Supporting Precise Garbage Collection in Java Bytecode-to-C Ahead-of-Time Compiler for Embedded Systems

Dong-Heon Jung Sung-Hwan Bae Jaemok Lee Soo-Mook Moon JongKuk Park

School of Electrical Engineering and Computer Science
Seoul National University, Seoul 151-742, Korea
{clamp,uhehe99,seasoul,jaemok,smoon}@altair.snu.ac.kr

Abstract

A Java bytecode-to-C ahead-of-time compiler (AOTC) can improve the performance of a Java virtual machine (JVM) by translating bytecode into C code, which is then compiled into machine code via an existing C compiler. Although AOTC is effective in embedded Java systems, a bytecode-to-C AOTC could not easily employ *precise* garbage collection (GC) due to a difficulty in making a *GC map*, which keeps information on where each root live object is located when GC occurs. This is one of the reasons why all previous JVMs using a bytecode-to-C AOTC employed *conservative* GC, which can lead to poorer GC performance with more frequent memory shortage, especially in embedded systems where memory is tight.

In this paper, we propose a way of allowing precise GC in a bytecode-to-C AOTC by generating additional C code which collects GC map-equivalent information at runtime. In order to reduce this runtime overhead, we also propose two optimization techniques which remove unnecessary C code. Our experimental results on Sun's CVM indicate that we can support precise GC for bytecode-to-C AOTC with a relatively low overhead.

Categories and Subject Descriptors

D3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Processors]: Compilers, Run-time environments; D.4.7 [Operating Systems]: Organization and Design-real-time and embedded systems;

General Terms

Design, Experimentation, Performance, Languages

Keywords

Bytecode-to-C, ahead-of-time compiler, precise garbage collection, Java virtual machine, J2ME CDC

1. Introduction

Java has been employed popularly as a standard software platform for many embedded systems, including mobile phones, digital TVs, and telematics [19]. The advantage of Java as an embedded platform is three folds. First, the use of a virtual machine allows a consistent runtime environment for a wide range of devices that have different CPUs, OS, and hardware (e.g., displays). Secondly, Java has an advantage in security such that it is extremely difficult for a malicious Java code to break down a whole system. Finally, it is much easier to develop software contents with Java due to its sufficient, mature APIs and its language features that increase software stability such as garbage collection and exception handling.

The advantage of platform independence is achieved by using the Java virtual machine (JVM), a program that executes Java's compiled executable called *bytecode* [1]. The bytecode is a stack-based instruction set which can be executed by *interpretation* on any platform without porting the original source code. Since this software-based execution is obviously much slower than hardware execution, compilation techniques for executing Java programs as native executables have been used, such as *just-in-time compiler* (JITC) [15] and *ahead-of-time compiler* (AOTC) [7]. JITC translates bytecode into machine code at runtime while AOTC translates before runtime. AOTC is more advantageous in embedded systems since it obviates the runtime translation overhead and the memory overhead of JITC, which consume the limited computing power and memory space of embedded systems.

There are two approaches to AOTC. One is translating bytecode directly into machine code, referred to as *bytecode-to-native* [2-6]. The other approach is translating bytecode into C code first, which is then compiled into machine code by an existing compiler, referred to as *bytecode-to-C* (b-to-C) [7-10]. The approach of b-to-C allows faster development of a stable, powerful AOTC by resorting to full optimizations of an existing compiler and by using its debugging and profiling tools. It also allows a portable AOTC that can work on different CPUs.

There is one important issue in designing a b-to-C AOTC related to garbage collection (GC) [11], though. A JVM that supports *precise* GC requires a *GC map* at each program point where GC can possibly occur, which is a data structure describing where each root live object is located at that point. Since a b-to-C AOTC normally translates such locations that have root references into C variables, it is difficult to know where the C compiler places those C variables in the final machine code. All previous JVMs using a b-to-C AOTC employed *conservative* GC [7, 9, 10], which is simpler to implement but can lead to poorer GC performance than precise GC.

This paper proposes a method to support precise GC in a b-to-C AOTC by generating C code that records references in a stack

This research was supported in part by Samsung Electronics.

CASES'06, October 23–25, 2006, Seoul, Korea.

frame whenever a reference-type C variable is updated. In order to reduce the runtime overhead caused by the additional C code, we also propose two optimization techniques. We evaluated the proposed technique on a CVM in Sun's J2ME CDC environment which requires precise GC. We developed a b-to-C AOTC in this environment and performed experiments with it.

The rest of this paper is organized as follows. Section 2 reviews precise GC and our b-to-C AOTC. Section 3 introduces the proposed solution to allow precise GC with a b-to-C AOTC. Section 4 describes two optimization techniques to reduce the overhead of our solution. Section 5 shows our experimental results. A summary follows in Section 6.

2. Precise GC and Bytecode-to-C AOTC

In this section, we briefly review our b-to-C AOTC and issues in precise GC. Then, we discuss why it is difficult to support precise GC with a b-to-C AOTC.

2.1 Overview of JVM and Our Bytecode-to-C AOTC

The Java VM is a typed stack machine [14]. Each thread of execution has its own Java stack where a new activation record is pushed when a method is invoked and is popped when it returns. An activation record includes state information, *local variables* and the *operand stack*. Method parameters are also local variables which are initialized to the actual parameters by the JVM. All computations are performed on the operand stack and temporary results are saved in local variables, so there are many pushes and pops between the local variables and the operand stack.

In our b-to-C AOTC, each local variable is translated into a C variable (which we call a *local C variable*). Also, each stack slot is also translated into a C variable (which we call a *stack C variable*). Since the same stack slot can be pushed with differently-typed values during execution, a type name is attached into a stack C variable name such that a stack slot can be translated into multiple C variables. For example, `s0_ref` is a C variable corresponding to a reference-type stack slot 0, while `s0_int` is a C variable corresponding to an integer-type stack slot 0.

Our AOTC first analyzes the bytecode and decides the C variables that need to be declared. It then translates each bytecode one-by-one into corresponding C statements, with the status of the operand stack being kept track of. For example, `aload_1` which pushes a reference-type local variable 1 onto the stack is translated into a C statement `s0_ref = l1_ref;` if the current stack pointer points to the zero_th slot when this bytecode is translated. Figure 0 shows an example.

Our AOTC can work with the interpreter, so AOTC methods and interpreter methods can co-exist during execution. This is useful for an environment where we also need to download class files dynamically (e.g., in digital TVs the Java middleware is AOTCed while the *xlets* downloaded thru the cable line is executed by the interpreter). How to pass parameters between AOTC methods and interpreter methods and the translation details are described in [20].

(a) **Java Method**

```
public int max(int a,
int b) {
  return (a>=b)? a:b;
}
```

(b) **Bytecode**

```
0: iload_0
1: iload_1
2: if_icmplt 9
5: iload_0
6: goto 10
9: iload_1
10:ireturn
```

(c) **Translated C Code**

```
Int      Java_java_lang_Math_max_II(JNIEnv
*env, int l0_int, int l1_int)
{
  int s0_int;
  int s1_int;

  s0_int = l0_int;              // 0:
  s1_int = l1_int;              // 1:
  if(s0_int < s1_int) goto L9;  // 2:
  s0_int = l0_int;              // 5:
  goto L10;                     // 6:
L9: s0_int = l1_int;            // 9:
L10: return s0_int;             // 10:
}
```

Figure 0. An example of Java code and translated C code

2.2 Precise GC

As an object-oriented language, Java applications tend to allocate objects at a high rate while the memory space of embedded systems is tight, so GC should reclaim garbage objects efficiently. Most GC techniques first trace all *reachable* objects on a directed graph formed by program variables and heap-allocated objects where program variables constitute the *root set* of the graph [11]. In Java, the roots are located in the operand stack slots and local variables of all methods in the call stack, whose types are object references. So, GC traces all reachable objects starting from the root set and reclaims all unreachable objects. There are two approaches to tracing reachable objects: *conservative* and *precise* [12].

Conservative GC regards all word-size data items saved in the JVM as references, which simplifies the identification of the root set and makes the GC module easily portable [13]. However, non-reference numeric values can be misunderstood as references, which might cause incomplete reclamation of garbage objects.

Precise GC can reclaim all garbage objects completely by computing the root set precisely with a help from the JVM or from the compiler. The GC module is thus more complicated to implement, yet there is no misunderstanding of numeric values as references [12].

The biggest advantage of precise GC is that it allows using a GC algorithm that can move objects during GC, such as copying GC [17], mark-and-compact GC [18], and generational GC [12]. These moving GC algorithms do not cause fragmentation. Also, they are known to be faster than non-moving GC algorithms and have a better locality of memory accesses [11].

Moving GC algorithms cannot be used with conservative GC since it is not safe to update potentially misunderstood references after moving the referenced object. In fact, non-moving GC can be a serious defect in embedded systems since it can easily cause memory shortage due to fragmentation. For example, the first version of Sun's CLDC reference implementation which did not

employ a compacting GC has often suffered from memory shortage. Since a compacting GC was employed from the CLDC 1.03 version, however, memory shortage rarely occurred.

Sun's CVM is designed to support precise GC only. CVM introduces the idea of a *GC-point* for precise GC, which means a point in the program where GC can possibly occur. Examples of GC-points are memory allocation requests, method calls, method entries, loop backedges, or synchronization points. The CVM can start GC only when every thread waits at one of its GC points since otherwise GC cannot find all reachable objects precisely [12]. So, when a thread's memory allocation request at a GC-point cannot be satisfied, it will request GC and wait at the GC-point. Other threads are supposed to check if there is any pending GC request whenever they pass through a GC-point, so they will also eventually wait at their GC-points. Then, the CVM starts GC by first computing the root set from the call stack of methods and class static fields, followed by tracing all reachable objects.

2.3 Precise GC with a Bytecode-to-C

As explained, we need to compute the root set for each thread when all threads are stopped at their GC-points. The root set includes the roots of all methods in the call stack. In order to compute the root set, GC needs a data structure describing the location of each root at the GC-point, which is called a *GC-map* in CVM. When GC occurs if only the interpreter is being used, the interpreter first analyzes the bytecode for each method in the call stack to compute the GC-map at every GC-point in the method. It then saves the GC-map at the method block so that it can be reused if GC occurs again in the future. The GC-map is actually a bit map describing which local variables and stack slots are references at that point (1 means a reference and 0 means a non-reference). GC consults with this bit map to decide which stack slots and local variables in the activation record are reference-typed, thus including root references [14].

If the AOTC translates the bytecode into machine code, these stack slots and local variables will eventually be translated into memory locations or registers. So, this time the compiler must prepare for the GC-map in order to help GC compute the root set. The bytecode-to-native AOTC can compute the GC-map easily since the AOTC itself allocates variables to registers or spills to memory, so at each GC point it knows where root references are located. In the C program translated by the b-to-C AOTC, however, roots are in local C variables and stack C variables whose types are object references. It is not easy for the AOTC to figure out where the C compiler will allocate these variables in the machine code, so it cannot prepare for the GC-map.

3. Supporting Precise GC for Bytecode-to-C AOTC

Unlike the precise GC environment where the interpreter or the bytecode-to-native AOTC can prepare for the GC-map, the b-to-C AOTC cannot provide the GC-map by itself. This section describes our proposed solution to cope with this lack of the GC-map.

3.1 Saving Reference Variables at the Java Stack

The simplest way to support precise GC without a GC-map would be using JNI (Java native interface) [1]. JNI provides an interface to declare references in the native code so that those referenced objects in the native code are not deallocated during GC. We can simply declare all reference variables in the translated C code using the JNI interface, which can achieve the effect of precise GC. However, this approach is not a viable solution because it is too slow due to indirect accesses of all referenced objects in JNI.

Our idea is having the AOTC generate additional C code that saves the values of reference variables in some area which the GC module can access, so that GC can easily get all roots within the translated methods. There are a couple of issues with this idea.

First, we need to decide where to save the reference values. Since GC is supposed to look for root references from the Java stack in the interpreter mode and our AOTC methods run concurrently with interpreted methods, saving references at the Java stack would be a natural choice. In fact, Java stack frame is needed anyway in CVM even for AOTC methods for other purposes, so we just need to allocate additional space for saving references.

Secondly, we need to decide which reference variables need to be saved at which locations. All live reference variables at each GC-point must have been saved at the Java stack before reaching the GC-point. One solution is saving all reference-type variables declared in the method just before entering each GC-point. This is obviously involved with too much overhead since only a subset of those variables will be live at a GC-point. Our solution is saving a reference variable at the stack frame *only when* it is updated. For each GC-point, this will ensure saving only those references that were newly created on the execution path from the method entry to the GC-point (when combined with our liveness analysis in Section 4.2, it will minimize useless saves).

In order to implement this solution, the AOTC generates the following additional C code.

- At the method entry, a Java stack frame must be allocated by calling pushJavaStackFrame() which is a macro for reserving a frame space (expands it if not enough space is available)
- Whenever a C statement that updates a reference-type variable is generated, another C statement that saves the variable into a stack frame is also generated.
- At a GC-point, a GC check code for a pending GC request is generated which stops the thread and waits if there is any.
- At the method exit, the Java stack frame is deallocated by calling popJavaStackFrame(), which is also a macro for releasing the stack frame space.

Figure 1 depicts the C code generated by the AOTC, where a statement that saves `s0_ref` at the stack frame is added at (3) when it is updated at (2). Macro calls to allocate and deallocate the stack frame are generated at (1) and (5), respectively. At the GC-point, GC-check code is generated as in (4) (more explanation on `s0_ref = frame[0];` will follow shortly).

For each reference variable in a method, there is a reserved slot in the stack frame, so when there is an update for a reference variable, it is saved at its reserved slot. The number of slots allocated at the beginning of a method thus should be the same as the number of reference variables in it. At each GC-point, the AOTC prepared for a GC map-equivalent information on which refernce C variables, hence which frame slots, are live at that point, so that GC regard only them as roots.

When GC occurs, all stack frames in the call stack are scanned by GC to compute the root set.

Additional C code added for a method	
	// Stack frame is allocated at the beginning of a method
(1)	`frame = pushJavaStackFrame(1);`
	….
	// There is an update for a reference variable s0_ref
(2)	`s0_ref = s0_ref->myField;`
	// Save the reference at the stack frame
(3)	`frame[0] = s0_ref;`
	…
	// At a GC-point

(4)	`if (pending_request) {`
	`  stop_the_thread_and_wait_GC();`
	`  s0_ref = frame[0];` // Copying back if moving GC is used
	`}`

	// Stack frame is deallocated at the end of a method
(5)	`popJavaStackFrame();`

Figure 1. An example of additional C code to support precise GC.

3.2 Dealing with Moving GCs

A moving GC algorithm can be employed with precise GC such that objects can be moved during GC. As objects are moved, GC updates their references (addresses) including root references. This requirement causes an issue in our proposed precise GC. Since all root references are copied from the reference C variables into the Java stack frames before GC occurs, if GC moves root objects, GC will update references in the stack frames, not the reference C variables. This means that reference variables may no longer have valid references after GC. Figure 2 depicts this scenario where after GC `so_ref` is not pointing the object any more.

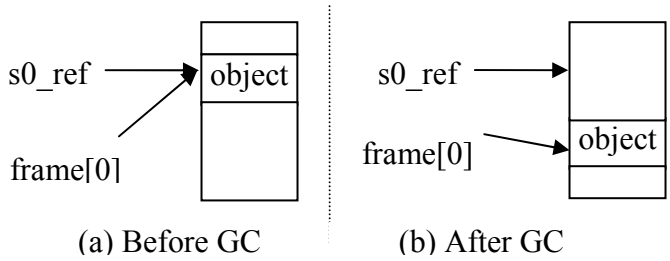

Figure 2. A problem when moving GC is employed

In order to handle this problem, updated references existing at Java stack frames should be copied back to the corresponding reference variables after GC. This can be done by adding C statement at the GC-point, and Figure 1 shows the copying C statement `s0_ref = frame[0]` in the GC-check code in (4). One caution is that we need to declare the frame[] as a volatile array; otherwise, the compiler might delete some added statements with optimizations.

4. Optimizations to Reduce the Runtime Overhead

Although we can support precise GC with a b-to-C AOTC by adding C code, the runtime overhead caused by the additional C code can be significant. There are three kinds of runtime overhead. The first one is saving the value of a reference variable into the Java stack frame whenever it is updated. The second one is copying the reference value from the stack frame into a reference variable when a moving GC algorithm is employed. The last overhead is allocating and deallocating the Java stack frame at the beginning and at the end of a method, respectively. In this section, we propose optimization techniques to reduce these overheads.

4.1 Copy Propagation

Since the JVM supports object-oriented computation on the operand stack, there are many pushes and pops of object references between local variables and the operand stack in the bytecode. In our b-to-C AOTC, these pushes and pops are translated into copy statements between the local C variables and the stack C variables. For example, a Java statement, `Object dest = src.f`, accesses a reference-type field `f` of an object referenced by a variable `src` and saves it to a variable `dest`. The bytecode for this statement is composed of `aload_1` (which pushes the local variable `src`), `agetfield f` (which pushes the field `f`), and `astore_2` (which pops and saves at the local variable `dest`). Finally, this bytecode sequence is translated into the following C statement sequence: `s0_ref =l1_ref; s0_ref =s0_ref->f; l2_ref= s0_ref;` as shown in Figure 3.

(a) Java source code	(b) Bytecode

`Object dest = src.f`	`aload_1`
	`agetfield f`
	`astore_2`

(c) Translated C code	

`s0_ref = l1_ref;`
`s0_ref = s0_ref->f;`
`l2_ref = s0_ref;.`	

Figure 3. An example Java source code, bytecode, and translated C code.

We are supposed to save a reference into the Java stack frame whenever a reference-type C variable is updated, as shown in Figure 4 (a). However, many of those updates are, in fact, copying of (already-saved) references, thus being useless. For example, the save of `l2_ref` `frame[1] = l2_ref` after the statement `l2_ref = s0_ref` is useless since the value of `s0_ref` have already saved in the stack frame in the previous statement `frame[0] = s0_ref`.

In order to reduce this useless reference saves, we need to remove as many copies as possible so as to keep only essential computations and copies. In fact, the optimizing compiler that translates the C code into machine code will remove many of these copies. Unfortunately, the AOTC does not know which copies will be removed and which will remain by the compiler, thus unable to add the reference-saving C statements selectively. So, we need to add C statements everywhere, as shown in Figure 4 (a). The final machine code does not have any copies as in Figure 4 (b), yet all the frame-saving store instructions remain since the compiler cannot easily optimize and remove them (also frame[] array is volatile).

(a) C code with reference saves	(b) Compiled machine code
`s0_ref = l1_ref;`	`// copy was deleted`
`frame[0] = s0_ref;`	**`sw $t1, $frame+0`**
`s0_ref = s0_ref->f;`	`lw $t2, $t1+offset_of_f`
`frame[0] = s0_ref;`	**`sw $t2, $frame+0`**
`l2_ref = s0_ref;`	`// copy was deleted`
`frame[1] = l2_ref;`	**`sw $t2, $frame+4`**

Figure 4. How the C code added with reference saves is compiled

Our idea is performing copy propagation by the AOTC in order to remove useless copies in the translated C code, thus removing useless reference saves. We use a conventional copy propagation algorithm based on simple data flow analysis [9]. If we do copy propagation for our example, there will be a single C statement `l2_ref = l1_ref->f` with a single reference save statement as shown in Figure 5, thus obviating useless savings.

(a) Copy-propagated C code

```
l2_ref = l1_ref->f;
frame[0] = l2_ref;
```

(b) Compiled machine code

```
lw $t2, $t1+offset_of_f
sw $t2, $frame+0
```

Figure 5. Optimization based on copy propagation

4.2 Removal of Reference Saves via Liveness Analysis

In addition to removing unnecessary reference copies, we can also reduce the reference saves via liveness analysis [16]. Although we save references whenever a reference variable is updated, a GC-point is supposed to keep references live *only* at that point. So, if an updated reference variable is not live at any GC-points, we do not have to save it. In fact, a stack variable generally has a short live range due to Java's stack machine model where the stack keeps a value temporarily, so it would be rare for a stack reference variable to be live across a GC-point.

We perform live variable analysis for this optimization such that if a reference variable defined at some location is not live at any GC-points, it is not saved at the stack frame there. Figure 6 shows an example where `s1_ref` does not need to be saved since it is not live at the GC-point.

(a) C code before optimization

```
s0_ref = l1_ref;
frame[0] = s0_ref;
s1_ref = l2_ref;
frame[1] = s1_ref;
…
[GC-point]
// s0_ref is live,
s1_ref is dead here
```

(b) C code after optimization

```
s0_ref = l1_ref;
frame[0] = s0_ref;
s1_ref = l2_ref;
// no need to save
s1_ref
…
 [GC-point]
// s0_ref is live,
s1_ref is dead here
```

Figure 6. Optimization based on liveness analysis

4.3 Removal of Stack Frame Allocation

If all reference saves in a method are completely eliminated by the two optimizations, the allocation of stack frame itself can be obviated. Since the stack frame allocation includes a significant amount of work in order to follow the CVM's stack allocation convention, it is very important to remove the overhead. In order to promote this optimization opportunity, when a method call includes a reference-type variable as a parameter, we pass a reference saved at the stack frame, instead of the variable itself. In Figure 7, for example, we pass `&frame[0]` as an argument instead of passing `s0_ref` in the caller. When `l0_ref` is initialized to the argument `*ptr` in the callee `foo(*ptr)`, we do not have to save `l0_ref` at the callee's stack frame since the `*ptr` is already at the caller's stack frame. Restoration of `l0_ref` at the GC-point after GC can also be made from the argument `*ptr` since it (`frame[0]` in the caller) should have already been updated during GC. If `l0_ref` is the only reference updated in foo(), we do not even have to allocate frame for foo() since there is no reference save, which will remove the allocation overhead (the actual code is somewhat different from Figure 7, which is just for illustration of the idea).

(a) Caller code

```
s0_ref = …
frame[0] = s0_ref
…..
foo       (&frame[0]);//insteadf       of
foo(so_ref)
if (GC occurred in foo()) {
  s0_ref = frame[0];
}
```

(b) Callee code

```
foo(*ptr)
 l0_ref= *ptr; // No need to save
l0_ref
 …
 // At a GC-point
 if (pending_request) {
  stop_the_thread_and_wait_GC();
  l0_ref = *ptr; // copy back after GC
 }
```

Figure 7. Optimized argument passing

5. Experimental Results

Previous sections described our proposed solution to allow precise GC with a bytecode-to-C AOTC by adding C code that saves live references in the stack frame. They also proposed optimization techniques to reduce the overhead of additional C code via copy propagation and liveness analysis on the translated C code. In this section, we evaluate our proposed solution and the optimization techniques.

5.1 Experimental Environment

We experimented with Sun's CVM for which we implemented a bytecode-to-C AOTC. The CVM employs generational GC which is a moving GC algorithm, so we need to add reference restoration code in Section 3.2. Since CVM supports precise GC only, we could not compare with conservative GC (comparing both GCs is beyond the scope of this paper).

The experiments were performed on an Intel Pentium4 2.40 GHz CPU with 512M RAM and the OS is Debian Linux with kernel 2.6.8-2. The translated C code is compiled by GNU C compiler (GCC) version 3.3.5. The CVM is constrained to have 32M memory. The benchmarks we used are SPECjvm98 (except for mpegaudio for which CVM cannot read its class files).

5.2 Performance Impact of Optimization

We first evaluate the effectiveness of the two optimization techniques. Tables 1 shows the number of dynamic reference saves at the stack frame for four cases with: no optimizations, copy propagation only, liveness analysis only, and both optimizations. It shows that the number of reference saves decreases significantly with either optimization. With both optimizations the number of reference saves decreases by an average of 88%.

We also examined the whole Java stack frame at each GC-check point during execution if there are any duplicate references.

Duplicate references would mean that there are more than one save of the same references, probably via copies. We found there are few instances of duplicate references (somewhat more in `javac` and `mtrt` which are unavoidable due to inherent duplication of `this` pointer using dup), meaning that reference saves were made efficiently.

As the reference saves in a method are completely removed by these optimizations, the allocation of stack frame can also be removed for that method. Table 2 shows the dynamic number of stack frame allocation requests for each benchmark for the four cases. The table shows that the number of stack frame allocation can be reduced by an average of 55% with both optimizations, although the reduction rate varies significantly from benchmark to benchmark.

Table 1. Number of dynamic reference saves at the Java stack frame (millions)

benchmarks	No optimization	Copy Propagation	Liveness Analysis	Both Optimizations
compress	4,073	1,469	238	217
jess	703	228	201	93
db	817	413	200	133
javac	650	160	182	79
mtrt	799	236	279	55
jack	283	78	98	45

Table 2. Number of dynamic Java stack frame allocation requests (millions)

benchmarks	No optimization	Copy Propagation	Liveness Analysis	Both Optimizations
compress	226	226	20	20
jess	104	83	58	54
db	72	65	61	58
javac	84	65	63	56
mtrt	278	102	34	12
jack	41	28	29	24

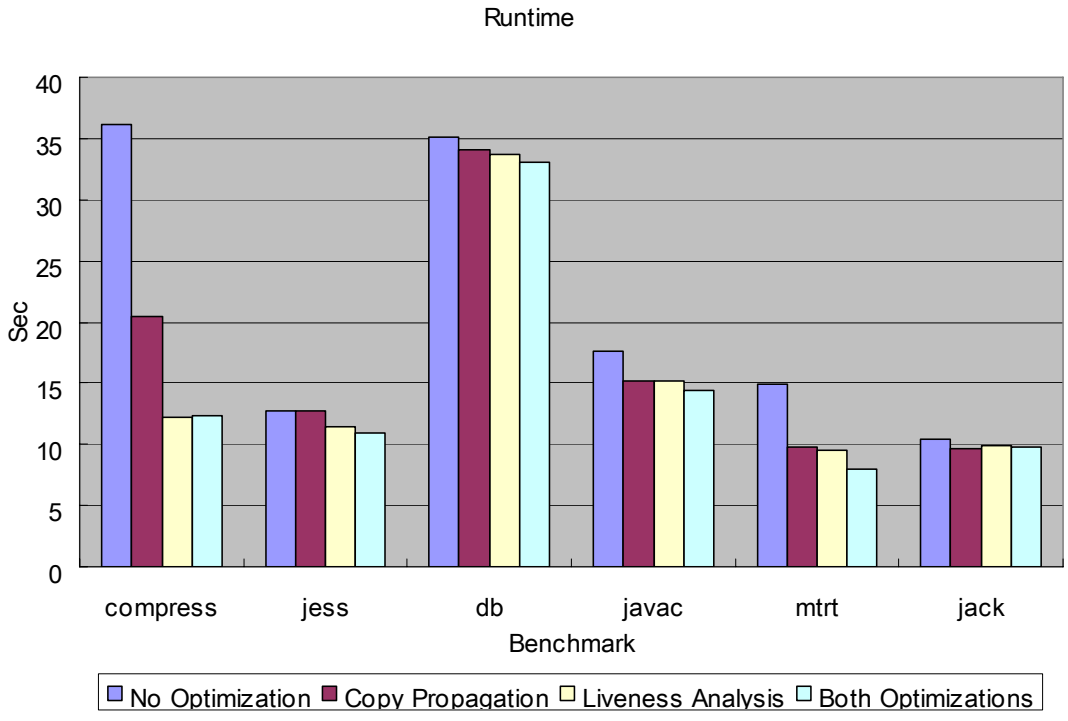

Figure 8. Running time of benchmarks for each case.

Figure 8 shows the total running time (in seconds) of each benchmark for the four cases. It shows that the performance improves with optimizations turned on, more significantly for `compress`, `javac`, and `mtrt`, which is consistent with the reduction of the number of reference saves and the number of frame allocation, depicted in Table 1 and Table 2, respectively.

We also estimated for the optimized code with both optimizations how much of its running time is spent for reference saves and stack frame allocation. For this analysis, we ran an experiment with both optimizations turned on, but with no removal of stack frame allocation. The difference of running time between this experiment and the original experiment in Figure 8 divided by the difference of their numbers of stack frame allocations gives the unit time spent for a single stack allocation. If this unit time is multiplied by the number of stack allocations left in the optimized in Table 2, it will give an estimate for stack allocation overhead within the total running time in Figure 8. Similarly, we can also estimate the overhead of reference saves.

Table 3 shows the percentage of those two overheads against the total running time. It shows that stack allocation overhead is much more serious than reference saves (except for `compress` where there is a very small number of hot spot methods unlike others). This is no wonder since a single stack frame allocation requires more than 20 x86 instructions for the most probable case while a reference save is a single store instruction. The combined overheads constitute from 4.3% to 21.4% of the running time with an average of 13.2%, which is still not trivial but would be something that we can pay if we can avoid fatal memory shortage by using precise GC.

Table 3. Percentage of overheads after optimization against total running time.

benchmarks	saving references	stack allocation
compress	7.1%	5.7%
jess	0.8%	17.0%
db	0.4%	12.7%
javac	0.9%	20.5%
mtrt	1.5%	2.8%
jack	0.7%	9.1%

We need to reduce these overheads further. As to the Java stack frame allocation requests, we found that many of them request just one or two frames as shown in Table 4. So, additional removal of reference saves are likely to reduce the overhead of stack frame allocation. Also, the frame allocation task itself should be more light-weighted considering the frequent requests of small number of frames. We are currently working on these issues.

Table 4. Percentage of one or two frame allocation requests

benchmarks	saving references	stack allocation
compress	0.0%	50.0%
jess	5.6%	33.9%
db	5.4%	0.1%
javac	15.3%	47.9%
mtrt	20.0%	7.3%
jack	20.3%	32.6%

6. Summary

A bytecode-to-C AOTC is one of the most promising approaches to embedded Java acceleration and embedded Java requires precise GC for the best utilization of limited memory. We proposed a solution to allow precise GC with a bytecode-to-C AOTC and two code optimization techniques to reduce the overheads of reference saves and stack allocation. Although the overhead after optimization is still not trivial, we could say that it can be compensated by complete reclamation of garbage objects as well as by using moving GCs, which might avoid fatal memory shortage in embedded systems.

We also found by analysis that the overhead of stack frame allocation is much more serious than that of reference saves. We are working on reducing this overhead.

References

[1] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification Reading*: Addison-Wesley, 1996.

[2] Instantiations Inc. , "Jove: super optimizing deployment environment for Java."

[3] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, "Marmot: An Optimizing Compiler for Java," *Software Practice and Experience*, vol. 30, pp. 199-232, 2000.

[4] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta, "Quicksilver: a quasi-static compiler for Java," *ACM SIGPLAN Notices*, vol. 35, pp. 66-82, 2000.

[5] D. J. Scales, K. H. Randall, S. Ghemawat, and J. Dean, "The Swift Java Compiler: Design and Implementation," *WRL Research Report 2000/2*, 2000.

[6] V. Mikheev, A. Yeryomin, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, and D. Leskov, "Overview of excelsior JET, a high performance alternative to java virtual machines," presented at Proceedings of the 3rd international workshop on Software and performance, 2002.

[7] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: Java for Applications A Way Ahead of Time (WAT) Compiler," presented at Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon, 1997.

[8] M. Weiss, X. Spengler, F. d. Ferrière, B. Delsart, C. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, and F. Siebert, "TurboJ, a Java Bytecode-to-Native Compiler," presented at Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, 1998.

[9] G. Muller and U. P. Schultz, "Harissa: A Hybrid Approach to Java Execution," *IEEE Software*, vol. 16, pp. 44-51, 1999.

[10] A. Varma and S. S. Bhattacharyya, "Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems " presented at Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Designers (DATE), 2004.

[11] R. Jones and R. Lins, *Garbage Collection Algorithms for Automatic Dynamic Memory Management*: JOHN WILEY & SONS, 1996.

[12] Sun Microsystems, "CDC Foundation Porting Guide."

[13] H. -J. Boehm and M. Weiser, "Garbage Collection in an Uncooperative Environment," *Software Practice and Experience* vol. 18, pp. 807-820, 1988.

[14] F. Yellin and T. Lindholm, *The Java Virtual Machine Specification*, 2nd ed: Addison Wesley, 1999.

[15] J. Aycock. "A Brief History of Just-in-Time", ACM Computing Surveys, 35(2), Jun 2003

[16] Aho, A., R Sethi, and J. Ullman. “Compilers-Principles, Techniques, and Tools”. Addison-Wesley, Reading, Mass., 1985

[17] R. Fenichel and J. Yochelson. “A lisp garbage-collector for virtual-memory computer systems”. Communications of the ACM, 12(11):611.612, November 1969.

[18] COHEN, J., AND NICOLAU, A. Comparison of compacting algorithms for garbage collection. ACM Transactions on Programming Languages and Systems 5, 4 (1983), 532.553.

[19] Sun Microsystems, White Paper “CDC: An Application Framework for Personal Mobile Devices”

[20] S. Bae. “Design and implementation of an ahead-of-time compiler for embedded systems”. MS Thesis, Seoul National University, Feb. 2005