Design, Implementation, and Evaluation of Optimizations in a Java™ 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, which are executed 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 a JIT compiler, such as exception check elimination, common subexpression elimination, simple type inclusion test, method inlining, and devirtualization of dynamic method call. We evaluate the performance benefits of these optimizations based on various statistics collected using SPECjvm98, its candidates, and two JavaSoft applications with byte code sizes ranging from 23000 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. It has excellent safety, flexibility, and reusability. The safety was achieved by introducing exception checks and disallowing interior object pointers. The flexibility and reusability were 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. Java 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 array elements and instance variables.

To improve the performance of the Java execution, two compiler solutions have been proposed: a static compilation model and a "Just-In-Time" (JIT) compilation model. 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, static compilation can use expensive optimizations. On the other hand, it does not support dynamic class loading, and does not take advantage of Java's support for program flexibility and reusability. The JIT compilation translates byte code into native code when a new method is invoked at runtime, and 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 that we developed to reduce various runtime overheads of the Java language without compromising its 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 array elements and instance variables. Our type inclusion test uses a simpler method than previous approaches, and is effective. Inlining of static method call increases the opportunity for other optimizations. Devirtualizing dynamic method call using direct binding with a class hierarchy analysis (CHA) is a new approach to reducing the overhead of dynamic method call in the

sense that we adapted direct binding with CHA to dynamic class loading. It also allows dynamic methods to be inlined, to increase the opportunity for other optimizations.

We validated our approach on the basis of various statistics collected by running fourteen large Java programs. We evaluated each of the optimizations by turning them off one by one. All the evaluations were carried out with the Java JIT compiler for the PowerPC architecture on AIX, whose product version will be shipped with JDK 1.1.8 for AIX 4.3.3 [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 for reducing 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 0 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 and dynamic method calls. Inlining of dynamic method call is applied using our direct binding with CHA. 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 array elements and instance variables. We note here that we extended the byte code to represent an object's interior pointer, with the aim of improving accesses to consecutive array elements.

Next, the JIT compiler maps each stack operand to either a logical integer or a 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, native code is generated 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 [4], cannot be used. Instead, a simple and fast algorithm is used 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 needed 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 the computational expense of searching for spill candidates. Live information on local variables, obtained from data flow analysis, is also used to avoid generation of inefficient spill code.

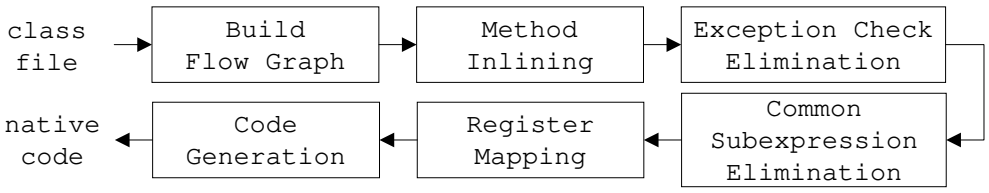


Figure 1: Overview of the JIT compiler

3. Optimization of Accesses to Arrays and Instance Variables

Many exceptions may be thrown in Java programs, for various reasons. An access to an array element 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 element with an out-of-bounds index causes an array-bounds exception.

In a typical implementation of a multi-dimensional array in Java, generating an 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. It has to generate the code for explicit null-pointer checks, because AIX permits address 0 to be read for a speculative load. The JIT compiler eliminates null-pointer checks by solving a data flow equation.

To eliminate array-bounds checks efficiently, we improved the elimination phase in Gupta's algorithm [5]. This phase propagates the information on 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* [6] by the previous algorithm. Therefore, more exact information on the checked exceptions can be propagated to the predecessor or successor statements. Furthermore, our algorithm can eliminate checks of array accesses with a constant index, which could not be eliminated by the previous algorithm.

Consequently, it can eliminate more exception checks for array accesses, especially for those with a constant index value. We give an example in Example 1 to show one advantage of our algorithm, where the array index exception check required is explicitly indicated by *italicized statement*s. The number of exception checks, which was originally 11 (the number of array accesses), was reduced to 6 by the existing method, and is further reduced to only 3 by our method.

It is important to eliminate exception null-pointer and array-bounds checks, because optimizations that use data flow analysis treat operations that may throw exceptions as *kill*. The elimination may improve effectiveness of common subexpression elimination.

```
t = a[i]+a[i-1]+a[i-2];
i++;
if (t < 0) {

   t = a[i]+a[i-1]+a[i-2]+a[3];
   i++;
}

t = a[i]+a[i-1]+a[i-2]+a[i-3];

(a) Original Program
```

```
index_check ( 0 <= i-2 );
index_check ( i <= ub );
t = a[i]+a[i-1]+a[i-2];
i++;
if (t < 0) {
    index_check ( i <= ub);
    index_check ( 3 <= ub)
    t = a[i]+a[i-1]+a[i-2]+a[3];
    i++;
}
index_check ( 0<= i-3);
index_check ( i <= ub);
t = a[i]+a[i-1]+a[i-2]+a[i-3];

(b) Result with Gupta's algorithm
```

```
index_check ( 0 <= i-2 );
index_check ( i+1 <= ub );
t = a[i]+a[i-1]+a[i-2];
i++;
if (t < 0) {

   t = a[i]+a[i-1]+a[i-2]+a[3];
   i++;
}
index_check ( i <= ub);

t = a[i]+a[i-1]+a[i-2]+a[i-3];

(c) Result with our algorithm
```

Example 1: Example of Exception Check Elimination

3.2 Lightweight Exception Checking

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

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. 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. To use a `trap` instruction effectively, the handler has to identify the cause of an exception. 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 the 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` in-

structions (`tw` and `twi`) are generated for three different conditions without register assignments in a critical path 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    LLT, r6, r4 ; 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_LLT(iar) {        // Is inst. 'tw LLT' ?
      process_ARRAYOUTOFINDEX_EXCEPTION()
   } else if IS_TRAPI_LLT(iar) {       // Is inst. 'twi LLT' ?
      process_ARITHMETIC_EXCEPTION()
   }
}
```

Example 2: Example of Lightweight Exception Checking

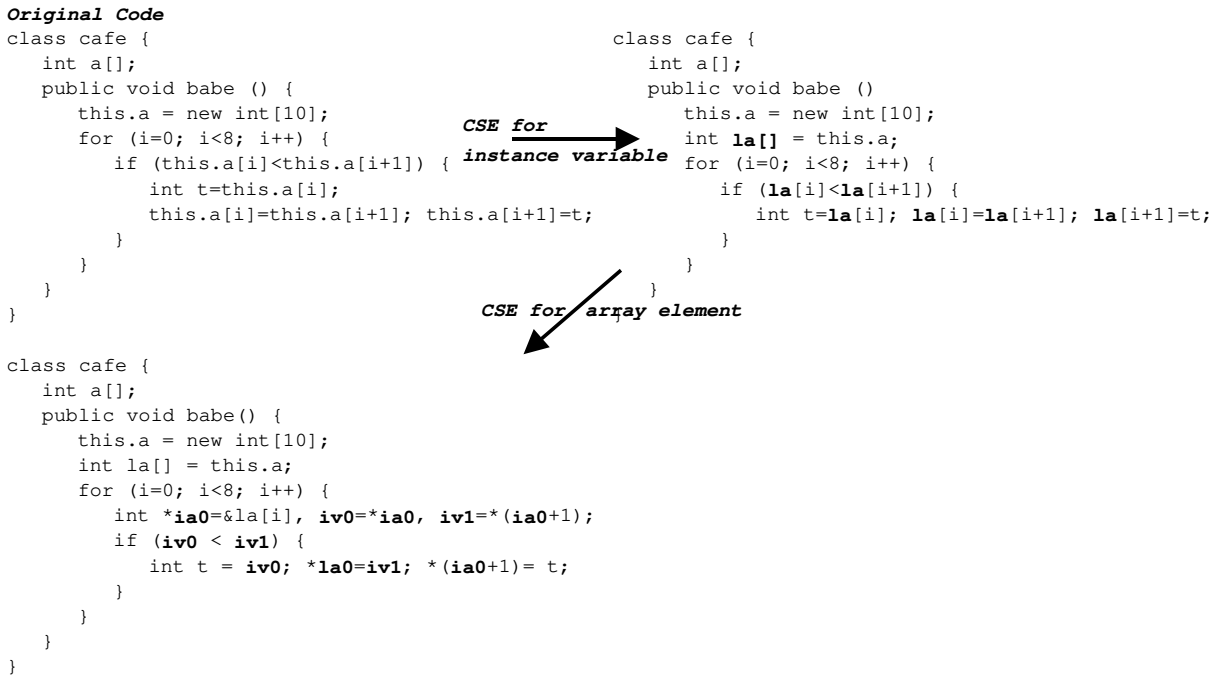
3.3 Common Subexpression Elimination

To reduce the overhead of accesses to array elements, the JIT compiler applies two techniques for common subexpression elimination (CSE). One is scalar replacement of array elements. The other is improvement of accesses to consecutive array elements using an interior pointer. The former generates a temporary local variable for an array element and replaces accesses to the same array element with this variable only if the array object and the index variable are not updated in a basic block. The latter introduces an instruction for generating an effective address, which is commonly used by accesses to consecutive array elements. In either case, the code will be moved out of the loop if it is loop invariant. 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 because of its performance.

To reduce accesses to instance variables, the JIT compiler uses partial redundancy elimination [7, 8]. It eliminates redundant accesses in a method by moving invariant accesses out of a loop and by eliminating identical accesses that are performed more than once on any execution path. The instance variable moved out of a loop can be mapped to a local variable, which can be allocated to a physical register.

An example of CSE is shown 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 replaced with the local variable '`la`'. Finally, the accesses to the arrays '`la[i]`' and '`la[i+1]`' are replaced with accesses using the 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 is only one access to the instance variable and four accesses to the array elements.

For correct and effective array bound checking, 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 6 in the original code to 2. Now, an access to an array element does not require any exception checks.



Example 3: Example of CSE

4. Optimization for Type Inclusion Test

In this section, we describe the implementation of type inclusion test. Previous approaches [9, 10] to type inclusion test in constant time encode 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 by following 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, in accordance with the result of 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 successful comparison in the referenced object is identical to the class of the operand expression. These three checks can avoid the overhead of expensive test, since each takes only two or three machine instructions. If all of them tests fail, then the C runtime library is executed in order to traverse a linked list of the class hierarchy. In this case, the cost is higher. The effectiveness of this simple implementation will be shown in Section 6.3.

5. Optimization of Method Call

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

5.1 Inlining of Static Method Call

In object-oriented languages, a typical program has small methods and method calls occur frequently. Furthermore, the constructor is invoked when a new class is created. Therefore, the JIT compiler inlines small method, 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 applies unrolling to the body of the method when a recursive call is detected.

5.2 Devirtualization of Dynamic Method Call

Dynamic method call is an important feature of object-oriented language because of the flexibility, and it is therefore used frequently. However, it degrades the performance of the program, because of the overhead of method lookup. Many techniques for resolving this performance problem have been proposed, such as type prediction [11, 12], polymorphic inline cache [13], and method test [14]. However, they incur overheads by requiring an additional runtime test. In our JIT compiler, we chose direct binding with class hierarchy analysis (CHA) [15, 16] 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 a static method call by direct binding at compile time, and can be executed without method lookup. Previously, direct binding with CHA has been investigated and implemented for languages that support 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.

We adapted direct binding with CHA to dynamic class loading. If class loading overrides a method that has not been overridden, the static method call must be replaced 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. We implemented this atomic updating by rewriting only one instruction as shown in Example 5. In the example, we assume an object layout that combines the class instance data and the header such as Caffeine [17], so that three load instructions are required to obtain the address of a compiled instruction.

```
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 Devirtualization 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 in order to undo direct binding. Consequently, the code sequence for the dynamic method call is now executed. The JIT compiler also uses a similar implementation for inlining of dynamic methods, as shown in Example 6.

```
Before overriding the method
    nop
   // inlined code
    jmp   after_call
dynamic_call:
    load  cp, (obj)
    load  mp, (cp)
    load  ca, (mp)
    call  (ca)
after_call:
```

→

```
After overriding the method
    jmp   dynamic_call   // static method call
    // inlined code
    jmp   after_call
dynamic_call:
    load  cp, (obj)      // load class pointer
    load  mp, (cp)       // load method pointer
    load  ca, (mp)       // load code address
    call  (ca)           // dynamic method call
after_call:
```

Example 6: Example of the Inlining of Dynamic Method Call

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 a class implements an interface class, a virtual call with a single method lookup can be generated by using the implementation class as a static type. Furthermore, if the target method is not overridden through the implementation class hierarchy, direct binding 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 devirtualization of dynamic method call. We used fourteen Java programs, seven of which (**compress**, **jess**, **raytrace** (a body of mtrt), **db**, **javac**, **mpegaudio**, and **jack**) are benchmarks in SPECjvm98 [18]. Five others (**cst**, **si**, **richards**, **tsgp**, **tmix**) were candidates considered for inclusion in SPECjvm98. The last two (**hotjava** and **swing**) are applications with GUIs, released by JavaSoft. SPECjvm98 was executed with a size of '100', and the results do not follow the official SPEC rules. HotJava 1.1.4 was executed while accessing a web page containing an applet and GIF data. Swing 1.0.3 was 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 768 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
cst	30943	340	1982	438	15	78	246	1312	3921
si	27873	310	1731	462	7	47	169	1216	3386
richards	40216	787	2103	644	36	85	202	1636	5120
tsgp	23155	269	1494	292	7	32	193	980	2845
tmix	28983	386	1919	378	7	44	167	1400	3619
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	3858087901
jess	108104957	35498836	706107	29204058	91339251	259063616	563228349
raytrace	278960441	26664017	147	3280212	81405118	334641372	779421109
db	96181237	1562479	14931186	85991464	153086345	333401516	738835422
javac	65204998	49808807	3531139	12099157	49642977	328850733	531646238
mpegaudio	103004068	9843381	181867	51989	1630911748	1099455343	4319453641
jack	35584857	13282175	3965412	7651521	149029390	758644986	1104184009
cst	37788715	69995954	1472547	11984846	152308744	280904613	698762117
si	63948130	156033719	147	2842724	125970940	782095505	1301374956
richards	223911200	390196676	21914040	26322814	36407310	897969456	1463035929
tsgp	10937	112140585	145	8257098	3769809366	3046100095	5139119054
tmix	11771091	17845514	145	3926999	779618561	917217742	2500397977
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.

The results show that our exception check elimination is very effective, especially for null-pointer checks, of which it eliminates 67% on average. It is also quite effective for array-bound checks, of which it eliminates 53% for **`mpegaudio`** and 69% for **`tmix`**.

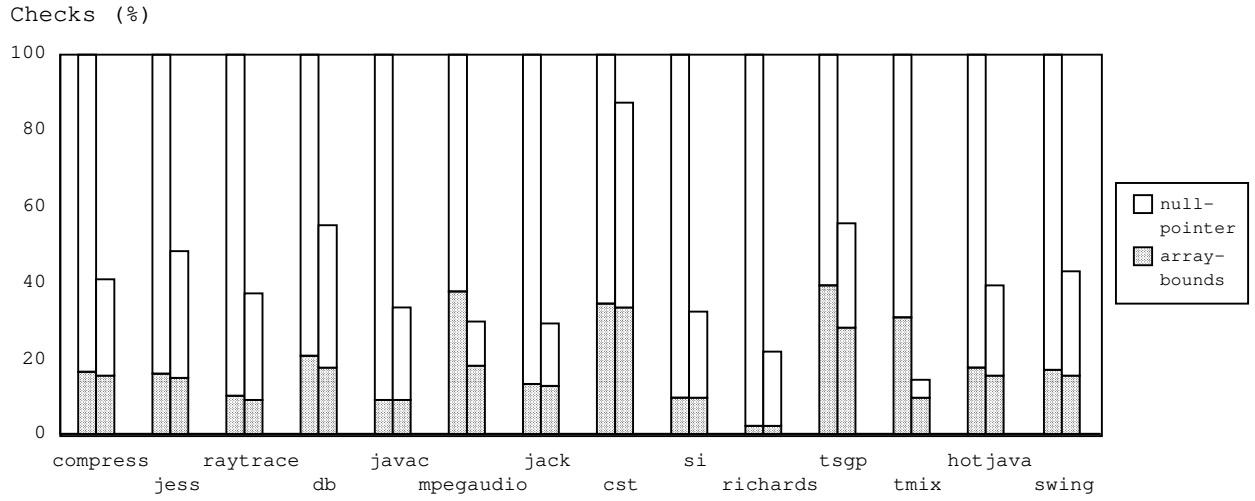


Figure 2: Results of Exception Check Elimination at Runtime

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 successful 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 87% of tests in the programs. The result shows that our simple implementation of inlined test code is effective for the Java environment. In **`tsgp`**, almost all tests are performed with array objects.

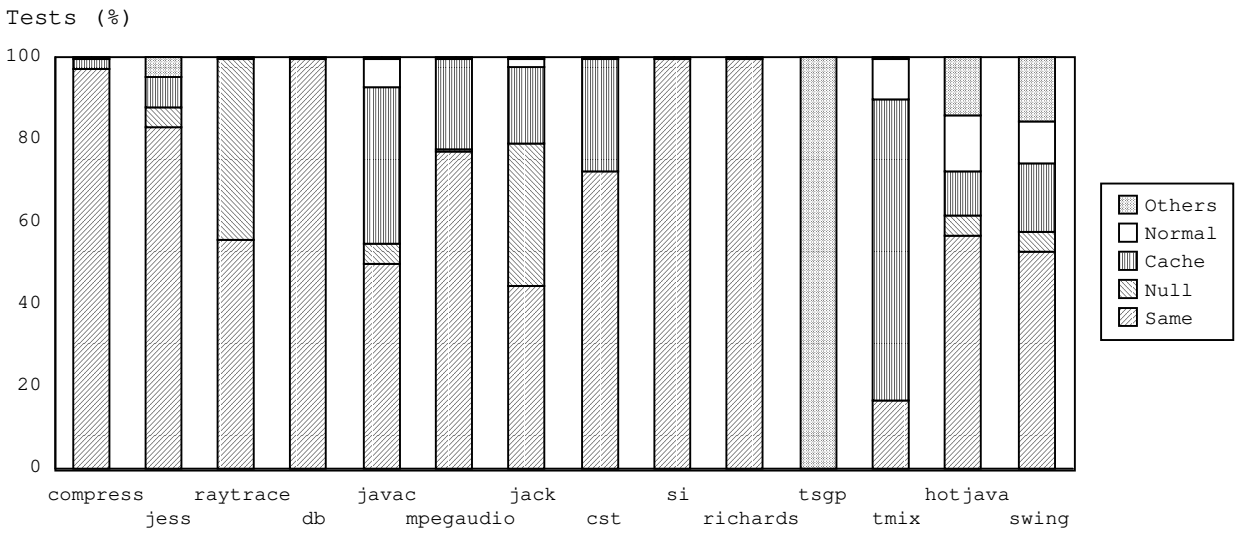


Figure 3: Distribution of Object Types in Type Inclusion Test at Runtime

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 array elements. The striped bar shows the number of accesses to array elements 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 array elements. Scalar replacement of accesses to array elements is particularly effective for **`mpegaudio`**, in which 25% of accesses are eliminated. Access to array elements using an interior pointer is effective for **`db`**, **`tsgp`**, and **`tmix`**. In each case, 14%, 28%, and 45% of the original accesses are used. In all cases, it optimizes accesses to array elements in order to swap array elements in the shell, quick, or bubble sort.

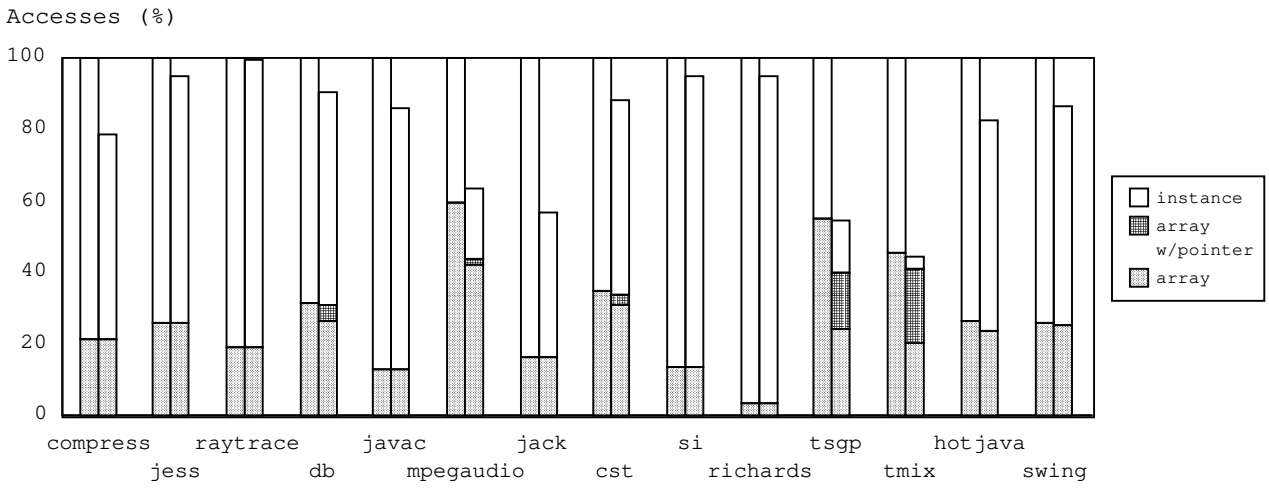


Figure 4: Results of Common Subexpression Elimination at Runtime

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`**, **`raytrace`**, and **`richards`**. An average of 52% 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.

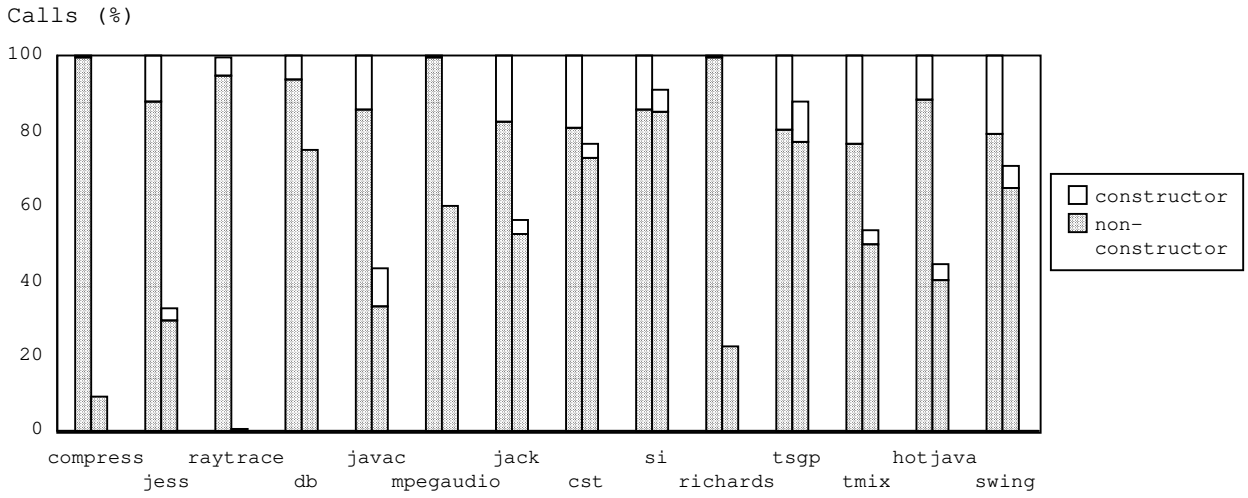


Figure 5: Counts of Static Method Calls at Runtime

6.6 Devirtualization of Dynamic Method Call

The performance in devirtualizing dynamic method call by using direct binding with 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. That is, it is devirtualized. The other three types of bars represent the cases in which dynamic

method calls are not devirtualized by direct binding with CHA. The dark dotted bars represent call sites or calls are not devirtualized by direct binding with CHA, but are monomorphic at runtime. 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 bars (`Revirtualized`) represent cases in which dynamic method call or inlining with method lookup is used. They replace static method call or inlining devirtualized by CHA with dynamic method call at runtime.

Figure 7 shows that direct binding with CHA is highly effective, since it devirtualized an average of 60% for all programs. Furthermore, it devirtualized more than 75% of dynamic method calls or inlinings for five out of fourteen programs. In the worst case '**`compress`**', the non-devirtualized methods do not affect the performance, since many static calls for kernel routines that use *final* classes occur. The optimization for interface call is also effective for **`db`** and **`richards`**, since more than 99% of the interface calls are translated into simple method calls. In **`db`**, the `java.util.Vector` class, which uses the implementation class of the interface class, is used very frequently. In **`richards`**, the class for the benchmark is used frequently. The results also show that all the programs except **`mpegaudio`** are surprisingly monomorphic. Therefore, there is still scope to improve the performance.

Since we have adapted direct binding with CHA to dynamic class loading, a static method call may be replaced with a dynamic method call by overriding the target method when a class is loaded dynamically. For inlining of dynamic method calls, 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`**, and fewer than 20 for the other programs. In our approach, the overhead of replacing the code is very small.

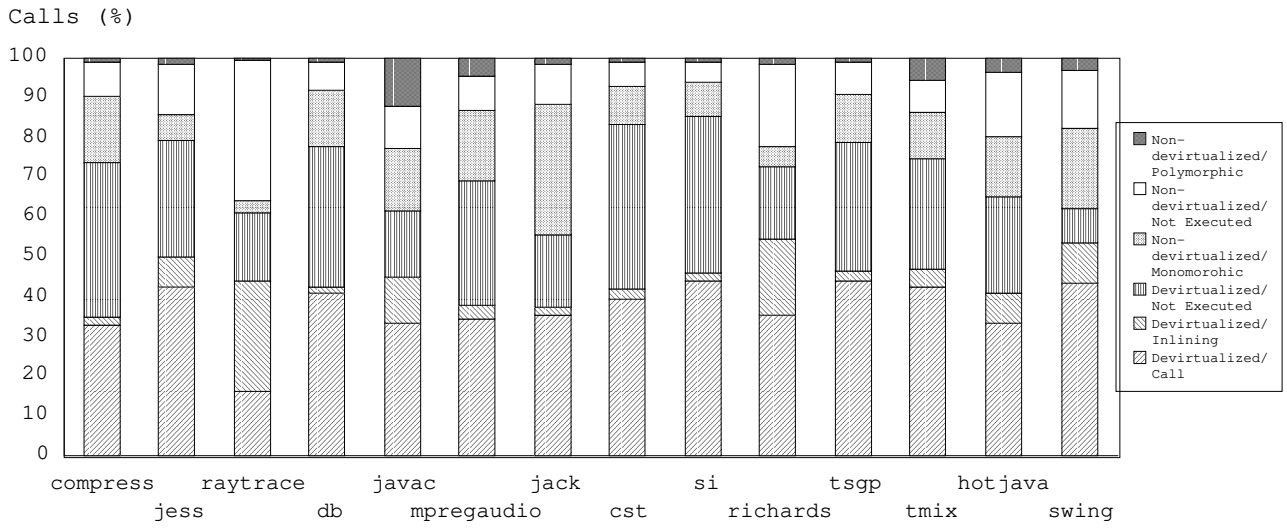


Figure 6: Devirtualization of Dynamic Method Call Sites at Compile Time

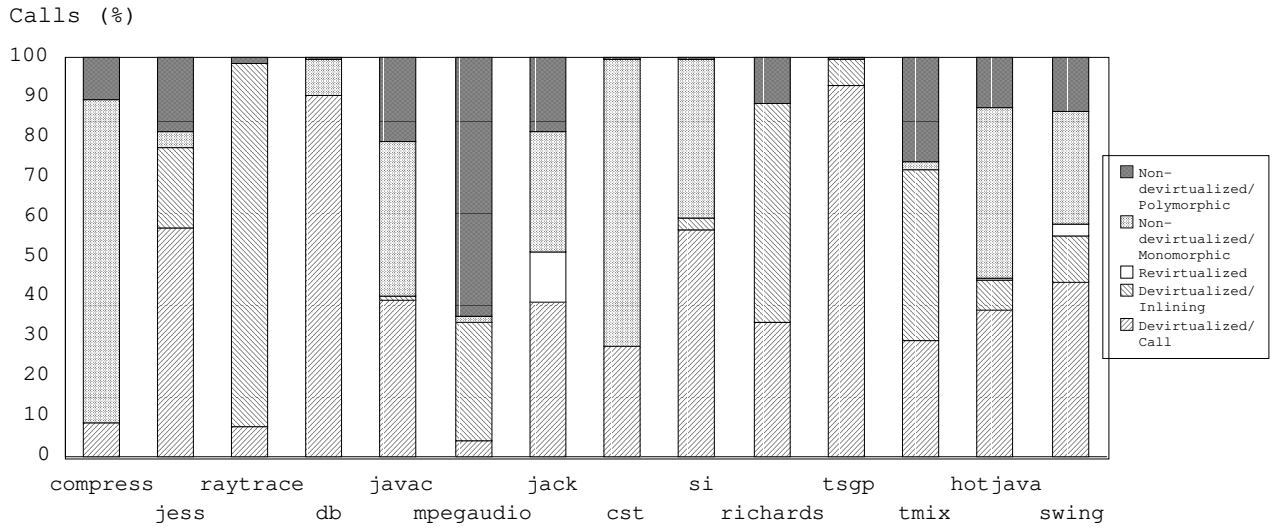
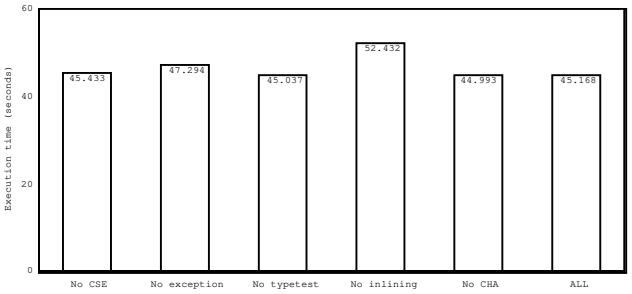


Figure 7: Devirtualization of Dynamic Method Call Sites at Runtime

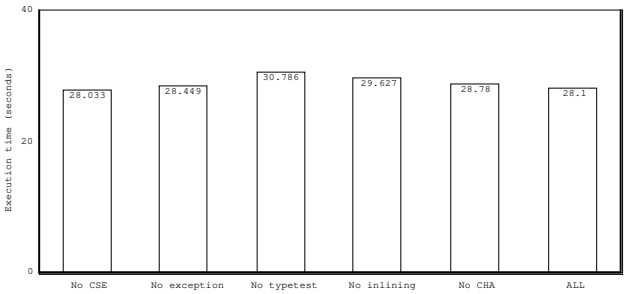
6.7 Performance

We measured the execution time of twelve of the programs; since the other two programs were difficult to measure because of their interactive nature. Figure 8 shows the performance improvements resulting from various optimizations. The white bar represents the best execution time. 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 devirtualization of dynamic method call (`No CHA`). The rightmost bar (`ALL`) shows the time with all optimizations enabled.

In Figure 8 (a), inlining of static method call improves the performance of **`compress`** by 14%. In Figure 8 (b), simple type inclusion test improves the performance of **`jess`** by 8%. In Figure 8 (c), devirtualization of dynamic method call improves the performance of **`raytrace`** by 20%. In Figure 8 (d), simple type inclusion test improves the performance of **`db`** by 17%. In Figure 8 (e), all optimizations make virtually no difference in the performance of **`javac`**. In Figure 8 (f), CSE improves the performance of **`mpegaudio`** by 13%. Inlining also improves the performance by 10%. In Figure 8 (g), both CSE and exception optimization improve the performance of **`jack`** by 10%. Simple type inclusion test also improves by 9%. In Figure 8 (h) and (i), all optimizations make virtually no difference in these benchmarks of **`cst`** and **`si`**. In Figure 8 (j), devirtualization of dynamic method call improves the performance of **`richards`** by 32%. Inlining of static method call also improves the performance by 16%. In Figure 8 (k), CSE improves the performance of **`tsgp`** by 18%. In Figure 8 (l), CSE improves the performance of **`tmix`** by 8%. Figure 8 thus shows that all optimizations contribute to an improvement in the performance.



(a) `compress`



(b) `jess`

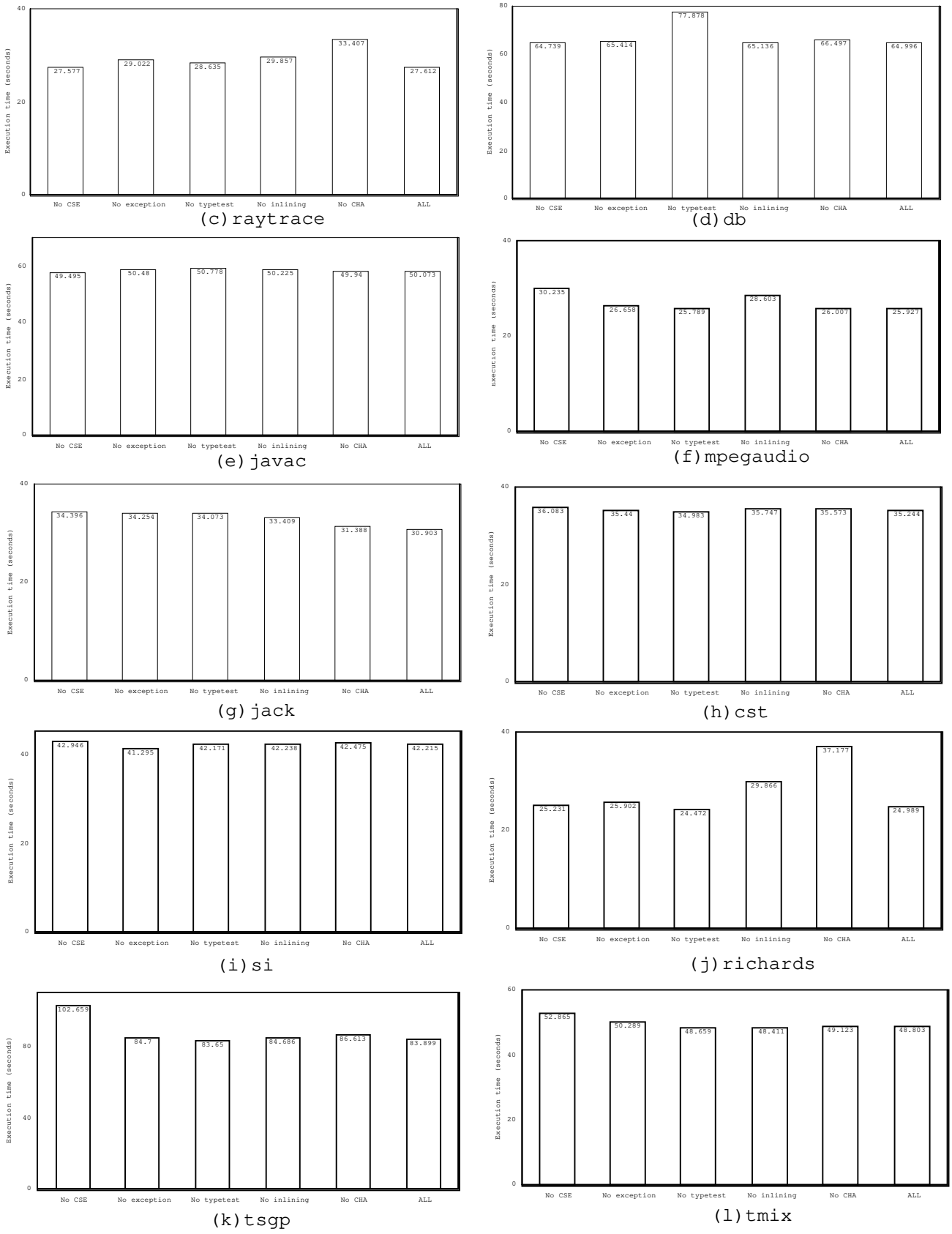


Figure 8: Execution Times of the JIT'ed code

7. Related Work

The Intel JIT compiler [19] 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 results of our experiment show that it is effective. Exception check elimination [5, 20] has been proposed as a 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 [9, 10] has been investigated for efficient type conformance test in an object-oriented language. In previous research, the class hierarchy was encoded in a small table, so that it could be tested in a constant time. The table may be reconstructed later 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 87% of all tests. Therefore, we chose a simple implementation. The Intel JIT compiler inlines the code for traversing the hierarchy up to two levels [21], as in our approach. For **`javac`**, the inlined codes of both approaches account for 92%. Our approach generates a smaller amount of inlined code than Intel's.

Polymorphic inline cache (PIC) [13] 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 [11, 12] and method test [14] have also been proposed, with type analysis for languages supporting dynamic class loading. Type prediction and method test predict the type of an object, which are called frequently, at compile time. PIC, type prediction, and method test introduce a runtime test newly, since they are executed on the basis of the cache mechanism with memory references. According to the results of simple experiments [22], type prediction without inlining at 100% accuracy cannot outperform devirtualization of dynamic method call without inlining. Type prediction with inlining must achieve 90% accuracy to outperform devirtualization without inlining. Finally, nothing can outperform devirtualization with inlining. In implementations of Java, the cost of dynamic method call is not so different from that of PIC, type prediction, and method test.

Direct binding with class hierarchy analysis (CHA) [15, 16] can replace a dynamic method call with a faster static method call at compile time. It has been investigated and implemented for languages that support static class loading. To avoid the runtime test overhead of PIC, type prediction, and method test, we developed a version of direct binding with 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.

The Java HotSpot compiler [23] adopts a recompilation approach that includes on-stack replacement [24]. Preexistence [14] is an approach for reducing the number of on-stack replacement. It requires recompilation of a whole method when a method is overridden. On the other hand, our approach has a lower overhead than other approaches because direct-binded code is undone by rewriting only a single instruction.

8. Conclusions

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

Acknowledgement

We are grateful to the people in our group at Tokyo Research Laboratory for implementing our JIT compiler and for participating in helpful discussions. We also thank Michael McDonald for checking the wording of this paper.

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 1.1.8,” Available at http://www.ibm.com/java/

[4] Frederick Chow and John Hennessy: “The priority-based coloring approach to register allocation,” ACM Transactions on Programming Languages and Systems, vol. 12, no. 4, pp. 501-536 1990.

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Urs Holzle, Craig Chambers, and David Ungar: “Optimizing Dynamically-Typed Object-Oriented Langages with Polymorphic Inline 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.

[14] David Detlefs and Ole Agesen: “Inlining of Virtual Methods,” to appear in ECOOP '99.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Aart Bik, Milind Girker, and Mohammad Haghighat: “JIT Compilation of Java for Intel Architechture,” ACM 1999 Java Grande Conference Tutorial, 1999.

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

[23] Sun Corp.: “The Java HotSpot Performance Engine Architecture,” Available at http://java.sun.com/products/hotspot/whitepaper.html

[24] Urs Holzle, Craig Chambers, and David Ungar: “Debugging optimized code with dynamic deoptimization,” In Proceedings of the ACM SIGPLAN ’92 Conference on Programming Language Design and Implementation, pp. 32-43, 1992.