Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler

Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani
IBM Tokyo Research Laboratory
1623-14, Shimotsuruma
Yamato, Kanagawa
242-8502, Japan
ishizaki@trl.ibm.co.jp

ABSTRACT

The Java language incurs a runtime overhead for exception checks and object accesses without an interior pointer in order to ensure safety. It also requires type inclusion test, dynamic class loading, and dynamic method calls in order to ensure flexibility. A "Just-In-Time" (JIT) compiler generates native code from Java byte code at runtime. It must improve the runtime performance without compromising the safety and flexibility of the Java language. We designed and implemented effective optimizations for the JIT compiler, such as exception check elimination, common subexpression elimination, simple type inclusion test, method inlining, and resolution of dynamic method call. We evaluate the performance benefits of these optimizations based on various statistics collected using SPECjvm98 and two JavaSoft applications with byte code sizes ranging from 20000 to 280000 bytes. Each optimization contributes to an improvement in the performance of the programs.

1. Introduction

Java [1] is a popular object-oriented programming language suitable for writing programs that can be distributed and reused on multiple platforms. Java is excellent because of safety, flexibility, and reusability. The safety is achieved by introducing exception checks and disallowing interior object pointers. The flexibility and reusability are achieved by supporting dynamic class loading and dynamic method call. As in typical object-oriented programs, there are many small methods, and calls without method lookup to find the target method, which we call *static method call*, occur frequently. This prevents intra-procedural optimizations by a compiler. The programs also include calls for virtual and interface methods with method lookup to find the target method, which we call *dynamic method call*. Furthermore, to ensure safety, Java contains runtime overheads, such as type inclusion tests and exception checks for accesses to arrays and instance variables.

To improve the performance of the Java execution, two compiler solutions are proposed: a static compilation model and a "Just-In-Time" (JIT) compilation model. The static compilation translates *Java byte code* [2] into native code before the start of program execution, and thus the compilation overhead can be ignored at runtime. Therefore, it can use expensive optimizations. On the other hand, it does not support dynamic class loading, and does not take advantage of Java that supports the flexibility and reusability of a program. The JIT compiler translates the byte code into native code when a new method is invoked at runtime. It allows classes to be loaded dynamically. On the other hand, the overall execution time of the program must include the JIT compilation time, and thus the JIT compiler must be much more efficient in both time and space than the static compiler.

In this paper, we present optimizations we developed to reduce various runtime overheads of the Java language without compromising safety and flexibility. Exception check elimination and lightweight exception checking reduce the overhead of exception checks, which are frequently executed in Java programs. Common subexpression elimination reduces the overhead of accesses to arrays and instance variables. Our type inclusion test has a simpler method than previous approaches. Inlining of static method call increases the opportunity for other optimizations. Resolving dynamic method call using our class hierarchy analysis (CHA) is a new approach to reducing the overhead of dynamic method call in the sense that we adapted CHA to dynamic class loading. It also allows dynamic methods to be inlined, to increase the opportunity for other optimizations.

We validate our approach on the basis of various statistics collected by running several large Java programs. We evaluate each optimization by turning off one by one. All the evaluations are carried out using the Java JIT compiler for the PowerPC architecture on AIX, whose product version has been shipped with JDK 1.1.6 for AIX 4.3.2 [3] with the "Java Compatible" logo.

The paper is structured as follows. Section 2 presents an overview of the JIT compiler. Section 3 describes optimizations to reduce the overhead of accesses to arrays and instance variables. Section 4 describes the implementation of type inclusion test. Section 5 describes how to reduce the overhead of static and dynamic method calls. Section 6 gives experimental results with statistics and performance data. Section 7 summarizes related work. Section 8 outlines our conclusions.

2. Overview

In this section, we outline the structure of the JIT compiler as shown in Figure 1. It translates the byte code into native code in six phases. First, it constructs the basic blocks and loop structure from the byte code. Next, it applies method inlining to both static

JAVA'99 San Francisco California USA

and dynamic method calls. Inlining of dynamic method call is applied using our class hierarchy analysis. The JIT compiler then applies exception check elimination, as well as other optimizations such as constant propagation and dead code elimination. After that, it applies common subexpression elimination to reduce the number of accesses to arrays and instance variables. We note here that we extended the byte code to represent an object's interior pointer for improving consecutive array accesses.

Next, the JIT compiler maps each stack operand to either a logical integer or floating-point register, and counts the numbers of uses of local variables in each region of a program. The regions are also decided based on the loop structure in this phase. Finally, it generates native code from the byte code along with a physical register allocator. Since the JIT compiler requires fast compilation, expensive register allocation algorithms, such as graph coloring, cannot be used. Instead, it uses a simple and fast algorithm to allocate registers without an extra phase. In each region, frequently used local variables are allocated to physical registers. The remaining registers are used for the stack operands used in computation. If the code generator requires a new register but no registers are available, the register allocator finds the least recently used register that can be used, to avoid expensive computation for searching spill candidates. Live information on local variables, obtained from data flow analysis, is also used to avoid generation of inefficient spill code.

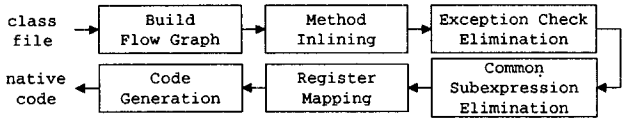


Figure 1: An overview of the JIT compiler

3. Optimization for Accesses to Arrays and Instance Variables

Many exceptions may be thrown from various causes in Java programs. An access to an array or an instance variable frequently causes an explicit exception check at runtime. An access with a null object causes a null-pointer exception. An access to an array with an out-of-bounds index causes an array-bounds exception.

In a typical implementation of a multi-dimensional array in Java, to generate a target effective address requires multiple array references and array-bound checks for each dimension, which requires more expensive implementation in Java than in C or Fortran. The implementation of access to an instance variable is also more expensive than that of access to a local variable, since a local variable can be allocated to a physical register.

In this section, we describe three optimizations: exception check elimination, lightweight exception checking, and common subexpression elimination.

3.1 Exception Check Elimination

The JIT compiler can eliminate null-pointer and array-bounds checks, if it can prove that the access is always valid or that the exception check has already been tested. The JIT compiler has to generate the code for explicit null-pointer checks because AIX permits to read address 0 for a speculative load. It eliminates null-pointer checks by solving a data flow equation.

To eliminate array-bounds checks efficiently, we developed a new algorithm by extending the elimination phase in Gupta's algorithm [4]. It propagates the information of checked exceptions forward and backward, using data flow analysis. Our algorithm computes the exact range of the checked index set by adding a constant to the index variable, an operation treated as *kill* [5] by the previous algorithm. Therefore, exact information on the checked exceptions can be propagated to the successor statements. Furthermore, our algorithm can also eliminate the checks of array accesses with a constant index, which cannot be eliminated by the previous algorithm. In short, our algorithm extends the capability for eliminating array-bounds checks. We give an example in Example 1. The exception checks for **bold references** are eliminated by both algorithms. The exception checks for ***italic references*** can be eliminated only by our algorithm.

```
a[i]=0;
a[i+2]=2;
if (j==k) a[i++]=0;
a[i+2]=a[i]+a[i+1];
a[i+1]=a[0]+a[1]+a[2];
```

Example 1: Example of Exception Check Elimination

3.2 Lightweight Exception Checking

Even after application of the above algorithm, many exception checks could remain. Therefore, we developed lightweight exception checking to reduce the overhead of runtime checks.

The PowerPC architecture provides a `trap` instruction to execute compare and branch to the handler, and this instruction requires only one cycle if it is not taken. To use the instruction, the handler must identify the cause of an exception in the handler. The exception checks are executed frequently, but they seldom throw an exception. If a register is used to identify the cause of an exception, the assignment for a rarely-thrown exception becomes an overhead on a critical execution path. Therefore, the JIT compiler generates only a `trap` instruction with a uniquely encoded condition corresponding to the cause of an exception. If an exception occurs, the `trap` instruction is decoded to identify the cause of an exception in the handler. The handler can tell from the instruction what exception has occurred. We give an example of generated native code in Example 2. Here, three `trap` instructions (`tw` and `twi`) are generated for three different conditions without register assignments to identify the cause of each exception.

Generated code

```
; r4 : array index
; r5 : array base
; r6 : array size
; r7 : divisor

twi   EQ, r5, 0   ; Check null-pointer
tw    GE, r4, r6  ; Check array-bounds
mulli r4, r4, 2
lwzx  r3, r4(r5)  ; Get array element
twi   LLT, r7, 1  ; Check divisor
divi  r3, r3, r7
```

The handler

```
void TrapHandler(struct context *cp)
{
      int *iar = cp->IAR;           // Get the address at which
                                    //   the exception occurs
      if IS_TRAPI_EQ(iar) {         // Is inst. 'twi EQ' ?
            process_NULLPOINTER_EXCEPTION()
      } else if IS_TRAP_GE(iar) {   // Is inst. 'tw GE' ?
            process_ARRAYOUTOFINDEX_EXCEPTION()
      } else if IS_TRAPI_LLT(iar) { // Is inst. 'twi LLT' ?
            process_ARITHMETIC_EXCEPTION()
      }
}
```

Example 2: Example of Lightweight Exception Checking

3.3 Common Subexpression Elimination

To reduce the overhead of array accesses, the JIT compiler applies two techniques for global common subexpression elimination (CSE). One is scalar replacement of array elements. The other is the improvement for consecutive array accesses using an interior pointer. The former generates a temporary local variable for array element and replaces the same array element accesses with the local variable only if the array object and the index variable are not updated in a loop. The latter introduces an instruction for an effective address generation, commonly used by consecutive array accesses. In either case, the code will be moved out of the loop if it is a loop invariant. These techniques are applied only to the loop, since value numbering [5] is too expensive. For a garbage collection, the top pointer of the object must be kept in the memory or register. Because CSE generates an interior pointer of an object and the garbage collector does not scan an interior pointer of an object.

To reduce accesses to instance variables, the JIT compiler uses partial redundancy elimination [6, 7]. It eliminates redundant accesses in a method by moving invariant accesses out of loops and by eliminating identical accesses that are performed more than once on any execution path. The instance variable moved out of loops can be mapped to a local variable, which can be allocated to a physical register.

We give an example of CSE in Example 3. We introduce C notation to represent an interior pointer of an object. The **bold local variables** are generated by each optimization. First, the accesses to the instance variable 'a' are moved out of the loop and are replaced with the local variable '`la`'. Finally, the accesses to the arrays '`la[i]`' and '`la[i+1]`' are replaced with accesses using interior pointer '`*ia0`'. The references to '`la[i]`' and '`la[i+1]`' are also replaced with the local variables '`iv0`' and '`iv1`' by scalar replacement. Consequently, there are only one access to the instance variable and four accesses to the array.

For correct and effective array bound check, the JIT generates the code for array-bounds checks between `i` and `i+1` for original references to '`la[i]`' and '`la[i+1]`' at '`*ia0=&la[i]`' in the example. It can reduce the number of array-bounds checks from six times in the original code to two times.

```
Original Code
class cafe {
   int a[];
   public void babe () {
      this.a = new int[10];
      for (i=0; i<8; i++) {
         if (this.a[i]<this.a[i+1]) {
            int t=this.a[i];
            this.a[i]=this.a[i+1]; this.a[i+1]=t;
         }
      }
   }
}

                    CSE for
                    instance variable

class cafe {
   int a[];
   public void babe () {
      this.a = new int[10];
      int la[] = this.a;
      for (i=0; i<8; i++) {
         if (la[i]<la[i+1]) {
            int t=la[i]; la[i]=la[i+1]; la[i+1]=t;
         }
      }
   }
}

                    CSE for array

class cafe {
   int a[];
   public void babe() {
      this.a = new int[10];
      int la[] = this.a;
      for (i=0; i<8; i++) {
         int *ia0=&la[i], iv0=*ia0, iv1=*(ia0+1);
         if (iv0 < iv1) {
            int t = iv0; *ia0=iv1; *(ia0+1)= t;
         }
      }
   }
}
```

Example 3: Example of CSE

4. Optimization for Type Inclusion Test

In this section, we describe the implementation of type inclusion test. Previous approaches [8, 9] on type inclusion test in constant time have been proposed encoding the class hierarchy in a small table, but they require recomputation of the table when a class is loaded or unloaded dynamically. They also require additional space for the table. We implemented type inclusion test in a completely different approach. To avoid time and space overheads, we generate a simple inlined code to test the most-frequently occurring cases, as in Example 4. This is based on our investigation, described in Section 6.3.

```
Java Code
   Type to = (Type)from;

Pseudo Code
   if (from == NULL) then to = from;
   else if (from.type == Type) then to = from;
   else if (from.type.lastsucc == Type) then to = from;
   else if (call expensive test in C) then {to = from; from.type.lastsucc = Type;}
   else throw exception
```

Example 4: Pseudo-code of a Simple Type Inclusion Test

The first case checks whether the referenced object (`from`) is NULL. The second case checks whether the class of the referenced object is identical to the class of the operand expression (`Type`). The third case checks whether the class cached by the latest comparison in the referenced object is identical to the class of the operand expression. These three checks can avoid overhead of the expensive test, since each takes only two or three machine instructions. If all of these tests fail, then the C runtime library is executed for traversing a linked list of the class hierarchy. Its cost is higher than the first three cases. This simple implementation is effective, as will be shown in Section 6.3.

5. Optimization for Method Call

In this section, we describe two optimizations of method calls: first, inlining of static method calls. Secondly, and then resolution of dynamic method calls.

5.1 Inlining of Static Method Call

In object-oriented languages, a typical program has smaller methods and method calls occur frequently. Futhermore, the constructor is invoked with the creation of a new class. Therefore, the JIT compiler inlines small methods to reduce the number of static method calls. The JIT compiler also optimizes tail recursion and recursive call. It replaces a tail recursion with a branch to the beginning of the method, and it expands the body of the method once when a recursive call is detected.

5.2 Resolution of Dynamic Method Call

In the object-oriented language, dynamic method call is an important feature for its flexibility, and thus it is used frequently. On the other hand, it penalizes the performance of the program because of the overhead of method lookup. Many techniques for resolving this performance problem, such as type prediction [10, 11] and polymorphic inline cache [12], have been proposed. However, they incur overheads by requiring an additional runtime test. In our JIT compiler, we chose class hierarchy analysis (CHA) [13, 14] to improve the performance of dynamic method call. We will discuss the choice in more detail in Section 7.

CHA determines a set of possible targets of a dynamic method call by combining a static type of object with the class hierarchy of a program. If it can be determined that there is no overridden method, the original dynamic method call can be replaced with the

static method call at compile time and can be executed without method lookup. Previously, CHA has been investigated and implemented for the languages supporting static class loading, in which the class hierarchy does not change at runtime. Java supports dynamic class loading, in which the class hierarchy may change in the future. To our knowledge, CHA has not yet been implemented for any language that supports dynamic class loading.

We adapted CHA to dynamic class loading. If class loading overrides a method that has not been overridden, the static method call must be replaced back with the original dynamic method call. Since Java is an explicitly multi-threaded language, all optimizations must be thread-safe. That is, the code must be modified atomically by rewriting only one instruction. We implemented this atomic updating as shown in Example 5. In the example, we assume the object layout that combines the class instance data and the header such as Caffeine [15] so that three load instructions are required to get the address of a compiled code.

```
Before overriding the method          After overriding the method

    call  imm_ca                  →       jmp   dynamic_call   // static method call
    jmp   after_call                      jmp   after_call
dynamic_call:                         dynamic_call:
    load  cp, (obj)                       load  cp, (obj)      // load class pointer
    load  mp, (cp)                        load  mp, (cp)       // load method pointer
    load  ca, (mp)                        load  ca, (mp)       // load code address
    call  (ca)                            call  (ca)           // dynamic method call
after_call:                           after_call:
```

Example 5: Example of the Resolution of Dynamic Method Call

At compile time, the top address of the dynamic method call sequence is recorded. The address is filled with a `call` instruction to call a method statically. When the method is not yet overridden in the left column in Example 5, the *italic code sequence* for the dynamic method call is not executed at all. When the method is overridden by dynamic class loading, the `call` instruction in the address is replaced with a `jmp` instruction to the dynamic method call by the class loader. Consequently, the code sequence for the dynamic method call is now executed. The JIT compiler also uses as similar implementation for inlining of dynamic methods.

Java provides an interface for implementation of multiple inheritance. The JIT compiler also optimizes an interface call by replacing it with a virtual call. If CHA finds that only one class implements an interface class, a virtual call with a single method lookup can be generated using the implementation class as a static type. Furthermore, if the target method is not overridden through the implementation class hierarchy, the JIT compiler can replace the interface call with a static method call. This optimization is much more efficient than a naive implementation of an interface call, which requires a loop to search for an implementation class.

6. Experiments

In this section, we evaluate the effectiveness of individual optimizations such as exception check elimination, simple type inclusion test, common subexpression elimination, inlining of static method call, and resolution of dynamic method call. We used nine Java programs, seven of which (**`compress`**, **`jess`**, **`raytrace`** (a body of mtrt), **`db`**, **`javac`**, **`mpegaudio`**, and **`jack`**) are benchmarks in SPECjvm98 [16]. The other two (**`hotjava`** and **`swing`**) are applications with GUIs released by JavaSoft. SPECjvm98 are executed with size '100'. Therefore, the results do not follow the official SPEC rules. HotJava is executed with an access a web page with an applet and GIF data. Swing is executed with clicks to all tabs. All the measurements were taken on an IBM RISC System 6000 Model 7043-140 (containing a 332-MHz PowerPC 604e with 512 MB of RAM) running AIX 4.3.1.

6.1 Benchmarks

Table 1 shows the static characteristics of the class files for each program at compile time.

Table 2 shows the dynamic characteristics of unoptimized code for each program at execution time.

Program	Compiled-Bytecode Size (bytes)	Number of Compiled Methods	Static Call Sites	Virtual Call Sites	Interface Call Sites	Type Inclusion Test Sites	Array Access Sites	Instance Variable Sites	Exception Check Sites
compress	23598	276	1525	280	7	41	183	1246	2964
jess	44548	704	3494	746	38	122	507	2716	6068
raytrace	33163	424	2879	1133	7	60	476	2489	4846
db	25605	291	1924	355	21	52	169	1005	3113
javac	91144	1068	5614	1833	72	406	412	6737	11730
mpegaudio	38204	441	2190	335	21	71	1237	2374	6838
jack	50573	522	3197	779	88	219	1152	2648	7879
hotjava	193868	3032	10190	4863	274	25322	2390	13607	27524
swing	282982	4822	9854	9732	1025	2647	2942	20837	40024

Table 1: Static (compile-time) characteristics

program	Static Calls	Virtual Calls	Interface Calls	Type Inclusion Tests	Array Accesses	Instance Variable Accesses	Exception Checks
compress	225935935	12765	93	2274	650483870	236458881	38580879
jess	108104957	35498836	706107	29204058	91339251	259063616	56322834
raytrace	278960441	26664017	147	3280212	81405118	334641372	77942110
db	96181237	1562479	14931186	85991464	153086345	333401516	73883542
javac	65204998	49808807	3531139	12099157	49642977	328850733	53164623
mpegaudio	103004068	9843381	181867	51989	1630911748	1099455343	43194436
jack	35584857	13282175	3965412	7651521	149029390	758644986	11041840
hotjava	683972	542498	35796	175767	1077044	2966944	5925880
swing	1741114	2903810	262501	809732	6107677	17204605	35719184

Table 2: Dynamic (runtime) characteristics

6.2 Exception Check Elimination

Figure 2 shows how our exception check elimination reduces the number of exception checks at runtime. All values are given as percentages of the non-optimized case. The left bar shows the number of exception checks without the elimination. The right bar shows the number of exception checks with the elimination. The dark bar shows the number of null-pointer checks. The white bar shows the number of array-bounds checks.

It is proved that our exception check elimination is very effective, especially for null-pointer checks, of which it eliminates an average of 60%. It is quite effective for array-bound checks, of which it eliminates 53%, particularly for **`mpegaudio`**.

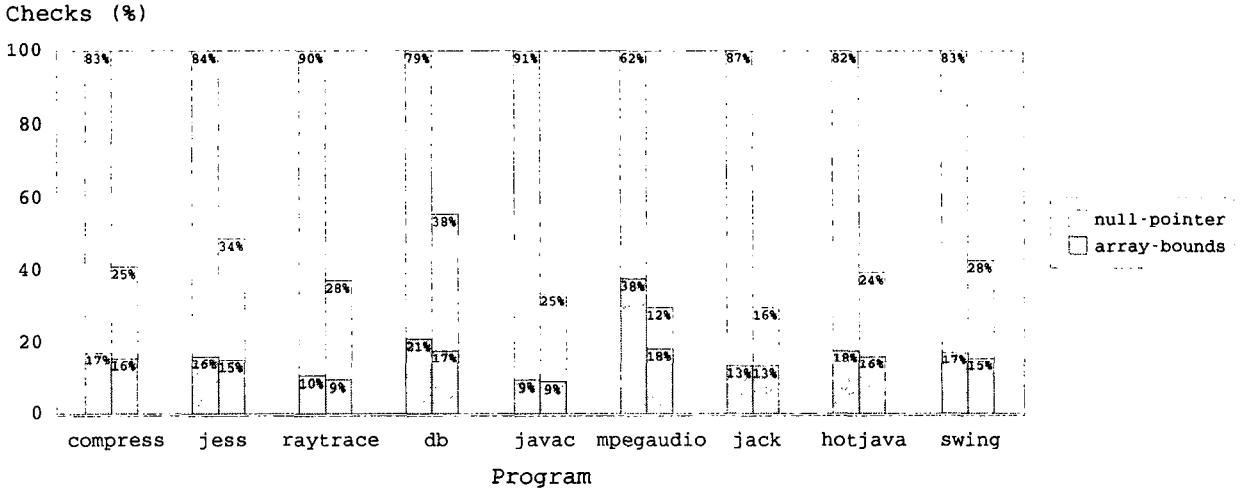


Figure 2: Results of Exception Check Elimination at Runtime

6.3 Simple Type Inclusion Test

Figure 3 shows the distribution of object types in type inclusion test at runtime. `Same` indicates the case in which the class of the referenced object is identical to the class of the operand expression. `Null` indicates the case in which the referenced object is NULL. `Cache` indicates the case in which the class cached by the latest comparison in the referenced object is identical to the class of the operand expression. These three cases are processed by inlined test code. `Normal` indicates the case in which a class hierarchy must be traversed to determine the result. `Others` indicates the case in which the class of the reference object or operand expression is either interface or array type. These two cases are processed in the C runtime library.

`Same`, `null`, and `cache` account for an average of 91% of tests in the programs. The result shows that our simple implementation of inlined test code is effective for the Java environment.

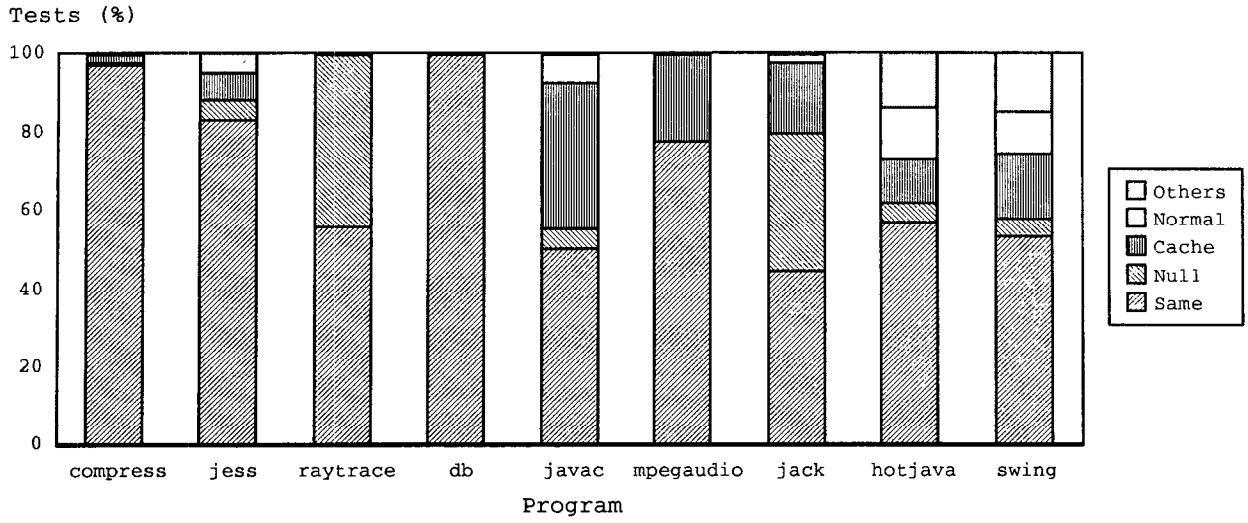


Figure 3: Distribution of Object Types in Type Inclusion Test at Runtime

6.4 Common Subexpression Elimination

Figure 4 shows how common subexpression elimination (CSE) reduces the number of accesses to arrays and instance variables at runtime. All values are given as percentages of the non-optimized case. The left bar shows the number of accesses without CSE. The right bar shows the number of accesses with CSE. The dark bar shows the number of accesses to instance variables. The white bar shows the number of accesses to arrays. The striped bar shows the number of array accesses using interior pointers.

Our CSE is effective except for **raytrace**. The elimination of accesses to instance variables is more effective than that of accesses to arrays. Scalar replacement of array accesses is particularly effective for **mpegaudio**, in which 25% of accesses are eliminated. Array access using an interior pointer is effective for **db**, in which 14% of the accesses are used. It optimizes array accesses to swap array elements in the shell sort.

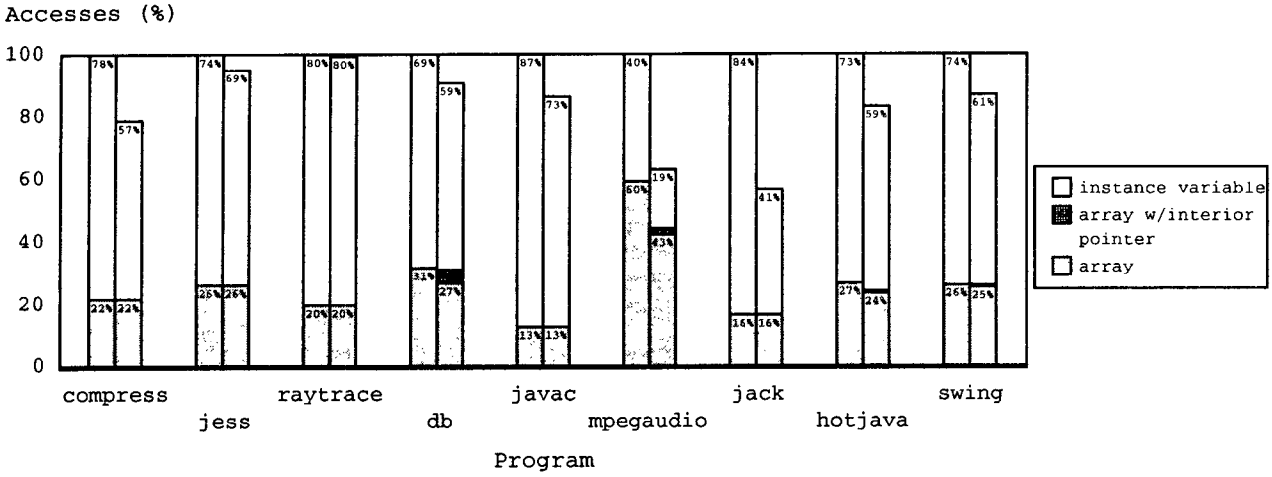


Figure 4: Results by Common Subexpression Elimination at Runtime

6.5 Inlining of Static Method Call

Figure 5 shows how method inlining reduces the number of static method calls at runtime. All values are given as percentages of the non-optimized case. The left bar shows the number of static method calls without inlining. The right bar shows the number of static method calls with inlining. The dark bar shows the number of calls for non-constructors. The white bar shows the number of calls for constructors.

Inlining is particularly effective for **compress** and **raytrace**. An average of 50% of static method calls are eliminated, further increasing opportunities for other optimizations. In all programs, there is a drastic reduction in the number of calls for the constructor.

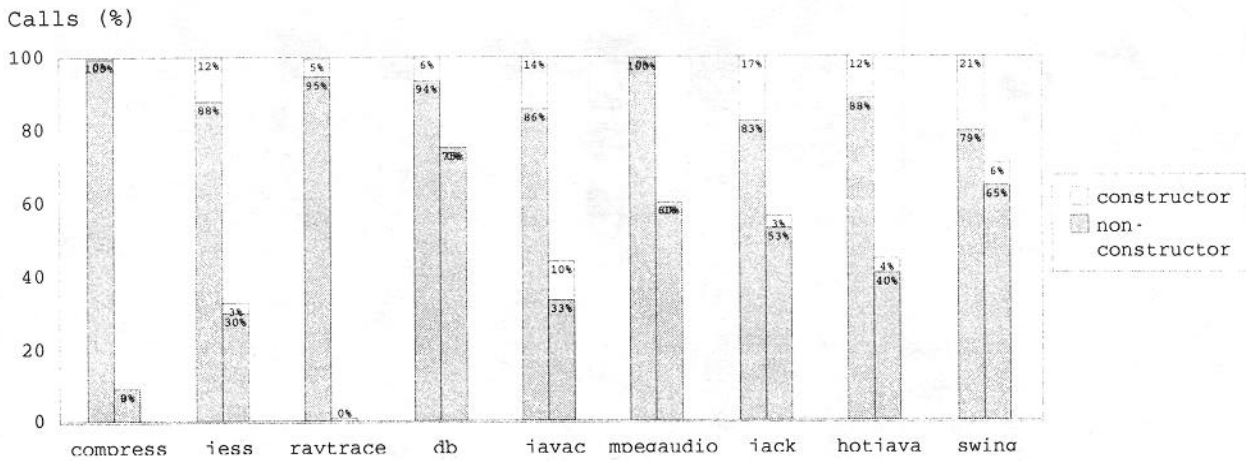


Figure 5: Counts of Static Method Calls at Runtime

6.6 Resolution of Dynamic Method Call

The performance in resolving dynamic method call using class hierarchy analysis (CHA) is shown in Figure 6 (for statistics on call sites at compile time) and Figure 7 (for statistics on calls at runtime). In both figures, `Call` contains both virtual and interface calls. The three types of striped bars represent the cases in which dynamic method calls are replaced with static method calls or inlinings. The other three types of bars represent the cases in which dynamic method calls are not resolved by CHA. The dark doted bars represent call sites or calls are dynamically monomorphic, but are not resolved by CHA. The black bars represent call sites or calls that are polymorphic. In Figure 6, white bars represent call sites whose calls are not executed at runtime. In Figure 7, white bar `(Deresolved)` represent cases in which dynamic method call or inlining with method lookup. The static method call or inlining resolved by CHA are replaced by them at runtime.

Figure 7 shows that CHA is highly effective, since it resolved an average of 85% of dynamic method calls or inlinings for three out of nine programs. Furthermore, it resolved an average of 40% for all of the other programs, except for compress. In **compress,** the unresolved methods do not affect the performance, since many static calls for the kernel routines that use *final* classes occur. The optimization for interface call is also effective for db, since the `java.util.Vector` class, which uses the implementation class of the interface class, is used very frequently. The results also show that all the programs except **mpegaudio** are surprisingly monomorphic. Therefore, we still have an enough room to improve the performance in the four programs.

Since we have adapted CHA to dynamic class loading, a static method call may be replaced back with a dynamic method call by overriding the target method when a class is loaded dynamically. For inlining of dynamic method call, method lookup may be also required. In our experiment, the numbers of replaced sites are 216 for **swing**, 107 for **hotjava**, 66 for **jack**, 22 for **db**, 12 for **javac**, and fewer than 10 for the other programs. In **jack** and **hotjava**, the dynamic method calls or inlinings are executed remarkably. The overhead of replacing the code is small.

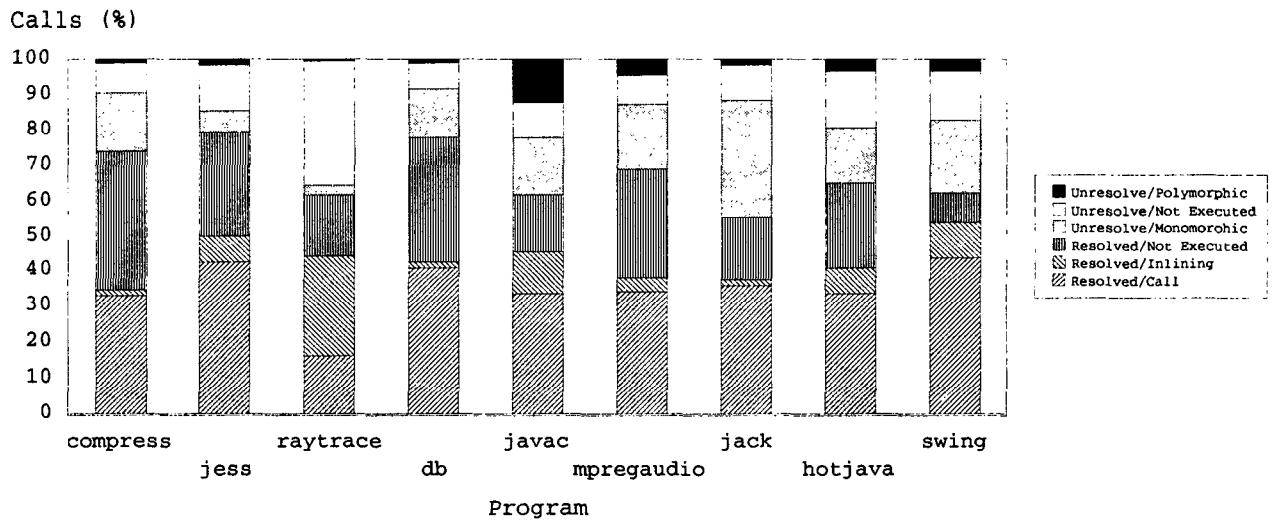


Figure 6: Resolution of Dynamic Method Call Sites at Compile Time

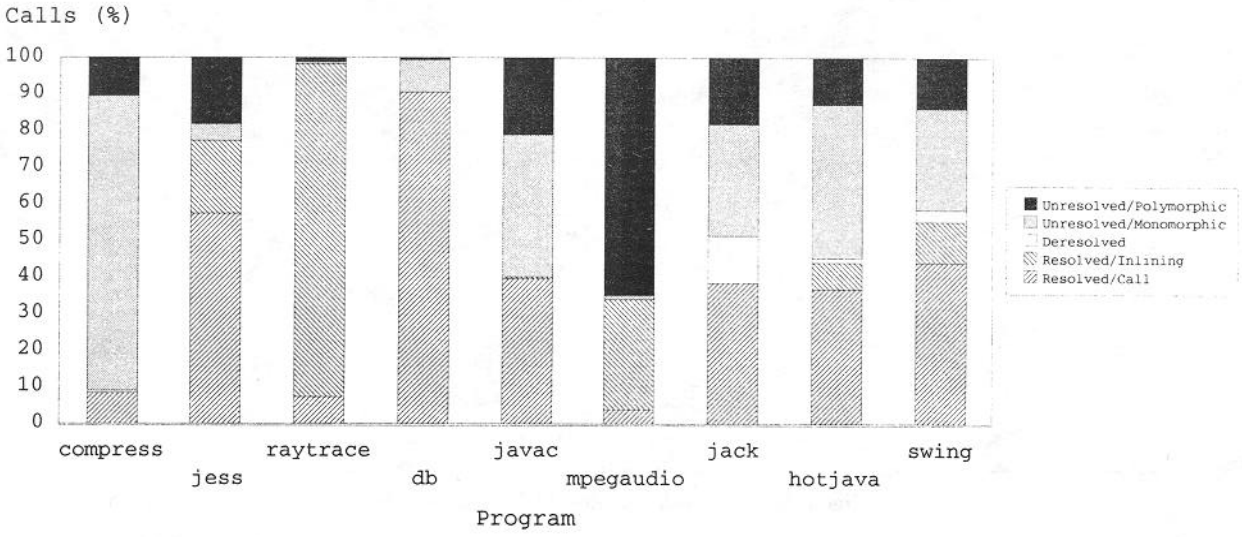
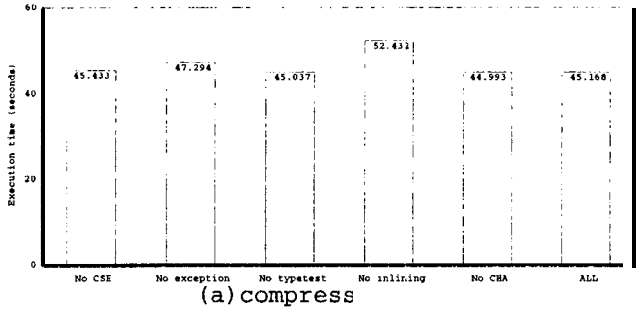


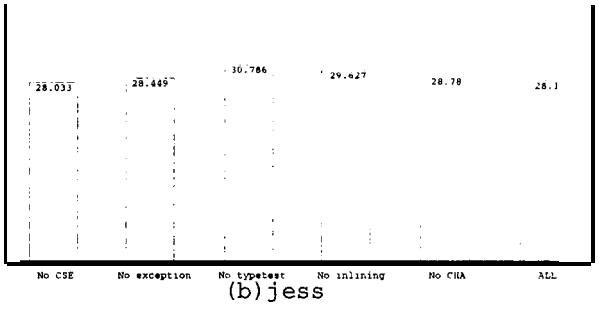
Figure 7: Resolution of Dynamic Method Call Sites at Runtime

6.7 Performance

We measured the execution time of seven of the programs, since the two other programs were difficult to measure because of their interactive nature. Figure 8 shows the performance improvements from various optimizations. The white bar represents the best execution time in five trials. All values are given in seconds. Each of the bars except the rightmost bar shows the effect of all but one optimization. The optimizations include common subexpression elimination **(No CSE),** exception elimination and lightweight exception checking (No exception), simple type inclusion test **(No** typetest), inlining of static method call (No inlining), and resolution of dynamic method call (No **CHA**). The rightmost bar **(ALL)** shows the time with all optimizations enabled.

Figure 8 (a) shows the performance of compress. Here, inlining of static method call improve the performance by 14%. Figure 8 (b) shows the performance of **jess**. Here, simple type inclusion test improves the performance by 8%. Figure 8 (c) shows the performance of raytrace. Here, resolution of dynamic method call improves the performance by 20%. Figure 8 (d) shows the performance of db. Here, simple type inclusion test improves by 17%. Figure 8 (e) shows the performance of **javac**. All optimizations make virtually no difference. Figure 8 (f) shows the performance of mpegaudio. Here CSE improves the performance by 13%. Inlining also improves by 10%. Figure 8 (g) shows the performance of **jack**. Here both CSE and exception optimization improve the performance by 10%. Simple type inclusion test also improves by 9%. It shows all optimizations contribute to an improvement in the performance.





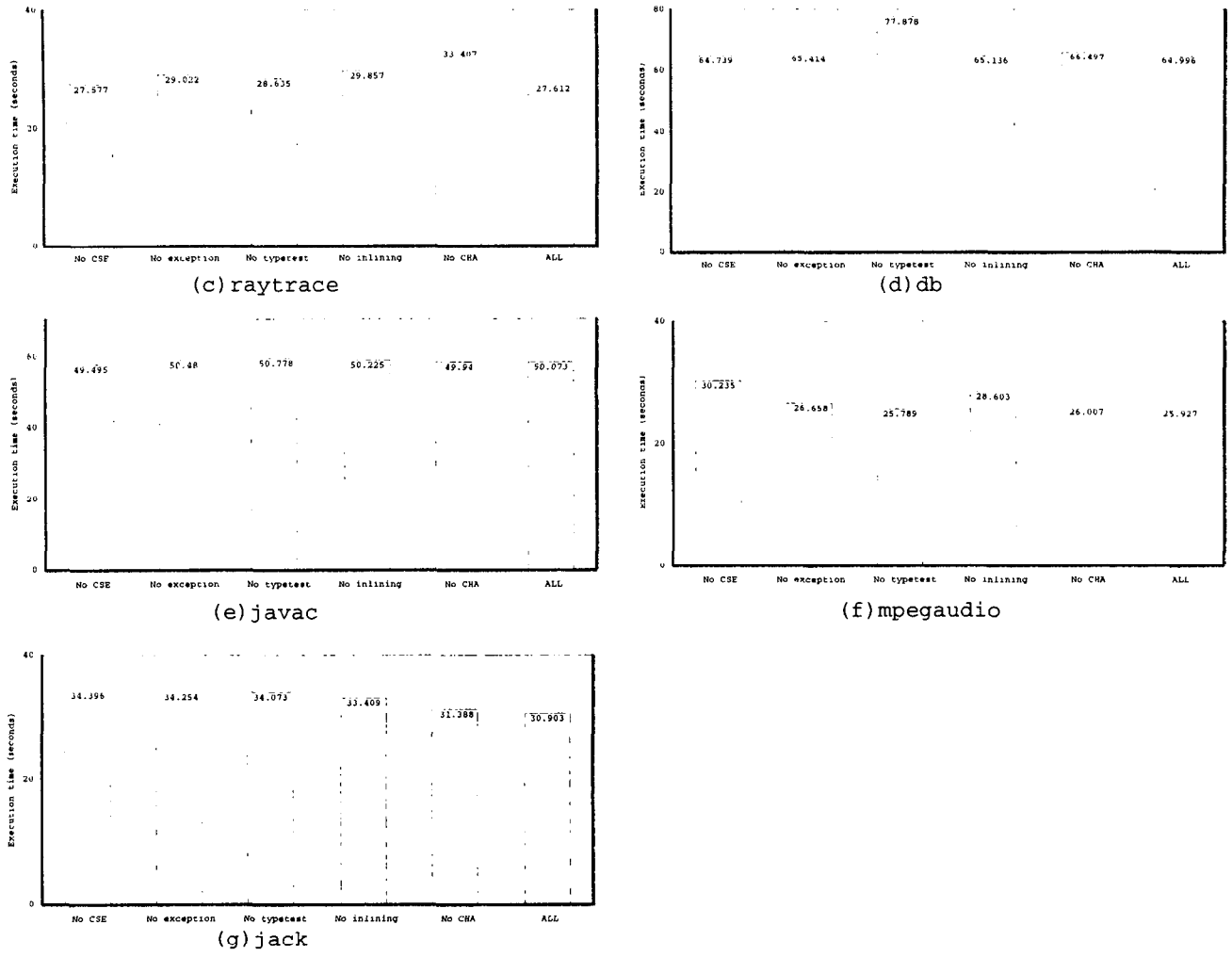


Figure 8: Execution Times of the JIT'ed code

7. Related Work

The Intel JIT compiler [17] applies simple array-bounds check elimination in the extended basic block. Our JIT compiler applies array-bounds and null-pointer check elimination to the whole method using our algorithm. The experiment shows it is effective.

IBM Research is developing another dynamic optimizing compiler [18]. Flow-insensitive optimizations are implemented in a faster fashion. Interprocedural optimizations will be implemented.

Exception check elimination [4, 19] has been proposed as means of reducing the overhead of certifying the correctness of a program. We have extended the elimination algorithm, using more exact program analysis.

Type inclusion test [8, 9] has been investigated for efficient type conformance test in an object-oriented language. In previous researches, the class hierarchy was encoded in a small table, so that it could be tested in a constant time. The table may be reconstructed in the future by dynamic class loading. To avoid the time and space overhead, we investigated the behavior of type inclusion test in Java. The results show that simple checks with the cache of the referenced object account for an average of 91% of tests. Therefore, we chose the simple implementation.

Polymorphic inline cache (PIC) [12] has been proposed as means of reducing the overhead of polymorphic method call. PIC compiles a dynamic method call as though it was being inlined into the context of the caller. The call site is patched to jump to a stub that conditionally executes the inlined code on the basis of the types of an object. Type prediction [10, 11] has also been proposed, with type analysis for languages supporting dynamic class loading. Type prediction predicts the type of an object, which are called frequently, at compile time. PIC and type prediction introduce a runtime test newly, since they are on the basis of the cache mechanism with memory references. According to the results of simple experiments [20], type prediction without inlining at 100% accuracy cannot outperform resolution of dynamic method call without inlining. Type prediction with inlining must achieve 90% accuracy to outperform against the resolution without inlining. Finally, nothing can perform the resolution with inlining. In implementations of Java, the cost of dynamic method call is not so different from that of PIC and type prediction.

Class hierarchy analysis (CHA) [13, 14] can replace a dynamic method call with a faster static method call at compile time. It has been investigated and implemented for languages supporting static class loading. To avoid the runtime overhead of PIC and type prediction, we developed a version of CHA adapted to dynamic class loading. It allows the JIT compiler to inline dynamic method call without a runtime execution overhead. Inlining increases the opportunity for other optimizations. The experimental results showed the effectiveness of our approach.

8. Conclusions

In this paper, we presented optimizations that we developed for a production JIT compiler to reduce the overhead of the Java language, which supports dynamic class loading without compromising flexibility and safety. We validated our approach on the basis of various statistics collected by running nine large Java programs. We evaluated each optimization by turning off one by one. Finally, by investigating the statistics collected in our experiment, we showed that there are still some rooms to further improve runtime Java performance.

Acknowledgement

We are grateful to our group's people of Tokyo Research Laboratory for implementing our JIT compiler and for participating in helpful discussion. We also thank Michael McDonald for his proofreading assistance.

References

[1] James Gosling, Bill Joy, and Guy Steele: "The Java Language Specification," Addison-Wesley, 1996.

[2] James Gosling: "Java Intermediate Bytecodes," ACM SIGPLAN Workshop on Intermediate Representations, 1995.

[3] International Business Machines Corp. "AIX Java Development Kit 1.1.6," Available at http://www.ibm.com/java/

[4] Rajiv Gupta: "Optimizing array bound checks using flow analysis," ACM Letters on Programming Languages and Systems, 2(1-4): pp. 135-150, 1993.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman: "Compiler: Principle, Techniques, and Tools," Addison-Wesley, 1986.

[6] Etienne Morel and Claude Renvoise: "Global Optimization by Suppression of Partial Redundancies," Communication of the ACM, vol. 2, no. 2, pp. 96-103, 1979.

[7] Jens Knoop, Ruthing Oliver, and Steffen Bernhard: "Lazy Code Motion," In Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pp. 224-234, 1992.

[8] Norman Cohen: "Type-extension type tests can be performed in constant time," ACM Transactions on Programming Languages and Systems, Vol. 13 No.4, pp. 626-629, 1991.

[9] Jan Vitek, R. Nigel Horspool, and Andreas Krall: "Efficient Type Inclusion Test," In Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '97, pp. 142-157, 1997.

[10] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers: "Profile-Guided Receiver Class Prediction," In Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '95, pp. 107-122, 1995.

[11] Gerald Aigner, and Urs Holzle: "Eliminating Virtual Function Calls in C++ Programs," In Proceedings of the 10th European Conference on Object-Oriented Programming – ECOOP '96, volume 1098 of Lecture Notes in Computer Science, Springer-Verlag, pp. 142-166, 1996.

[12] Urs Holzle, Craig Chambers, and David Ungar: "Optimizing Dynamically-Typed Object-Oriented Langages With Polymorphic Inlin Caches," In Proceedings of the 5th European Conference on Object-Oriented Programming – ECOOP '91, volume 512 of Lecture Notes in Computer Science, Springer-Verlag, pp. 21-38, 1991.

[13] Jeffery Dean, David Grove, and Craig Chambers: "Optimization of object-oriented programs using static class hierarchy," In Proceedings of the 9th European Conference on Object-Oriented Programming – ECOOP '95, volume 952 of Lecture Notes in Computer Science, Springer-Verlag, pp. 77-101, 1995.

[14] Mary F. Fernandez: "Simple and Effective Link-Time Optimization of Modula-3 Programs," In Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, pp. 103-115, 1995.

[15] Cheng-Hsueh A. Hiesh, John C. Gyllenhaal, and Wen-mei W. Hwu: "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results," In 29th Annual IEEE/ACM International Symposium on Microarchitecture, 1996.

[16] Standard Performance Evaluation Corp. "SPEC JVM98 Benchmarks," Available at http://www.spec.org/osg/jvm98/

[17] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth: "Fast, Effective Code Generation in a Just-In-Time Java Compiler," In Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pp. 280-290, 1998.

[18] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, and Harini Srinivasan: "The Jalapeño Dynamic Optimizing Compiler for Java, " to appear in JavaGrande, 1999

[19] Priyadarshan Kolte and Michael Wolfe: "Elimination of Redundant Array Subscript Range Checks," In Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, pp. 270-278, 1995.

[20] David F. Bacon: "Fast and Effective Optimization of Statically Typed Object-Oriented Languages," Ph.D. thesis, University of California at Berkeley, 1997.