization at

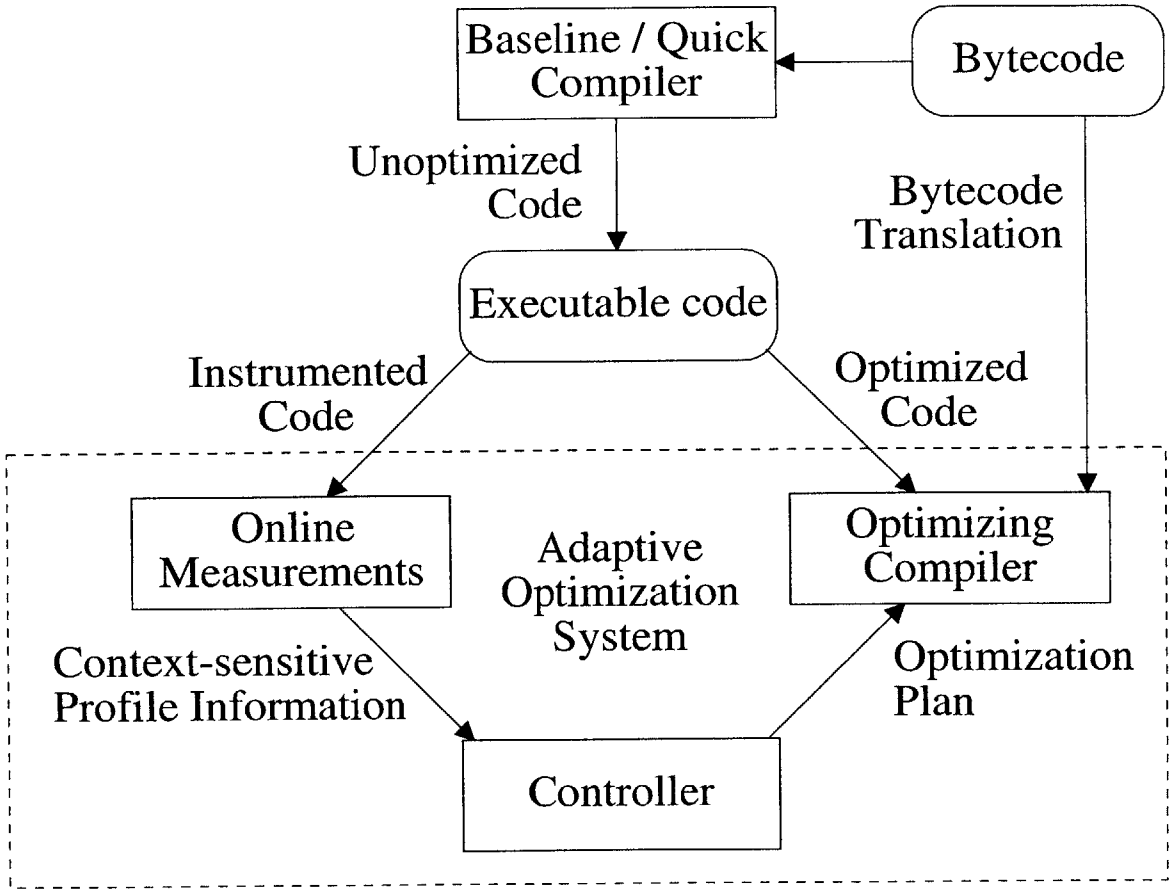


Figure 3-1: Overview of the dynamic compilation system

run time has not yet been explored. The context of the dynamic compiler also played a part in the decision; a Java Virtual Machine dynamically loads code that is often unoptimized and not in a form that is amenable to efficient execution, therefore a more substantial dynamic compiler is required for good performance.

But although we have made the design decision to use a more heavyweight dynamic compilation strategy, luxuries such as independent compiler passes are very difficult to justify. For this reason, many of the compilation passes that are typically separate in traditional compilers are combined into a single pass in our dynamic compiler. Because the amount of resources consumed by a compiler pass is almost always at least linear with the size of the code that is being compiled, enabling some optimizations that reduce code size, such as dead code elimination and null pointer check elimination actually reduces the total compilation time. Combining many optimizations into a single pass also has the benefit of exposing more optimization opportunities, without the need to iterate over optimization passes.

Because the compiler was designed from the beginning to be dynamic, many of the compilation stages, especially the conversion process, make use of any available dynamic information. They also support more dynamic features such as specialization with respect to data values (section 3.4.13).

The fact that the dynamic compiler is operating within the constraints of a Java Virtual Machine that can dynamically load classes means that the compiler has to be able to deal with incomplete knowledge of the program. Because this was known to be a design criteria from the beginning, the system was designed with support for incomplete information, and makes conservative assumptions whenever necessary.

3.1 Overview

Figure 3-2 shows an overview of the stages of the dynamic compiler. Java source code is compiled with an offline static compiler into bytecode, which is distributed in a machine-independent class file format. The dynamic compiler starts by converting the bytecode into an easy to manipulate intermediate representation (IR). Many

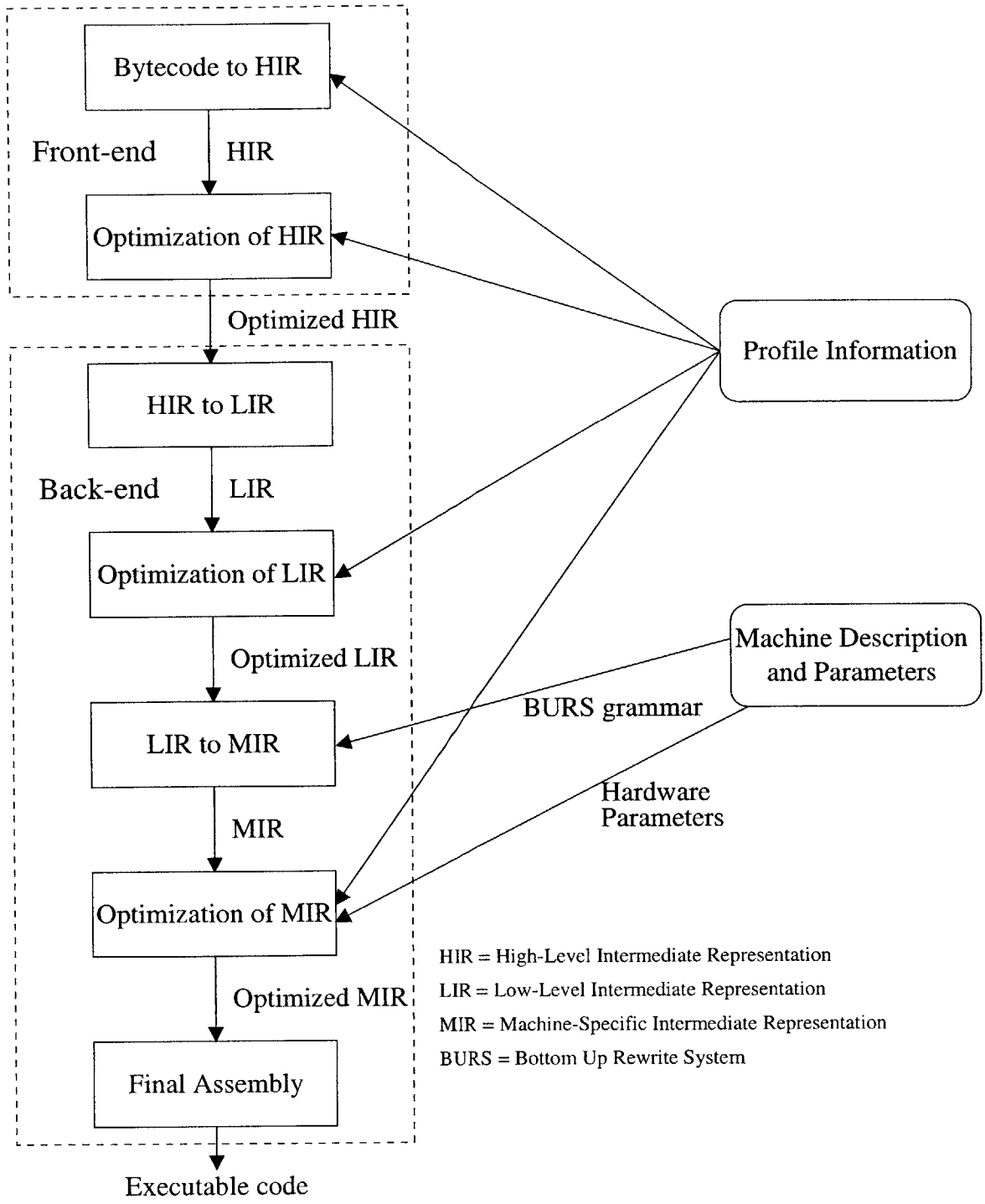


Figure 3-2: Overview of the compilation stages in the heavyweight dynamic compiler

Operator	Operand(s)	Meaning
`int_add`	$t1_{int}$, $t2_{int}$, $t3_{int}$	$t1 := t2 + t3$
`long_return`	42_{long}	return from method with (constant) value 42
`ifeq`	$t5_{cond}$, ⟨target⟩	if t5 is equal to zero, branch to ⟨target⟩
`call`	$t8_{ref}$, **toString**, $t7_{ref}$	call method **toString** with an argument given in $t7$, and put the result in $t8$
`bounds_check`	$t9_{ref}$, 10_{int}	do a array bounds check on the array $t9$ and the (constant) index 10.

Figure 3-3: Examples of IR **Instructions**

optimizations and analyses happen as part of this conversion process. Optimizations are performed on the IR, and target code is generated. This thesis focuses on the conversion step (called BC2IR) because it is the most unique piece; other steps are only briefly outlined in section 3.5. Section 3.2 describes the intermediate representation that our compiler uses. Section 3.3 describes the basics of the BC2IR step, while section 3.4 describes the optimizations and analyses that can be performed as a part of BC2IR. Section 3.5 briefly outlines the compiler steps after BC2IR.

3.2 Intermediate Representation

We first describe the format of our intermediate representation (IR), and give an example. Section 3.2.1 describes the essential pieces of the IR, while section 3.2.2 describes some of the auxiliary information that the IR can contain.

3.2.1 Core IR

For reasons of simplicity, efficiency, and versatility, the IR was designed to be modular. The core IR is simply a set of **Instructions**. An **Instruction** consists of an **Operator** and some number of **Operands**. See Figure 3-3.

Instructions can be organized in a number of different data structures, depending on what manipulations are expected to occur. They are commonly organized into a

Operand type	Description
Register	symbolic register
IntConst	integer constant
LongConst	long constant
FloatConst	float constant
DoubleConst	double constant
StringConst	string constant
NullConst	null constant
CaughtException	location of exception in an exception handler
Method	target of a method call
FieldAccess	an object or static field access
Location	a location in memory, used to track memory aliasing
Type	represents a type in a type check instruction
Branch	target of a branch
BasicBlock	used in a LABEL instruction to point to the basic block
Condition	condition code for a branch

Figure 3-4: **Operand** types

doubly-linked list for ease of manipulation. If code motion transformations are not expected to occur, they can also be organized into an array-like structure. If a control flow graph is available, **Instruction**s can also be organized into their respective basic blocks. Most optimizations are not concerned with the format of the instruction stream because all access is through interface calls.

Operators are divided into ranges. Very roughly, there is a *high level* range for the HIR, which has operators similar to Java bytecode, a *low level* range for the LIR, which replaces higher-level constructs by lower level operations, and a *machine-specific* range for the MIR, which corresponds to the instruction set architecture of the target machine. The output of the BC2IR step consists only of HIR operators, *viz.*, most of the non-stack-manipulation operators of Java bytecode, plus a few notable additions: there are `move` operators of different types to move values between symbolic registers, and there are separate operators to check for run time exception conditions (for example, `null_check`, `bounds_check`, etc.) Run time exception checks are defined as explicit check instructions so that they can be easily moved or eliminated. Keeping exception checks as separate instructions also makes it easier for code motion optimizations to obey exception semantics.

```
class t1 {
     static float foo(A a, B b, float c1, float c3)
     {
        float c2 = c1/c3;
        return(c1*a.f1 + c2*a.f2 + c3*b.f1);
     }
}
```

Figure 3-5: An example Java program.

The **Operand** types are described in Figure 3-4. The most common type of **Operand** is the **Register**, which represents a symbolic register. There are also **Operand**s to represent constants of different types, branch targets, method signatures, types, etc.

```
 0   LABEL0                 B0@0
 2   float_div              l4(float)  = l2(float), l3(float)
 7   null_check             l0(A, NonNull)
 7   getfield_unresolved    t5(float)  = l0(A), < A.f1>
 10  float_mul              t6(float)  = l2(float), t5(float)
 14  getfield_unresolved    t7(float)  = l0(A, NonNull), < A.f2>
 17  float_mul              t8(float)  = l4(float), t7(float)
 18  float_add              t9(float)  = t6(float), t8(float)
 21  null_check             l1(B, NonNull)
 21  getfield               t10(float) = l1(B), < B.f1>
 24  float_mul              t11(float) = l3(float), t10(float)
 25  float_add              t12(float) = t9(float), t11(float)
 26  float_return           t12(float)
     END_BBLOCK             B0@0
```

Figure 3-6: HIR of method `foo()`. l and t are virtual registers for local variables and temporary operands, respectively.

Figure 3-5 shows an example Java source program of `class t1`. Figure 3-6 shows the HIR for method `foo` of that class. The number on the first column of each HIR instruction is the index of the bytecode from which the instruction is generated. Before loading `class t1`, `class B` was loaded, but `class A` was not. As a result, the HIR instructions for accessing fields of `class A`, bytecode indices 7 and 14, are `getfield_unresolved`, while the HIR instruction accessing a field of `class B`, bytecode index 21, is a regular `getfield` instruction. Also, notice that BC2IR's on-

the-fly optimizations eliminated a redundant `null_check` instruction for the second `getfield_unresolved` instruction.

3.2.2 Auxiliary information

The IR includes space for the caching of optional auxiliary information, such as a control flow graph, reaching definition sets, or a data dependency graph. Auxiliary information is computed on demand — optimizations that require auxiliary information compute it if it is not available and cache the result. When transformations are performed that potentially invalidate a piece of auxiliary information, the cached copy is marked as invalid. This design makes it possible to easily add, remove, or reorder compilation stages to suit the particular situation. It also simplifies implementation because optimizations are not required to maintain auxiliary information through transformations.

Some of the auxiliary information is kept in side tables, whereas some is stored in the IR itself. Information that is not used often or that is easily invalidated, such as dominator information, is kept in side tables. This has the benefit of keeping the IR small. However, a few pieces of auxiliary information are stored in the IR itself. This includes information such as the control flow graph, because it is small and convenient, program transformations that make local code transformations can update the control flow graph with little hassle.

Control flow graph

The control flow graph is represented as a set of basic blocks, where each basic block has a start instruction, an end instruction, a set of in edges, a set of out edges, and a set of exception edges. The start instruction is always a `label` instruction, and the end instruction is always a `bbend` instruction. Exception edges point to any exception handlers in the current method that protect the given basic block. To facilitate backward-pass analyses, there is an edge from every block that ends with an explicit `return` (or an uncaught `throw`) instruction to a special exit basic block.

Control flow may only enter at the start of the block, but can potentially leave in the middle of the block through a trap or a method call. (In other words, method calls and potential trap sites do not end basic blocks. This is similar to superblocks [109] and traces [71].) Basic blocks are linked together using normal edges and exception edges. See [37] for a more complete description of the control flow graph structure and its benefits.

The decision to use extended basic blocks was based on the fact that because a large number of instructions can potentially throw exceptions, forcing basic blocks to end at all such instructions would greatly increase the number of control flow edges and greatly reduce the size of basic blocks. Many optimizations are more effective and/or have shorter running times when basic blocks are larger and there are fewer control flow edges. Forward pass analyses can support the exception edges with very few modifications. (Backward analyses can also support exception edges with a little more work [37].)

Dominators

Dominators are represented in a separate tree structure, where basic blocks are nodes in the tree and a node dominates all nodes in its subtree. The dominator information is used by some optimizations, mostly code motion transformations.

Last use information

Register operands contain an *is last use* bit, which, if the *last use* information is valid, corresponds to whether or not it is the last use of the given register. This is useful for register allocation and can make some optimizations more efficient, because they can free the resources allocated to keeping track of a register when its last use is reached.

Reaching definitions

Register operands that are uses contain the set of defs that reach them. (If the register is in SSA form, it is a singleton set.) This information is used by a variety of optimizations.

Upwardly-exposed uses, downwardly-exposed defs

Basic blocks have a set of *upwardly-exposed uses*, which are the register operands in the basic block with reaching defs outside of the basic block. They also have a set of *downwardly-exposed defs*, which are defs that are exposed to other blocks. These are used when computing reaching definitions.

Branch/Trace frequency information

There are side tables that contain frequency information, collected by the online measurement system, about the number of times that a trace or branch is taken. This information is used by the controller and trace scheduler to make decisions about "hot" traces, and by the code generator to organize code to improve branch prediction hit rates and instruction cache hit rates, and by a few other optimizations. The information is not required — if it is not available, the optimizations use static prediction routines.

Type/Value frequencies

There are tables that contain type frequencies for dynamically dispatched method calls and frequencies for method parameter values. These are also collected by the online measurement system. They are used by the controller and some of the dynamic optimizations.

SSA Form

SSA form can be very useful; it can simplify many analyses [111]. However, maintaining a full SSA form is not always desirable in a dynamic compiler — when we are not performing optimizations and/or analyses that benefit from SSA, the time spent converting to and maintaining SSA form is wasted. Also, using SSA form greatly increases the number of registers used, and therefore usually necessitates a register allocation step, whereas the non-SSA version may simply fit into registers without having to perform allocation. Furthermore, we wanted to use the same IR form for

post-register allocation, for uniformity and to facilitate reordering of optimizations. Therefore, the IR does not force a particular convention with regard to SSA.

The IR typically uses a hybrid approach, where some registers are in SSA form (compiler generated temporaries) while others are not (local variables.) Each individual register is marked as SSA or non-SSA. Furthermore, all non-SSA registers are numbered consecutively, so that analyses can efficiently build tables based on the register numbers.

Method-wide information

Some information is stored at a per-method level. Information used for inter-procedural analysis — for example, a method's side effects or the results of escape analysis — falls into this category. This gives us an easy way to perform simple inter-procedural analysis. Assumptions that were made during code generation are also stored at a per-method level, to allow multiple specialized copies of a single method to exist and be distinguishable, and to aid in the invalidation process when new classes are loaded, etc.

3.3 Converting from bytecode to IR

The first order of business in the dynamic compiler is to convert the input (Java bytecode) into a form that is more amenable to optimization, namely, the IR described in the previous section. There are many ways of converting from stack-based Java bytecode to register-based IR (see sections 8.1.1 and 8.1.2). However, all of these methods are time-consuming, mostly due to the fact that they are multi-pass. Through experimentation, we discovered that parsing the bytecodes takes a significant amount of time. (See Chapter 7.) Because the conversion process must occur at run time, its efficiency is of the utmost concern. Therefore, to minimize the parse time, we decided to make BC2IR operate in a single pass over the bytecodes.

3.3.1 Overview

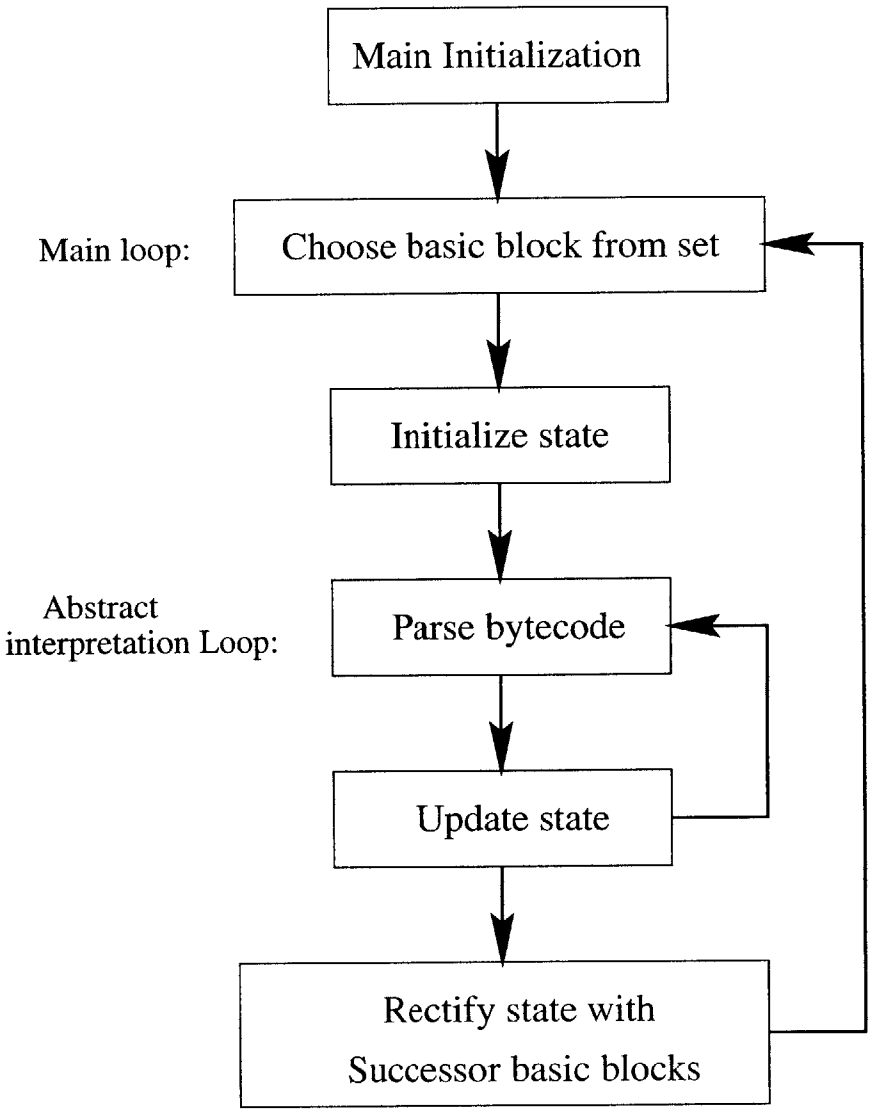


Figure 3-7: Graphical overview of the BC2IR algorithm

BC2IR is based on an abstract interpretation engine. Bytecodes are *abstractly interpreted*,[1] performing actions on a *symbolic state*. As a side effect of the abstract interpretation, BC2IR generates an instruction stream and control flow information. See Figure 3-7 for an graphical overview of the algorithm. Section 3.3.2 covers the symbolic state, Sections 3.3.3 through 3.3.5 give an overview of each of the steps of the algorithm, and sections 3.3.6 through 3.3.9 go into more detail on a few aspects of the algorithm.

3.3.2 Symbolic state

The only symbolic state that must be maintained in BC2IR is the state of the symbolic stack. The symbolic stack state is stored as an array of **Operands**, with an index variable indicating the top of the stack. The maximum stack depth for each method is available from the Java class file, so the array can be pre-allocated to the correct size and never needs to be grown.

Extensions to BC2IR can require maintaining other symbolic state information. Performing bytecode verification and effective type analysis requires keeping track of the types of the local variables. Performing constant propagation through local variables, effective method inlining, and basic block specialization requires keeping track of not only the types, but also the values of the local variables. When performing these optimizations, we maintain an array of **Operands** indexed by local variable number. Various other extensions to the symbolic state are described in section 3.4.

3.3.3 Initialization

BC2IR begins by initializing a basic block set. This set is implemented as a balanced tree of basic blocks, indexed by starting bytecode index. The set contains all basic blocks discovered thus far. As the abstract interpretation continues and more basic blocks are discovered, they are added to the basic block set. Basic blocks also contain

[1]Some authors [111] prefer to use the term *symbolic execution*, but I use *abstract interpretation* because it emphasizes the fact that there is an interpretation loop.

an initial state, which corresponds to the symbolic state of the machine at the start of the basic block. If some of the initial state is not known, it can be marked as unknown.

Before parsing any bytecodes, only a few basic blocks are known. There is a basic block beginning at bytecode 0, with an empty initial stack. The bytecode indices of exception handlers also start basic blocks; for these, the initial stack state is known to contain a single exception operand. The start and end indices of try ranges also denote basic block boundaries, but for these, nothing is known about the initial state.

3.3.4 Main loop

After initialization, BC2IR enters the main loop. The main loop searches in the basic block set for an as-of-yet ungenerated basic block with a fully known initial state. (For more information on the order that the basic block set is searched in, see section 3.3.8, below.) If no such block is found, generation is complete. If it finds one, it initializes the state to the initial state stored in the basic block, and enters the abstract interpretation loop.

3.3.5 Abstract interpretation loop

The abstract interpretation loop is the core piece of BC2IR. Each iteration of the abstract interpretation loop symbolically executes the current bytecode and updates the current state accordingly. The loop terminates when the current bytecode index reaches a basic block boundary. As a side effect of the symbolic execution, instructions are generated and added to the current basic block, new basic blocks that are discovered are added to the basic block set, the initial states of basic blocks are updated, and the control flow graph is updated.

The effect of each bytecode on the symbolic state follows straightforwardly from the Java bytecode specification [102]. See Figure 3-8 for table containing the effects of each of the bytecodes on the symbolic state and the IR instructions that they generate.

Bytecode	Effect on stack	Instructions generated	Ends basic block?
pop, pop2, dup, ...	Perform appropriate action on stack	None	No
iconst_1, aconst_null, ...	Push constant	None	No
iload, ...	Push local variable	None	No
istore, ...	Pop value, replace copies of local with new temps	For each copy on stack: move *new temp*, *local* move *local*, *popped val*	No
iaload, ...	Pop index, array reference, push new temp	null-check *array ref* bounds-check *array ref*, *index* *new temp* = array load *array ref*, *index*	No
iadd, fmul, ...	Pop operands, push new temp	*new temp* = op {...operands...}	No
idiv, imod, ldiv, lmod	Pop operands, push new temp	zero-check *denominator* *new temp* = op {...operands...}	No
iinc	Replace copies of local with new temps	For each copy on stack: move *new temp*, *local* *local* = add *local*, *amount*	No
i2f, l2d, checkcast, ...	Pop value, push new temp	*new temp* = conv_op *value*	No
instanceof	Pop value, push new temp	*new temp* = instanceof *value*, *type*	No
ireturn, ...	Pop value	return *value*	Yes
athrow	Pop value	null-check *value* athrow *value*	Yes
getstatic	Push new temp	*new temp* = getstatic *field*	No
putstatic	Pop value	putstatic *field*, *value*	No
getfield	Pop obj ref, push new temp	null-check *obj ref* *new temp* = getfield *obj ref*,*field*	No
putfield	Pop obj ref, value	null-check *obj ref* putfield *obj ref*,*field*, *value*	No
invokestatic	Pop args, push new temp	*new temp* = call *method*,{*args*}	No
invokevirtual, ...	Pop args, obj ref, push new temp (if not void)	null-check *obj ref* *new temp* = call *method*,*obj ref*, {*args*}	No
new	Push new temp	*new temp* = new *type*	No
newarray, ...	Pop size, push new temp	*new temp* = newarray *type*, *size*	No
arraylength	Pop array ref, push new temp	null-check *array ref* *new temp* = arraylength *array ref*	No
monitorenter, monitorexit	Pop obj ref	null-check *obj ref* monitor___ *obj ref*	No
goto, ...	None	goto *target*	Yes
ifeq, if_icmpeq, ...	Pop value(s)	if__ *value*, *target*	Yes
tableswitch, ...	Pop value	switch *value*, {*target*}	Yes
jsr, ret	See section 3.3.9.		Yes

Figure 3-8: Java bytecodes and their effects on the symbolic state

Java source code	Java bytecode
...	...
`do {`	`loop:` *loop body here*
loop body here	`iload x`
`} while (x != 0);`	`ifne loop`
...	...

Figure 3-9: Example of how the layout of do-while statements forces basic blocks to be split. BC2IR does not know that the branch to loop exists until after it has parsed the loop body and incorrectly appended its instructions to the previous basic block.

3.3.6 Splitting basic blocks

Due to backward branches, a basic block boundary may not be known until after the bytecodes at that boundary have been parsed and instructions generated. This can happen when the source code uses a `do-while` statement. (See Figure 3-9.) If we can guarantee that the state at the backward branch is identical to the state of the abstract interpretation loop when it reached the target bytecode, then BC2IR can split the basic block. If BC2IR is only keeping track of the stack state and the stack is empty, then the states must be identical and therefore the basic block can be split.

Instructions have a bytecode index, which corresponds to the bytecode from which they came. When BC2IR encounters a backward branch into the middle of an already-generated basic block, it searches from the end of the basic block for the instruction that has a bytecode index that is lower than the target. The basic block is split at that point. If the stack is not empty on a backward branch to the middle of an already-generated basic block, the basic block must be regenerated because the states may not match. However, current Java compilers never generate code that performs backward branches with a non-empty stack, so this situation is currently not encountered in practice.

3.3.7 Rectifying state at control flow joins

At a control flow join, the state may be different on different incoming edges. We define a *meet* operation on the state. The *meet* of a number of states is the result of

an elementwise *meet* of their components. We use the lattice pictured in Figure 3-10, where C_{type} refers to constants of the given type, and R_{type} refers to registers of the given type.

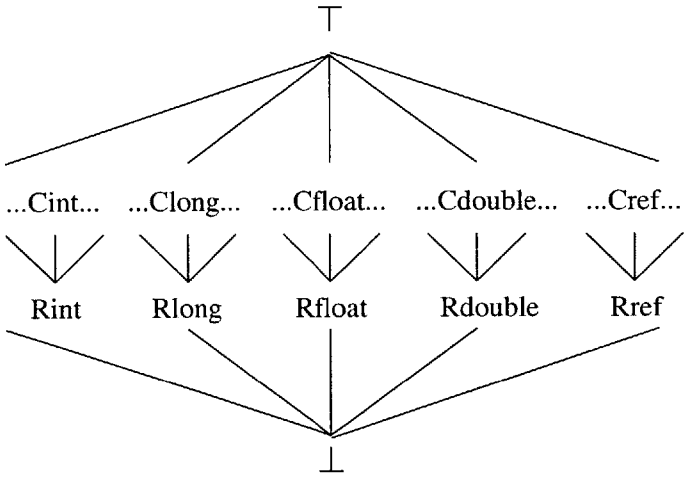


Figure 3-10: Lattice for **Operand**s

Thus, the meet of two stacks A and B is a stack whose elements are the results of the meet operation on the corresponding elements of A and B. See Figure 3-11 for an example.

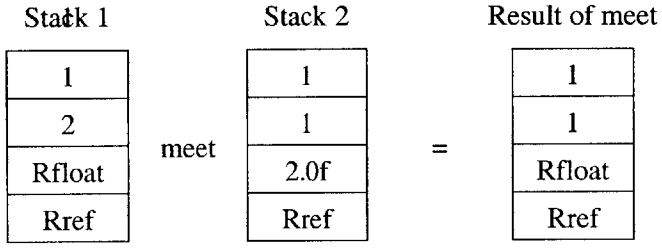


Figure 3-11: Example of the *meet* operation on two stacks

3.3.8 Greedy ordering for basic block generation

BC2IR generates code for a basic block assuming an initial state, which may be incorrect due to as-of-yet-unseen control flow joins. In some cases, the basic block

may have to be regenerated in order to take into account a more general initial state. We would like to choose basic blocks in an order such that the number of basic blocks that must be regenerated is minimized.

When the state is empty on backward branches, visiting the basic blocks in a topological order will avoid regeneration because the correct initial state will always be known. However, when BC2IR is keeping track of state information other than the stack, the state may not be empty on backward branches, and therefore regeneration may be necessary. We would like to minimize this regeneration by traversing back edges as early as possible so that the state from the back edge will "corrupt" a minimal number of basic blocks. See Figure 3-12 for an example.

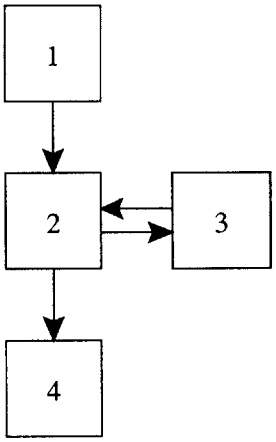


Figure 3-12: Choosing the topological order 1 2 4 3 may result in having to regenerate blocks 2, 3 and 4. The refined topological order 1 2 3 4 allows us to avoid having to regenerate block 4.

The optimal order for basic block generation is a *refined topological order*, which is a topological order where edges that lead into loop paths are taken before edges that exit loops.[2] However, because BC2IR computes the control flow graph in the same pass, it cannot compute the optimal order a priori.

Somewhat surprisingly, for programs compiled with current Java compilers, a simple greedy algorithm can always find the optimal ordering (ignoring exceptional

[2]This is similar to the reverse post order.

control flow which uses finally clauses or exception catches.) When selecting which block to generate out of the set of valid blocks, choose the block with the lowest starting bytecode index.

The reason that the simple greedy algorithm will always find the optimal ordering is as follows. Ignoring loops, all non-exceptional control flow constructs are generated in topological order. This means that, for non-loops, the bytecode order is the optimal order. Also, ignoring exceptional control flow, the control flow graph is always reducible (there are no loops with multiple headers.) The control flow graph can be thought as a "string" in a "graph language", where the reductions of the "graph language" grammar correspond to the reduction of the control flow graph into the Java language control flow construct that generated it.

Now, because non-loops are in topological order, the only edges that branch to prior bytecode indices must have arisen from a Java loop construct. Furthermore, because the graph is reducible, any loop has only a single entry and therefore a single header. The greedy algorithm specifies that the loop edge will be traversed before other edges, and therefore the algorithm will complete the loop before continuing. Now, because the graph is reducible, the loop can be summarized by its header in the control flow graph (with additional out edges corresponding to loop exits.) After collapsing the node, the graph will still be reducible and still correspond to a "string" in the "graph language" grammar, and therefore the algorithm continues as if the loop were a single node. Because the algorithm makes the correct decision for all loop constructs (entering loops before non-loops) and non-loop constructs (because they are in topological order and any loops that they contain can be summarized as a single node,) the algorithm finds the optimal order.

3.3.9 Control-flow and Java subroutines

Java supports the use of intra-method subroutines through the `jsr` and `ret` bytecodes. The existence of these subroutines complicates control-flow and data-flow analyses. Some suggested resolutions to this problem are to duplicate the entire subroutine at each `jsr` instruction [120], or even to change the Java virtual machine specification [4,

5]. However, the following constraints on Java bytecode make it possible to compute the control flow information for intra-method subroutines efficiently and in the same pass [102] [sections 4.8.2 and 4.9.6]:

1. The instruction following each `jsr` (or `jsr_w`) instruction only may be returned to by a single `ret` instruction.

2. No `jsr` or `jsr_w` instruction may be used to recursively call a subroutine if that subroutine is already present in the subroutine call chain.

3. Each instance of type **returnAddress** can be returned to at most once. If a `ret` instruction returns to a point in the subroutine call chain above the `ret` instruction corresponding to a given instance of type **returnAddress**, then that instance can never be used as a return address.

4. When executing the `ret` instruction, which implements a return from a subroutine, there must be only one possible subroutine from which the instruction can be returning. Two different subroutines cannot "merge" their execution to a single `ret` instruction.

These four constraints imply that a subroutine is defined by a `jsr` target address, and that each `ret` instruction corresponds to exactly one subroutine. Thus, each `ret` instruction maps to one and only one `jsr` target address.

Although it is not specified explicitly in the JVM specifications, BC2IR assumes that subroutines are distinct; *i.e.*,two subroutines do not share regions of code. If this were not the case, the verification of constraint 4, above, would be an undecidable problem.[3] Two subroutines can, however, "share" a region of code through a third subroutine, because the third subroutine will return to a different location depending upon where it was called from, and so constraint 4 is verifiable in this case.

For each subroutine, we keep track of a *return site set*, which contains the locations that the subroutine can return to; a *ret block*, which is the basic block containing the

[3]Nonetheless, some current Java compilers generate bytecode that does not have this property. We consider this to be due to either an inaccurate specification or broken compilers. When such code is encountered, it is easily detected and compilation reverts to the baseline compiler.

`ret` instruction for the subroutine; and an *ending state*, which is the state after interpreting the `ret` instruction. We define a subtype of the basic block type with these extra fields (a subroutine basic block) and make the basic blocks that mark the start of a subroutine use this subtype.

The `jsr` (and `jsr_w`) bytecodes are handled as follows: first, it looks up the basic block corresponding to the bytecode after the `jsr`. If it doesn't exist, it is created, and its initial stack state is marked as unknown. Then, it looks up the basic block corresponding to the target of the `jsr`. If it doesn't exist, then this is the first call to the subroutine that we have encountered, so the subroutine basic block is created and its return site set is initialized to contain one element: the basic block after the `jsr`. If it does exist, the algorithm has already encountered a call to the subroutine and so the states are rectified. If the subroutine basic block has a valid *ret block*, then a control flow edge is added from the *ret block* to the next basic block, and the ending state is copied from the subroutine basic block into the next basic block. If it doesn't have a valid *ret block*, then it simply adds the next block to the *return site set*. A subroutine operand is pushed onto the stack and the state is rectified with the initial state contained in the subroutine basic block. Finally, it ends the current basic block.

Similarly, for the `ret` (and `wide ret`) bytecodes, it starts by setting the current block as the `ret` block of the current subroutine. Then, it adds each of the blocks contained in the *return site set* of the subroutine as successors of the current block. Finally, it ends the current basic block.

3.4 Combining optimizations and analyses with IR generation

This section describes some of the extensions to BC2IR. Because the conversion is based on a general abstract interpretation framework, it was straightforward to extend BC2IR to concurrently perform a number of forward pass analyses and optimizations. Six such analyses and thirteen such optimizations have been implemented thus far.

Java bytecode	Generated IR *(optimization off)*	Generated IR *(optimization off)*
iload x iconst 5 iadd istore y	`int_add` t_{int}, x_{int}, 5 `int_move` y_{int}, t_{int}	`int_add` y_{int}, x_{int}, 5

Figure 3-13: Example of limited copy propagation and dead code elimination

Traditional optimizers found in static compilers perform optimizations independently. Because compile time is not critical, they often make design choices that favor simplicity of implementation over efficiency. For example, static compilers often recompute analysis information after code transformations rather than update it incrementally, even when an incremental update would be more efficient. Some optimization stages leave code in a suboptimal state, knowing that a later pass will clean it up. Such luxuries are difficult to justify in a dynamic compiler. Performing optimizations and analyses incrementally and in the same pass can be much more efficient.

Note that many of these optimizations and analyses have near-identical versions that operate independently, on the IR directly. These versions are used when the method has already been converted to IR — for example, after the dynamic compiler has already compiled the method once, but wants to perform more analyses or optimizations.

3.4.1 Limited copy propagation and dead code elimination

Java bytecode often contains sequences that perform a calculation and store the result into a local variable. See Figure 3-13. A simple peephole optimization can eliminate most of the unnecessary temporaries. When storing from a temporary into a local variable, BC2IR looks back at the most recently generated instruction. If its result is the same temporary, the instruction is mutated to write the value directly to the local variable instead. (When a stack manipulation bytecode is encountered that duplicates a temporary on the stack, the pointer to the most recently generated

instruction is reset to temporarily disable the optimization.) This simple technique catches almost all cases of unnecessary copy instructions, and it actually speeds up IR generation because of the reduction in the number of instructions that have to be generated. See section 7.2.3.

3.4.2 Unreachable code elimination

Because the algorithm only generates code that it has encountered a reference to, it obviously does not generate unreachable code. Less obviously, if an exception handler protects a range and no exception can be thrown in the range that will be caught by that exception handler, the exception handler is considered unreachable, and code is not generated for it.

This has limited use by itself, because java compilers will typically not allow you to compile programs with unreachable code. When used in conjunction with other optimizations such as constant propagation, branch optimizations, and method inlining, however, it becomes more useful.

3.4.3 Constant propagation and folding

Constant propagation and folding is easy for values on the stack. Constants are implicitly propagated on the stack and when an arithmetic operation is performed on operands that are constant, the result of the operation is simply pushed back onto the stack; no instruction is generated.[4]

Constant propagation through local variables requires extending the state to include the values of local variables. The current state of the local variables is stored in a local variable file. By default, the entries in the local variable file contain the register operand corresponding to each of the local variables. When a constant is stored into a local variable, however, the variable's entry in the local variable file is replaced by that constant. When pushing the value of the local variable onto the

[4]Operations that will throw an exception (divide by zero) are simply changed into an explicit throw instruction, and the remainder of the basic block is considered unreachable.

stack, BC2IR uses the operand contained in the local variable file, and therefore, will use the constant. The meet operation for the local variable state is similar to that of stacks — the pairwise meet of each entry in the local variable file. Constant propagation through local variables is very useful when used along with basic block specialization and method inlining.

3.4.4 Strength reduction

Strength reduction is another optimization that is straightforward to perform on the fly. Transformations that always result in better code — for example, replacing division by a constant power of two by a right-shift-arithmetic — are performed here.

3.4.5 Calculation of last use information

Keeping track of the last use of a compiler-generated temporary is very straightforward. When a temporary is pushed onto the stack, it has its *last use* flag set, signifying that it is the last use of that temporary. Stack manipulation operations simply maintain the invariant that if there are multiple copies of a temporary on the stack, only the one closest to the bottom has its *last use* flag set.

Computing last use information for local variables is a backwards analysis, and therefore is not performed in BC2IR. It is performed in a later step.

3.4.6 Control flow optimizations

BC2IR can replace unnecessary branches by storing a "forwarding address" in basic blocks that do nothing but unconditionally branch. This optimization does more than improve code quality — it also speeds up data flow calculations because it limits superfluous basic blocks. It is more efficient to perform the optimization in BC2IR rather than in a separate, later pass, although performing this optimization later may catch more cases.

3.4.7 Reaching definitions

BC2IR can also concurrently calculate reaching definitions. The algorithm used is similar to the forward pass analysis described in [7]. Calculating reaching defs for compiler generated temporaries is very straightforward — because they are in SSA form, they only have a single reaching def, and when the operand is generated, the definition is known, so the reaching def set can be initialized to the correct value.

Calculating reaching defs for local variables is more complicated. We add a new piece of state information — a *reaching def set array* for local variables. This is an array, indexed by local variable number, of the set of definitions that can reach the current program point. We also keep track of the *downwardly-exposed def array* for each basic block, which are the downwardly exposed defs for each of the local variables. This is very simple to do — it is an array indexed by local variable number, initialized to all nulls, and definitions of a local variable write their definition into the slot in the array, overwriting whatever was there.

At the start of BC2IR, the reaching def set array of the first basic block is initialized to refer to the parameters of the method. In the main loop, the reaching def set is initialized by copying from the initial state contained in the basic block. Uses of local variables use the set of reaching defs in the appropriate slot in the reaching def set array. Defs of local variables overwrite the slot with a new set containing the single new definition. Because reaching defs are implemented using sets and all upwardly-exposed uses of a given local variable refer to the same set, updating the initial state of the reaching def set array in a basic block simultaneously updates the reaching defs of all upwardly-exposed uses.

If the control flow graph has loops, it may be necessary to iterate over the control flow graph. Iteration is fast because the sets have already been computed, so it can deal with an entire basic block at a time. Since there is no such thing as a 'may-def' of a local variable, the downwardly-exposed def set also acts as the 'kill' set — if an entry in the downwardly-exposed def set is non-null, then that local variable is defined within the basic block and therefore all other definitions of that local variable

```
aload a
invokevirtual foo()
aload a
getfield x
aload b
ifrefnull label
aload b
putfield x
label:  ...
```

Figure 3-14: Assuming that there are no other incoming edges, only the first `invokevirtual` can throw a null pointer exception. The `getfield` and `putfield` cannot possibly throw null pointer exceptions in this context.

are killed. The data flow equations are identical to those found in [7].

3.4.8 Null pointer check elimination

As noted in section 3.2.1, the IR contains explicit *exception check* instructions. This gives the compiler a uniform way of keeping track of the locations where exceptions can possibly be thrown, so that code motion optimizations will not inadvertently violate exception semantics [102]. Also, it is possible to do analysis to prove that many `null-check` instructions are unnecessary. By having a separate instruction, we can simply remove it so that it will not constrain other optimizations.

There would appear to be a large number of locations that null pointer exceptions can be thrown. Any field access, array access, non-static method call, or exception throw has the potential to throw a null pointer exception. The large number of potential exception sites severely constrains potentially profitable code reordering optimizations like instruction scheduling. Because there is a separate `null-check` instruction for each one of these, it also greatly expands the size of the IR, slowing down optimizations and analyses. Therefore, it is in the best interest of the compiler to avoid generating and/or eliminate unnecessary `null-check` instructions.

Certain `null-check` instructions are superfluous. For example, see Figure 3-14. When you have a sequence of operations in a single basic block on the same variable

OUT (s = `null_check x`) =	x ∪ IN(s)
OUT (s = `c = string_const`) =	c ∪ IN(s)
OUT (s = `ref = new object`) =	ref ∪ IN(s)
OUT (s = `ref1 = ref2`) =	if (ref2 ∈ IN(s)) ref1 ∪ IN(s)
	otherwise IN(s) − ref1
OUT (s = `ifnull ref1`) =	IN(s) − ref1 along **true** branch
	IN(s) + ref1 along **false** branch
OUT (s = `ifnonnull ref1`) =	IN(s) + ref1 along **true** branch
	IN(s) − ref1 along **false** branch
IN (s) =	intersection over all OUT(PRED(s))
IN (Exception Handler with caught exception e) =	e
IN (Start of instance method) =	'this' pointer

Figure 3-15: Data flow equations for null pointer check elimination

and that variable does not change, only the first `null-check` is necessary — the fact that control flow had progressed past the first `null-check` implies that the variable is not null, so later null checks are superfluous. Performing a null check on a variable is not the only way to imply that a variable is not null — there are a number of other interesting cases. For example, bytecodes that perform allocations (new, etc.) can never return null, the *this* pointer in a virtual method is never null, a caught exception is never null, and string constants and certain static final fields are never null. Conditional branches that are based on comparing a variable against null imply that the variable is non-null on one of the branches.

The *non-null* property can be modeled as a data-flow problem specifying out sets in terms of in sets, where the sets refer to the "set of non-null references". See Figure 3-15 for the data flow equations.

The implementation of `null-check` elimination in BC2IR follows straightforwardly from the data-flow equations. Although this analysis is conservative, it eliminates over 90Java code. (See section 7.2.1.)

This optimization can benefit code quality in other ways, too: for example, conditionals based on a comparison of a value to null can sometimes be eliminated. Performing this optimization also decreases the time spent in IR generation, because the extra time spent in performing the optimization is less than the time that would

have been spent generating the superfluous instructions. (See section 7.2.4.)

3.4.9 Type analysis

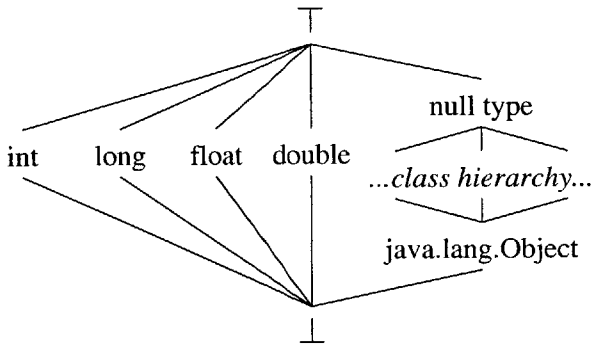


Figure 3-16: Lattice for type information

BC2IR can concurrently perform simple type analysis. We extend operands to include a type field. Instructions that push operands on the stack set the type field to be the most specific type possible. Performing a checkcast (run-time type test) operation on an operand causes its field to be updated to the more specific type, and conditionals based on instanceof tests propagate refined type information for the operand to one of the target basic blocks.

We extend the meet operation to include the type information. The lattice for computing the type information is shown in Figure 3-16. There are four primitive types — int, long, float, and double — and a set of reference types. The reference types include the class hierarchy (without exceptions) and a special *null type*, which corresponds to the type of a null pointer. We also include a bit that signifies whether or not the given type is exact.

Type analysis allows more method calls to be devirtualized and inlined. It also removes many redundant checkcast operations. For example, see Figure 3-17. In this case, the type of x at the checkcast instruction is already known to be Bar, so the redundant checkcast is ignored. This instanceof-checkcast pattern is very common in Java bytecode, so a large number of run-time type tests can be eliminated.

Java source	Java bytecode
if (x instanceof Bar)	instanceof Bar
	ifeq lab1
... = (Bar)x;	aload x
	checkcast Bar

Figure 3-17: Example of redundant checkcast operation

`checkstore`[5] operations can also be eliminated when the array type is known exactly.

3.4.10 Extended basic blocks

Although there may be a limitless number of possible paths through the code of a method, in typical programs only a handful are taken with any significant frequency. We can use dynamic profile information to discover the frequently executed (or dominant) paths through a method, and generate an IR where branches off of the dominant path are considered to be traps and therefore do not end basic blocks. (See section 3.2.2.) Many optimizations work better on larger basic blocks, and by performing trace scheduling on the dominant path, we can further improve performance on the typical case. Because we can avoid compiling code on non-dominant paths, dynamic compilation time is also reduced.

Extending BC2IR to support extended basic blocks was simple. In the abstract interpretation loop, when a forward conditional branch instruction is encountered and code for neither of the cases has been generated yet, the auxiliary data structure (described in section 3.2.2) that holds dynamic profile data is consulted. If one condition has occurred much more often than the other condition, then the conditional branch is changed to a conditional trap (inverting the conditional if necessary) and the loop continues along the dominant path without ending the basic block.

In the case of a backward conditional branch instruction, if the fallthrough case is taken much more often than the other case, then the conditional branch is changed

[5] `checkstore` operations are run time type checks that occur when an object is stored into an object array.

into a conditional trap and the loop continues along the dominant path.

3.4.11 Full SSA form

BC2IR makes compiler generated temporaries SSA by default, but leaves local variables as they were. Sometimes it is useful to have all registers, including locals, in SSA form. A simple extension allows BC2IR to concurrently convert local variables to SSA form with ϕ nodes where necessary.

Converting straight line code to SSA form is straightforward. Each slot in the local variable file contains, instead of a reference to the local variable, a reference to an SSA temporary that represents that local variable. Loads from a local variable use that temporary. When storing into a local variable, a new SSA temporary is generated, which replaces the old value in the slot.

At control flow joins, if an element of the local variable file differs and there is no current ϕ node for that variable, a ϕ node is inserted on the original value and the new value. If the ϕ node already exists, the new value is added to the ϕ node. If the target block has already been generated, all "upwardly-exposed uses" of the original value are modified to point to the result of the ϕ function. Because ϕ nodes are added only when necessary, they are relatively rare.[6]

Variables have to be modified at most once per basic block to take into account heretofore unknown control flow edges. By recognizing some common Java constructs during abstract interpretation, we can pessimistically add ϕ nodes, and thereby avoid a large number of cases that would have otherwise required modifying upwardly-exposed uses. This is identical to the pessimistic generalization of the state described in section 3.4.13.

[6]This algorithm for ϕ node placement does not come up with the optimal ϕ node placement [54, 55, 130, 56], but comes reasonably close in practice.

3.4.12 Method inlining

One of the most beneficial optimizations in Java is method inlining. Most Java programs are written in a heavily object-oriented fashion, with a large number of calls to methods with small bodies. Method inlining can yield very substantial performance increases due to the reduction of dynamic dispatch/call overhead and improved optimization opportunities.

When we know what we want to inline, method inlining is trivially straightforward. When we encounter a method invokation that we want to inline, we simply save away the current state, initialize a new state corresponding to the start of the inlined method, and begin abstractly interpreting the bytecodes of the inlined method. After it completes, we change return instructions to store the return value into a register and branch back into the calling method. Because code is more or less generated as if the method invokation were replaced by the bytecodes for the method, all of the other optimizations and analyses apply to the inlined method, and so the method body is automatically specialized to the call site. For example, constant parameters to an inlined method call can be propagated and folded in the inlined method body, more null pointer checks can be eliminated, and type information from outside of the method can propagate inside, improving optimizations opportunities.

3.4.13 Basic block specialization and loop peeling

While keeping track of more state information (such as the local variable file, type information, or "non-null" bits) allows us to do a better job at code generation, it also increases the chance of overspecialization. The more specific the information that we assume about the state, the more likely that an assumption that we make based on incoming edges that have been seen will not hold for an as-of-yet unseen incoming edge. In these cases, our prior solution was to regenerate the basic block, assuming a more general initial state.

However, there is no reason that we have to throw away the overspecialized version of the basic block that we worked so hard to generate. A basic block that is generated

assuming more constraints will be more optimized. We can keep the more optimal code around, and incoming edges that satisfy the tighter constraints can use the more specialized version, while others can use the less-specialized version.

We cannot indiscriminately specialize all basic blocks, however, because that can lead to explosive code growth. Code growth not only increases dynamic compile time, but can also make the resulting code less efficient because it may disrupt cache locality.

By recognizing common patterns in Java bytecode, it is possible to gain a fine control over how basic block specialization is performed. For example, current Java compilers output the `for` and `while` loop constructs in a specific order — the entrance to the loop is an unconditional branch to the loop condition, which is at the bottom of the loop. These are the only constructs that are laid out as such, and therefore when BC2IR encounters an unconditional forward branch past a block that it has never seen before, it can assume that it is entering a loop construct and pessimistically generalize the state at that point.[7]

By refraining from generalizing the state, we can also peel iterations out of a loop, specializing the peeled iterations with respect to any constant values, information propagated from the loop condition, or any other state information available. Peeling an iteration out of a loop also allows a common subexpression elimination optimization to implicitly perform loop invariant code motion.

3.4.14 Object escape analysis

A significant portion of JVM run time is spent in garbage collection. In typical Java programs, most of the objects die quickly. By performing escape analysis, we can determine whether objects can *escape* the current method. An object is said to *escape* if, when the method exits, there is still a live reference to the object. If an object cannot escape a method, it can be allocated on the stack instead of the heap,

[7]Generalization of the state usually involves converting constants to registers and reseting non-null attributes to possibly null. Other possibilities include changing register types to a more general form, but this is not done because the register type rarely changes during a loop iteration.

and thus will be implicitly garbage collected when the method returns. This saves both on the number of garbage collections that have to be performed and on the speed of those collections. Also, synchronization operations on non-escaping objects are unnecessary and therefore can be removed. (For more information on escape analysis in Java and the optimizations that it facilitates, see [135].)

We implemented a simple object escape analysis as part of the BC2IR framework. Object creation sites are tracked, and registers include attributes stating whether or not they can refer to objects created at a given site. If a location that can contain an object created at a particular object creation site is used in a `putfield` or `putstatic` operation, then objects created at that site are said to escape.

The escape analysis also implements a simple interprocedural analysis.[8] If a location that can refer to objects created at a given site is used in a `return` instruction, then objects created at that site are said to escape through the returned value.[9] Then, each method call is modeled as a function on its parameters. Each of the method parameters can be of one of three types. The first type is that objects passed into a particular parameter can escape outright. The second type is that objects passed into the parameter can escape through the return value. The third type is objects passed into that parameter can not escape. When a method call is encountered where all targets are known and have been analyzed, the algorithm uses the summary information for the target method(s). When analysis of a method completes, summary information for the method is stored.[10]

3.4.15 Side effect analysis

We also implemented a simple side effect analysis. It keeps track of which fields are read and written in the method, using profile information (or static estimates, if profile information is unavailable) to estimate the number of reads/writes to each

[8]Some sort of interprocedural analysis is necessary due to the fact that all created objects have constructor methods called on them.

[9]For simplicity, anything used as an arugment to an `athrow` instruction is said to escape.

[10]When a cycle is encountered in the call graph, the algorithm pessimistically assumes the worst case — *i.e.*,, everything escapes outright.

field.

3.4.16 Specialization benefit prediction

Compiling dynamically allows us to take advantage of actual run time information to selectively specialize with respect to a data value. Specialization with respect to values can allow order-of-magnitude increases in performance [118, 74]. However, specialization has a significant cost in time and memory, and therefore we must be careful what we attempt to specialize and on what values.

We implemented an extension to BC2IR that allows us to analyze a method and make a prediction as to how much better we could do if we knew that a particular parameter or parameter field were constant, or if we knew that a parameter or parameter field had a specific type. This information is used by the controller (chapter 5) along with frequency information to make decisions about when to specialize with respect to data values and what to specialize on.

For predicting the benefit of specializing with respect to a parameter value or type, the algorithm is as follows. Along with each register operand, we include extra information that specifies whether the value contained in that register would be constant if a parameter were constant, and if so, the parameter(s) that would need to be constant. This attribute can be modeled as a simple data flow problem; the modeling is straightforward so we do not go into any more detail here.

For predicting specialization benefit if the parameter were constant, a benefit number is associated with each parameter.[11] Whenever a register that is derived from a parameter is used in an operation, the parameter's benefit number for the parameter(s) it is derived from is incremented by an amount relative to the expected savings on the cost of the operation if the specified operand were constant. For example, for an addition where one of the operands was from a method parameter, the increment is small, but for a division operation where the denominator was from a method parameter, the increment is much larger.

[11]Benefit numbers are also associated with combinations of parameters. These are allocated as needed, and are very rarely used.

When a branch is discovered that depends entirely on registers that come from parameters, the algorithm calculates the expected benefit gained if the conditional branch were either eliminated or changed into a unconditional branch. It does this as follows. On each branch, it propagates information that the code is dominated by the outcome of the branch. The costs of all operations along that branch are scaled by the estimated frequency that the outcome of the branch will occur. (It uses dynamic information if it is available, otherwise it uses a static prediction.) This scaled value is added to the benefit number. When a control flow merge occurs where an incoming edge is not dominated by a particular branch outcome, the code after the merge is not dominated by the outcome (*i.e.*,, the merge function for the dominator information is intersection.)

Predicting the benefit if a parameter were a known type works in a similar fashion, but in this case, the benefit number is updated for operations that could be optimized based on the type (such as checkcast operations or virtual method calls.)

Predicting specialization benefit if a parameter field is similar. Performing a get-field operation on a register operand that comes from a parameter causes that value to be tracked analogous to the tracking of parameter values.[12] However, synchronization points kill information, because fields must be reloaded at synchronization points.

3.4.17 Bytecode verification

The abstract interpretation engine performs a number of steps that are very similar to the Java bytecode verifier as described in [102]. For example, the manner in which type information is propagated and the tracking of which local variables are used in a subroutine are identical. Therefore, adding bytecode verification support to BC2IR was relatively simple. Instead of implicitly assuming that constraints are true, it explicitly verifies them. Because BC2IR is much more optimized than the Sun verifier implementation and it traverses the basic blocks in an optimal order, it can

[12]Assuming that the field is not marked as volatile.

sometimes perform bytecode verification and concurrently generate IR more quickly than the Sun verifier can perform just the bytecode verification. See chapter 7.

3.5 Later compiler stages

This section briefly describes the later stages of the compiler. Because the focus of this thesis is on BC2IR, they are not covered in detail. See [20] for a more complete treatment of these compiler stages.

3.5.1 Lowering of IR

After high-level analyses and optimizations are performed, HIR is lowered to LIR. The LIR expands instructions into operations that are specific to the virtual machine's object layouts and parameter-passing conventions. For example, operations in HIR to invoke methods of an object consist of a single instruction, closely matching the `invokevirtual` bytecode. These single-instruction HIR operations are lowered (*i.e.*,, converted) into multiple-instruction LIR operations that invoke the methods based on the virtual-function-table layout. These multiple LIR operations expose more opportunities for low-level optimizations.

```
0   LABEL0               B0@0
2   float_div            l4(float)  = l2(float), l3(float)  (n1)
7   null_check           l0(A, NonNull)                     (n2)
7   getfield_unresolved  t5(float)  = l0(A), <A.f1>         (n3)
10  float_mul            t6(float)  = l2(float), t5(float)  (n4)
14  getfield_unresolved  t7(float)  = l0(A, NonNull), <A.f2>(n5)
17  float_mul            t8(float)  = l4(float), t7(float)  (n6)
18  float_add            t9(float)  = t6(float), t8(float)  (n7)
21  null_check           l1(B, NonNull)                     (n8)
21  float_load           t10(float) = @{  l1(B), -16 }      (n9)
24  float_mul            t11(float) = l3(float), t10(float) (n10)
25  float_add            t12(float) = t9(float), t11(float) (n11)
26  return               t12(float)                         (n12)
    END_BBLOCK           B0@0
```

Figure 3-18: LIR of method `foo()`

Figure 3-18 shows the LIR for method `foo` of the example in Figure 3-5. Notice

that the LIR is very similar to the HIR of Figure 3-6. The only differences are that the `getfield` of bytecode 21 has been lowered to a `float_load` and `return_float` has been lowered to `return`. (The labels (n1) through (n12) on the far right of each instruction indicate the corresponding node in the data dependence graph shown in section 3.5.2, Figure 3-19.)

3.5.2 Building dependence graphs

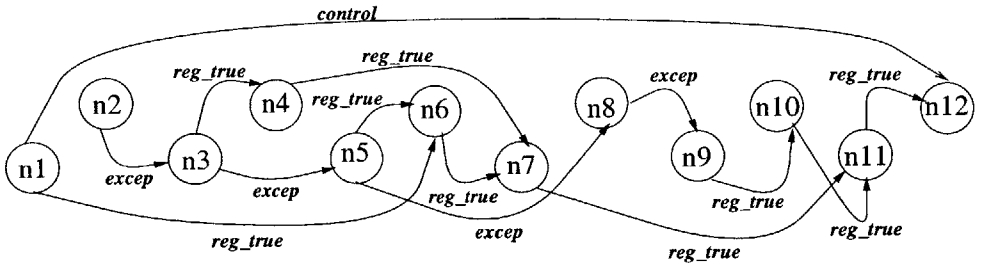


Figure 3-19: Dependence graph of basic block in method `foo()`

An instruction-level dependence graph for each basic block is constructed that captures register true/anti/output dependences, control dependencies, and other dependencies that preserve the Java memory model and exception semantics. This graph is used during BURS code generation (section 3.5.3).

Synchronization constraints are modeled by introducing *synchronization dependence* edges between synchronization operations (`monitor_enter` and `monitor_exit`) and memory operations. These edges prevent code motion of memory operations across synchronization points. Java exception semantics [102] is modeled by *exception dependence edges*, which connect different exception points in a basic block. Exception dependence edges are also added between register write operations of local variables and exception points in the basic block. Exception dependence edges between register operations and exceptions points need not be added if the corresponding method does not have catch blocks. This precise modeling of dependence constraints allows us to perform more aggressive code generation.

Figure 3-19 shows the dependence graph for the single basic block in method `foo()` of Figure 3-5. The graph, constructed from the LIR for the method, shows register-

true dependence edges, exception dependence edges, and a control dependence edge from the first instruction to the last instruction in the basic block. There are no memory dependence edges because the basic block contains only loads and no stores, and we do not not need any load-load dependencies because the fields are not marked as *volatile* [102]. An exception dependence edge is created between an instruction that tests for an exception (such as `null_check`) and an instruction that depends on the result of the test (such as `getfield`).

3.5.3 BURS code generation

Machine specific code is generated from the optimized LIR. The dependence graph for a basic block is partitioned into trees that are input to a tree-pattern-matching system based on a bottom-up rewriting system (BURS) [53]. Unlike previous approaches [70] to partitioning DAGs for tree-pattern-matching, this approach considers partitioning in the presence of memory and exception dependences (as well as register-true dependences). The partitioning algorithm also incorporates code duplication [129].

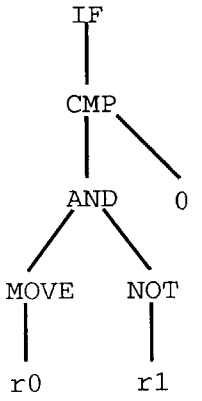
input LIR:

```
move r2=r0
not  r3=r1
and  r4=r2,r3
cmp  r5=r4,0
if   r5,!=,LBL
```

emitted instructions:

```
andc. r4,r0,r1
bne   LBL
```

DAG/tree:



input grammar (relevant rules):

```
RULE PATTERN                 COST
---- -------                 ----
1    reg: REGISTER             0
2    reg: MOVE(reg)            0
3    reg: NOT(reg)             1
4    reg: AND(reg,reg)         1
5    reg: CMP(reg,INTEGER)     1
6    stm: IF(reg)              1
7    stm: IF(CMP(AND(reg,      2
           NOT(reg)),ZERO)))
```

Figure 3-20: Example of tree pattern matching for PowerPC

Figure 3-20 shows a simple example of pattern matching for the PowerPC. The data dependence graph is partitioned into trees before using BURS. Then, pattern matching is applied on the trees using a grammar (relevant fragments are illustrated in the figure.) Each grammar rule has an associated cost, in this case the number of

[0]This footnote tests a conjecture that no one ever reads M.Eng theses after they are submitted.

instructions that the rule will generate. (Rule 2, for example, has a zero cost because it is used to coalesce register moves.) Although rules 3, 4, 5, and 6 could be used to parse the tree, the pattern matching selects rules 1, 2, and 7 as the ones with the least cost to cover the tree. Once these rules are selected as the least cover of the tree, the selected code is emitted as MIR instructions. Thus, only two PowerPC instructions are emitted for five input LIR instructions. Figure 3-21 shows the MIR for method `foo` in Figure 3-5. Notice that the null-pointer checks have been eliminated since these are implemented as hardware traps in the virtual machine.

```
     LABEL0                B0@0
2    ppc_fdivs             l4(float)  = l2(float), l3(float)
7    getfield_unresolved   t5(float)  = l0(A, NonNull), < A.f1>
10   ppc_fmuls             t6(float)  = l2(float), t5(float)
14   getfield_unresolved   t7(float)  = l0(A, NonNull), < A.f2>
17   ppc_fmuls             t8(float)  = l4(float), t7(float)
18   ppc_fadds             t9(float)  = t6(float), t8(float)
21   ppc_lfs               t10(float) = @{ -16, l1(B, NonNull) }
24   ppc_fmuls             t11(float) = l3(float), t10(float)
25   ppc_fadds             t12(float) = t9(float), t11(float)
26   return                t12(float)
     END_BBLOCK            B0@0
```

Figure 3-21: MIR of method `foo()` with virtual registers

3.5.4 Instruction scheduling

Instruction scheduling allows the compiler to improve code quality by reordering instructions to increase the utilization of processor resources and eliminate stalls. We use a priority list scheduling algorithm with bitmapped resource management [111].

The target machine is specified as a collection of independent resources of different types. Instructions are grouped into instruction classes such that all instructions in the same class have the same resource usage pattern. A resource usage pattern is a list of resource reservations, each of which consisted of a resource type, and start and end time of usage relative to instruction issue time.

The compiler builds all possible resource patterns for each instruction class in an offline step, also adding latency information between instruction classes. Each

instruction has associated with it a list of resources that are required. The scheduler will reserve particular resources for a given instruction, as long as there are no resource conflicts.

The current target architecture for the compiler is PowerPC 604e, which contains the following types of functional units: FXU (fixed point unit), FPU (floating point unit), BRU (branch unit), LDST (load/store unit), and FXUC (complex fixed point unit). The architecture contains multiple instances of the FXU. To model a multiple-issue machine, we added a pseudo-resource class ISSUE (issue slot). To deal with certain peculiarities of the architecture, we also added two more artificial resource classes: CR (control register) and RESERVE (memory reservation).

Most instructions have straightforward resource allocation. For example, most floating point arithmetic instructions belong to the `float_arith` instruction class, and reserve the floating point unit for one cycle. However, since the pipeline is of length 3, the results of these instructions can only be available 3 cycles later. Some instructions, however, required special treatment. For example, the integer divide instruction (`DIVW/DIVWU`) is non-pipelined, which means that it will reserve the fixed point unit that it is executing on for the entire time of execution (19 cycles). The load/store-multiple instructions (`LMW/STMW`) have a variable latency that depends on the operand. For these instructions, we use a conservative approach, because such instructions occur rarely in practice. Some instructions that manipulate the data or instruction cache have effects that are difficult to model, we conservatively reserve all resources for a long time to avoid any possible intermixing of these with other instructions.

The scheduling algorithm used is a greedy list scheduler (see Figure 3-22), with instructions arranged in a list by priority. The algorithm for constructing the priority list is deliberately left unspecified in the scheduling algorithm. The scheduler picks the next available instruction from the priority list, and finds the first available slot in the schedule starting with the instruction's earliest start time. The dependence graph (see section 3.5.2) is used to compute earliest start times for all instructions.

The schedule itself is just a two-dimensional bitmap, so that bitwise operations

```
build dependence graph for bb;
initialize schedule;
set earliest start time for each instruction to 0;
for each instruction i {
   propagate earliest start time for i;
}
build priority list pl for bb;
for each instruction i in pl {
   t = earliest start time of i;
   while (i cannot be scheduled at time t)
      t++;
   set scheduling time of i to t;
   propagate earliest start time for i;
}
rearrange instructions in bb based on their scheduling times;
```

Figure 3-22: Scheduling Algorithm

can be used to check resource usages. The algorithm is run on each basic block of a method.

3.5.5 Register allocation

The dynamic compiler's register-allocation framework supports different allocation schemes, according to the available time that can be spent in optimizing a method.

A linear scan register allocator [119] is currently employed. The linear scan algorithm is not based on graph coloring, but allocates registers to variables in a single linear-time scan of the variables' live ranges in a greedy fashion. This algorithm is several times faster than algorithms based on graph coloring, and results in code that is almost as efficient as that obtained using more complex allocators [119].

The LIR that reaches the register allocator contains two types of symbolic registers: temporaries, obtained from converting stack simulation into registers, and locals, obtained from Java locals specified in the bytecode. Higher priority is given to allocating physical registers to those temporaries whose live range does not span a basic block.

Once registers for temporaries are allocated, a second pass allocates the remaining

registers consecutively to locals without performing any flow analysis. This simple scheme is appropriate for small methods, as it is typical of object-oriented programs. (We have observed that for the PowerPC architecture with 32 registers, most methods do not need spill locations.) Large methods (either in the source program or after inlining) may merit a better register allocator to avoid spills.

Figure 3-23 shows the `foo` method of Figure 3-5 after register allocation. The output of the register allocator also includes a prologue at the beginning, and an epilogues at the end, of each method.

```
0  LABEL0                B0@0
0  ppc_stwu              FP(int),    @{-24, FP(int) }
0  ppc_ldi               R0(int)   = 4021
0  ppc_stw               R0(int),    @{  4, FP(int) }
0  ppc_mfspr             R0(int)   = LR(int)
0  ppc_stw               R0(int),    @{ 32, FP(int) }
2  ppc_fdivs             F3(float) = F1(float), F2(float)
7  getfield_unresolved   F4(float) = R3(A, NonNull), < A.f1>
10 ppc_fmuls             F1(float) = F1(float), F4(float)
14 getfield_unresolved   F4(float) = R3(A, NonNull), < A.f2>
17 ppc_fmuls             F3(float) = F3(float), F4(float)
18 ppc_fadds             F1(float) = F1(float), F3(float)
21 ppc_lfs               F3(float) = @{ -16, R4(B, NonNull) }
24 ppc_fmuls             F2(float) = F2(float), F3(float)
25 ppc_fadds             F1(float) = F1(float), F2(float)
0  ppc_lwz               R0(int)   = @{ 32, FP(int) }
0  ppc_mtspr             LR(int)   = R0(int)
0  ppc_addi              FP(int)   = FP(int), 24
26 ppc_blr               LR(int)
   END_BBLOCK            B0@0
```

Figure 3-23: MIR of method `foo()` with physical registers

3.5.6 Outputting retargetable code

The final phase of the compiler emits binary executable code into an (`int`) instruction array (called a method body). The assembly phase also finalizes the exception table and the stack map of the instruction array, by converting offsets in the IR to offsets in the machine code. A reference to the instruction array is stored into a field of the object instance for the method. The method object can hold multiple method bodies

for the same bytecodes (specialized based on factors such as the call-site contexts or the values of the parameters.) Selection of a particular method body to be invoked at a particular invocation site can be made during compile-time when LIR is generated or at the actual invocation time.

Chapter 4

Weighted calling context graph

ran no ka ya — An orchid's perfume
chō no tsubasa ni — transfers incense to the wings
takimono su — of the butterfly.

— *Matsuo Bashō, seventeenth century.*

A dynamic compiler must walk a delicate line. Because compilation occurs at run time, it must be very selective in what it decides to compile and how it decides to compile it. A dynamic compiler should only spend extra time analyzing and compiling a piece of code if there is a reasonable chance that the extra time spent on compilation will be made up in run time. However, if the dynamic compiler is too timid in its decision making, it may miss good opportunities for optimization and lead to overall slow performance. An aggressive dynamic compiler even has the ability to beat the best static compilers because information about the actual run time performance of the system is available, and it can specialize the code and data to suit the situation.

To guide decisions on what and how to dynamic compile and optimize, we use the prior behavior of the system with the expectation that the past will provide some indication of the future. This information is gathered by an *online measurement system* through the use of timer-based sample profiling and code instrumentation. The information gathered by the online measurement system must be stored in a

form that allows fast updates and that also allows the salient information to be easily extracted.

The primary data structure used by the online measurement system is the *weighted calling context graph*, or WCCG. It maintains context-sensitive profile information for method calls. It is similar to the Calling Context Tree data structure introduced in [9], with the major distinction being that a WCCG may have multiple roots since the entire call stack may not be analyzed at a sample point due to time constraints.

This chapter describes the WCCG. Section 4.1 describes the data structure, while section 4.2 describes how the WCCG is built and maintained.

4.1 Description of data structure

The weighted calling context graph is a graph (not necessarily connected) whose edges go from *call sites* to *methods*. Each edge corresponds to calls from the call site that resolve to the method. (Different call sites are treated as different nodes for greater precision.) The edges have weights, which correspond to approximate frequencies with which each call is made.

In addition to the weights on the edges, each node can also contain extra information. Each method node can contain a number that corresponds to the approximate time spent in the body of method. (This does not include time spent in methods called by this method.) It can also contain the results of the specialization benefit prediction algorithm described in 3.4.16, and the results of other pertinent analyses like side-effect analysis.

Edges can also contain tables that break down the edge frequencies into more precise information. For example, a table associated with an edge can contain combinations of parameter types and associated frequencies. The numbers associated with each entry correspond to the approximate frequencies with which the edge was traversed when the types of the parameters matched the types listed in the entry. There can also be tables that record frequency information for combinations of parameter values or parameter field values.

4.2 Building and maintenance

Information in the WCCG can come from four different places. The first is from a timer-tick based profiler, which periodically samples the currently executing context. The second is from instrumented code that records method entries, exits, etc. The third is from analyses such as those found in BC2IR. Finally, the WCCG can be preloaded with values. The following sections cover each of these in turn.

4.2.1 Timer-tick based profiler

The first source for WCCG information is the timer-tick based sample profiler. Timer-tick based sample profiling is a very inexpensive way to quickly determine where the most time is being spent in the system. However, it is not very precise. The timer-tick based profiler is used as the default profile strategy upon virtual machine startup.

The timer-tick based profiler works as follows. An interrupt is set up to fire at a defined frequency (approximately every millisecond.) When the interrupt fires, the interrupt service routine looks at the program counter and the stack pointer to determine which method is currently executing. It also begins walking the stack, constructing the calling context. It continues walking the stack until it reaches a cycle or the top of the stack, or runs out of budgeted time. It then updates the WCCG with the new information — namely, that a timer tick occurred in a given calling context, constructing new nodes and edges when necessary.

4.2.2 Instrumented code

The second source for WCCG information is instrumented code. When the controller decides that it would like more precise information for some of the methods, it introduces instrumentation to measure information such as the exact number of times that an edge in the WCCG is traversed, the exact amount of time spent in a method, or other conditions present during method calls. (See section 5.4.1.)

The instrumented code makes a call to the appropriate routine in the online measurement system to record the pertinent information. The routine analyzes the

stack to find the calling context, traverses the WCCG to find the correct edge or node, and updates the information stored there.

4.2.3 Analyses

The third source for WCCG information is code analyses. Some analyses, such as specialization benefit prediction and side effect analysis, record their results in the WCCG (as well as in other locations.)

4.2.4 Preloaded WCCG

While the online measurement system was still in development, we constructed the WCCG in an offline step. This functionality still exists in the system, and we use it extensively for testing purposes. One could easily envision using this functionality to allow profile information from prior executions to be used in the current execution.

4.2.5 Invalidating the WCCG

Program profiles typically change over time, as the program goes through different modes of operation: initialization, input, computation, output, etc. The program profile may be entirely different from one mode to another, and the decisions made assuming the conditions for one mode may be suboptimal when executing in a different mode.

Furthermore, as the system runs and more data is collected, newer data becomes less and less important due to the fact that as data is collected, recent data becomes a smaller and smaller portion of the whole. If anything, more recent data should be weighted more heavily than earlier, potentially-more-obsolete data.

Rather than attempt to age data or invalidate portions of the data, we decided to simply throw away the WCCG whenever it gets old or obsolete due to new information, code transformations, etc. We decided that it would be too complicated and time consuming to implement aging and/or invalidation of data. Besides, changes in the system will often cause system performance to change; by throwing away the

data, we are insured that future decisions will only be based on the new data. This also avoids the problem of the "incredible expanding data"; because data is thrown away, the memory consumption of the WCCG never grows large.

Chapter 5

The controller

kaze fukeba — White waves of blossoms
hana no shiranami — cascade over the rocks
iwa koete — when the breezes blow.
watariwazurau — How difficult to ford
yamakawa no mizu — the waters of the mountain stream!

— *Lady Nijō, Towazugatari (The Confessions of Lady Nijō), forthteenth century.*

The controller is at the heart of our dynamic compilation system. It has three main functionalities. First, it makes the decisions of when and how to invoke the dynamic compiler. Second, it drives the online measurement system by selectively introducing instrumented code for profiling and data gathering purposes. Third, it gives hints to the garbage collector about the placement of code in memory.

The first section (5.1) covers how the controller decides which call sites to inline. The second section (5.2) covers how the controller decides which methods to specialize with respect to parameters. The third section (5.3) describes how the controller decides which methods to specialize with respect to field values. The fourth section (5.4) covers how the controller directs the online measurement system, and the fifth section (5.5) describes how the controller gives hints to the garbage collector for code relocation purposes.

5.1 Deciding which call sites to inline

An object oriented language such as Java has many small method calls. The BC2IR also has support for "on-the-fly inlining" (see section 3.4.12), which has the benefit that all the on-the-fly optimizations performed by BC2IR will automatically be performed on the inlined code; it is not necessary to repeat those optimizations after inlining. Therefore, inlining is one of the most important optimizations in our system. However, needlessly inlining call sites that are rarely executed or that don't considerably improve code quality is wasteful; because inlining increases code size and therefore compilation time, and compilation time counts against run time in a dynamic compiler, one must be very careful about what one inlines.

The controller decides where and how to apply inlining using the information in the WCCG along with static information about the method and the call site. The controller traverses the WCCG and looks for edges with execution weights that are greater than a certain threshold.[1] For all such edges, it calculates a utility function to decide whether or not to request that the call site be inlined. The utility function is the combination of a number of metrics with different weighting functions. The strongest weighted factor is the method size (smaller is better,) followed by the execution frequency (larger is better.) If specialization benefit prediction information is available and the call site has one or more of its parameters as a constant or as a known type, the expected benefit from specializing the method with respect to that constant/type is also added.[2] If the result of the utility function is greater than a certain threshold, the call site is added to the "inline plan" of the caller method. See Figure 5-1 for the algorithm.

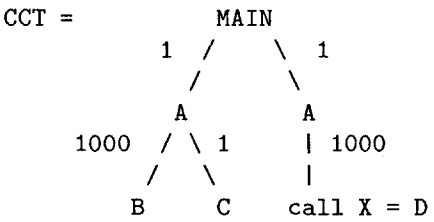
[1] If only timer tick profile information is available, execution weights are synthesized by looking at the number of timer ticks in the caller vs. the callee.

[2] If one or more parameters is a constant but the actual specialization benefit prediction information is not available, it uses a default estimate for each of the constant parameters.

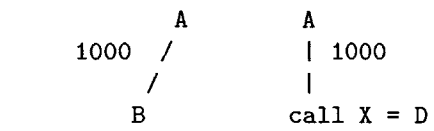
```
OUTPUT:

WORKLIST = list of root methods to be compiled with inlining.  For
each method M in WORKLIST there is an "inlining plan" that specifies
what inlining should be done when compiling M

ALGORITHM:

* Select a subset of WCCG edges for inlining, based on edge frequency,
  target method size, and how many parameters to the call are
  constant, along with specialization benefit prediction information,
  if it is available.

* Define an Inlining Graph (IG) to be the subgraph of the WCCG that
  only contains edges selected for inlining.  In general, the IG will
  be a forest of small trees.

* Initialize WORKLIST to an empty list

* FOR EACH tree T in the IG such that T contains >= 2 nodes DO
      Let M := method corresponding to root node of T
      IF ( there exists a WORKLIST entry with M as the root method ) THEN
          Merge decision to inline all edges in tree T into existing plan for
          method M
          (This is the case in which there are multiple trees in the IG that
          have the same method as root.  We are not performing cloning -
          rather, we are effectively merging all IG trees that have the same
          root method.)
      ELSE
          Create a new WORKLIST entry with M as the root method;
          Set PLAN field for method M to an inlining plan that inlines all
          call sites corresponding to edges in tree T
      END IF
  END FOR
```

Figure 5-1: Algorithm for building inlining plan

Assume that the CCT returned by online measurements is as follows
(call X is a virtual method call; all other calls are direct calls):



Assume that the subset of edges selected for the Inlining Graph is as
follows:



Then the planning phase will create a WORKLIST with a single entry
that has the following "inlining plan":

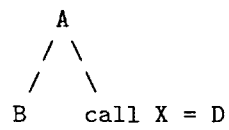


Figure 5-2: Example of inlining plan

```
ALGORITHM:

FOR EACH method M with time spent > threshold DO
    initialize totals to zero
    FOR EACH edge E with target M in WCCG DO
        FOR EACH combination C of parameter types/values associated with E DO
            Let S := specialization benefit prediction for C
            increment total associated with C by frequency * S
        END FOR
    END FOR
END FOR

FOR EACH method M with total > threshold DO
    add method M to list of methods to specialize, sorted by total
END FOR
```

Figure 5-3: Algorithm for choosing when to specialize with respect to method parameters

5.2 Specialization with respect to method parameters

The controller can also select to specialize with respect to method parameters. Method specialization is somewhat similar to inlining, but there is a key difference — a single specialized method can be shared by multiple call sites; an inlined call, by definition, is only at a particular call site. This also means that call sites can be added to point to a specialized method later.

After the controller is finished making inlining decisions, it chooses which methods to specialize and how to specialize on them. It uses the extra table information associated with the edges in the WCCG. (See section 4.1.) Section 5.2.1 describes how the controller detects opportunities to specialize on parameter types and values.

5.2.1 Specializing on parameter types and values

The controller detects opportunities for specialization based on parameter types/values using the information about the frequency of combinations of parameter types/values associated with the WCCG graph edges, along with the results of the specializa-

tion benefit prediction algorithm. The controller first selects methods that have the amount of time spent in them above a certain threshold. For these methods, it looks at all edges into the method, and for each combination of parameter types/values it scales the frequency by the results of the specialization benefit prediction algorithm for that parameter combination. This number is added to a total associated with that combination of parameter types/values for the method. After the controller finishes, if any of the totals are above a certain threshold, the controller chooses to specialize those methods. After the dynamic compiler finishes specializing those methods, it patches the call sites of the edges who have the frequency with the given types/values to check their parameter types/values and conditionally branch to the new version.

5.3 Specialization with respect to fields

Many values that we would want to specialize on are not passed as method parameters, but are rather stored in static fields or object fields. Therefore, the controller also specializes methods with respect to object fields.

5.3.1 Static fields

Specializing a method with respect to a static field, unlike the other forms of method specialization, does not use the specialization benefit prediction algorithm. It uses the execution frequencies from the WCCG along with side-effect analysis results (see section 3.4.15).

The controller only attempts to specialize with respect to static fields that it is *confident* of. The controller is *confident* of a static field if it has performed side-effect analysis on all recently executed methods that can potentially access the static field and has added profile instrumentation to those which side-effect analysis has stated can store to the specified static field.

As the controller traverses the WCCG, it keeps track of a number for each static field that it is *confident* of and that is reported as used in the side effect analysis for each method. For each method, it scales the number of estimated uses of the given

```
ALGORITHM:

initialize totals to zero
FOR EACH method M in WCCG > threshold DO
    Let C := execution frequency of M
    FOR EACH confident field F used in M DO
        Let N := expected number of uses of F in M
        increment total for F by C * N
    END FOR
END FOR

FOR EACH confident field F DO
    IF (total for F > threshold) THEN
        FOR EACH method M that uses F DO
            Let N := expected number of uses of F * execution frequency of M
            IF N > threshold THEN
              add to list method M to be specialized w.r.t. current value of F
            END IF
        END FOR
    END IF
END FOR
```

Figure 5-4: Algorithm for choosing when to specialize with respect to a field

static field by the execution frequency of that method and adds it to the total for that static field. If a method is ever encountered in the WCCG that modifies the static field as a side effect, the number for the static field is removed and that static field is no longer tracked.

After the controller has traversed the WCCG, the number associated with each field signifies the approximate dynamic number of uses throughout the system of the given static field. The controller then considers each of these static fields in turn. If the number is above a certain threshold, the controller specializes all of the methods encountered that have an estimated number of dynamic uses of that field higher than a certain threshold with respect to the current value of the static field.

5.3.2 Object fields

The controller decides to specialize with respect to object fields in two different ways. The first method is analogous to deciding when to specialize with respect to method parameters types/values — by using the extra frequency information associated with

the edges in the WCCG.

The second method is similar to the method used to decide when to specialize with respect to static fields. The analysis proceeds in a similar manner, but the actual specialization proceeds differently — the value that the method is specialized on is taken from frequency information on the WCCG edges, and a run time check is introduced, which checks the value of the object field and branches to the correct version.

5.4 Directing the online measurement system

There is a certain synergy between the controller and the online measurement system. Not only does the controller base its decisions on the information provided by the online measurement system, but it also directs the online measurement system on what information it should try to collect.

5.4.1 Adding instrumentation

The controller adds instrumentation to method headers for information-gathering purposes. The information gathered by the instrumentation can range from a simple counter that counts the number of executions to a table with frequency information on the values of method parameters.

When the virtual machine starts up, the timer-tick based profiler is the only method of gaining information. The controller uses the information from the timer-tick based profiler to decide which methods to add instrumentation to. The controller only adds instrumentation to "hot" methods and the subgraph of methods below "hot" methods in the static call graph, up to a certain depth. This instrumentation records number of invocations of a method and amount of time spent in a method in the WCCG (see section 4.1.)

5.4.2 Adding instrumentation to evaluate specialization opportunities

The controller also adds instrumentation when the results of static analysis look promising, and it needs a way to justify performing a specialization optimization. The controller adds instrumentation to methods whose anticipated specialization benefit is high. This instrumentation records the types/values of the potentially beneficial parameters and uses the information to update frequencies stored in tables attached to the edges in the WCCG (see section 4.1.)

Another situation where the controller adds instrumentation based on a static analysis is when the controller detects that it may be beneficial to specialize with respect to a field, but it does not have the confidence to make the specialization (see sections 5.3.1 and 5.3.2, above.) In this case, it adds instrumentation to count invokation frequencies of methods that have a side effect of modifying the specified field.

5.4.3 Adding basic block level/trace level profiling

The controller also adds instrumentation to measure intramethod information. When the execution time is dominated by a small number of methods, the controller adds basic block level profiling or trace profiling to those methods. These profiling methods store the frequency information in the IR for the method (see section 3.2.2) and the information is used by the trace scheduler and the basic block reordering algorithm, among other optimizations.

For the placement of instrumentation code in basic block level profiling, we use an algorithm similar to that presented by Ball and Larus [13]. We use the maximal spanning tree on the method control flow graph to determine the optimal placement for instrumentation counters.[3] Placement of instrumentation for trace profiling also uses the Ball and Larus algorithm [14].

[3]We use a priority-first search algorithm for finding the minimum spanning tree [50].

5.4.4 Removing instrumentation

Although in the thesis we assume that instrumentation code comes free of charge, in our actual implementation this is not the case. Our system does not support the explicit removal of instrumentation code. Rather, instrumentation is implicitly removed when methods are dynamically recompiled and optimized.

5.5 Directing the garbage collector to reorder code

The last functionality of the controller is to give "hints" to the garbage collector about which pieces of code call each other often and therefore should be located near each other in memory to maximize instruction cache locality.

As the controller traverses the WCCG, it keeps track of total call frequencies between methods in a separate graph. After the controller completes, the graph is passed to the garbage collector and used during the next garbage collection as a guide on how to relocate code in memory. (The garbage collector uses a relocation algorithm similar to that presented in [117].)

Chapter 6

Performing the dynamic optimizations

ashibe kogu	For how many times —
tananashiobune	how many dozens of trips —
ikusotabi	might the tiny boat
yukikaeruran	go and return among the reeds
shiru hito mo nami	with none to know of it?

— *Ariwara no Narihira, Ise monogatari (Tales of Ise), tenth century.*

Once the controller identifies opportunities for dynamic optimization, the compiler needs to perform them. One interesting aspect of our system is the fact that it can speculatively perform specialization operations. This stems from the fact that the dynamic compiler is in the context of a Java Virtual Machine that can dynamically load classes, and therefore must be able to support incomplete information. If the dynamic compiler always made the most pessimistic decisions, then it wouldn't be able to make many optimizations at all. Instead, it speculatively performs the optimization, but leaves a way to *back out* of the optimization if necessary.

This ability to *back out* of optimizations is necessary for adequate performance and correct functionality in the presence of dynamic class loading. But because

```
ALGORITHM:

  FOR EACH root method M in WORKLIST DO
      Compile method M while obeying inlining decisions in plan for M ;
      If a call site to be inlined is a virtual call, then generate
      dual-path code with a target test and perform inlining only for the
      case where the target test corresponding to the desired WCCG edge
      returns true.
  END FOR
```

Figure 6-1: Algorithm for performing inlining based on an inlining plan from the controller

this mechanism exists in the system, we can also use it to speculatively perform specialization optimizations that selectively ignore pieces of the program that execute rarely. The process of *backing out* can be time-consuming, but if the event is rare enough and the specialization benefit is good enough, it may be worthwhile.

This chapter describes the speculative dynamic optimizations performed by our system, and the back out mechanism. Section 6.1 describes speculative inlining. Section 6.2 describes method specialization on parameters, while section 6.3 describes specialization on fields.

6.1 Speculative inlining

Inlining is performed according to an *inlining plan* (see section 5.1). The dynamic compiler takes the inlining plan and performs the inlining decisions listed in the plan. See figure 6-1 for the algorithm.

If the inlining is *speculative*, meaning that the given call site can potentially call a method body that is different than the inlined body, then a run time check of the method target is inserted. The run time check is implemented as a CHECK_TARGET instruction. The CHECK_TARGET is similar to other instructions that maintain exception semantics, like NULL_CHECK, BOUNDS_CHECK, etc., in that it does not end a basic block. One can think of the target being different as an 'exceptional event', just like a run time exception, but instead of walking the stack for an exception handler when the

exception occurs, it branches to code that calculates the correct method target and branches there. See figure 6-2 for an example written in high level code.

Some of the CHECK_TARGET instructions are eliminated due to the more precise type information propaged from prior CHECK_TARGET instructions. For example, in many nested inlining cases, only the initial call requires the CHECK_TARGET instruction, because the first target matching implies that other targets will match.

6.2 Method specialization on parameters

The dynamic compiler also supports method specialization with respect to parameter types/values. See section 5.2.1 for a description of how methods are selected to be specialized.

Specializing a method with respect to some parameter types/values works as follows. The dynamic compiler compiles a version of the method assuming that the parameters have the given constraints. (See section 3.4.) Then, for all call sites that the specialization plan specifies, it rewrites the call site to instead call a routine that checks the constraints on the parameters. If the constraints are satisfied, the specialized version is called, otherwise the (more) general version is called.

6.3 Method specialization on fields

Method specialization on fields is more complicated due to the Java memory model. The specification states that fields must be reread at every synchronization point [102]. Thus, when specializing with respect to field values, we compile a specialized version of the method assuming that the given field does not change values, but we include a run time check of the value at every (potential) synchronization point where the field value is potentially used after that point. If the value does change, we back out to the general version. See the next section.

```
ORIGINAL CODE FOR METHOD A:

... A ( ... ) {
    . . .
    while (...) {
      call B() ;
    }
    . . .
    call C() ;
    . . .
    while (...) {
      call this.X() ;
    }
}

STRUCTURE OF COMPILED CODE FOR METHOD A (AFTER INLINING):

... A ( ... ) {
    . . .
    while (...) {
      /* inlined copy of B() goes here */
    }
    . . .
    call C() ;  // this call does not get inlined
    . . .
    while (...) {
      try {
          if ( this.X != D ) throw WrongTarget;
          /* inlined copy of D() goes here */
      } catch (WrongTarget e) {
          call this.X() ;
      }
    }
}
```

Figure 6-2: Example of profile-directed speculative inlining

6.4 How we back out

When we are executing a method that is specialized with respect to a field having a particular value and that value changes during the execution of the method, we need to "back out" to a version of the method that is not specialized with respect to that field.

Backing out is accomplished by enforcing a mapping between synchronization points in the specialized method and the original method. Each synchronization point in the specialized method corresponds to exactly one synchronization point in the unspecialized method, and all variables that are live in the unspecialized method are also marked as live in the specialized method. When a "back out" needs to occur, affected stack frames are rewritten to match the layout of the unspecialized version, and all return addresses are updated to point to the unspecialized version. Control flow then continues in the unspecialized version.

Backing out of specialization when the program profile changes A back out mechanism is not (strictly) necessary for speculative inlining and specialization with respect to parameters, because even if the assumptions change, the code is still correct. However, as the program profile changes, the code may become inefficient. In this case, recompilation is triggered by the basic block level instrumentation (see section 5.4.3.) The controller adds basic block level profiling to a method when it begins to take a significant portion of time. If a large number of mispredictions occur due to the fact that an earlier assumption was incorrect, the time in the method will increase. If the time spent in the method is still small, then although there are a large number of mispredictions, it does not significantly adversely affect run time because only a small portion of the time is spent in the method anyway. If it does significantly increase the method time, however, then the controller will eventually recompile the method and use the updated edge weights in the WCCG, and make a decision that is more optimal for what is currently occurring in the program.

Chapter 7

Results

ayamegusa	Eagerly I await
oinishi kazu o	the Fifth Month Festival,
kazoetsutsu	the time when people
hiku ya satsuki no	pull up the sweet flags, counting
sechi ni mataruru	the sum of their well-grown roots.

— *Michitsuna's Mother, Kagerō nikki (Gossamer Journal), tenth century.*

In the chapter we present the effectiveness of our compiler on a large transacting processing benchmark program called "Portable BOB" (Portable Business Object Benchmark) [22]. Section 7.1 provides a description of the benchmark we used and a justification of why we used it and section 7.2 shows the effectiveness of the concurrent optimizations in BC2IR. Section 7.3 provides some preliminary performance results of the effectiveness of our dynamic optimizations.

7.1 Description of benchmark

"Portable BOB", or the "Portable Business Object Benchmark" [22], is a benchmark designed to quantify the performance of simple transactional server workloads written in Java. The source code, minus libraries, is approximately 18,000 lines. It models

a typical electronic order entry business scenario. Its business model is based on the business model used by TPC Benchmark C^{TM}, Standard Specification, Revision 3.3, April 8, 1997.[132]

This benchmark was chosen because it is a good model of the typical application that a dynamic compilation system such as this would be used for — a long running, dynamic server-type application. It is also written like a typical modern business application — it was written to emulate common coding practices, not necessarily practices providing the best possible performance. Another reason that it is an interesting benchmark is that it does not have any obvious way of benefitting from dynamic compilation. Some applications can have huge gains from dynamic compilation through obvious specialization opportunities [118, 74]. However, by showing the performance and effectiveness on a benchmark that does not have obvious specialization opportunities, we can better evaluate whether such a dynamic compilation system is useful in general, rather than in special, specific cases only.

7.2 Effectiveness of concurrent optimizations

This section shows the effectiveness of the concurrent optimizations in BC2IR on actual code. For each of the following sections, the stated numbers are for all of the code for Portable BOB benchmark, minus the libraries, with different BC2IR options.

7.2.1 Null pointer checks eliminated

Figure 7-1 shows the number of null pointer checks in the IR with and without the null pointer check elimination optimization (described in section 3.4.8). As the figure shows, over 90% of the null pointer checks are eliminated by the simple null pointer check elimination optimization. The optimization is especially effective because it reduces compile-time, due to the reduction in the size of the IR.

7.2.2 Type checks eliminated

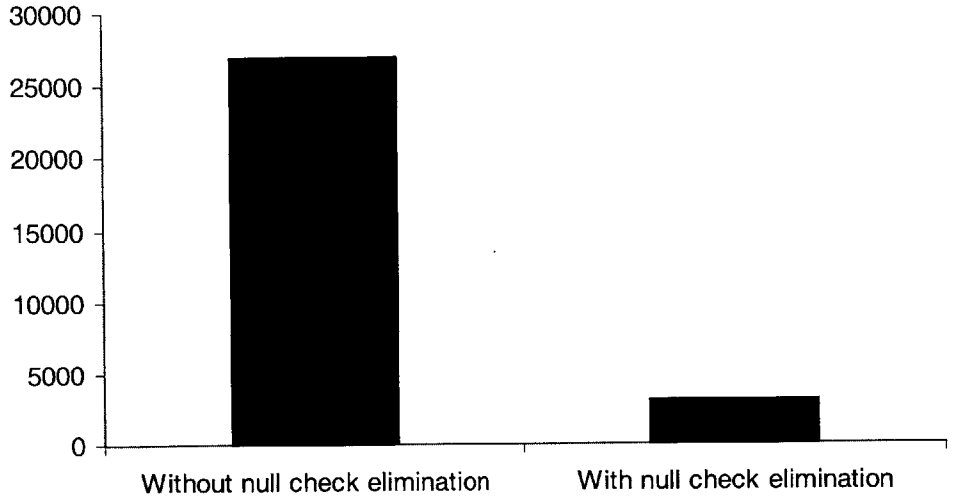


Figure 7-1: Static number of null pointer checks eliminated by the null pointer check elimination optimization in BC2IR

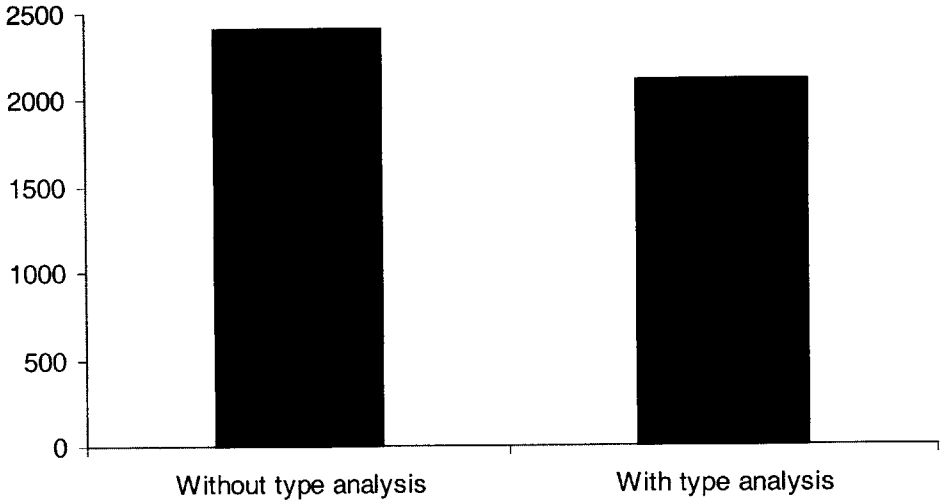


Figure 7-2: Static number of run time type checks eliminated by type analysis in BC2IR

Figure 7-2 shows the static number of runtime type checks in the IR with and without type analysis in BC2IR (see section 3.4.9). As the figure shows, a significant number of type checks (mostly `checkcast` instructions) are eliminated due to the type analysis performed by BC2IR.

7.2.3 Reduction in size of IR

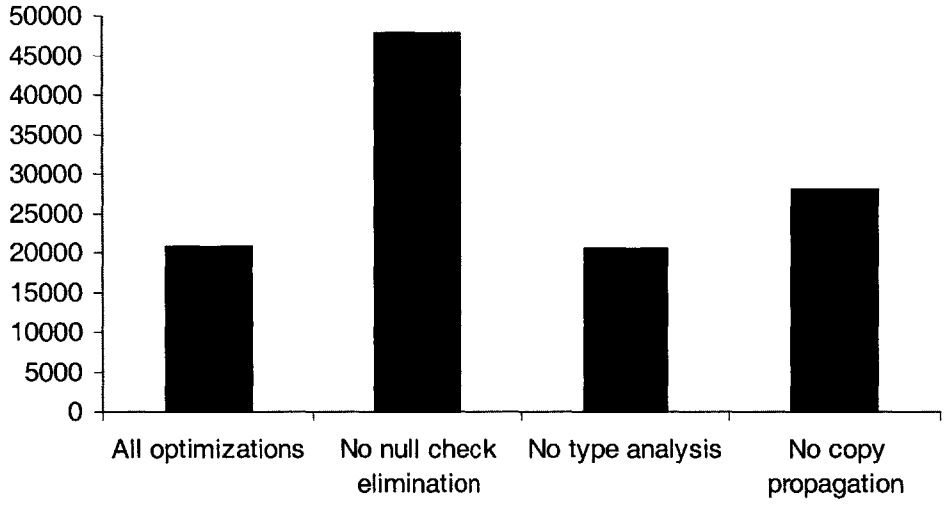


Figure 7-3: Size of the generated IR (in number of instructions) with different BC2IR optimization options

Figure 7-3 shows the size of the IR (in number of instructions) with different BC2IR optimization options enabled. Null pointer check elimination gives the largest gain. Copy propagation is also very effective.

7.2.4 Reduction in code generation time

Figure 7-4 shows the amount of time taken to compile all of the methods in the benchmark, with different BC2IR optimization options. Because the compilation time is strongly dependent on the size of the IR, optimizations that reduce the size of the IR also reduce compilation time.

7.2.5 Reduction in run time

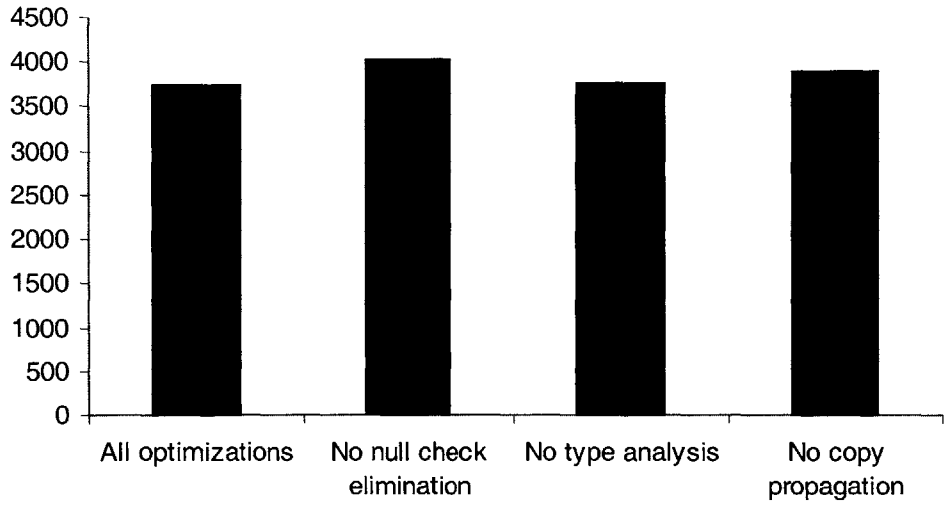


Figure 7-4: Time (in ms) spent in dynamic compilation of pBOB with different BC2IR optimization options

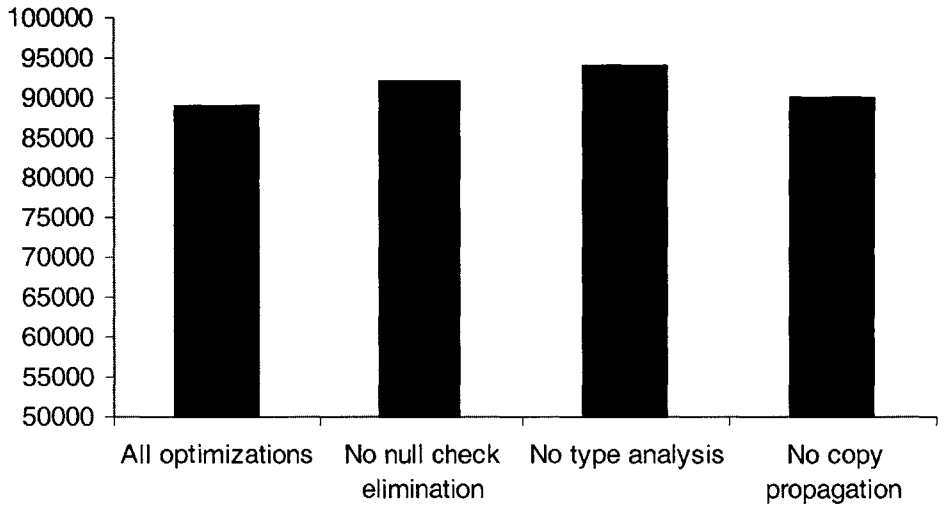


Figure 7-5: Execution times (in ms) of pBOB compiled with different BC2IR optimization options

Figure 7-5 shows the reduction in execution time when different optimization options are enabled in BC2IR. This shows that the optimizations not only reduce dynamic compile time, but also improve code quality.

7.3 Effectiveness of dynamic optimizations

This section presents some preliminary performance results on the effectiveness of the dynamic optimizations. More results will appear in future publications.

7.3.1 Effectiveness of speculative inlining

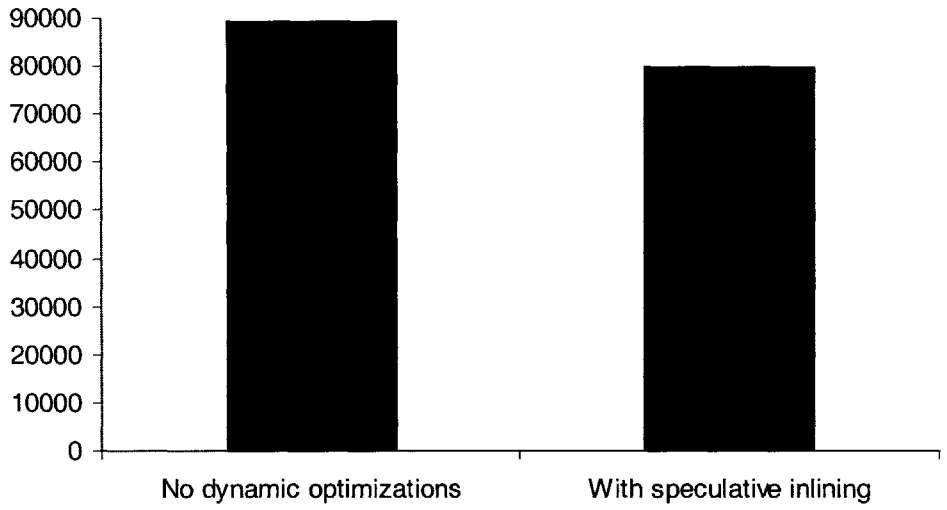


Figure 7-6: Execution times of pBOB (in ms) with and without speculative inlining enabled

Figure 7-6 shows the effect of the speculative inlining optimization described in section 6.1. Enabling speculative inlining has a major impact because of the object-oriented style of our benchmark — most calls are virtual, even though they only have a single target, and most method bodies are small.

Chapter 8

Related Work

The sound of the Gion Shōja bells echoes the impermanence of all things...
The color of the *śāla* flowers reveals the truth that the prosperous must decline.
The proud do not endure; they are like a dream on a spring night;
The mighty fall at last, they are as dust before the wind....

— *Heike monogatari (The Tale of the Heike), thirteenth century.*

8.1 BC2IR

The BC2IR step (see chapter 3) incorporates ideas from many different areas. This section compares BC2IR to prior work in related areas.

8.1.1 Converting stack based code to register based code

The problems of compiling stack-based code into efficient code for register machines are well known in the programming language Forth [67, 68]. However, Forth uses a more general stack machine model in which the stack depth can differ depending on which path through the code is taken, and so the stack cannot always be eliminated. Because the Java specifications state that at a given program point the stack will always be the same size regardless of the path, the stack can always be eliminated

and the process of translation is greatly simplified.

8.1.2 Java compilers

This section compares current work on Java compilers to the BC2IR algorithm described in chapter 3. CACAO is a Just-in-Time compiler for the DEC Alpha [95, 69]. It translates Java bytecode into a simple intermediate representation, performs register allocation, and then emits target code. Like BC2IR, it keeps track of a stack state as bytecodes are parsed, and attempts to reduce the number of superfluous copy instructions generated and registers used. Unlike BC2IR, it requires multiple passes over the code: once to extract the bytecodes and place them into fixed length instructions, another to compute basic block and control flow information, and a third to generate the IR. It does not support combining other analyses with IR generation. The resulting IR is designed for rapid code generation, rather than for performing optimization. DAISY is a VLIW architecture that supports JIT compilation of different architectures, including Java bytecode [63]. It supports single pass translation by performing a depth first traversal of the bytecode, but it does not perform any concurrent analyses or optimizations. Briki [39, 38] is an extension to the freely available Kaffe JIT compiler [136] to allow it to perform more advanced optimizations. It starts from the Kaffe IR representation and computes a control flow graph and def-use information.

Caffeine is a static Java compiler that includes a stage that generates IR from bytecode [85, 86]. It supports both a simple translation scheme that maintains the stack and a more sophisticated one that uses registers exclusively. The Cream system is a static optimizer for Java bytecode [40]. It transforms bytecode in a nine-step process, including construction of a control flow graph and inference of the stack depth at all points. It eliminates the stack abstraction by treating different stack depths as different registers. Toba [121] and Harissa [113, 112] convert Java class files to C, using a similar technique to remove the stack abstraction. Because they are designed for static compilation, none of these systems attempt to perform any optimizations during translation. They count on later optimization phases to remove

extra registers and redundant copy instructions.

8.1.3 Abstract Interpretation

Abstract interpretation (or alternatively, symbolic execution) has mainly been used in logic and functional languages. It is often used to develop provably correct program analyses [21, 51]. It also has been recognized as a powerful technique for performing flow sensitive data flow analyses [78] and has been used in highly optimizing compilers [93, 77]. As far as we know, BC2IR is the first abstract interpretation engine that operates on Java bytecode.

8.1.4 Combining analyses and negative time optimization

Various sources remark on the fact that combining analyses can not only be more efficient, but lead to better results than running the analyses separately [41, 111]. A system for intra- and interprocedural data flow analysis presented in [25] allows one to run multiple analyses "in parallel" to achieve the precision of a single monolithic analysis while preserving modularity and reusability.

In [41], Click mentioned that finding constants and common subexpressions while parsing reduced the peak intermediate representation size, which reduced total compilation time by 10% due to the fact that later global analyses were faster. The fact that some optimizations can reduce the amount of time spent in later optimizations is also sometimes mentioned in discussions of the order in which to perform optimizations [7, 111]. We are not aware of any other work on negative-time optimization.

8.2 Speculative inlining

8.2.1 Specialization benefit prediction

We believe that the specialization benefit prediction algorithm outlined in section 3.4.16 is unique. There has been some work on attempting to predict the benefit of inlining [58, 61] based on 'inlining trials'. The compiler experimentally performs inlining

and records the result in a database that is consulted to guide future inlining decisions. There has also been work on designing accurate static predictors of profile information [134, 116].

8.2.2 Backing out of specialization optimizations

There has been some work on partial program invalidation with specialization optimizations and selective recompilation [24]. Our system is different — it incorporates run-time checks to validate assumptions and has the ability to revert a currently executing optimized method to an unoptimized version.

Chapter 9

Conclusion

sakite toku	It is painful
chiru wa ukeredo	that blossoms must scatter so soon,
yuku haru wa	but please view them next year,
hana no miyako o	when the spring that now departs
tachikaerimi yo	shall have returned to the city.

— *Ō-no-myōbu, Genji monogatari, eleventh century.*

Dynamic compilation is an attractive research area because it holds the promise of significant performance improvements for some applications. Automatically identifying and exploiting opportunities for dynamic optimization is a difficult problem, but holds incredible promise. It is one more step towards the eventual goal of allowing programmers to concentrate on system aspects other than performance, such as maintainability, reliability, etc.

This thesis presented a dynamic compilation system that is suitable for relatively long-running dynamic Java applications, such as server applications. The system is centered around an abstract interpretation engine that converts Java bytecode into a register-based intermediate representation in a single pass, concurrently performing numerous optimizations and analyses. The conversion process is single pass due to efficiency considerations. This thesis also presented data structures and routines for gathering and organizing runtime performance data, and techniques for using that

information to improve system performance. Finally, it presented some preliminary results that show that the dynamic compilation techniques are reasonable.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast and effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices*, 33(5):280–290, May 1998.

[2] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type Inference SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In O. Nierstrasz, editor, *European Conference on Object-Oriented Programming*, LNCS 707, pages 247–267, Kaiserslautern, Germany, July 1993. Springer.

[3] Ole Agesen. Design and implementation of Pep, a Java just-in-time translator. *Theory and Practice of Object Sytems*, 3(2):127–155, 1997.

[4] Ole Agesen and David Detlefs. Finding references in Java stacks. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[5] Ole Agesen and David Detlefs. Garbage collection and live variable type-precision and liveness in Java Virtual Machines. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[6] Ole Agesen and Urs Hoelzle. Type feedback vs. concrete type analysis: A comparison of optimization techniques for object-oriented languages. Technical Report TRCS 95-04, Computer Science Department, University of California, Santa Barbara, March 1995.

[7] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[8] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño — a compiler-supported Java^{TM} virtual machine for servers. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software (WCSSS'99)*, Atlanta, Georgia, May 1, 1999. ACM Press.

[9] Glen Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.

[10] P. H. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[11] Wing Yee Au, Daniel Weise, and Scott Seligman. Automatic generation of compiled simulations through program specialization. In ACM-SIGDA; IEEE, editor, *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 205–210, San Francisco, CA, June 1991. ACM Press.

[12] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. *International Journal of Parallel Programming*, 26(1):43–64, February 1998.

[13] Ball and James Larus. Optimal profiling and tracing of programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 59–70. ACM Press, January 1992.

[14] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[15] L. Beckman et al. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[16] A. Berlin. Partial evaluation applied to numerical computation. In *Conference on Lisp and Functional programming*, pages 139–150. ACM SIGPLAN, SIGACT, SIGART, 1990.

[17] M. R. Blair. Descartes: A dynamically adaptive compiler and run-time system using continual profile-driven program multi-specialization.

[18] R. G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme.* PhD thesis, Indiana University, Bloomington, IN, February 1997.

[19] Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proceedings of the IEEE Conference on Computer Languages (ICCL).* IEEE, April 1998.

[20] 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^{TM}. In *Proceedings of the ACM SIGPLAN '99 Java Grande Conference*, San Francisco, CA, June 12–14, 1999. ACM Press.

[21] G. L. Burn. The abstract interpretation of functional languages. In G. L. Burn, S. J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*, Workshops in Computer Science, Isle of Thorns Conference Centre, Chelwood Gate, Sussex, UK, March 1993. Springer-Verlag.

[22] Business object benchmark for java. http://www.as400.ibm.com/developer/performance/bob/jbob400paper.pdf.

[23] Gary Carleton, Knud Kirkegaard, and David Sehr. Programmer's toolchest: Profile-guided optimizations. *j-DDJ*, 23(5):98, 100–103, May 1998.

[24] C. Chambers, J. Dean, and D. Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. In *Proceedings of the*

17th International Conference on Software Engineering, pages 221–230, April 1995.

[25] C. Chambers, J. Dean, and D. Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report TR-96-11-02, University of Washington, Department of Computer Science and Engineering, November 1996.

[26] C. Chambers, J. Dean, and D. Grove. Whole-program optimization of object-oriented languages. Technical Report TR-96-06-02, University of Washington, January 28, 1997.

[27] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF .. *Lisp and Symbolic Computation*, 4:243–281, 1991.

[28] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Object-Oriented Programming Language.* PhD thesis, Stanford University, Palo Alto, California, March 1992.

[29] Craig Chambers, David Grove, Greg DeFouw, and Jeffrey Dean. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, October 1997.

[30] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for Java. In *Proceedings of the 1999 ACM Workshop on Languages for Compilers and Parallel Computing (LCPC'99)*. ACM Press, 1999. submitted.

[31] Craig Chambers and David Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *ACM SIGPLAN Notices*, 24(7):146–160, July 1989.

[32] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *Lisp and Symbolic Computation*, 4(3):283–310, July 1991.

[33] Pohua P. Chang, Daniel M. Lavery, and Wen mei W. Hwu. The effect of code expanding optimizations of instruction cache design. Technical Report CRHC-91-18, Coordinated Science Lab, University of Illinois, January 1992.

[34] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–369, May 1992.

[35] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 1991.

[36] W. Chen, R. Bringmann, S. Mahlke, S. Anik, T. Kiyohara, N. Warter, D. Lavery, W.-M. Hwu, R. Hank, and J. Gyllenhaal. Using profile information to assist advanced compiler optimization and scheduling. *Lecture Notes in Computer Science*, 757:31–??, 1993.

[37] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, Toulouse, France, September 6–10, 1999. ACM Press.

[38] M. Cierniak and W. Li. Optimizing Java bytecodes. *Concurrency - Practice and Experience*, 9(6):427–444, June 1997.

[39] Michal Cierniak and Wei Li. Briki: A flexible Java compiler. Technical Report TR621, University of Rochester, Computer Science Department, May 1996. Thu, 17 Jul 97 09:00:00 GMT.

[40] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. In Geoffrey C. Fox and Wei Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997. ACM.

[41] Jr. Click, Clifford Noel. Combining analysis, combining optimizations. Technical Report TR95-252, Rice University, April 24, 1998.

[42] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, 1990.

[43] C. Consel, L. Hornof, F. Noel, and J. Noye. A uniform approach for compile-time and run-time specialization. *Lecture Notes in Computer Science*, 1110:54–??, 1996.

[44] C. Consel and S. C. Khoo. Semantics-directed generation of a prolog compiler. Technical Report YALEU/DCS/RR-781, Yale University, New Haven, Connecticut, May 1990.

[45] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In ACM, editor, *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages, January 21–23, 1991, Orlando, FL*, pages 14–24, New York, NY, USA, 1991. ACM Press.

[46] Charles Consel and Olivier Danvy. Partial evaluation: Principles and perspectives. A previous version of this tutorial appeared in the proceedings of the 20th Annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, Jan 1993, South Carolina, January 1993.

[47] Thomas M. Conte, Kishore N. Menezes, and Mary Ann Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 36–45, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[48] Thomas M. Conte, Burzin A. Patel, and J. Stan Cox. Using branch handling hardware to support profile-driven optimization. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 12–21, San Jose,

California, November 30–December 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[49] Thomas M. Conte, Burzin A. Patel, Kishore N. Menezes, and J. Stan Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24(2):187–206, April 1996.

[50] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[51] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282. ACM, ACM, January 1979.

[52] P. Cregut. *Machines a environnement pour la reduction symbolique et l'evaluation partielle.* PhD thesis, Universite Paris VII, 1991.

[53] R.R. Henry C.W. Fraser and T.A. Proebsting. Burg — fast optimal instruction selection and tree parsing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[54] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. Technical Report CS-88-16, Department of Computer Science, Brown University, October 1988. Sun, 13 Jul 1997 18:30:15 GMT.

[55] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing Static Single Assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[56] Ron K. Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, May 1995.

[57] O. Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37(6):315–322, 1991.

[58] Dean and Chambers. Training compilers to make better inlining decisions. Technical Report TR 93-05-05, University of Washington, 05 1993.

[59] J. Dean, C. Chambers, and D. Grove. Identifying profitable specialization in object-oriented languages. Technical Report TR-94-02-05, University of Washington, Department of Computer Science and Engineering, February 1994.

[60] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–??, 1995.

[61] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Conference on Lisp and Functional programming*, pages 273–282. ACM SIGPLAN, SIGACT, SIGART, LISP Pointers, July-September 1994.

[62] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices*, 30(6):93–102, June 1995.

[63] Kemal Ebcioglu, Erik Altman, and Erdem Hokenek. A Java ILP machine based on fast dynamic compilation. In *MASCOTS '97 — International Workshop on Security and Efficiency Aspects of Java*, 1997.

[64] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26,4 of *ACM SIGCOMM Computer Communication Review*, pages 53–59, New York, August 26–30 1996. ACM Press.

[65] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. *ACM SIGPLAN Notices*, 31(5):160–170, May 1996.

[66] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–272, San Jose, California, October 4–7, 1994. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[67] M. Anton Ertl. A new approach to Forth native code generation. In *euroForth '92 Conference Proceedings*, pages 73–78, MPE Ltd., 133 Hill Lane, Southampton. SO1 5AF UK, October 1992. Forth Interest Group.

[68] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines.* Dissertation, Technische Universität Wien, Austria, 1996.

[69] M. Anton Ertl. Fast high-quality JavaVM compilation. Seminar given at IBM Watson Research Center, Yorktown Heights, August 1997.

[70] M. Anton Ertl. Optimal code selection in DAGs. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1999.

[71] J. A. Fisher. Trace scheduling : A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, 1981.

[72] J. A. Fisher. *Trace Scheduling-2, an extension of trace scheduling.* Hewlett-Packard Laboratories, 1992.

[73] C. D. Garrett, J. Dean, D. Grove, and C. Chambers. Measurement and application of dynamic receiver class distributions. Technical Report TR-94-03-05, University of Washington, Department of Computer Science and Engineering, March 1994.

[74] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report TR 97-03-03, University of Washington, 03 1997.

[75] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. *ACM SIGPLAN Notices*, 30(10):108–123, October 1995.

[76] Sheila Harnett and Margaret Montenyohl. Towards efficient compilation of a dynamic object-oriented language. In *Proceedings of the 1992 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 82–89, San Francisco, U.S.A., June 1992. Association for Computing Machinery.

[77] L. Harrison. Abstract interpretation in a production C/C++ compiler. *Dagstuhl-Seminar on Abstract Interpretation*, August 1995.

[78] L. Harrison. Can abstract interpretation become a mainstream compiler technology? *Lecture Notes in Computer Science*, 1302:395–??, 1997.

[79] R. R. Heisch. Trace-directed program restructuring for AIX executables. *IBM Journal of Research and Development*, 38(5):595–603, September 1994.

[80] T. H. Hickey and D. A. Smith. Toward the partial evaluation of CLP languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 43–51. New York: ACM, 1991.

[81] Urs Hölzle and Ole Agesen. Dynamic versus static optimization techniques for object-oriented languages. *Theory and Practice of Object Sytems*, 1(3):167–188, 1995.

[82] Urs Holzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. *Lecture Notes in Computer Science*, 512:21–??, 1991.

[83] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Notices*, 27(7):32–43, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[84] Hotspot compiler. http://java.sun.com/javaone/sessions/slides/TT06/title.htm.

[85] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In IEEE, editor, *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, December 2–4, 1996, Paris, France*, pages ??–??, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[86] Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C. Gyllenhaal, and Wen mei W. Hwu. Optimizing NET compilers for improved Java performance. *Computer*, 30(6):67–75, June 1997.

[87] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In Michael Yoeli and Gabriel Silberman, editors, *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242–251, Jerusalem, Israel, June 1989. IEEE Computer Society Press.

[88] Intel C/C++ compiler plug in. http://developer.intel.com/design/perftool/icl24/icl24wht.htm.

[89] J. S. Cox, D. P. Howell and T. M. Conte. Commercializing profile-driven optimization. In Trevor N. Mudge and Bruce D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 221–228, Los Alamitos, CA, USA, January 1995. IEEE Computer Society Press.

[90] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.

[91] N.D. Jones, P. Seshoft, , and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2:9–50, 1989.

[92] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Proceedings of the 1990 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 177–195, London, August 13–15 1991. Springer Verlag.

[93] A. Kelly, A. Macdonald, K. Marriott, H. Søndergaard, P. Stuckey, and R. Yap. An optimizing compiler for CLP(R). In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming—CP'95*, Lecture Notes in Computer Science 976, pages 222–239. Springer-Verlag, 1995.

[94] S. C. Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 211–222, New Haven, CN, June 1991.

[95] Andreas Krall and Reinhard Grafl. CACAO – A 64 bit JavaVM just-in-time compiler. In Geoffrey C. Fox and Wei Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997. ACM.

[96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, Pennsylvania, 21–24 May 1996.

[97] M. Leone and R. K. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report #490, Indiana University Computer Science Department, Indiana University, Bloomington, IN, September 1997.

[98] Mark Leone and Peter Lee. Deferred compilation: The automation of run-time code generation. Technical Report CMU-CS-93-225, Carnegie-Mellon, Department of Computer Science, Pittsburgh, PA 15212, December 1993.

[99] Mark Leone and Peter Lee. Lightweight run-time code generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.

[100] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[101] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):??–??, September 1998. Article 23.

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

[103] L. A. Lombardi and Bertram Raphael. LISP as the language for an incremental computer. Report MAC-M-142, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, March 1964.

[104] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[105] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS–005–91, Columbia University, 1991.

[106] Henry Massalin and Calton Pu. Threads and input output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, Litchfield Park AZ USA, December 1989. ACM.

[107] Henry Massalin and Calton Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, Winter 1990.

[108] Henry Massalin and Calton Pu. Reimplementing the Synthesis kernel on the Sony NeWS workstation. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 177–186, Seattle WA (USA), April 1992. Usenix.

[109] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, May 1993.

[110] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

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

[112] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, June 16–20 1997. Usenix Association.

[113] Gilles Muller and Ulrik Pagh Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, 16(2):44–51, March/April 1999.

[114] Gilles Muller, Eugen-Nicolae Volanschi, and Renaud Marlet. Scaling and partial evaluation for optimizing the Sun commercial RPC protocol. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–126, Amsterdam, The Netherlands, 12–13 June 1997.

[115] Vivek Nirkhe and William Pugh. Partial evaluation of high-level imperative programming languages with applications in hard real-time systems. In ACM, editor, *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the*

symposium, Albuquerque, New Mexico, January 19–22, 1992, pages 269–280, New York, NY, USA, 1992. ACM Press.

[116] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. *ACM SIGPLAN Notices*, 30(6):67–78, June 1995.

[117] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Notices*, 25(6):16–27, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[118] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 109–121, New York, June 15–18 1997. ACM Press.

[119] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM TOPLAS*, 1999. To appear.

[120] Todd Proebsting. Personal communication, 1997.

[121] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 41–54, Berkeley, June 16–20 1997. Usenix Association.

[122] Calton Pu and Henry Massalin. AN OVERVIEW OF THE synthesis OPERATING SYSTEM. Technical Report CUCS-470-89, University of Columbia, 1989.

[123] Calton Pu and Henry Massalin. Quaject composition in the Synthesis kernel. In Luis-Felipe Cabrera, Vince Russo, and Marc Shapiro, editors, *1991 International Workshop on Object Orientation in Operating Systems*, pages 29–34, Palo Alto CA (USA), October 1991. IEEE, IEEE Computer Society Press.

[124] Calton Pu, Henry Massalin, and John Ioannidis. THE SYNTHESIS KERNEL CUCS-259-87. Technical report, University of Columbia, 1987.

[125] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. In USENIX Association, editor, *Computing Systems, Winter, 1988.*, volume 1, pages 11–32, Berkeley, CA, USA, Winter 1988. USENIX.

[126] C. Queinnec and J.-M. Geffroy. Partial evaluation applied to pattern matching with intelligent backtracking. In M. Billaud et al., editors, *WSA '92, Static Analysis, Bordeaux, France, September 1992. Bigre vols 81–82, 1992*, pages 109–117. Rennes: IRISA, 1992.

[127] A. Dain Samples. Compiler implementation of ADTs using profile data. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction, 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 72–87, Paderborn, Germany, 5–7 October 1992. Springer.

[128] Alan Dain Samples. Profile-driven compilation. Technical Report UCB//CSD-91-627, University of California Berkeley, Department of Computer Science, 1991.

[129] Vivek Sarkar, Mauricio J. Serrano, and Barbara B. Simons. "Retargeting Optimized Code by Matching Tree Patterns in Directed Acyclic Graphs", Patent Application, submitted in December 1998.

[130] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 62–73, San Francisco, California, January 22–25, 1995. ACM Press.

[131] Swaptuner. http://www.microquill.com/prod_st/index_st.html.

[132] Tpc benchmark c^{TM}, standard specification, revision 3.3. http://www.benchmarkresources.com/handbook/tpca.pdf.

[133] David Ungar and Randall B. Smith. SELF. the power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, July 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241.

[134] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. *ACM SIGPLAN Notices*, 29(6):85–96, June 1994.

[135] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, Denver, CO, 2–5 November 1999. To appear.

[136] T. Wilkinson. Kaffe v0.8.3 - a free virtual machine to run Java code. http://www.kaffe.org, March 1997.