Memory Usage Improvement Using Runtime Alias Detection

Ryo HANAI, Tomoharu UGAWA, Masashi YONEDA, Masahiro YASUGI, and Taiichi YUASA

Graduate School of Informatics, Kyoto University

Abstract

Tofte and Talpin proposed a novel method for memory management called region inference in typed, higher-order languages[7]. In their method, memory is composed of blocks called regions, and when to allocate and deallocate regions are determined automatically by the compiler using a type system, and source programs are translated into target programs with region annotations.

Their method realizes fairly economical use of memory for many programs. However, there are some kinds of programs which are not friendly to region inference. When executed on region-based systems, these programs can cause large amount of memory leak and in the worst case, they cannot finish their execution because of memory shortage. Therefore, some people proposed methods to improve memory usage of Tofte/Talpin system and others proposed different region inference systems.

In this paper, we propose another technique to improve memory usage of Tofte/Talpin system. Our technique adds some changes to Storage Mode Analysis, which is a succeeding phase of their system, and delays some of decisions till runtime as to whether or not it is possible to overwrite existing objects. Our method is especially useful for a program compiled separately, where we cannot see the contexts in which top-level funtions are called. We implemented this technique to MLKit and confirmed that the amount of memory used during execution is reduced for some programs.

1 INTRODUCTION

Tofte and Talpin proposed a novel method for memory management called region inference in typed, higher-order languages. In their method, memory is composed of blocks called regions, and when to allocate and deallocate regions are determined automatically by the compiler using a type system, and source programs are translated into target programs with region annotations.

Their method realizes fairly economical use of memory for many programs. However, there are some kinds of programs which are not friendly to region inference. When executed on region-based systems, these programs can cause large amount of memory leak and in the worst case, they cannot finish their execution because of memory shortage. executed to the end. Therefore, some people proposed methods to improve memory usage of Tofte/Talpin system and others proposed different region inference systems.

In this paper, we propose another technique to improve memory usage of Tofte/Talpin system. Our technique adds some changes to Storage mode analysis (SMA)[2], which is a succeeding phase of their system, and delays some of decisions till runtime as to whether or not it is possible to overwrite existing objects. We also implemented this technique to MLKit[6][3] and confirmed that the amount of memory used during execution is reduced for some programs.

Here is an example of ML-like program.

```
letrec f(x, y) =
    let a = if x < 0 then - x else x in
        let b = if y < 0 then - y else y in a + b
    end end
in
    let u = 1 and v = 2 in f(u, v) + f(v, u) end
end end
```

This is translated into the program below with `letregion` constructs and allocation directives (`attop` ρ, `atbot` ρ, and `sat` ρ) in it by Tofte/Talpin system. When a program is translated by region inference, all allocation directives in the translated program are `at` ρ and then SMA rewrites these directives into one of the three directives `attop` ρ, `atbot` ρ, and `sat` ρ.

There are two kinds of annotations. `letregion` ρ `in` *e* binds a new region to the region variable ρ. The scope of ρ is the expression *e*. Upon completion of the evaluation of *e*, the region bound to ρ is deallocated. At the same time, any values it contains are discarded. The expressions *e* `attop` ρ, *e* `atbot` ρ, and *e* `sat` ρ evaluate *e* and write the result in ρ. We assume all values including integers are each stored in some region.

The difference among these three directives is how to allocate an object. When an object is created with `attop` ρ directive, the size of the region bound to ρ is increased by the size of the object, and the object is allocated. However, when an object is created with `atbot` ρ directive, first the region is reset, meaning that all objects in the region are deallocated, and then the new object is allocated.

Functions defined by `letrec` construct are region polymorphic. They can take extra region parameters, which are bound to actual regions by a region instantiation construct. This region polymorphism is very important from the view point of memory usage because it allows expressions to perform operations on different regions depending on the context. However, it can cause region aliases, i.e., region variables bound to the same region at runtime.

In the translated program, values of -`x` and -`y` are both stored without resetting. This is because both of the region parameters ρ_1 and ρ_2 can be aliases of the same region bound to ρ_9 (or ρ_{10}). Informally, SMA first makes a graph expressing the possibility of aliases by analyzing the whole program, and then generates one definition for each function. Therefore, even if a region in which an object will be stored is not actually aliased at runtime by any of region variables which may be accessed by the remaining computation, SMA gives up region-resetting.

This fact means that we can increase the region-resetting by comparing at run-time the regions bound to possible region aliases (ρ_1 and ρ_2 in the example). Our method is especially useful for a program compiled separately where we cannot see the contexts in which top-level functions are called.

In the next section, we pick up some features of Tofte/Talpin system, on which our method is based, and then present our method in section 3. Evaluation of our method is in section 4, and a discussion of related work is in section 5. Finally we conclude this paper in section 6.

```
letrec f [ρ1, ρ2, ρ3] (x, y) attop ρ4 =
  let a = letregion ρ6
            in if letregion ρ5 in x < 0 atbot ρ5 end atbot ρ6
               then - x attop ρ1 else x end
  in let b = letregion ρ8
              in if letregion ρ7 in y < 0 atbot ρ7 end atbot ρ8
                 then - y attop ρ2 else y end
    in a + b sat ρ3
  end end
in
  letregion ρ9, ρ10
  in let u = 1 atbot ρ9 and v = 2 atbot ρ10
     in letregion ρ11, ρ12
        in (letregion ρ13
              in f [attop ρ9, attop ρ10, atbot ρ11] atbot ρ13 (u , v) end
             + letregion ρ14
               in f [atbot ρ10, atbot ρ9, atbot ρ12] atbot ρ14 (v , u)) end
             attop ρ0
end end end end end
```

2 TOFTE/TALPIN SYSTEM

In this section, we give a short description of Tofte/Talpin system especially of SMA, on which our proposal is based. Please refer to [2][5][7] for more details.

2.1 Source and Target Language for SMA

SMA is a successive phase of region inference in Tofte/Talpin system. SMA takes a program with all allocation directives being `at` ρ as its input, and replaces those directives into extended allocation directives `attop` ρ, `atbot` ρ, `sat` ρ. Therefore, the source and target languages for SMA are the same, and it includes both allocation directives.

$$
\begin{array}{rcl}
e & ::= & c\ a \mid x \mid \lambda x.e\ a \mid e_1\, e_2 \\
  & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \\
  & \mid & \texttt{letrec}\ f[\vec{\rho}](x)\ a = e_1\ \texttt{in}\ e_2\ \texttt{end} \\
  & \mid & \texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end} \\
  & \mid & f\ [\vec{a}]\ a \\
a & ::= & \texttt{at}\ \rho \mid \texttt{attop}\ \rho \mid \texttt{atbot}\ \rho \mid \texttt{sat}\ \rho
\end{array}
$$

This language is essentially conventional polymorphically typed lambda calculus[4], but it has several constructs to express region operations explained section 1.

2.2 Storage Mode Analysis

Regions introduced by `letregion` construct cannot be deallocated until the control flow of the program exits their scope. As a result, objects created in a region during successive function calls cannot be deallocated until the program returns to its outermost call even if those objects get useless. This means that the size of some regions and/or the number of regions may keep growing and in the worst case the program may use up memory.

To alleviate this situation, Tofte et al. use an optimization after the region inference phase. They call this optimization storage mode analysis, which is abbreviated to SMA. SMA is a kind of flow analysis and changes some object creations into the ones with region-resetting. Here, region-resetting means that all values in the region are discarded but the region itself is not deallocated. For this purpose, they extend allocation directives `at` ρ to two different kinds of allocation directives, `attop` ρ and `atbot` ρ. The allocation directive `attop` ρ is the same as `at` ρ operationally, but `atbot` ρ means a value is created after the region is reset. They also add another allocation directive `sat` ρ, meaning “somewhere at” for `letrec` bound region variables to realize more efficient memory usage. Generally, `letrec` bound region variables are bound to regions passed by their callers. Therefore, when a function creates a new object in a region, it cannot tell if the region still has objects which will be used by the caller only from the function-local analysis. So `sat` ρ in a function decides whether it uses reset or not using the information passed by its caller.

SMA first analyzes the whole program and make a directed graph G expressing the dependency of region and effect variables. This graph is called region flow graph. There is one node in G for every region variable and every effect variable which occurs in the program. Thus, we can identify variables with nodes. Whenever the program has a `letrec` bound program variable f with type scheme:

$$\pi \quad = \quad \forall \cdots \rho_i \cdots \alpha_j \cdots \varepsilon_k \cdots . \underline{\tau}$$

and whenever there is an applied occurrence of f instantiated by a substitution:

$$S = (\{\cdots \rho_i \mapsto \rho_i' \cdots\}, \{\cdots \tau_j \mapsto \tau_j' \cdots\}, \{\cdots \varepsilon_k \mapsto \varepsilon_k'.\varphi_k \cdots\})$$

there is an edge from ρ_i to ρ_i', and from ε_k to ε_k'. Similarly for `let` bound variables. Finally, for every effect $\varepsilon.\varphi$ occurring anywhere in the program, there is an edge from ε to every region and effect variable which occurs free in φ. Intuitively, region variables reachable from a region variable ρ can be aliases of ρ, and region variables reachable from an effect variable ε can be accessed by calling a function whose arrow effect is $\varepsilon.\varphi$ (for some φ). Next, at each allocation point, SMA identifies live program variables, and then identifies region variables which may be accessed by the rest of the computation using types of the live variables and the graph G.

3 RUNTIME ALIAS DETECTION

This section describes our technique using runtime alias detection. The third storage mode `sat`(somewhere at) for `letrec` bound region variables accomplishes polymorphism in storage mode and enables regions to be reset in more allocation points. However, because of region aliases generated by region polymorphic functions, it is difficult to see exactly if a region in which we are to allocate an object through a region variable is aliased by other region variables. Therefore, SMA translates allocation directives in a program conservatively to `sat` only if the region is never accessed after that allocation point by any region closure instantiation of the corresponding function in the program.

However, a region bound to each region variable is determined at runtime. So, by checking the existence of the problematic aliases at runtime, we can improve memory usage of Tofte/Talpin system.

3.1 Extending `sat` allocation directive

We want to delay some of decisions till runtime as to whether or not it is possible to overwrite existing objects. Therefore, we first extend the allocation directive `sat`.

$$a \ ::= \ldots \mid \texttt{sat}\ \rho\ \texttt{unless}\ \{\rho_1, \ldots, \rho_n\}$$

This extended `sat` directive is a version of `sat` and takes a set of region variables after a keyword `unless`. It behaves as `attop` if the target region in which we store an object is aliased by some region variables in the following set. If not, it behaves as original `sat`. In other words, the region is reset if it is not used by the caller. Region variables in the following set are candidates of aliases at the program point.

3.2 Translation Rules

Next, we present translation rules to replace allocation directives in program. These are based on the rules of SMA.

Let E be a conventional global expression context, and R be a local allocation context. Please refer to [2] for the full definition. For simplicity, we assume that region polymorphic functions take only one region parameter. Let $LV(R)$ be the set of live program variables in context R. Let $\mathrm{frv}(A)$ be the set of free region variables for a semantic objects A such as expressions, types, and type schemes. Similarly, let $\mathrm{fev}(A)$ be the set of free effect variables. For every node n in the region flow graph G, let $\langle n \rangle$ denote the set of variables that are reachable in G starting from n, including n itself. This definition is naturally extended to the set of nodes.

We define $LR(R)$ to be the set of all region variables which appear free in the type with place of a live program variable in R, $\bigcup \{\mathrm{frv}(T,\rho) \mid x : (T,\rho) \in \mathrm{LV}(R)\}$. We also define $LE(R)$ to be a set $\bigcup \{\mathrm{fev}(T) \mid x : (T,\rho) \in LV(R)\}$. We define $CS(R,\rho)$ to be a set of alias candidates, $\{\rho' \mid \rho' \in LR(R), \langle\rho\rangle \cap \langle\rho'\rangle \neq \emptyset\}$. When an object is stored in the region bound to ρ in a local allocation context R, the elements of $CS(R,\rho)$ are free region variables of R, and by tracing the graph G, we can reach a node which is also reachable from ρ.

In our scheme, all `at` ρ allocation directives for `letrec` bound region variables can be extended to `sat` ρ directives (rule 4). However, every live region cannot always be checked by using region variables within the current scope. Suppose that we are to store an object in a region bound to a region variable ρ, and we can reach a node starting from ρ and a live effect variable in the graph G defined in the previous section. In this case, there may be a live closure which might access the region bound to ρ, but we cannot generate an enough set to confirm that there is no problematic aliases because the current region environment and the region environment of the closure may bind same region variable (by name) to different regions. Therefore we translate these directives conservatively into `attop` (rule 3). The other rules are the same as original ones of SMA.

$$\frac{\rho \in \langle LR(R)\rangle \cup \langle LE(R)\rangle}{E[\mathtt{letregion}\,\rho\,\mathtt{in}\,R[\mathtt{at}\,\rho]\,\mathtt{end}]} \Rightarrow E[\mathtt{letregion}\,\rho\,\mathtt{in}\,R[\mathtt{attop}\,\rho]\,\mathtt{end}] \quad (1)$$

$$\frac{\rho \notin \langle LR(R)\rangle \cup \langle LE(R)\rangle}{E[\mathtt{letregion}\,\rho\,\mathtt{in}\,R[\mathtt{at}\,\rho]\,\mathtt{end}]} \Rightarrow E[\mathtt{letregion}\,\rho\,\mathtt{in}\,R[\mathtt{atbot}\,\rho]\,\mathtt{end}] \quad (2)$$

$$\frac{\rho \in LR(R) \ \vee\ \langle\rho\rangle \cap \langle LE(R)\rangle \neq \emptyset}{E[\mathtt{letrec}\ f[\rho](x)\ a_0{=}R[\mathtt{at}\,\rho]\ \mathtt{in}\ e\ \mathtt{end}]} \Rightarrow E[\mathtt{letrec}\ f[\rho](x)\ a_0{=}R[\mathtt{attop}\,\rho]\ \mathtt{in}\ e\ \mathtt{end}] \quad (3)$$

$$\frac{\rho \notin LR(R) \quad \langle\rho\rangle \cap \langle LE(R)\rangle = \emptyset}{E[\mathtt{letrec}\ f[\rho](x)\ a_0{=}R[\mathtt{at}\,\rho]\ \mathtt{in}\ e\ \mathtt{end}]} \Rightarrow E[\mathtt{letrec}\ f[\rho](x)\ a_0{=}R[\mathtt{sat}\,\rho\ \mathtt{unless}\ CS(R,\rho)]\ \mathtt{in}\ e\ \mathtt{end}] \quad (4)$$

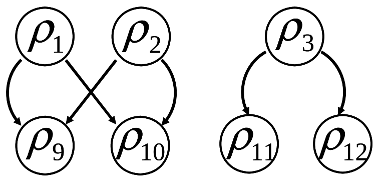

FIGURE 1. A region flow graph for the example program

$$\frac{\rho \text{ bound non-locally in } E[R]}{E[R[\texttt{at}\ \rho]] \Rightarrow E[R[\texttt{attop}\ \rho]]} \qquad (5)$$

3.3 Translation of the Example

Here we consider the example again. The region flow graph of the example is shown in figure 1. Isolated nodes are abbreviated to make the graph simple. When the value of the expression - x is stored in the region bound to ρ_1, the program variable - y is alive. Since the value bound to y is in ρ_2 and we can reach nodes ρ_9 and ρ_{10} starting from both ρ_1 and ρ_2 in the graph, we translate “- x at ρ_1” into “- x `sat` ρ_1 unless $\{\rho_2\}$”. Similarly, we translate “- y at ρ_2” into “- y `sat` ρ_2 unless $\{\rho_1\}$”. In this case, a live program variable is a.

3.4 Typical Case of Memory Usage Improvement

In this subsection, we consider more concretely when memory usage is improved by using our method. First of all, when a function f satisfies the next two conditions, the return value of f is allocated by the extended `sat` ρ directive.

1. When f allocates the return value in a region bound to ρ, there is another region variable ρ' bound to a region which might contain a live object.
2. It is possible that ρ and ρ' are bound to the same region.

The first condition is typically satisfied by functions which return objects including objects passed by their caller. For example, in a function “fun f x y = (x, y)”, when the tuple (x, y) is created, we cannot reset the region in which the tuple is to be stored because values bound to x and y are still alive. The second condition means that there is a node in the graph G which is reachable from both ρ and ρ'. This is typically caused by the next three cases.

- f is instantiated at more than two contexts and ρ and ρ' can be bound to regions introduced by the same `letregion` in different instances. For example, (f (f x)) is translated to (f [ρ_1, ρ_2] (f[ρ_2, ρ_3] x)). Here, the first region parameter is for the return value and the second region parameter is for the parameter. In this case, the region bound to ρ_2 is passed to the region parameter for the return value at the inner application of f as well as for the parameter at the outer application.

- There is an instantiation of f which may bind ρ and ρ' to the same region. For example, in an expression "if e then x else f x", values of x and f x are stored in the same region because of the constraint of typing. This means f must be instantiated with its input and output region parameter bound to the same region in the expression.

- f is defined at top level. Since functions defined at top level can be called by other compile modules, we cannot determine which region variables are bound to the same region. This means that each pair of region parameters has the possibility of an alias.

In the first case, ρ and ρ' are bound to different regions. In the second case, there can be other instantiations of f, and in those instantiations, ρ and ρ' may be bound to different regions. In the last case, there can also be instantiations of f which bind ρ and ρ' to different regions.

When a function f satisfies conditions 1, 2 and ρ and ρ' are not bound to the same region, memory usage is improved by runtime alias detection. Besides, when this function is repeatedly called, the effect of our method can be much larger.

Some programs are translated to programs which does not accumulate useless objects by optimizations such as unboxing and inline expansion even if they satisfy the conditions above, but this is not true of every program. For example, the figure 2 is a program which calculates interest needed to increase principal ten times larger in ten years. This program satisfies the above two conditions and *power* cannot be expanded because it calls itself recursively. The function *power* returns the value of "acc + 0.0", and when the result of addition is created, the variable acc is still alive. Although you may think it strange to add 0.0, this is necessary to store the values of "acc * r" in regions allocated within *power*. This kind of rewriting is often used for region inference system. Note that floating-point numbers are not unboxed in MLKit, and the value of acc and the return value of *power* can be stored in the same region, since *power* can be called from other compile modules.

The function *loop* calls *power* repeatedly and returns the return value of *power* finally. Therefore, *power* stores its return value in the region allocated outside of *loop* and this value is accumulated. However, our method enables to reset the region before allocation of the return value of *power*.

3.5 Code Generation

Allocation directives are used for two purposes in the language given 2.1. One is to specify how to create object, and the other is to pass the information as to whether a region is used by caller or not when region polymorphic functions are instantiated. Figure 3 and 4 show the code we generate in each case for the extended `sat` directive. Here, let $\mathrm{reg}_i (i = 1, 2)$ to be registers, and $[\rho]$ to be a pointer to the region bound to ρ, and *foreach* statement means the body is generated for each elements of the comparison set *CS*.

```
fun power 0 acc r = acc + 0.0
  | power n acc r = power (n - 1) (acc * r) r

val _ =
  let fun loop i =
    let val r = (1.0 + 0.0000001 * (real i))
        val x = power 10 1.0 r
    in
        if x > 10.0 then (r, x) else loop (i + 1)
    end
  in
    let val (r, x) = loop 0
    in
        print ((Real.toString r) ^ " ^ 10 = " ^
             (Real.toString x) ^ " > 10\n")
  end end
```

FIGURE 2. An example where memory usage is significantly improved

As you can see from the figures, both cases are the same until comparisons of regions. In case of allocation, when the region in which we are to allocate an object is different from any of regions in the comparison set, we reset the region and allocate the object, and otherwise we allocate the object without resetting. In case of instantiation, when the region in which we are to allocate an object is different from any of regions in the comparison set, we pass the flag as it is to the region polymorphic function, and otherwise we change the flag to `attop` expressing that this region might be used in the rest of the computation.

MLKit encodes this flag in the second least significant bit of the pointer to the region (zero expresses that the region might be used). Note that there is no overhead for the object allocation, when the flag meaning in use is passed from the caller because we check the flag before comparing regions and Tofte and Talpin system also needs this check.

MLKit analyze the size of each region after SMA, and by using size information we can see that some region variables in the comparison set never bind the same region as the region variables in which an object is to be allocated. This means we can remove those region variables from the comparison set.

```
      mov [ρ0], reg1
      bt $1, reg1
      jnc L1
      foreach ρ in CS{
          mov [ρ], reg2
          cmp reg1, reg2
          je L1
      }
      call resetregion
L1 :  call allocate
```

FIGURE 3. Code for allocation

```
      mov [ρ0], reg1
      bt $1, reg1
      jnc L2
      foreach ρ in CS{
          mov [ρ], reg2
          cmp reg1, reg2
          je L1
      }
      jmp L2
L1 :  btr $1, reg1
L2 :  push reg1
```

FIGURE 4. Code for region parameter passing

4 EVALUATION

We implemented our scheme in MLKit 4.1.4, which is an implementation of Tofte and Talpin system and evaluated memory usage and some other features for several benchmark programs *dangle, Knuth-Bendix, FFT, Mandelbrot,* and *compound interest*. *Compound interest* is a program explained in previous section. The other benchmark programs are supplied with MLKit.

Table 1 shows the number of elements in the following set of `sat` directives. This table says that the number of element is less than two for the programs we used for evaluation. Table 2 shows the number of comparisons, resets executed during the execution and an increased ratio of resets. Since there is no extended `sat` directive inserted in *Mandel*, there is no comparison or increased reset. *Knuth-Bendix* has extended `sat` directive, but they have little effect on the number of resets. There is a large increase in the number of resets for the other three programs. *Compound interest* has an extended `sat` directive, but it does not make comparisons, because we can remove some region variables from the comparison set as we explained in subsection 3.5.

Figure 5 shows maximum memory size and execution time compared with those of SMA. Memory usage is improved for *compound interest* and *dangle*. *compound interest* accumulates useless objects in proportion to the number of calls of loop in SMA. However, those objects are deallocated within each call in our method. As a result, *Compound interest* which uses too much memory in original SMA can be executed with reasonable memory usage (in the figure, less than

TABLE 1. The number of elements in the following set of `sat`

number of elements	0	1	> 2
dangle	0	1	0
Knuth-Bendix	11	7	0
FFT	15	7	0
Mandelbrot	0	0	0
compound interest	1	0	0

TABLE 2. The number of comparisons and increased percentage of reset

	# of comparisons	# of resets (SMA)	# of resets (Our Method)	increased ratio of resets
dangle	3,000	3,006	6,006	99.8%
Knuth-Bendix	33,134	2,916,741	2,920,233	1.2%
FFT	1,245,184	262,147	655,362	150%
Mandelbrot	0	6	6	0%
compound interest	0	2,589,256	5,178,512	100%

0.03%). For *Knuth-Bendix* and *FFT*, memory usage is improved but there is no effect on the maximum size.

Execution time is a little larger than SMA for most programs because of the overhead of comparisons and resets. The overhead of resets is considered to be dominant. This is because a comparison of regions can be done by only a few assembly instructions, but the reset of a region calls a function implemented by C-language and it needs comparatively heavy work including management of pages of which regions are composed. Though the more elements are in the comparison set, the more time the comparison takes, the number of elements in the comparison set is not so large generally as the table 1 shows.

5 RELATED WORK

Aiken et al. [1] extended the Tofte/Talpin system, decoupling dynamic region allocation and deallocation from the introduction of region variables with the `letregion` construct. As a result, they realized better memory behavior and enabled the tail call optimization for more programs.

Our method is an improvement of SMA, and it doesn't deallocate regions themselves earlier. Therefore, we cannot expect programs which keep allocating new regions during continuing function calls to be optimized, but we can expect more

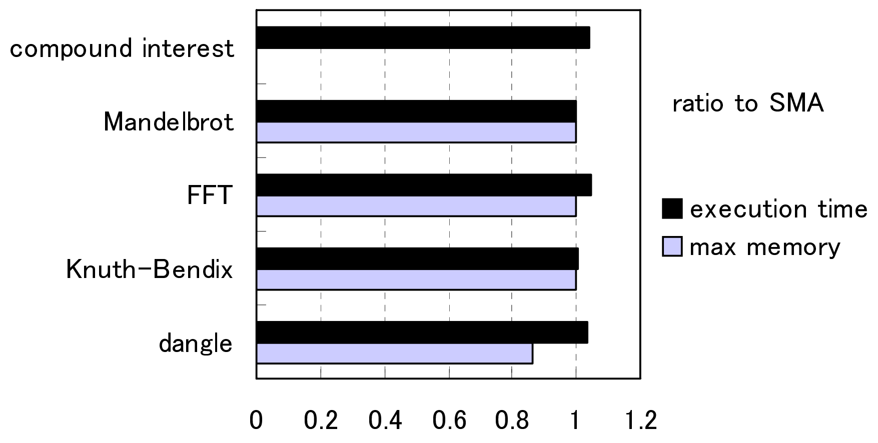

FIGURE 5. Maximum memory size and execution time

programs which keep adding objects in some regions to be optimized.

Our method is also orthogonal to the AFL system as well as SMA. The AFL system uses an abstract region environment to analyze the lifetimes of regions, which has more information about region aliases than the region flow graph. Therefore we think that we can refine our method by using this abstract region environment. With the abstract region environment, region aliases can be identified more accurately. Actually, in the example given in section 1, because there is no instantiation of the region polymorphic function `f` which binds ρ_1 and ρ_2 to the same region in the program. Therefore, we can see by static analysis that ρ_1 and ρ_2 are never bound to the same region. Even if so, our method is still useful, because programs are often compiled separately and when compiling, top level functions always have the possibility of problematic instantiations in other compilation modules.

6 CONCLUSION

We proposed a technique to improve memory usage in systems using static regions as a method of managing memory. In our technique, objects can be created by overwriting at some points where overwriting was impossible by the original Tofte/Talpin system because of the possibility of aliases of the target region. We also implemented our technique to MLKit, which is a system with region inference and is compliant with a subset of Standard ML and confirmed that our technique really improves memory usage for some programs. By this fact, we expect more programs will run with reasonable memory usage.

REFERENCES

[1] Alexander Aiken, Manuel Fahndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1995.

[2] Lars Birkedal, Mads Tofte, and Jean-Pierre Talpin. From region inference to von neumann machines via region representation inference. In *the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1996.

[3] Martin Elsman and Niels Hallenberg. An optimizing backend for the ml kit using a stack of regions. Technical report, Department of Computer Science, University of Copenhagen, July 1995.

[4] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[5] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, 1998.

[6] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, October 2001.

[7] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.