Opencj: A research Java static compiler based on Open64

Keqiao Yang, Zhemin Yang, Zhiwei Cao, Zeng Huang, Di Wang, Min Yang, Binyu Zang

Parallel Processing Institute, Fudan University, Shanghai, China

{kqyang,yangzhemin,zwcao,hz,wang_di,m_yang, byzang}@fudan.edu.cn

Abstract

As Java becomes more pervasive in the programming landscape even in HPC applications, it is very important to provide optimizing compilers and more efficient runtime systems. To this end, we try to leverage the synergy between static and dynamic optimization to exploit more optimization chances and improve Java runtime performance especially for server applications. This paper presents our first achievement of implementing a Java static compiler Opencj which can perform fully optimization for Java applications.

Opencj takes Java source files or class files as inputs and generates machine dependent executable code for Linux/IA32. It is developed based on Open64 with some optimizations implemented for Java. Efficient support for exception handling and virtual method call resolution fulfills the demands which are imposed by the dynamic features of the Java programming language. Due to the same optimizer in Opencj and Open64, the performance gap between Java and C/C++ programs can be evaluated. The evaluation of the scientific SciMark 2.0 benchmark suite shows they have a similar peak performance between its Java and C versions. The evaluation also illustrates that the performance of Opencj is better than GCJ for SPECjvm98 benchmark suite.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors-Compilers, Optimization, Code generation

General Terms Algorithms, Languages, Performance

Keywords Java, Java Static Compiler, Java Exception Handling, Bounds Check Elimination, Synchronization Elimination, Java Devirtualization

1. Introduction

The Java programming language enjoys widespread popularity on different platforms ranging from servers to mobile phones due to its productivity and safety. Many optimization approaches have been applied to improve Java runtime performance. In many applications, Java's performance is similar to other programming languages (including *C++/Fortran*)[16]. There are two ways to run Java programs: running bytecode within a Java virtual machine (JVM) or running its executable code directly.

JVM can either interpret the Java bytecode or compile the bytecode into native code of the target machine. Because interpretation incurs a high runtime overhead, JVMs usually rely on compilation. The most popular compilation approach is to perform dynamic or just-in-time (JIT) compilation, where translation from bytecode to native code is performed just before a method is executed. To reduce the overhead of runtime compilation, JVM usually operates in a hybrid mode, which means that bytecode of a method is initially executed by an interpreter until the JVM determines its suitability for further optimization. Examples of such systems include IBM JVM[34], Sun Hotspot compiler[21], Hotspot™for Java 6[17], and Jalapeño adaptive optimization system[2]. Nevertheless, JVMs using even the most sophisticated dynamic compilation strategies in their server compiler still suffer from two serious problems, *Large memory footprint* and *Startup overhead* of JVMs. However, static compilation achieves greatly reduced cost of startup, reduced usage of memory, automatic sharing of code by the OS between applications, and easier linking with native code.

This paper introduces a static Java compiler Opencj which is developed based on Open64 compiler. Open64 is an open-source static compiler for C/C++ and Fortran with excellent optimizer. Opencj benefits from Open64's backend, especially IPA which enables us to generate advanced optimized Java native code and evaluate the performance gap between Java and C/C++.

Due to the lack of precise runtime information, it is hard to predict runtime behavior for static compilation. Although offline profiling technique has been adopted by some static compilers including Open64, it may not be accurate or representative. As a result, compilers are forced to make conservative assumptions to preserve correctness and to avoid performance degradation. So, we argue that static and dynamic optimizations are not distinct and competitive. And

CGO 2009 Open64 Workshop March 22, Washington.

we try to find a way to integrate the benefits of static and dynamic optimizations to improve runtime performance. We take DRLVM, a JVM of Apache Harmony[13], as a platform to reveal how to leverage static optimization benefits.

In section 7, we picture a Java compilation framework mixing the static and dynamic optimization techniques to further reduce runtime overhead. In this framework, a dynamic compilation module is introduced to collect runtime information and apply dynamic optimizations guided by the profiling results. Besides, a complete Java runtime environment based on Harmony will be integrated into Opencj to further accelerate Java runtime speed.

This paper makes the following contributions:

- Design and implementation of a Java static compiler Opencj: we introduce the infrastructure of Opencj that compiles Java source files or Class files into optimized executable code. In particular, we focus on the Java exception handling and some optimizations in Opencj.
- We compare the runtime performance of Java applications between running in JVM and running executable code. Meanwhile we give an evaluation of the performance gap between Java and C in scientific applications.
- We evaluate the performance of Opencj at Linux/IA32 comparing to GCJ 4.2.0, Harmony DRLVM and Sun hotspot of JDK 1.6.
- We give a big picture of how to combine static optimization and Java runtime technique to improve Java runtime performance.

The rest of the paper is organized as follows. Section 2 gives an overview of Opencj compiler. Section 3 presents the frontend migration for Open64. Section 4 describes the optimizations which are designed and implemented for Java applications. Section5 gives the experimental evaluation of Opencj. Section 6 pursues related works. Section 7 highlights the future work of our research and, finally, section 8 concludes the paper.

2. Overview of Opencj

The main components of Opencj include the Java frontend migrated from GCJ[12] and the Optimizer of Open64[25]. The frontend is used to read Java source files or class files and transform them into WHIRL[14] of Open64. And then Open64 Optimizer performs optimization on the WHIRL and generates machine dependent executable code for IA32 or Itanium.

Open64 is open sourced by Silicon Graphics Inc. from its SGI Pro64 compiler targeting MIPS and Itanium processors. Open64 is a well-written, modularized, robust, state-of-the-art compiler with support for C/C++ and Fortran 77/90. The major modules of Open64 are the multiple language frontends, the interprocedural analyzer(IPA) and the middle end/back end, which is further subdivided into the loop nest optimizer(LNO), global optimizer(WOPT), and code generator (CG). These modules interact via a common tree-based intermediate representations, called WHIRL(Winning Hierarchical Intermediate Representation Language). The WHIRL has five levels to facilitate the implementation of different analysis and optimization phases. They are classified as Very High, High, Mid, Low, and Very Low levels, respectively. And each optimization is implemented on a specific level of WHIRL. For example, IPA and LNO are applied to High level WHIRL while WOPT operates on Mid level.

The C/C++, Java frontends are based on GNU technology. The Fortran90/95 frontend is the SGI Pro64 (Cray) Fortran frontend. This paper presents the detail of Java frontend in Open64 in section 3. Each frontend produces a Very High level WHIRL for the input program units, stored as a so-called *.B* file. This representation is available for the subsequent phases. The driver of Open64 controls the execution of the compiler, deciding what modules to load and what compilation plan to use. The Driver is responsible for invoking the frontends, the stand-alone procedure inliner, IPA, back-end modules and the linker. Figure 1 is the compiler executing pattern.

3. Frontend migration

3.1 Java frontend

The C/C++ frontend of Open64 is inherited from GCC, so the frontend of GCJ compiler which is a Java static compiler of GCC is chosen to develop the Java frontend of Open64 when designing Opencj. In figure 1, the frontend has two modules, *gspin* which outputs AST of GCC as language independent *spin* file and *wgen* which takes spin file as input and converts it into WHIRL. The objective of *gspin* module is to keep the independency and persistency of *wgen* module in the compiler version updating. Spin file is equal to AST file except a few modifications in AST to remove the language dependent tree nodes, such as renaming the tree node type.

We migrate the frontend of GCJ 4.2.0 version into Opencj. After GCC 4.0 version, it redefined the AST tree into *GENERIC* and *GIMPLE*[20] tree for performing optimization at the tree level. The purpose of *GENERIC* is simply to provide a language-independent way of representing an entire function in trees. *GIMPLE* is a simplified subset of *GENERIC* for use in optimization. We used to try to generate spin file at *GIMPLE* tree instead of *GENERIC* to benefit the GCJ optimization. The evaluation of SciMark 2.0 Java benchmark shows that it caused about 30% performance degradation since some high level structure has been transformed into low level, e.g. loop structure has been transformed into *goto* structure, thus preventing further optimization in Opencj.

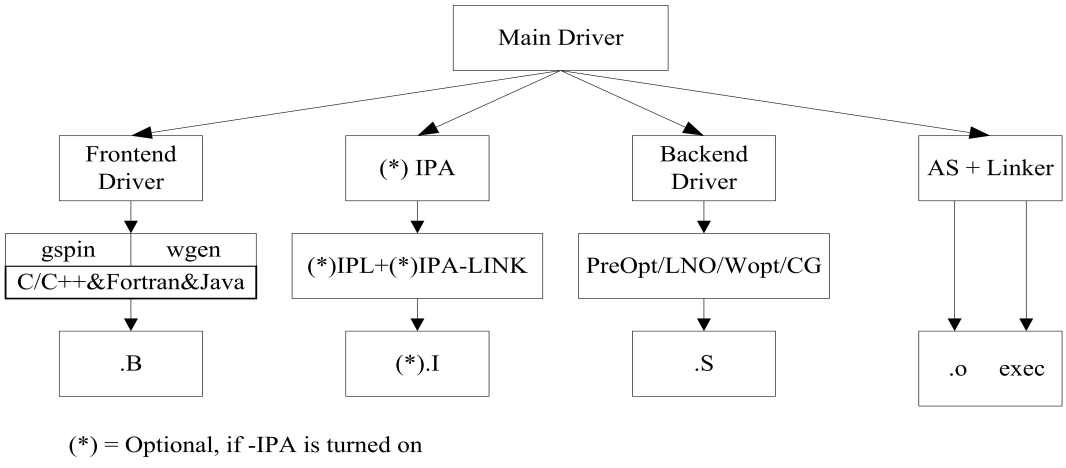

Figure 1. Compilation Model of Open64

3.2 Handling Java exception

Open64 has handled C++ exception. Although Java exception is similar to C++ exception, there are several differences between them:

1. Java exception could throw runtime exceptions, such as throwing an **ArithmeticException** when a *div* or *rem* instruction takes *zero* as its second operand, and throwing a **NullPointerException** when an indirect load instruction takes *zero* as its base address.
2. C++ exception has no restriction on the type of exception objects, which means objects of any type could be thrown out, while Java exception restricts all its exception objects to be of Throwable class or its subclass.
3. C++ exception uses "catch-all" handler to catch exceptions which have no corresponding handler, while Java has an alternative way to get the same result. Java uses a catch handler which catches the exception objects of Throwable class.
4. When a statement throws an exception in a try block, C++ exception requires to destruct the objects which are defined within a try block before this statement. Java exception has no such a requirement since all objects in Java are managed by the garbage collector.
5. Java exception has the "finally" mechanism, that is, no matter what happens in a try block, the finally block following this try block must be executed. This mechanism makes Java exception more complex than C++ exception. As figure 2[32] shows, there could be 15 kinds of execution routes during a Java exception handling process, while C++ exception just has 7 execution routes corresponding to the route 1, 3, 5, 7, 9, 13 and 15.

Therefore, an new algorithm need to be designed to handling Java exception in Opencj. The Java exception handling can be divided into four sub-procedures:

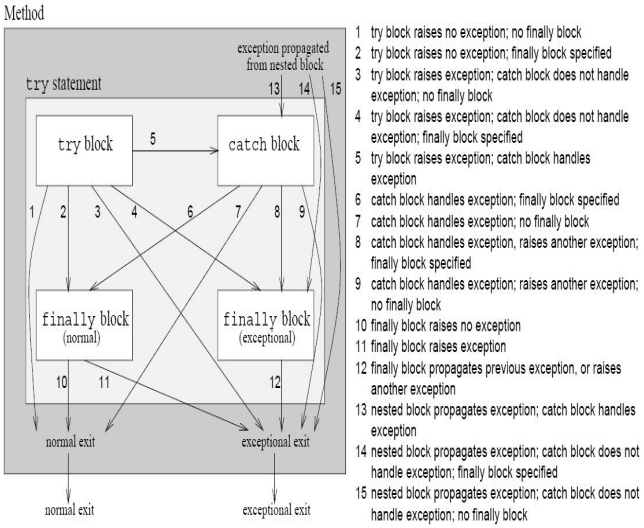

Figure 2. Intraprocedural control flow in Java exception-handling constructs

1. **To recognize the expressions which may throw an exception**. In wgen module, the expressions which may throw an exception will be surrounded in an indicative REGION. In C++ exception, only the CALL expression may throw an exception, but in Java exception, the *ILOAD*, *DIV* and *REM* expressions can throw exceptions as well.
2. **To analyze the relationships of try statements.** There are many kinds of execution routes in a Java exception handling process. To implement these routes and make the exceptions thrown from different parts of a program be handled correctly, the critical problem is how to make the compiler understand the relationship among *try/catch/finally* blocks.
3. **To find the landpad of the expressions which may throw an exception.** The landpad means the start point

of a piece of the exception handling code. Basing on the knowledge of the relationship among *try/catch/finally* blocks, the compiler will locate the corresponding exception handling codes of each exception throwing point.

4. **To build the exception handling relative code.** The layout of all the exception handling codes will be set down in this step.

Take a program presented in figure 3(a) as an example. The *wgen* phase will generate the WHIRL with Java exception handling code as illustrated in figure 3(b). Each block in this Java program may contain expressions which may throw an exception, and the landpads of these expressions may not be the same. In figure 3(b), the Java exception handling algorithm sets the landpad of the inside try block with label L3, sets the landpad of catch block A and catch block B with $L6$, sets the landpad of the inside finally block, code block a and code block b with $L7$, set the landpad of catch block c with $L9$ and leaves the landpads of other blocks unset. Finally, each exception thrown in this program can be processed by its corresponding exception handling code.

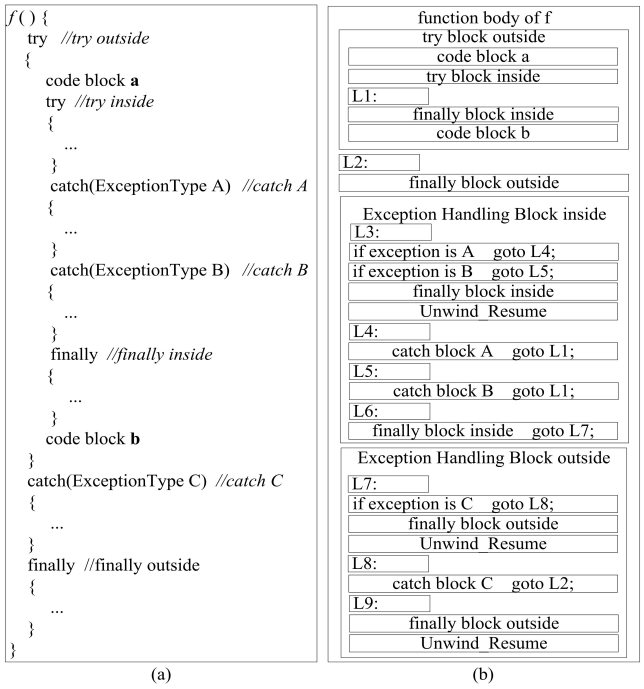

Figure 3. A simple Java program and its exception handling code

4. Optimization for Java

The backend of Opencj comes from the optimizer of Open64 as figure 1 shows. It performs advanced optimizations at WHIRL and generates the machine code of target platform. The remainder of this section outlines the main optimizations for Java applications which we implemented in Opencj: the virtual call resolution, redundant synchronization elimination and the array bounds check elimination.

4.1 Virtual method call resolution for Java

A major advantage of Java is abstraction, which supports dynamic dispatch of methods based on the run-time type of an object. Virtual functions make code easier for programmers to reuse but harder for compilers to analyze. Of cause, many devirtualization techniques[33][15] have been proposed to reduce the overhead of dynamic method calls for various object-oriented languages through statically determining what methods can be invoked. Many optimizations can benefit from devirtualization. It can provide an accurate graph which can be used to compact applications by removing dead methods and improve the efficiency and accuracy of subsequent interprocedural analysis.

Opencj adopts *class hierarchy analysis*[9] and *rapid type analysis*[4] to resolve Java virtual method call. The devirtualization algorithm has four following steps:

1. Identifying whether a indirect call is a virtual function call which is called through vtable, and recording the offset of the function in the vtable.
2. Rapid type analysis: checking the initialized type of the object with the type table analysis. For example, $A\ q = new\ B()$;, the declared type of q is A, but the initialized type is B. According to Java specification, the class B must be a subclass of A. And then all virtual functions called through q can be found in the vtable of class B. This is simple case. When the initialized type of q depends on runtime control flow, e.g. $q = foo()$, the type analysis can only achieve the declared type of function $foo()$. If the $foo()$ has a declared class type B, the possible runtime return type includes B plus subclasses of B. So we need to build class hierarchy graph to resolve Java virtual function call.
3. Building class hierarchy graph at inter-procedure analysis phase. When handling a virtual function call, the declared type of object q has no subclass, this function can be resolved, otherwise, we adopt a conservative way.
4. Resolving the virtual function call into direct call with the vtable and the offset and updating the call graph.

In the SciMark 2.0 Java benchmark test, Opencj can resolve all 21 user defined virtual function calls.

4.2 Synchronization elimination

One important characteristics of the Java programming language is the built-in support for multi-threaded programming.There are two synchronization constructs in Java, *synchronized* methods and blocks. When a thread executes a synchronized method against an object or a synchronized block with an object, the thread acquires the object's lock before the execution and releases the lock after the execution. Thus, at most one thread can execute the synchronized method or the synchronized block. As a result, Java programs perform many lock operations. Many techniques have

been proposed for optimizing locks in Java, which can be divided into two categories, runtime techniques and compile-time techniques. The former attempts to make lock operations cheaper[10], while the later attempts to eliminate lock operations[29]. Opencj tries to eliminate redundant lock operations to reduce the synchronization overhead and exploit more optimization chance.

We implement synchronization optimization in Opencj based on escape analysis[6] which is flow-sensitive and interprocedural. Escape analysis checks whether an object escapes from the method or the thread. In Opencj, we focus on the thread escape. If a synchronized object doesn't escape from thread, then the synchronized operation must be redundant, it can be removed; otherwise, conservatively identified it as a needed synchronization. In fact, if do inter-thread analysis and make sure no other threads operate on this object, the synchronization operation can be removed too. The synchronization optimization implemented in Opencj can be divided into three following steps:

To build connection graph. The connection graph abstraction captures the connectivity relationship among objects and object references. Performing reachability analysis on the connection graph can easily determine whether an object is local to a thread. Because connection graph just focuses on objects in the program, only five following kinds of statements need to be traced in building connection graph:

1. $p = new\ P()$
2. $p = return\_new\_P()$ where returns a class type to p
3. $p = q$
4. $p = q.f$
5. $p.f = q$

These five statements will update connection graph. Figure 4 shows a example to illustrate the connection graph computation:

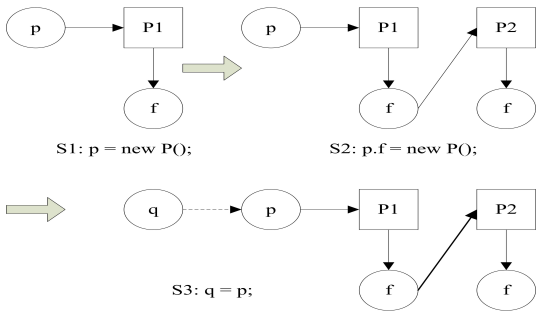

Figure 4. An example illustrating connection graph computation. Boxes indicate object nodes and circles indicate reference nodes (including field reference nodes). Solid edges indicate points-to edge, dashed edges indicate deferred edges, and edges from boxes to circles indicate field edges.

Intraprocedural analysis. The objective of this step is to record synchronization operations and synchronized objects, and set initial escape state for each node in the connection graph.

We define four kinds of escape states with an ordering among them: GlobalEscape $>$ ArgEscape $>$ OutEscape $>$NoEscape.

- *GlobalEscape*: static variable which is thread escape.
- *ArgEscape*: formal arguments which are method escape or thread escape.
- *OutEscape*: parameters and return values which is method escape or thread escape.
- *NoEscape*: local variables which are not escape from method.

An object node maybe pointed by many reference nodes which have different escape states. So the ordering among escape states is necessary and let A $\in$ EscapeSet = $\{$*NoEscape, OutEscape, ArgEscape, GlobalEscape*$\}$, then A $\wedge$ *NoEscape* = A, and A $\wedge$ *GlobalEscape* = *GlobalEscape*.

If a node is marked *NoEscape*, the synchronization operation can be removed. For GlobalEscape, the synchronization operation must be preserved. If it is marked *ArgEscape* or *OutEscape*, it needs interprocedural analysis to identify whether is redundant or not.

Interprocedural analysis. Interprocedural analysis can get the escape state from other methods. For example, A() has a statement $a = B()$ where a receive return value of method B(), so a is marked *OutEscape*. If return variable of B() is *NoEscape*, a will be updated into *NoEscape*. This analysis starts at entry point of program, then traverses call graph in depth-first order.

The task of interprocedural analysis is to match escape states for caller and callee pair. The process is:

1. Caller sends escape states of actual parameters and their field nodes to callee;

2. Callee updates escape states of its parameters and related nodes in its own connection graph, and then submits the escape states to caller.

3. Caller updates escape states and connection graph according to callee's feedback.

There are four synchronization operations in figure 5, and three of them(#1, #2, #4) can be removed by Opencj. In the building connection graph step, four statements (line 5, 6, 7, 23) will be used to construct connection graph. Since this example is simple that every object node is pointed by only one reference node. synchronized object of #1, #2, #3, #4 is d, b, a and $this$.

Intraprocedural analysis can easily set escape states of a, b, d and $temp$ with *GlobalEscape*, *OutEscape*, *OutEscape* and *OutEscape* correspondingly.

In interprocedural analysis phase, d receives the return value of default constructor of Class C, so d is *NoEscape*, and #1 can be removed. b receives the return value of return_new_C() method, so its escape state is equivalent to the variable *temp* in return_new_C(). The *temp* is *NoEscape*

same as d, then #2 can be removed. For #4, the formal parameter variable *this* is a synchronized object, and the actual parameter of caller $main$ is d. d is *NoEscape*, so *this* is also *NoEscape* and #4 can be removed. For #3, a is static variable, so it is *GlobalEscape*, #3 need to be kept.

```
1  public class Test{
2    static C a;
3
4    public static void main(String[] args){
5      a = new C();
6      C d = new C();
7      C b = d.return_new_C();
8      synchronized(d){            //#1
9              System.out.println("synchronized #1 for d");
10     }
11     synchronized(b){            //#2
12             System.out.println("synchronized #2 for b");
13     }
14     synchronized(a){            //#3
15             System.out.println("synchronized #3 for a");
16     }
17   }
18 }
19
20 public class C{
21   public final synchronized C return_new_C(){    // #4
22           System.out.println("synchronized #4 for this");
23           C temp = new C();
24           return temp;
25   }
26 }
```

Figure 5. An simple example illustrating synchronization optimization

4.3 Array bounds check elimination

Array bounds check elimination removes checks of array indices that are proven to be within the valid range.When an index variable is guaranteed to be between 0 and $array.length-1$, the check can be completely omitted (*Fully Redundant* Check). When the check is in a loop, the array length is loop invariant, and the index variable is an induction variable, the check can be moved out of the loop (*Partially Redundant* Check) to reduce the total number of executed checks. The semantics must stay the same when eliminating or moving checks in Java programs.

In contrast to other approaches that perform in JVMs, (see e.g. [5], [28] and [36]), we adhere to the design principle of the static compiler to optimize scientific Java applications. The algorithm tries to eliminate redundant array bounds check especially in a loop to exploit more optimization chance. It takes advantage of the SSA form and requires an inequality graph to recode value range constraint for each array index variable, and it handles more cases, such as index variable which is a multiplication or division expression and two-dimension array.

In SSA form, each variable is assigned only at a single point. When a variable is defined, its value will never change again. To build inequality graph for a bounds check, we need to access the dominator tree and get the path from the root to the current bounds check block. To determine the value range of the array index, the ABCD[5] algorithm builds two inequality graphs for the current PU, one is used to determine the upper bound value of the array index and the other is used to determine the lower bound value. Different from the ABCD algorithm, our algorithm merges these two graphs together. An inequality graph is a constraint system of an array index. It is a weighted directed graph built from a root-path in the dominator tree. The building process of the inequality graph is dynamic, updated when entering or exiting a block. If a block is pushed into the stack, the constraint information (e.g. nodes or edges for the graph) contained in the current block will be added to the inequality graph and if a block is popped from the stack, the information generated by this block will be removed as well. The nodes in the inequality graph represent the variables or expressions with the int type. Unlike the ABCD and the constraint graph in the paper[28], our inequality graph does not contains any constant node. The edge in the inequality graph represents a condition. An edge from i to j with a constant weight c stands for a constraint condition $i + c \leq j$.

A condition is generated by an assignment or a branch statement or an array bounds check statement. For example, an int type assignment statement $i = j + c$ generates conditions $i - c \leq j$ and $j + c \leq i$, a branch statement $i < j$ generates a condition $i + 1 \leq j$ for the true branch and the check statement for $a[i]$will generate conditions $0 \leq i$ and $i + 1 \leq a.length$ after the array access $a[i]$.

In ABCD algorithm, it needs a shortest path algorithm to determine the relationship between the array index and 0 or array length. Our algorithm solves this problem in a different way: recording the value range information for each variable node in the inequality graph. The last step is *Elimination*. For the PRC, ABCE adopts loop versioning[22]. It clones the original loop and sets some trigger conditions before and after the optimized loop. This tactic can guarantee the exception semantic for Java. When a check fails, the exception can be thrown at the correct code position of the failing array access.

5. Evaluation

This section presents the evaluation of the Java runtime performance of SciMark 2.0 Java benchmark suite[27] and SPECjvm98 benchmark suite[8] at Linux/32 platform. We have two objectives in the experiment. One is to evaluate the performance gap between Java and C in the scientific applications with the same optimizer, the other is to compare the peak performance of Opencj, GCJ 4.2.0, Sun HotSpot of JDK 1.6 and Harmony[13].

All experimental results are obtained on an IA32 platform with four Intel(R) Xeon(TM) 2.8GHz CPUs and 3.5 GB main memory. The operating system is Linux-2.6.18.

For the evaluation, two benchmark suites SPECjvm98 and SciMark 2.0 are executed. The first one consists of eight benchmarks derived from real-world client applications ex-

cept *_200_check*, while the second one performs scientific computations which has Java and C versions. Five scientific computing applications which are widely used in Java programs are included in SciMark 2.0 benchmark.

1. FFT: A complex 1D fast Fourier transform algorithm;
2. SOR: Solving of the Laplace equation in 2D by successive over-relaxation;
3. MC: Computing by Monte Carlo integration;
4. MV: Sparse matrix-vector multiplication;
5. LU: Computing the LU factorization of a dense N x N matrix.

Each kernel except MC has small and large problem sizes. The small problems are designed to test raw CPU performance and the effectiveness of the cache hierarchy. The large problems stress the memory subsystem because they do not fit in cache. In the experiments, we only test its small problem size.

5.1 Performance gap between Java and C

Five kernels in the SciMark 2.0 benchmark suite are loop-intensive programs. We test their Java version and C version with the same optimizer to evaluate the peak performance gap between Java and C.

In the evaluation, all compilers compile the source code with -O3 flag, while opencc(C driver of open64) and opencj enable their IPA, moreover opencj and gcj enable -fno-bounds-check flag to eliminate all array bounds check in Java programs. Although, we implemented a ABCE algorithm in Opencj and can eliminate most redundant array bounds check in SciMark 2.0 test, the achieved 28.4% speedup is lower than we expected, mainly due to phase ordering. Currently the eliminating array bounds check optimization is added after the SSA-PRE since it may benefit from this optimization. However Open64 does LNO optimization before PRE and the bounds checks limit some optimizations in the LNO phase. In another word, the LNO optimization can benefit from ABCE. We will move the ABCE phase before LNO in the future.

Figure 6 shows the Java performance is similar to C performance in SciMark 2.0 benchmark whatever opencj vs opencc, or gcj vs gcc. The result also shows the opencc has the best performance, followed by opencj which is better than gcj and gcc. There is a big speedup in MC comparing opencj to gcj. The MC has a synchronized method which the synchronization is unnecessary. However gcj does not have synchronization optimization to remove this lock operation while opencj has. If disable this optimization, the opencj gets a similar performance score as gcj. The synchronization optimization of Opencj achieved about 3.94X speedup in MC test case.

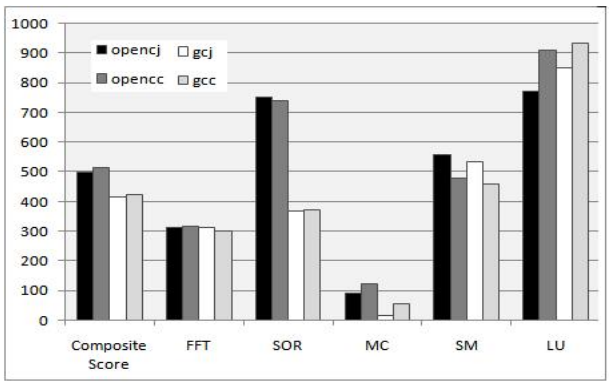

Figure 6. Performance test among four compilers in SciMark 2.0 test. The y-axis indicates the performance score and taller bars are better.

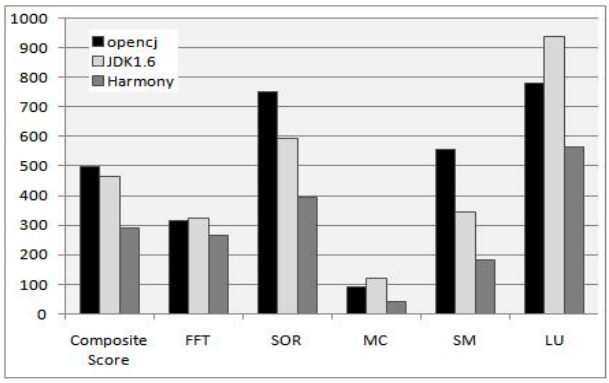

Figure 7. Performance of SciMake 2.0 in two running modes. One is to run bytecode in Sun Hotspot of JDK1.6 and DRLVM of Harmony, the other is run executable code compiled by Opencj. The y-axis indicates the performance score and taller bars are better.

5.2 Static compilation vs JVM in Java runtime performance

Two popular approaches for compiling Java programs are Just-In-Time (JIT) compilation and static compilation. It would be wrong to say one approach is definitely better than the other, since they are suited for different situations[37]. We measure the Java runtime performance in these two modes comparing the Opencj,Sun JDK1.6 and Harmony. Harmony is a Java SE project of the Apache Software Foundation. We test its latest Apache Harmony 6.0 M8 version.

In JVM running mode, we test its server mode comparing to Opencj with full optimization. Currently, the Opencj just can correctly compile seven benchmarks of SPECjvm98 except *_228_jack* as GCJ does, since the Java frontend of Opencj is migrated from gcj 4.2.0 version. *_200_check* is not used to evaluate Java performance, so the evaluation results just have six test cases of SPECjvm98. Figure 7 shows Opencj is a little better than JDK1.6 in composite score of the SciMark 2.0 test cases while more better than Harmony

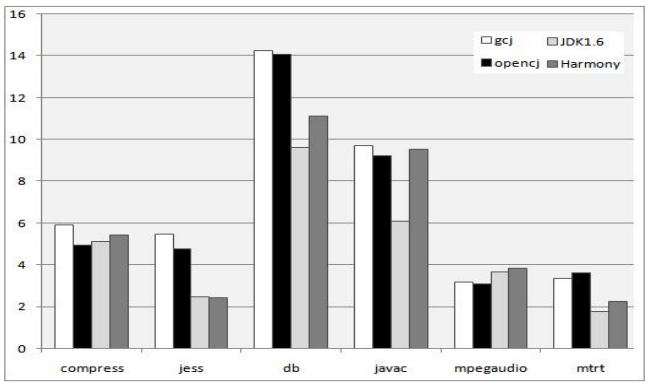

Figure 8. Performance of SPECjvm98 in two running modes. The y-axis is the running time(sec.) and lower bars are better.

in all 5 test cases, on average about 72.7% performance gap. There are some bugs in compiling SPECjvm98 test cases when enable IPA, so Opencj compiles SPECjvm98 test cases at $-O3$. Figure 8 illustrates the result of SPECjvm98 benchmark test. The y-axis in the graph is the running time(sec.) and JDK1.6 is more better than Opencj except *compress* and *mpegaudio*.

The test results show that sometimes dynamic optimizer can do more effective optimization to enhance Java runtime performance. So we adhere to that online profiling mechanisms and feedback directed optimizations will become mainstream, and multi-language runtime systems as the divide between static and dynamic environments will dissolve.

6. Related work

As Java becomes more and more popular, many dynamic and static compilation techniques were applied to compile Java code and improve the runtime performance and security of Java programs. With the support of runtime profiling techniques, many dynamic compilers were developed in these years, such as Sun Hotspot[21], Microsoft JIT[7], JRockit[26], Apache Harmony, JikesRVM[1] and OpenJIT[24]. These dynamic compilers, apply optimizations and analysis actions at runtime. Such characteristic of dynamic compilation uncovers a lot of optimization chances to the compiler, but it also introduces a significant compilation overhead at runtime. Such overhead not only greatly raises the startup time of Java programs, but also neutralizes the effect of optimizations.

On the other hand, several static compilers for java are also available, such as HPCJ[31], Marmot[11], TowerJ[35] , BulletTain[23] and GCJ. Static compilation easily eliminates the runtime overhead which bothers the dynamic optimization designers. But, due to the lack of runtime profiling feedback, many aggressive optimizations cannot be directly applied in these compilers. Current generation of Opencj uses static compilation to compile Java code, and we attempted to take the advantages of static interprocedural analysis to exploit the optimization chance for Java codes in Opencj. And evaluation result shows the performance of Opencj is better than GCJ.

Azevedo, Nicolau, and Humme have developed the AJIT compiler[3]. It annotates the bytecode with machine independent analysis information that allows the JIT to perform some optimization without having to dynamically perform analysis. This approach reduces the runtime analysis overhead, but runtime optimization overhead cannot be reduced. Besides, considering portability, those optimizations that require machine dependent information cannot benefit from this framework.

Recently, several research groups attempt to mix static compilation and dynamic compilation together in order to achieve better performance behavior for Java language. Quicksilver[30] bypass the runtime optimization overhead through using a static compilation phase instead of the dynamic one. It generates a version of object code, and uses these object codes as an alternative in target JVMs. However, it makes the JVM sophisticated and weakens the portability of these object codes. LLVM[19] is a compiler infrastructure that can be used as Java static compiler as well as C/C++ compiler. It also provides a Just-In-Time compiler for runtime optimizations. In the next generation framework for Opencj, we aim to construct a general framework that not only support static and dynamic optimization, but also provides other Java runtime environment such as garbage collector to uncover the chances for improving Java runtime behavior.

7. Future work

As a static compiler, Opencj achieves a relative high performance comparing to other static Java compilers. As the evaluation result shows, the performance of Opencj can be further improved. There are two following aspects can be taken into account:

- Higher-level languages must have interface with lower-level languages, typically C, to access platform functionality, reuse legacy libraries, and improve efficiency. For example, most Java programs use native function calls, since several methods of even class Object at the root of Javaś class hierarchy are written in C. Java uses Java Native Interface (JNI)[18] to incorporate native code. However, the using of JNI in Java programs will cause a large overhead at runtime because the JNI instructions introduce indirect calls which block the optimizations in compile time. Most of the JNI instructions can be promoted into direct calls with the help of runtime profiling result.
- With the multicore/manycore architecture becoming popular, the efficient usage of idle resources may significantly improve the performance of Java programs. With the help of machine information, the jobs can be properly de-composited in JVM and assigned to different cores by

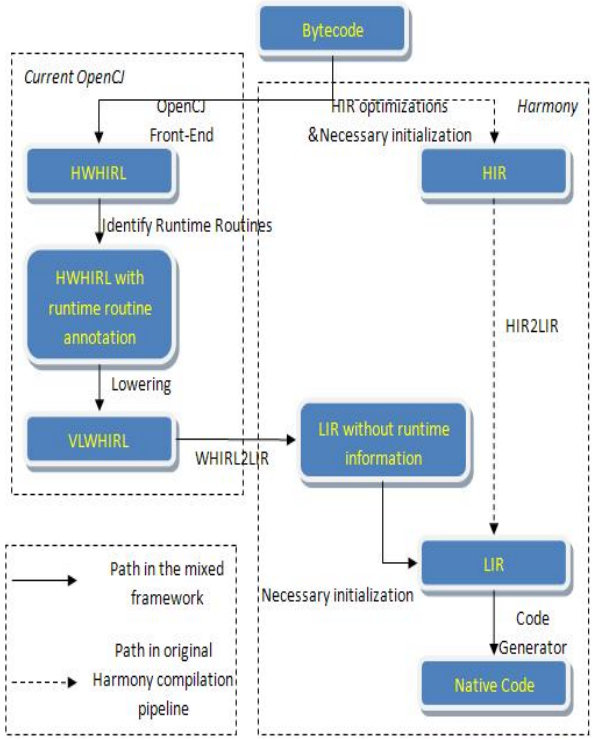

Figure 9. The framework of next generation Opencj for further improving the performance

operating systems, thus improve the overall performance of Java code.

Both JNI calls and machine information can be easily collected at runtime. In the next generation of Opencj, we plan to introduce a Java compilation framework which combines the static and dynamic compilation together. In this framework, profiling results can be used to guide the dynamic optimizations, thus achieving better runtime performance. Figure 9 depicts the overall framework.

In the next generation framework, Opencj will be further modified to translate WHIRL files into IR used by dynamic compiler after fully optimization. The dynamic compiler is modified from one of the Harmony's JVMs, named DRLVM. DRLVM has two intermediate data structures, called the high-level intermediate representation (HIR) and the low-level intermediate representation (LIR). We only use LIR in DRLVM. Java bytecode will be first pre-compiled by Opencj to generate LIR files with some annotations. DRLVM will take LIR files as input and compile them into native code after some runtime initializations.

This framework can greatly benefit the optimizations of Opencj and DRLVM. The profiling information can be collected at runtime to guide the further optimization of DRLVM. Such information can guide the optimizations at runtime and further improve the performance of Java applications. Moreover, by integrating Harmony JVM into Opencj, it is possible to construct a general framework leveraging the cooperation between static and dynamic optimizations to improve the performance of other programming languages. Now the implementation of this framework is in progress, more evaluation will be presented in the future.

8. Conclusion

In this paper, we present a static compiler named Opencj which compiles Java code offline. Opencj utilizes the Open64 backend to achieve more higher quality optimizations. We migrate the frontend of GCJ into Opencj, and handle Java exception in Opencj. Some optimizations which have great effect on Java performance, such as redundant array bounds check elimination and synchronization optimization, have been implemented in Opencj.

In the future, there are still much work for Opencj. Opencj will be integrated with dynamic compilers to obtain a more flexible control over Java programs. We can apply more optimistic optimizations by this integration. For example, by dealing with Java and C cross-calls in Opencj, we can break the boundary between Java and C programs in JVMs which will cost a significant performance degradation at present.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, 2005.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000.

[3] Ana Azevedo, Alex Nicolau, and Joe Hummel. An annotation-aware java virtual machine implementation. concurrency - practice and experience, June 1999.

[4] David F. Bacon and Peter F. Sweeny. Fast Static Analysis of C++ Virtual Function Calls. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[5] Rastislav Bodk, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *ACM Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

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

[7] Microsoft Corporation. MicroSoft SDK for Java 4.0. http://www.microsoft.com/java/vm.htm.

[8] Standard Performance Evaluation Corporation. The SPECjvm98 Benchmarks. http://www.spec.org/jvm98/. 1998.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In *ECOOP'95–Object-oriented Programming, 9th European Conference*, pages 77–101, 1995.

[10] D. Dice. Implementing Fast Java Monitors with Relaxed-Locks. In *Proceedings of USENIX JVM '01*, pages 79–90, 2001.

[11] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. Technical report, 1999.

[12] GNU. GCJ: The GNU Compiler for Java. http://gcc.gnu.org/java/. 2007.

[13] Apache Harmony. Open Source Java SE. http://harmony.apache.org/. 2008.

[14] SGI Inc. WHIRL Intermediate Language Specification. http://open64.sourceforge.net, 2006.

[15] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 294–310, 2000.

[16] Michael Klemm, Ronald Veldema, Matthias Bezold, and Michael Philippsen. A Proposal for OpenMP for Java. In *the Second International Workshop on OpenMP(IWOMP'06)*. Reims, France, June, 2006.

[17] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™client compiler for Java 6. volume 5, pages 1–32, New York, NY, USA, 2008.

[18] Dawid Kurzyniec and Vaidy Sunderam. Efficient cooperation between Java and native codes–JNI performance benchmark. In *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[19] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04)*, 2004.

[20] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May, 2003.

[21] Sun Microsystems. The Java hotspot performance engine architecture. http://java.sun.com/products/hotspot/whitepaper.html. 1999.

[22] Vitaly V. Mikheev, Stanislav A. Fedoseev, Vladimir V. Sukharev, and Nikita V. Lipsky. Effective Enhancement of Loop Versioning in Java. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 293–306, London, UK, 2002.

[23] Inc. NaturalBridge. BulletTrain Description. http://www.naturalbridge.com/bullettrain.html. Technical report.

[24] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. OpenJIT: an open-ended, reflective JIT compiler framework for Java. In *ECOOP 2000 Conference Proceedings*, pages 362–387. Springer-Verlag, 2000.

[25] Open64. Overview of the open64 Compiler Infrastructure. http://open64.sourceforge.net, 2006.

[26] Helena Aerg Östlund. JRA: offline analysis of runtime behaviour. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 16–17, New York, NY, USA, 2004.

[27] R. Pozo and B. Miller. Scimark 2.0. http://math.nist.gov/scimark2/. 1999.

[28] Feng Qian, Laurie J. Hendren, and Clark Verbrugge. A Comprehensive Approach to Array Bounds Check Elimination for Java. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 325–342, London, UK, 2002.

[29] Erik Ruf. Effective synchronization removal for Java. *SIGPLAN Not.*, 35(5):208–218, 2000.

[30] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. Quicksilver: a quasi-static compiler for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 66–82, New York, NY, USA, 2000.

[31] V. Seshadri. IBM High Performance Compiler for Java. In *AIXpert Magazine. http://www.developer.ibm.com/library/aixpert*, Sep. 1997.

[32] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception-handling constructs. volume 26, pages 849–871, 2000.

[33] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vall Ee-rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.

[34] T. Ogasawara T. Suganuma, T. Yasue M. Takeuchi, K. Ishizaki M. Kawahito, and T. Nakatini H. Komatsu. Overview of the IBM Java just-in-time compiler. volume 39, pages 175–193, 2000.

[35] Tower Technology. TowerJ3–A New Generation Native Java Compiler And Runtime Environment. http://www.towerj.com/products/whitepapergnj.shtml. Technical report.

[36] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the Java HotSpot™client compiler. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 125–133, New York, NY, USA, 2007.

[37] Dachuan Yu, Zhong Shao, and Valery Trifonov. Supporting Binary Compatibility with Static Compilation. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 165–180, 2002.