Just-in-Time Compiler Assisted Object Reclamation and Space Reuse*

Yu Zhang[1,2], Lina Yuan[1], Tingpeng Wu[1], Wen Peng[1,2], and Quanlong Li[1,2]

[1] School of Computer Science and Technology,
University of Science and Technology of China, Hefei 230027, P.R. China
[2] Software Security Laboratory, Suzhou Institute for Advanced Study,
University of Science and Technology of China, Suzhou 215123, P.R. China
yuzhang@ustc.edu.cn

Abstract. Garbage collection consumes significant overhead to reclaim memory used by dead (*i.e.,* unreachable) objects in applications. This paper explores techniques for compiler assisted object reclamation and allocation on an actual JVM. Thereinto, the just-in-time compiler identifies dead objects using pointer and escape analysis combining liveness information and inserts calls to free them. The garbage collector provides runtime support for explicit reclamation and space reuse. Our approach differs from other compiler assisted GC in two crucial ways. First, it identifies not only the objects that are no longer referenced directly by the program, but also the objects that are referenced only by those identified to-be-freed objects. Second, it modifies a parallel garbage collector, and not only frees the identified dead objects, but also tries to reuse their space immediately. The experimental results show that the JIT-assisted GC improves the memory utility and the performance efficiently.

Keywords: Compiler assisted garbage collection, pointer and escape analysis, live variable information, parallel garbage collector.

1 Introduction

Garbage collection (GC) [1] is a technology that frees programmers from the error-prone task of explicit memory management. However, it consumes significant overhead to find dead (*i.e.,* unreachable) objects in the managed heap and to reclaim the memory used by them. Accordingly, GC has become one of the dominant factors influencing performance of the runtime systems such as Java virtual machine (JVM). For example, SPECjbb2005 [2] usually spends 10% of its total execution time in GC.

In order to reduce the cost of GC, other than improving GC algorithms [1, 3, 4], a more effective approach is compiler assisted memory management, including stack allocation [5–7], region allocation [8–11], compile-time free [12–15] and reuse [15–17].

* This research was supported in part by the National Natural Science Foundation of China under Grant No. 90718026 and gifts from Intel Corporation.

Stack or region allocation reduce the load of GC through allocating some objects in a method stack frame or a region, and all objects in a stack frame or in a region should be reclaimed simultaneously even if some of them became dead before. However, stack allocation may induce stack overflow, while region allocation needs sophisticated region management, neither has delivered improvements on garbage collectors.

Compile-time free and reuse belong to compiler assisted GC, they improve the collection or reuse of objects allocated in heap through compiler efforts. Some works insert free instructions to free dead objects [12–15], thus reduce GC the load of identifying dead objects. Others automate compile-time object merging or reuse [15–17] to decrease the number of objects allocated in heap.

We explore techniques on compiler assisted object reclamation and space reuse on an actual JVM, *i.e.,* Apache Harmony DRLVM [18], and implement them as a system called just-in-time compiler assisted garbage collection (JIT-assisted GC). The novel contributions we made are as follows:

– We design a novel object lifetime analysis algorithm which is field-sensitive and context-sensitive. The analysis combines pointer and escape analysis with flow-sensitive liveness information to identify not only the objects that are no longer referenced directly by the program, but also the objects that are referenced only by those identified objects.
– We collect the free instrument information from the dead object information based on the dominance relationship in control flow. Various strategies are used to ensure the validity and flexibility of the instrumentation.
– We modify GCv5 [19], a parallel garbage collector, not only adding *gc_free* interface for explicit reclamation but also improving *gc_alloc* to try to reuse the explicitly reclaimed space immediately.

The JIT-assisted GC system can handle multi-threaded programs. The experimental results show that the memory utility and the performance of the whole runtime system are improved efficiently.

2 Overview of the JIT-Assisted GC

In this section we first give an overview of the framework of JIT-assisted GC, then take a simple example to illustrate the compiler analysis and transformation for explicit object deallocation.

2.1 The Framework of the JIT-Assisted GC

The JIT-assisted GC is built on DRLVM, involving several components of DRLVM, such as VMCore, EM(Execution Manager), Jitrino.OPT (a JIT optimizing compiler), and GCv5 garbage collector, etc.

VMCore concentrates most of the JVM control functions. EM selects a compiler or an interpreter for compiling/executing a method, handles profiles and the dynamic recompilation logic. Jitrino.OPT features two types of code intermediate representation (IR): platform-independent high-level IR (HIR) and platform-dependent low-level IR (LIR). Both of them are graph-based structures denoting the control flow of a program. Jitrino incorporates an extensive set of code optimizations for each IR type, and defines the compilation process as a pipeline, which is a linear sequence of steps. Each step stores a reference to an action object (*e.g.,* an optimization pass), its parameters and other information. GCv5 is a fully parallel garbage collector including various algorithms, and can work in generational and non-generational modes.

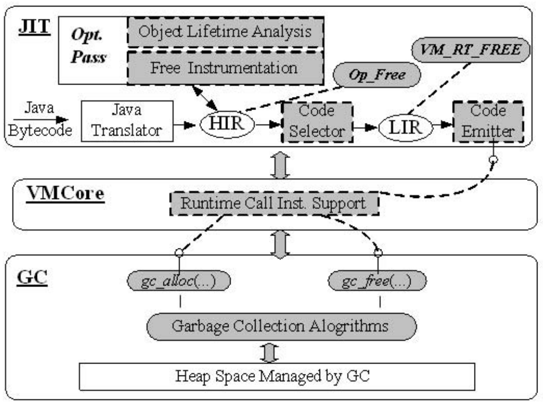


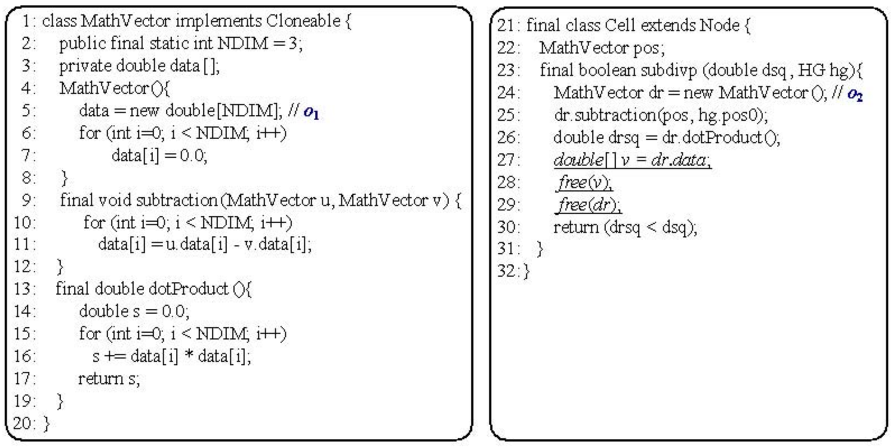
Fig. 1. Framework of the JIT-assisted GC

Fig.1 shows the framework of JIT-assisted GC, which mostly refers to the shadowed areas in the figure. On the JIT side, the compilation pipeline loads the bytecode of current to-be-compiled method, first translates it into the HIR via the translator, then transforms the HIR into the LIR via the Code Selector, and last emits the native code via the Code Emitter. We currently explore techniques for explicit object reclamation and space reuse based on the framework and they work as follows:

JIT side. To support explicitly object deallocation, an Op_Free instruction and a VM_RT_FREE runtime call instruction are extended into HIR and LIR, respectively. And the Code Selector and the Code Emitter are modified to support translating the extended instructions. Then algorithms on compiler analysis and transformation for explicit object deallocation are designed and implemented as an optimization pass for HIR type.

GC side. A *gc_free* interface is added to support explicitly reclaiming object space, and the implementation of *gc_alloc* is modified to try to reuse the explicitly reclaimed space immediately.

VMCore side. Runtime support for mapping VM_RT_FREE instruction to *gc_free* interface is implemented. The mapping needs to push the right parameters onto the runtime stack, and to ensure the consistency of the stack pointer before and after the call of *gc_free* interface.



```
 1: class MathVector implements Cloneable {
 2:    public final static int NDIM = 3;
 3:    private double data [];
 4:    MathVector(){
 5:       data = new double[NDIM]; // o1
 6:       for (int i=0; i < NDIM; i++)
 7:            data[i] = 0.0;
 8:    }
 9:    final void subtraction (MathVector u, MathVector v) {
10:       for (int i=0; i < NDIM; i++)
11:         data[i] = u.data[i] - v.data[i];
12:    }
13:    final double dotProduct (){
14:       double s = 0.0;
15:       for (int i=0; i < NDIM; i++)
16:         s += data[i] * data[i];
17:       return s;
19:    }
20: }
```

```
21: final class Cell extends Node {
22:    MathVector pos;
23:    final boolean subdivp (double dsq , HG hg){
24:       MathVector dr = new MathVector (); // o2
25:       dr.subtraction(pos, hg.pos0);
26:       double drsq = dr.dotProduct();
27:       double[] v = dr.data;
28:       free(v);
29:       free(dr);
30:       return (drsq < dsq);
31:    }
32: }
```

Fig. 2. Code fragment from BH, one of the Jolden benchmarks. The code in italics is inserted by the compiler.

Beyond the explicit reclamation and space reuse discussed in this paper, more techniques for compiler assisted GC can be explored on the framework. *e.g.,* as to some allocation sites in loops we can let them produce objects at the first iteration and reset such objects in subsequent iterations to reduce the allocation overhead of GC.

2.2 A Simple Example

Fig.2 shows a code fragment from BH of Jolden. The code underlined is inserted by the compiler. Lines 1-20 show a class MathVector containing a static field *NDIM* and an array field *data*. The constructor at lines 4-8 builds a double array object o_1 and initializes each element of o_1. Lines 23-31 show a method *subdivp* in class Cell which creates an object o_2 of type MathVector at line 24. Note that after line 26, *dr* is not live, thus o_2 only referenced by *dr* is dead.

If the compiler maintains the field reference information across procedures, it can further check whether objects referenced by the fields of the identified dead objects are dead. In the example object o_1 is only referenced by field *data* of o_2 in method *subdivp*, so once o_2 is dead, the compiler can detect that o_1 is also dead according to the field reference information, and can decide that the dead points (*i.e.,* program point where an object is dead) of o_1 and o_2 are the same.

Although the compiler identifies o_1 and o_2 are dead after line 26, there are still two problems to be considered. One is does the allocation site of an object dominates its dead point? The other is how to get the reference to the dead object? Here the allocation site of o_2 dominates the point after line 26, so its free instruction can be inserted after line 26. However, there is no reference to o_1 in original *subdivp* code, so the compiler has to generate instructions to obtain the reference of o_1, the load of *dr.data* in Fig.2 (line 27) will correspond to several instructions in HIR level.

3 Object Lifetime Analysis and Program Transformation

In this section we first give the compiler analysis and transformation for explicit object deallocation in a nutshell, then describe some key parts in it.

3.1 Analysis and Transformation in a Nutshell

When executing a Java application on DRLVM, VMCore controls the class loading, and interacts with the compiler to compile the bytecode of a to-be-executed method into native code, and then executes it. The compilation process of a Java method is defined as a pipeline specified in the EM configuration file, and our analysis and transformation is developed as an HIR optimization pass which can be configured into a pipeline.

Due to the limitation of the pipeline management framework, the pass can only directly obtain the HIR of the current compiling method M, and have to insert free instructions into M only when it is the compiling method of the pipeline.

```
01 if ($M$ is not analyzed){
02   $G$ = init($M$);
03   $\mathcal{L}$ = calculateLiveInfo($M$);
04   $B$ = getRevRuntimeBasicBlocks($M$);
05   foreach $b$ in reverse iterator of $B$ {
06     foreach instruction $i$ in $b$
07       transGwithInst($i$, $G$);
08     transGwithLiveInfo($b$, $\mathcal{L}$, $\mathcal{D}$);
09     genInstrumentInfo($\mathcal{D}$, $\mathcal{I}$)
10   }
11   addResult($M$);
12 }
13 $\langle\mathcal{M},\mathcal{I}\rangle$ = getResult($M$);
14 transHIR($\langle\mathcal{M},\mathcal{I}\rangle$);
```

Fig. 3. Flow of the object lifetime analysis and transformation

Fig.3 is the flow of the pass, where method M is in the HIR of static single assignment (SSA) form; $\mathcal{L}$, $\mathcal{D}$, $\mathcal{I}$, and $\mathcal{M}$ represent the liveness information, the dead object information, the free instrument information and the summary of M, respectively. Line 03 calculates $\mathcal{L}$. Line 04 gets the reversed pseudo-runtime basic block sequence of M, which consists of all reachable basic blocks of M in reverse topological order, where the basic block exited from a loop is located before all basic blocks in the loop, and the exception handling block edged from a basic block b is located before other blocks edged from b. Lines 05-10 include operations on identifying $\mathcal{D}$ through the intra-procedural and inter-procedural analysis based on a program abstraction called points-to escape graph (PEG), and collecting $\mathcal{I}$ from $\mathcal{D}$. Line 11 records the analyzing result of M, and line 14 transforms the HIR of M according to the analyzing result of $\mathcal{M}$.

In the following subsections we present the details of the PEG, the intra-procedural and the inter-procedural analysis, and the instrument information collection in turn.

3.2 Points-to Escape Graph

Definition 1. *Suppose M is a method, and V, P denote the set of variables and the set of formal parameters of method M, respectively. The PEG of M is a directed graph, denoted as $G = (N_o \uplus N_r, E_p \uplus E_f)$ ($\uplus$ represents disjoint union) where:*

– *$N_o = N_c \uplus N_p$ represents the set of objects accessed in M.*
  - *N_c represents the set of objects created by allocation sites in M.*
  - *$N_p = N_{fp} \cup N_{in}$ represents the set of objects created outside M, called phantom objects, where N_{fp} represents the set of objects created in the direct or indirect callers of M and passed into M via formal parameters of M and their fields, N_{in} represents the set of objects created in the direct or indirect callees of M and passed into M via the return value receivers and their fields or fields of the actual parameters at each call site in M.*
  - *$N_{ret} \subseteq N_o$ represents the set of objects returned from M.*

– *N_r is the set of reference nodes in M. Each variable with reference type in M corresponds to a reference node, i.e., $N_r \subseteq V$.*

– *$E_p \subseteq N_r \times N_o$ represents the set of points-to edges. $\langle v, o\rangle \in E_p$ denotes that reference node v may point to object o.*

– *$E_f \subseteq N_o \times F \times N_o$ represents the set of field edges where F represents the set of non-static fields in M. $\langle o_1, f, o_2\rangle \in E_f$ denotes that field f of object o_1 may point to object o_2.*

Each object o in a PEG G ($o \in N_o$) has an associated escape state, denoted as $\xi(o)$. The range of $\xi(o)$ is a lattice $\mathcal{E}$ consisting of two elements: $\mathcal{E}_N \prec \mathcal{E}_G$. $\mathcal{E}_G$ means the object escapes globally and may be accessed by multiple threads, $\mathcal{E}_N$ means that the object may not escape globally.

If an object o does not escape globally, that is, the object can be accessed only by a single thread, and no other variables or object fields refer to object o after a program point p, then o can be reckoned as a *dead object* at point p, we call p the *dead point.*

3.3 Intra-procedural Analysis

The identification of dead objects are accompanied by building and transforming the PEG of M according to each instruction in the pseudo-runtime basic block sequence, and the live variable information, *i.e.,* lines 07-08 in Fig.3. We first discuss the analysis process neglecting call instructions in this subsection.

Transforming the PEG According to the Basic Instructions. Given an instruction i in the HIR, the PEG at entry to i (denoted as $G_{(\bullet i)}$) and that at exit from i (denoted as $G_{(i \bullet)}$) are related by the standard data flow equations:

$$G_{(i\bullet)} = f^i(G_{(\bullet i)}) \tag{1}$$

$$G_{(\bullet i)} = \wedge_{i' \in Pred(i)} G_{(i'\bullet)} \tag{2}$$

where f^i denotes data flow transfer function of instruction i, $Pred(i)$ is the set of predecessor instructions of i and operator $\wedge$ is a merge of PEGs. Table 1 shows the transfer function f^i for each kind of basic instructions i, where the A_c and A_p operations are defined in Definitions 2 and 3.

Table 1. The transfer functions for each kind of basic instructions

HIR instruction i		$G_{(i.)} = f^i(G_{(.i)})$
defineArg:	$fp \in P$	$o :=$ newObject(); $N_{fp} := N_{fp} \cup \{o\}; E_p := E_p \cup \{\langle fp, o\rangle\}$; $\xi(o) := \mathcal{E}_G$.
new:	$v =$ new C $v =$ new C[]	$o :=$ newObject(); $N_c := N_c \cup \{o\}, E_p := E_p \cup \{\langle v, o\rangle\}$; **if** ($o$ is a thread object) $\xi(o) := \mathcal{E}_G$ **else** $\xi(o) := \mathcal{E}_N$.
copy:	$v_1 = v_2$	$E_p := E_p \cup \{\langle v_1, o\rangle \mid \langle v_2, o\rangle \in E_p\}$.
phi:	v=phi(v_1,v_2)	$E_p := E_p \cup \{\langle v, o\rangle \mid \langle v_1, o\rangle \in E_p \vee \langle v_2, o\rangle \in E_p\}$;
putField:	$v_1.f = v_2$	suppose $X = \{x \mid \langle v_1, x\rangle \in E_p\},\ Y = \{y \mid \langle v_2, y\rangle \in E_p\}$ $E_f := E_f \cup \{\langle x, f, y\rangle \mid x \in X, y \in Y\}$; $\forall x \in X, \forall y \in Y . A_c(x, y)$; **if**$(\xi(y) = \mathcal{E}_G)\ A_p(y)$.

getField:	$v_1 = v_2.f$	suppose $X = \{x \mid \langle v_2, x\rangle \in E_p\},\ Y = \{y \mid \langle x, f, y\rangle \in E_f, x \in X\}$
		if$(Y \neq \emptyset)\{$ $o :=$ newObject(); $N_{in} := N_{in} \cup \{o\}$; $\xi(o) := \mathcal{E}_N$;
		$E_f := E_f \cup \{\langle x, f, o\rangle \mid x \in X\}$; $E_p := E_p \cup \{\langle v_1, o\rangle\}\}$
		else$\{E_p := E_p \cup \{\langle v_1, y\rangle \mid y \in Y\}\}$

putStaticField:	C.sf=v	$\forall \langle v, o\rangle \in E_p . \xi(o) := \mathcal{E}_G$; $A_p(o)$.
getStaticField:	v=C.sf	$o :=$ newObject(); $N_{in} := N_{in} \cup \{o\}$; $\xi(o) := \mathcal{E}_G$; $E_p := E_p \cup \{\langle v, o\rangle\}$.
return:	return v	$N_{ret} := N_{ret} \cup \{o \mid \langle v, o\rangle \in E_p\}$.

Definition 2. *Given two object nodes $o_1, o_2 \in N_o$ in the PEG G, the escape state combination operation $A_c(o_1, o_2)$ which propagates $\xi(o_1)$ to $\xi(o_2)$ is defined as:*

$$\frac{e \in \mathcal{E} \quad e = \xi(o_1) \quad \xi(o_2) \prec e}{\xi(o_2) := e} \tag{3}$$

Definition 3. *Given an object node $o \in N_o$ in the PEG G where $\xi(o) = \mathcal{E}_G$, operation $A_p(o)$ sets the escape state of each object reachable from object o via a path of field edges to be $\mathcal{E}_G$.*

The A_c operation is used when there is a field assignment (*i.e., putField* or *getField* in Table 1) or inter-procedural information combination, while the A_p operation is used when there is a static field assignment (*i.e., putStaticField* or *getStaticField*).

Transforming the PEG Combining with the Live Variable Information. After analyzing all instructions in a basic block, combining with the live variable information, if a variable v is not live, the out points-to edges of v will be clipped, thus objects only pointed to by v can be regarded as dead. Furthermore, if an object o dies, the out field edges of o will be clipped, thus objects only referenced by the fields of o can also be regarded as dead. The clip operation A_D is based on the live variable information and produces the dead object information at the end of each basic block.

Definition 4. *Given an object $o \in N_o$, a reference node $v \in N_r$, G and G' denote the PEGs before and after the A_D operation respectively. The A_D operation is defined as the following two rules.*

$$\frac{G = (N_o \cup N_r, E_p \cup E_f) \quad v \in N_r \quad E_p^v = \{\langle v, o\rangle | o \in N_o\}}{G' = (N_o \cup N_r', E_p' \cup E_f) \quad N_r' = N_r - \{v\} \quad E_p' = E_p - E_p^v} \tag{4}$$

$$\frac{G = (N_o \cup N_r, E_p \cup E_f) \quad o \in N_o \quad E_f^o = \{\langle o, f, o'\rangle | o' \in N_o\}}{G' = (N_o' \cup N_r, E_p \cup E_f') \quad N_o' = N_o - \{o\} \quad E_f' = E_f - E_f^o} \tag{5}$$

3.4 Inter-procedural Analysis

When analyzing a method M, only objects with $\mathcal{E}_N$ state may be explicitly freed. If such an object is referenced by a formal parameter or the return value of M or reachable from their fields, the object cannot be freed in M because M's callers may use it. The object lifetime analysis needs to record them into a summary of M, and update the PEG of M's caller using M's summary when analyzing a call instruction to invoke M.

Definition 5. *Given a method M and its PEG $G = (N_o \cup N_r, E_p \cup E_f)$, the object lifetime analysis result of M is a 2-tuple $\langle \mathcal{M}, \mathcal{D}\rangle$ where:*

- *$\mathcal{M} = (N_{fp} \cup N_{ret}, E_f')$ is a summary of M. It records all objects referenced by the formal parameters or the return value of M,* i.e., *$N_{fp} \cup N_{ret}$, and the set of field edges starting from them,* i.e., *$E_f' = \{\langle o, f, o'\rangle | o \in N_{fp} \cup N_{ret} \wedge \langle o, f, o'\rangle \in E_f$.*
- *$\mathcal{D}$ describes the dead object information in M. It is a set of triples, each triple is denoted as $\langle o, r, p\rangle$, where*
  - *$o \in N_o$ is dead after the point p in M,*
  - *$r = \langle v, f\rangle$ represents the reference to o. If f is null, then v is the reference to o, otherwise, $v.f$ is the reference to o.*

Given a method M and its PEG G, suppose there is a call instruction $v = v_0.\mathsf{m}(v_1, ..., v_n)$ in M and the summary of m is $\mathcal{M}_\mathsf{m} = \langle N_{fp}^\mathsf{m} \cup N_{ret}^\mathsf{m}, E_f^\mathsf{m}\rangle$. The process of dealing with the call instruction is as follows:

1. Combine the formal parameters and the actual parameters. For each $\langle v_i, o\rangle$ in E_p, perform $A_c(fp_i, o)$ where $fp_i \in N_{fp}^\mathsf{m}$ is the corresponding formal parameter of m.
2. Combine the return value and the return value receiver. For each $\langle v, o\rangle$ in E_p, perform $A_c(r, o)$ where $r \in N_{ret}^\mathsf{m}$ is the return value of m.
3. Map field edges. For each edge in E_f^m, add a corresponding edge in E_f.
4. Propagate escape states. If the escape state of an object o referenced by one of the actual parameters or the return value receiver becomes $\mathcal{E}_G$, then perform $A_p(o)$.

The above inter-procedural combination takes a callee summary as precondition. However, a callee of M may not be analyzed when analyzing M. If so, there are two optional ways to deal with the call site. One is to neglect the callee and to make a decision conservatively. The other is to start up a new pipeline to compile the unanalyzed callee to obtain its summary. This special pipeline only includes a few basic passes translating bytecode into non-optimized HIR of SSA form and the object lifetime analysis pass in order to obtain the analyzed result of the callee and not the native code of the callee. The latter way is more precise but consumes more overhead. We introduce an argument to control the depth level starting up the special compilation of unanalyzed callees, thus users can use it to trade off between precision and performance.

3.5 Collecting Free Instrument Information

The dead objects and their dead points in $\mathcal{D}$ cannot be directly used as the instrument information generating free instructions. Sometimes instrumenting directly at a dead object o's dead point may bring compile-time or runtime errors. *e.g.,* if o is created in a branch and dies outside the branch, freeing o at the dead point might induce a runtime error. Another problem is how to free an object o that dies in method M and has no explicit reference to o in M, *e.g.,* o_1 referenced by *dr.data* in method *subdivp*.

Therefore, we need to collect instrument information from the dead object information. The structure of the instrument information $\mathcal{I}$ is quite similar to that of dead object information, the only difference is that the dead point in the latter is changed into the instrument point in the formal.

When collecting instrument information, a rule must be followed: given a dead object o and its dead point p, the basic block in which the allocation site of o appears must dominate the dead point p of o. Otherwise, it means the compiler is trying to free an object that may not be allocated before, thus causing runtime errors. The dominance relationship can be obtained from the dominance tree of HIR.

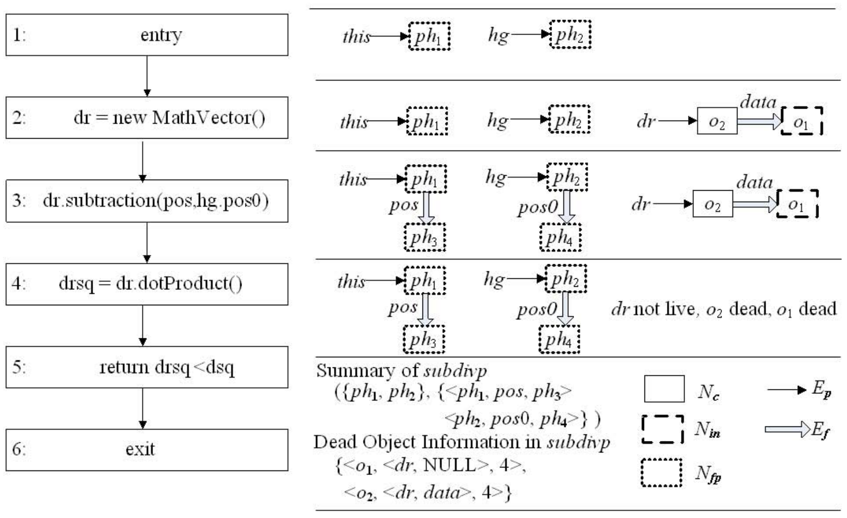
Two key steps of the collecting process are as follows:

Confirm dead object reference: for an object o, there are two kinds of references: one is at its allocation site, *e.g.,* A a = new A( ), where a is the reference to the newly created object o here; the other is brought by phi, $assign$, $getField$, or $putField$ instructions. We preferentially choose the reference at allocation site for a dead object. If the dead object has no explicit reference in the method, we can make use of other object's field to generate its reference indirectly.

Confirm instrument point: for a dead object o, we preferentially choose the basic block which contains a $return$ instruction (denoted $return$ node) to insert instructions to free o. If there is not any exception when executing the program, the $return$ node must be executed. If the confirmed reference point of dead object o cannot dominate $return$ node, and the dead point of o is in a branch, then we have to insert instructions to free o at the dead point because such a branch may not be executed.

According to the reference and the instrument point information of each dead object provided in $\mathcal{I}$, the code transformation of free instrumentation can easily create instructions to explicitly free object.

Fig.4 shows the analysis process of method *subdivp* in Fig.1. We give source-level statements for the sake of brevity. The `new` expression in block 2 will implicitly invoke the constructor, so object o_1 created there will be passed into *subdivp* and become an element of N_{in} in the PEG of *subdivp*. At the end of



(a) CFG of *subdivp* in Fig.1 (b) Analysis process and analysis result of *subdivp*

Fig. 4. An example illustrating the analysis process

block 4, variable *dr* is not live, so the analysis determines that o_2 only referenced by *dr* and o_1 only referenced by *dr.data* are both dead. At last the analysis will record the analysis result.

3.6 Special Design Tradeoff

Thread Object. Thread objects are distinguished from other ordinary objects based on class hierarchy diagrams, and their escape states can be initialized as $\mathcal{E}_G$. If an object o is assigned to a thread object field, then $\xi(o) := \mathcal{E}_G$. If a thread object o_t has ended its execution, o_t will be treated as an ordinary object and $\xi(o_t)$ will be reset according to the escape states of objects referring to o_t in the current PEG. However, it is difficult to identify when and where a thread has ended execution. Our analysis only judges this case by $join()$ invocations of thread objects.

Loop. Instructions in loops are analyzed only once, which makes the analysis simpler and cheaper since the analysis overhead is a part of the whole program runtime overhead. The analysis is also correct and conservative because according to the rules in section 3.5, 1)assuming the allocation site p_a of an object o occurs before a loop entry and o dies in the loop, if p_a dominates a *return* node p_r, then select p_r as the instrument point, else might select some point after the loop exit; 2) assuming the allocation site p_a of o occurs in a loop, if o dies in the loop, then select the dead point in the loop which can be dominated by p_a as the instrument point, otherwise indicating any reference to o is live at all basic blocks of the loop, and not freeing o.

Array. All elements of an array are abstracted as an object with a special field, and accesses to an element are treated as accesses to the special field. It may reduce the size of explicitly freed objects but save analysis overhead.

Recursion. Our inter-procedural analysis can handle recursion. It maintains a chain of method invocation when meeting an unanalyzed callee, if the current unanalyzed callee has already existed in the chain (*i.e.,* there is a recursion), the loop in the chain is cut and the inter-procedural combination could be done conservatively based on the current method summaries.

4 Explicit Reclamation and Space Reuse

GCv5 [19] is a parallel GC which support multiple *collector*s running collaboratively. We choose GCv5-MS to implement explicit reclamation and space reuse (denoted as JIT-GCv5-MS), because it uses free-list to organize heap space and is convenient to add or acquire a free space from the heap.

4.1 Brief Overview of GCv5-MS

Each thread in an application (called *application thread*) corresponds to a *mutator* thread in GCv5-MS. Each *mutator* takes charge of the allocation of the corresponding application thread. GCv5-MS classifies objects into two kinds, *i.e.,* small objects (less than 1KB) and large objects (greater than or equal to 1KB), and provides *Free Block Pool* (FBP) and *Free Area Pool* (FAP) shown in Fig.5 for the allocation of the two kinds, respectively. Each pool is organized as an array of segregated free lists, where each free list contains *blocks*/*areas* of the same size or class size.

The FBP has 254 segregated free *block* lists shared among all *mutator*s, and *block*s in the same list provide objects of the same size (from 8B to 1020B, aligned in 4B). Each *block* comprises a header and a data area. The header depicts information on the data area, *e.g.,* a bitmap marking the status of each slot in the data area, such as in use or free. Each *mutator* requests a free block from the pool and its acquired blocks are local to the *mutator*. When a *mutator* receives a request of allocating a small object, it searches its local block of the requested size. If there is a free slot of the requested size then the *mutator* can return one; otherwise it need to request a free block of the requested size from the pool. Operations on the pool must be synchronized while operations on the mutator-local blocks need not.

The FAP has 128 segregated free area lists. The last list contains free areas of the size greater than or equal to 128KB. All *mutator*s share the pool and must request memory for large objects with synchronization. Generally speaking, there are relatively few large objects in applications, so the synchronization overhead of parallel large object allocations is not high.

4.2 Allocation and Explicit Reclamation in JIT-GCv5-MS

In order to support explicit reclamation and space reuse, we modify GCv5-MS as JIT-GCv5-MS to add *gc_free* and to modify the implementation of *gc_alloc*.

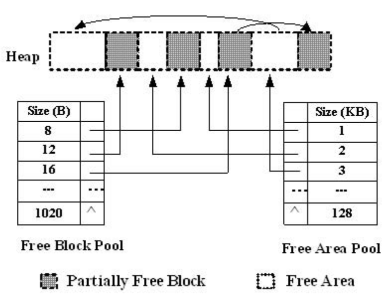


Fig. 5. Heap space management of GCv5-MS

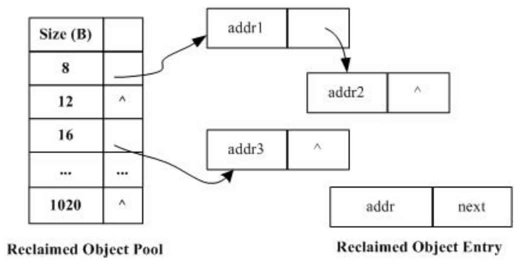


Fig. 6. Heap space management of the explicitly reclaimed objects

Each *mutator* in JIT-GCv5-MS handles not only allocation requests but also explicit free ones from its corresponding application thread. Due to the different memory management mechanisms between the small and the large objects in GCv5-MS, we take different methods.

Handling with Small Objects. If a *mutator* m_f receives a request to free a small object o, slot occupied by o must belong to a mutator-local block of some mutator m_a, where m_f may not be m_a, that is, o may not be thread-local. If m_f directly modifies the mark bits of o in the block header as free status to reclaim the slot, and lets the original allocation algorithm control the reuse, accesses to the word containing the mark bits by m_f need to be synchronized, because the word contains other slots' mark bits, which may be accessed by m_a to handle an allocation request or by other mutator to handle another explicit free request, simultaneously. Thus allocation operations on the mutator-local blocks which need not be synchronized originally, have to be synchronized, which brings more synchronization overhead.

In order to avoid such synchronization, we introduce a *Reclaimed Object Pool* (ROP) (shown in Fig.6) for each *mutator* to collect its explicitly reclaimed object spaces. When *mutator* m_f reclaims an object o, it does not modify the mark bits of o, but forces the object slot into a node of type *Reclaimed Object Entry* and inserts the node into a list of the same size in m_f's local ROP. *gc_alloc* need be modified to try to reuse the explicitly reclaimed object space immediately. That is, it first searches its local ROP for free space of the requested size. If there are none, it continues to allocate as the original strategy in GCv5-MS.

Handling with Large Objects. Because all *mutator*s share the FAP for allocating large objects and the synchronization on these operations cannot be neglected, we keep the implementation on allocating large objects as original. When *mutator* m_f receives a request to free a large object o, it directly insert the memory area occupied by o into the free list of the matched size in the FAP, thus the subsequent object allocation of the same size will reuse the memory area. It is noticed that explicit reclamation of large objects need be synchronized, and we cannot easily obtain the reuse rate of large reclaimed object space.

5 Experimental Results

We have implemented the above work in DRLVM and evaluated it with Jolden and SPECjbb2005. The experiments were performed on 2.1GHz AMD Athlon dual core machine with 896MB of memory running Windows XP.

5.1 Effectiveness of the JIT-Assisted GC

First, we check whether the JIT-assisted GC frees still reachable objects or frees dead objects at wrong program points. In order to perform the correctness validation, we modify the implementation of *gc_free*, mark the explicitly reclaimed object as un-useable and un-reusable. In addition, the pipeline performs many checks in LIR, such as variable liveness checking. In this way if done a wrong free action, the system will throw exception at the next access to a freed object or at the access to a potential undefined variable. The experiments show that there are no such exceptions and errors in compile time or runtime.

Table 2 presents the statistics on allocation, free and reuse for our JIT-assisted GC system at the default heap size 256MB of the VM. The first four programs are from Jolden, the JIT-assisted GC explicitly frees 66% of all objects on average and up to 96% in Jolden. We find that the free instructions inserted in loops or recursive methods can bring considerable income, and these explicitly reclaimed object spaces can be reused easily because the same allocation site will be executed many times, *e.g.,* Health reclaims 14MB and almost all the space is from such free instructions.

Table 2. Memory freed and reused by JIT-assisted GC.

Application	Total Alloc Mem	Free Mem on *free(x)*	Free Mem on *free(x.f)*	Total Free Mem.	Total Reuse Mem	%Free Mem	%Reuse Mem
BH	67MB	14MB	46MB	60MB	60MB	90%	100%
Health	60MB	14MB	0B	14MB	14MB	23%	100%
Power	24MB	23MB	100B	23MB	23MB	96%	100%
TSP	51MB	28MB	88B	28MB	27MB	55%	96%
SPECjbb2005	1419MB	104MB	0B	104MB	104MB	7%	100%

The last column of the table shows the explicitly reclaimed memory reuse ratio. For the programs in Jolden, the ratios are high and the explicitly reclaimed objects are all small objects, this illustrates that the JIT-GCv5-MS can reuse almost all these small objects. The reuse ratio of SPECjbb2005 is relatively low because the system reclaims many large objects, and the system does not count statistics for the large object space reuse due to not increasing the synchronization cost.

5.2 Time Cost

Table 3 presents the statistics on the time cost of our object lifetime analysis and transformation pass and the total compilation time, we can see that the pass cost less than 10% of the total compilation time.

Table 3. Analysis time and total compilation time.

Application	Objlife Time	Total Comp. Time	%Objlife
BH	23ms	537ms	4.3%
Health	14ms	309ms	4.5%
Power	12ms	332ms	3.6%
TSP	11ms	207ms	5.3%
SPECjbb2005	738ms	19011ms	3.9%

5.3 Performance Improving of JIT-Assisted GC

To evaluate the performance impact, we compared the GC execution times of the benchmark programs. Fig.7 presents the GC execution time comparison of programs in Jolden. The x-axis is the heap size and the y-axis is the GC execution time. We can see that GC execution time of JIT-GCv5-MS is less than that of GCv5-MS. Along with the increase of the heap size, the performance improvement becomes small. This is because the numbers of the explicitly reclaimed objects and those of the reused objects are fixed, the larger heap size relatively decreases the performance improvement, as Health and TSP in Fig. 7. As to BH and Power, even if the heap size is set to the least 16MB, the execution time of GC in JIT-GCv5-MS is zero, since more than 90% of the allocated space can be explicitly reclaimed.

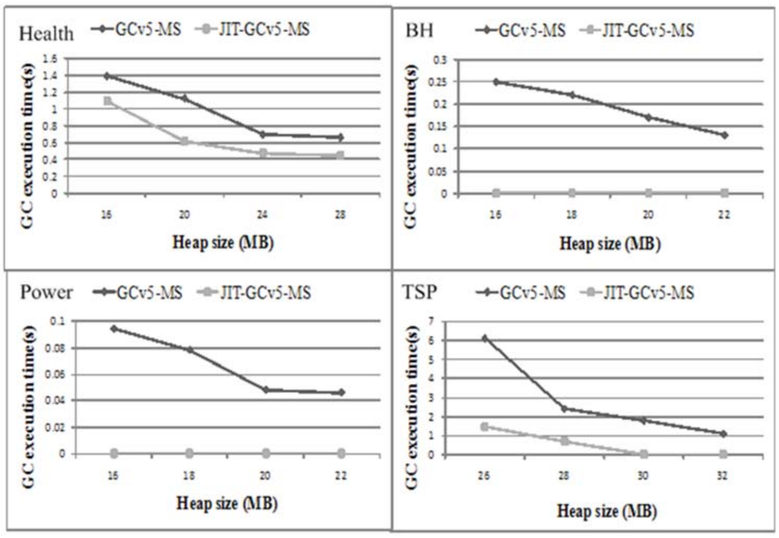
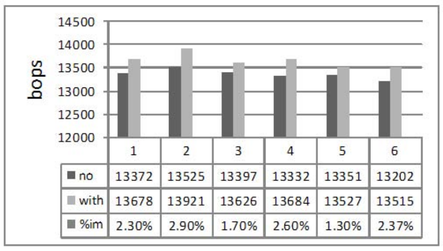


Fig. 7. Performance comparison of 4 programs in Jolden

Fig.8 shows the throughput comparison of SPECjbb2005 with and without JIT-assisted GC optimization. It lists the collectively throughputs of 6 group experiments. The dark column and the first row in the data table illustrate the throughput without JIT-assisted GC. The tint column and the second row in the



	1	2	3	4	5	6
■ no	13372	13525	13397	13332	13351	13202
■ with	13678	13921	13626	13684	13527	13515
■ %im	2.30%	2.90%	1.70%	2.60%	1.30%	2.37%

Fig. 8. The throughput comparison of SPECJbb2005

data table illustrate the throughput with JIT-assisted GC. The third row in the table is the improving ratio. We can see the improving ratio is about 1.3∼2.9%.

6 Related Work and Conclusions

Guyer *et al.* propose a *free-me* analysis [12] which is closest to our work. They combine a light-weight pointer analysis with liveness information that detects when short-lived objects die, and insert calls to free dead objects. However, their method cannot identify the lifetime of objects referenced by fields due to its field-insensitive property. Cherem *et al.* present a uniqueness inference and can free objects with unique reference in the whole heap through free instructions and destructors [13, 14], the work needs to modify libraries to add destructors, this method is complex and difficult and not fit for the system built in virtual machine because the latter need to exucute applications accompanied by just-in-time compilation. Both of the works do not support the reuse of the explicitly reclaimed space.

Lee and Yi's analysis inserts free instructions only for immediate reuse, *i.e.,* before an allocation of the same size [15]. Marinov *et al.* present Object Equality Profiling (OEP) [16] to discover opportunities for replacing a set of equivalent object instances with a single representative object. Gheorghioiu *et al.* present an inter-procedural and compositional algorithm for finding pairs of compatible allocation sites [17], which have the property that no object allocated at one site is live at the same time as any object allocated at the other site. All these works focus on object merging and reuse only for the same size objects with lifetime homogeneity only on the compiler end.

Our work can identify some short-lived objects not limited in method scope or other special features like [15], it also detects objects only referenced by the fields of the identified dead objects. The PEG based analysis seems similar to [7], however, the definition of the escape lattice and the rules on building and transforming the PEG are very different. In addition, our work not only frees the identified dead objects, but also tries to reuse them immediately. Although our current work in GCv5 is on Mark-sweep algorithm, we can easily extend the work to other algorithms in GCv5.

Based on the JIT-assisted GC framework, we can explore more optimization on memory management. We are analyzing the benefit of each free instruction inserted by JIT and the memory utility of each allocation site in loop or recursive method by developing a log system with the cooperation among JIT, VMCore and GC. According to the analysis results, we will find more chances on memory management optimization.

References

1. Jones, R., Lins, R.: Garbage collection: algorithms for automatic dynamic memory management. John Wiley & Sons, Chichester (1996)
2. Specjbb2005 benchmark (2005), `http://www.spec.org/jbb2005/`
3. Kero, M., Nordlander, J., Lundgren, P.: A correct and useful incremental copying garbage collector. In: Proc. 6th Int'l Symp. on Memory Management, October 2007, pp. 129–140. ACM Press, New York (2007)
4. Blackburn, S., McKinley, K.: Immix garbage collection: mutator locality, fast collection, and space efficiency. In: Proc. 2008 ACM Conf. on Prog. Lang. Design and Impl., pp. 22–32. ACM Press, New York (June 2008)
5. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for java programs. ACM SIGPLAN Notices 34(10), 187–206 (1999)
6. Gay, D., Steensgaard, B.: Fast escape analysis and stack allocation for object-based programs. In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 82–93. Springer, Heidelberg (2000)
7. Choi, J.D., Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P.: Stack allocation and synchronization optimizations for java using escape analysis. ACM Trans. on Programming Languages and Systems 25(6), 876–910 (2003)
8. Gay, D.E., Aiken, A.: Language support for regions. In: Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl., June 2001, pp. 70–80. ACM Press, New York (June 2001)
9. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: Proc. 2002 ACM Conf. on Prog. Lang. Design and Impl., June 2002, pp. 282–293. ACM Press, New York (June 2002)
10. Salagnac, G., Yovine, S., Garbervetsky, D.: Fast escape analysis for region-based memory management. In: Proc. 1st Int'l Workshop on Abstract Interpretation for Object-Oriented Languages, January 2005. ENTCS, vol. 141, pp. 99–110. Elseiver, Amsterdam (January 2005)
11. Stefan, A., Craciun, F., Chin, W.N.: A flow-sensitive region inference for cli. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 19–35. Springer, Heidelberg (2008)
12. Guyer, S.Z., McKinley, K.S., Frampton, D.: Free-me: a static analysis for automatic individual object reclamation. In: Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl., June 2006, pp. 364–375. ACM Press, New York (June 2006)
13. Cherem, S., Rugina, R.: Compile-time deallocation of individual objects. In: Proc. 5th Int'l Symp. on Memory Management, June 2006, pp. 138–149. ACM Press, New York (June 2006)
14. Cherem, S., Rugina, R.: Uniqueness inference for compile-time object deallocation. In: Proc. 6th Int'l Symp. on Memory Management, October 2007, pp. 117–128. ACM Press, New York (October 2007)

15. Lee, O., Yi, K.: newblock Experiments on the effectiveness of an automatic insertion of memory reuses into ml-like programs, October 2004, pp. 97–108. ACM Press, New York (October 2004)
16. Marinov, D., O'Callahan, R.: Object equality profiling. In: Proc. 18th ACM SIGPLAN Conf. on Object-Oriented Prog. Systems, Lang., and Applications, October 2003, pp. 313–325. ACM Press, New York (October 2003)
17. Ovidiu Gheorghioiu, A.S., Rinard, M.: Interprocedural compatibility analysis for static object preallocation. In: Proc. 30th ACM Symp. on Principles of Prog. Lang., January 2003, pp. 273–284. ACM Press, New York (January 2003)
18. Apache harmony drlvm (2006), `http://harmony.apache.org/subcomponents/drlvm/index.html`
19. Apache harmony gcv5 (2008), `http://harmony.apache.org/subcomponents/drlvm/gc-v5.html`