Creating Efficient Systems for Object-Oriented Languages

Norihisa Suzuki and Minoru Terada

The University of Tokyo
7-3-1 Hongo, Bunkyo-ku,
Tokyo, Japan 113

ABSTRACT

Increasingly computer science research has been done using workstations with high-resolution bitmap display systems. Smalltalk-80† is a very attractive programming language for such computation environments, since it has very sophisticated graphical systems and programming environments. Unfortunately there are still very few computer systems on which Smalltalk-80 can run with satisfactory speed, and furthermore they are quite expensive. In order to make Smalltalk-80 accessible to a large group of people at low cost, we have developed compiler techniques useful to generate efficient code for standard register machines such as MC68000. We have also extended Smalltalk-80 to include type expressions, which allow compilers to generate efficient code.

1. INTRODUCTION

More and more computer science research has been conducted on high-performance personal workstations. The reasons are that we can in general get more computation power per person if we use workstations instead of crowded time-sharing systems, and workstations generally have bitmap display screens that allow us to communicate with computers with much higher bandwidth.

There are, however, very few excellent programming systems that utilize the full capabilities of the workstations. Smalltalk-80 is one of such programming systems; it has a number of attractive features for experimental programming such as polymorphism, late binding, and object-oriented system structures.

Smalltalk-80 has only been successfully implemented on a very few computer systems, most of which are microcoded machines. These microcoded machines are generally very expensive and only a very few researchers can afford them. In order to make Smalltalk-80 programming systems widely available, they have to run on standard microprocessors such as MC68000. As the preliminary process for releasing Smalltalk-80, Xerox licensed Smalltalk-80 to a number of companies, which tried to implement it on a standard machines such as VAX and MC68000 based machines. These experiments [6] were generally very disappointing.

Therefore, we decided to put our efforts in generating efficient systems on MC68000. Even though most of the attempts in [6] to put Smalltalk-80 on conventional computers are not successful, they have obtained extensive performance measurements, which are very useful to the people like us who want to implement Smalltalk-80 systems.

They have found out that there are three major bottlenecks in the Smalltalk-80 systems that have been implemented according to the Virtual-Machine specification written in [4]; the bottlenecks are caused by the necessity to find the method, which corresponds to a procedure, for each message send, which corresponds to a procedure call, dynamically, by the cost of context allocation and deallocation since the contexts, which are the execution environments of procedures, are allocated on the heap, and by the cost of reference counting garbage collection used for memory management.

We have invented techniques to solve these problems; we allocate contexts on the stack as long as the retention of contexts is not necessary, we elide much of reference counting by using a transaction-based garbage collection, and we avoid method search by compare and branch and link instructions exploiting some statistical evidence.

We have also concluded that type information is absolutely necessary for both documentation and generating efficient code. Type information may be obtained both by declarations [2] and compiler inference [7]. We allow the programmer to supply as many type declarations as they wish. This new language and the system is called Kiku (chrysanthemum).

We will describe the problems and the outline of the techniques to solve these problems in section 2. In section 3 we describe the language features of Kiku. In section 4 we describe the details of the compilation techniques.

† Smalltalk-80 is a trademark of Xerox Corporation.

2. BOTTLENECKS AND SOLUTIONS

There are three major performance bottlenecks in the Smalltalk-80 systems that are implemented according to the Virtual Machine specification written in [5]. They are the dynamic method search, the context allocation in the heap, and the reference counting garbage collection.

2.1. Dynamic Method Search Overhead

The primitive computation mechanism of Smalltalk-80 is called a message send. For example, an expression to tell a table to be moved from location locA to location locB is

table moveFrom: locA to: locB

where table points to an object, which is called the receiver of the message, and moveFrom: locA to: locB is the message sent to table. The entire expression is called a message send. The concatenated string moveFrom:to: is called the message selector. The program that is going to be activated for this message send is not determined at compile time.

These special concepts and jargons of Smalltalk-80 can be described in terms of ordinary programming language concepts. A message send is a procedure call, a receiver of a message is the first argument of a procedure call, a message selector is a procedure name, and a method is a procedure. Therefore, the above Smalltalk-80 message send can be written as

moveFrom:to:(table, locA, locB)

in Pascal.

The binding between a message and a method is done at run time. Therefore, depending on the classes of the receivers of the messages, the messages are interpreted differently.

Each object belongs to a class, which is a specification of a group of objects. In particular a class has a method dictionary, which is a hash table with the message selector as a key and the method body as a value. When a message is sent to a receiver, the method dictionary of the class of the receiver is searched with the message selector. If it is found in the method dictionary, the method body that is the value of the message selector is invoked. If it is not found, then the superclass of the class of the receiver becomes the class to be searched, and the method dictionary is searched again. Thus, the method body to be invoked for a message send is determined dynamically.

The method search is optimized in Smalltalk-80 Virtual Machine by providing a large hash table. For every message send a hash key is created from the message selector and the hash table is looked up. Each entry in the hash table has two values: the class and the method body. If an entry with the same selector exists, the class is compared with the class of the receiver. If it is the same, the method body is activated.

Another optimization in the current Smalltalk-80 Virtual Machine is to speed up the method search for special messages such as arithmetic operators. When an arithmetic operator + is sent, the probability that the class of the receiver is Integer is very high. When the bytecodes for arithmetic messages are executed, the classes of the receivers are first checked. If they are Integer, addition is performed without invoking a message send.

Our strategies are the generalization of this philosophy. We assume that in each message send classes of the receivers that the receiver expression denotes are very unevenly distributed. This assumption is confirmed by many statistics taken at Xerox PARC, at UC Berkeley, and by us.

A special case of this is that an expression always denotes an object from a unique class. In such a case, we can statically bind the message send and the method so that a message send can be a simple branch and link operation. However, without a powerful type inference algorithm, there is no way for a compiler to predict that an expression always denotes objects of a unique class. Therefore, we are introducing type declarations. We specify the classes of variables, methods, and blocks. By these declarations we can tell the classes of expressions, and tell the methods involved at compile time.

We are creating special object code if the messages are sent to pseudo-variables. Pseudo-variables are identifiers that always denote some special objects. In Smalltalk-80 *self* is a pseudo-variable that denotes, within the method body, the receiver of the message that has invoked the current method. A variable *super* is another pseudo-variable that again denotes the receiver of the message that has invoked the current method, but forces the method search from the superclass of the class where the method appears.

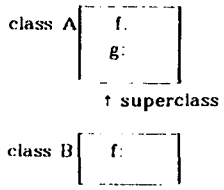
If a message is sent to *self* or *super*, it is relatively straightforward to find the corresponding method at compile time. Suppose a message

self f: arg

appears in the class A, then it is easy to think that f: has to invoke a method in the class A, or in the superclasses of A if f: does not appear in A. This is, however, not true. Suppose this message is defined in the method body of g: in the class A, as

g: aB || self f: arg

and this method is invoked by sending a message g: to an object of the class B that is a subclass of A, but g: is not defined in B.



Then g: invoked is the method in class A, but the self in the body of g: denotes the object of class B, so f: invoked is the one in class B. Therefore, if f: is defined in the subclasses of class A, it is not always true that f: sent to self invokes the method defined in A or in the superclasses of A.

The compiler strategy is, therefore, as follows. Any messages sent to self are compiled to direct branch and link instructions, if the methods are not defined in the subclasses. Whenever a method with the same name is defined in a subclass, all the message sends with the same message selector to self in the superclasses have to be recompiled.

On the other hand message sends to super can be compiled easily. Suppose there is a message send

super f: x

in the class A. Then f: has to be in the superclasses of A. We can trace the superclass chain starting from the superclass of A at compile time, and can find the location of the method f:.

As we explained before, in order to generate efficient code for message sends to *self* we need to maintain a database of methods and to perform recompilation time to time. This can be done, but may require a lot of computation. We, therefore, introduced another pseudo-variable *here*, which stands for the previous receiver just like *self* or *super*, but forces the method search to start from the class where the pseudo-variable *here* appears. We can statically determine the binding and do not have to recompile.

The third technique to speed up method invocation is to exploit statistical facts. It was reported by the measurements at Berkeley that it is very likely that the message sends in a class will invoke methods in the same class. We describe the code generation techniques that exploit this fact in section 4.

2.2. Context Allocation and Deallocation Overhead

In Smalltalk-80 a local execution environment created every time a method is invoked is called a context. It contains a pointer to the previous context called a caller, a pointer to the object code of the method, an instruction counter, parameters, local variables, which are called temporary variables, and an evaluation stack. These information are stored in the execution stack in the implementation of ordinary languages, but in Smalltalk-80 contexts are treated as objects and allocated in the heap.

There are a number of advantages for doing this way. Complex control structures such as processes and coroutines can very easily be implemented because contexts are objects. Utility systems such as a debugger can be easily implemented, because the history of the execution can be easily traced and manipulated by sending messages to contexts.

However, all the overhead associated with object creation and destruction has to be paid by a method activation and return. For each method activation, a storage area with the size of a context is allocated, and the reference count is incremented; for each method return the reference count is decremented and the storage is returned to the free list.

It was reported that more than 80% of objects allocated are contexts in the Smalltalk-80 Virtual Machine implementation [6]. Therefore, the overall overhead associated with context allocation and deallocation is tremendous.

Our philosophy in solving this problem is to allocate contexts linearly in the stack as long as possible. When it becomes necessary to treat the current context, which is the top context in the stack, as an object, or to change the current context because of a process switch, we make all the contexts in the stack objects in the heap.

2.3. Reference Counting Overhead

Automatic allocation and deallocation of objects is a must for easy to use object-oriented languages. However, the ordinary mark-and-sweep garbage collection has a long pause when the garbage collector starts, and is not suited for real-time, interactive applications.

Smalltalk-80 Virtual Machine uses reference counting garbage collection in order to improve the real-time response. Reference counting garbage collection generally performs better than mark-and-sweep garbage collection since it requires less disk I/O.

However, the overhead associated with reference counting is substantial. In a stack machine like Smalltalk-80 Virtual Machine most of the operations are stack pushes and pops. In order to push an object, the reference count of the object must be incremented, and in order to pop and object the reference count must be decremented. According to our calculation, 80% of the computation time for pushes and pops spent in reference counting.

We reduce the reference counting overhead by not counting if the pointers are in the stack. This is a modification to Deutsch and Bobrow [3] transaction garbage collection algorithm. The details will be explained in a later section.

3. LANGUAGE FEATURES OF KIKU

The syntax of Kiku is similar to Smalltalk-80; however, we felt that parentheses-less syntax of message send expressions is very confusing if message sends are nested. Therefore, we adopted conventional notations with parentheses. However, our compiler will accept both syntax.

The syntax for a message send is

<expression> ! <selector> [(<parameter list>)]

where the <selector> is an identifier and the <parameter list> has the following BNF-syntax:

<parameter list> ::= <parameter>
<parameter list> ::= <parameter> , <parameter list>

3.1. Typing

Some work have been done to introduce types to Smalltalk. Borning and Ingalls [2] introduced type expressions; their compiler understands types and checks type consistency, but it does not use the information for

code generation. Suzuki [7] has devised an algorithm to infer types from the completely untyped programs. His goal was to create efficient code as well as to locate potential bugs by type inference. This research, however, showed that in the language like Smalltalk-80 that allows union types, many inferences are not tight enough to be useful for generating optimized code. We still believe, however, that in Smalltalk-80 the ultimate optimization for message sending can only be attained by knowing the classes of the expressions at compile time either by type inference or by explicit typing. So we decided to introduce type expressions into Smalltalk-80.

Smalltalk-80 gets much of its power from the type hierarchy system, polymorphism, and function overloading. We would not like to destroy the flexibility by introducing a rigid type system. Therefore, the declarations are optional in our system; objects still carry type codes with them. Type expressions are decorations to the program. They are used by an optimizing compiler to create a tightly bound code.

The locations where we put type expressions are with variable declarations, method declarations, and block statements.

3.1.1. Type Expressions

A type expression can be a class name or a set of class names. A set of class names is used to denote a union type. To be precise, a type expression is a name of a class, or subclasses of a class, or a union of them. BNF-syntax for type expressions are:

```
<primitive type expression> ::=
        <class name> | sub <class name>
<type expression> ::= <primitive type expression> |
        (<primitive type expression> , ... ,
         <primitive type expression>)
```

3.1.2. Type Designations

We associate type expressions with variable declarations, method declarations, and block statements. We call the declarations with type expressions type designations.

We first explain the syntax for type designations by some examples. A declaration of Class Array is:

```
Class Array
   cv -- list of class variables --
   cm -- list of class methods --
   iv -- list of instance variables --
   im at(index: SmallInteger) ||
         -- method body to extract an element
         -- of the array
     -- the rest of instance methods.
```

Here Array is the class name. The header im marks the beginning of the declarations of instance messages. An instance message "at" is declared with type designations. It takes a parameter "index" of type "SmallInteger", which declares that "index" can only denote an instance of Class SmallInteger. In addition to being a class, SmallInteger serves as a type expression.

We may want to declare the class of the value returned by this call, which is the class of the element of the array. However, Smalltalk Array can take any objects as its elements; therefore, we cannot specify the resulting type of at. The best we can do is to specify that the result will belong to a subclass of Class Object:

```
at(index: SmallInteger): sub Object,
```

which is equivalent to have not type expression at all. In order to create an array which takes objects of one class we have to define special classes such as Class SmallIntegerArray, and specify that the result of array access is of Class SmallInteger and that the value to be assigned in the array assignment is of Class SmallInteger.

However, since the number of classes that may be defined is unbounded, it is not practical to introduce array classes for all the element classes. Therefore, we have to introduce parameterized classes in order to cover all the cases. Then the result type of at can be defined by the parameter of the class name used when the object is created. We have, however, not implemented parameterized classes, because we have to introduce new mechanisms to the compiler and the run-time system. In the near future we will be incorporating this feature.

BNF-syntax for type designations are:

```
<class variable declarations> ::=
      cv | cv <variable designations>
<variable designations> ::=
      <typed variable> |
      <typed variable> <variable designations>
<typed variable> ::=
      <identifier> |
      <identifier> : <type expression>
<instance variable declarations> ::=
      iv | iv <variable designations>
<class method declarations> ::=
      cm | cm <method declarations>
<method declarations> ::=
      <method header>
      | <variable designations> | <expression list>
```

where the regular vertical bar (|) is a metasymbol denoting an alternative choice, and the bold vertical bar (|) is a symbol that appears in Smalltalk-80 programs. Blocks also have type expressions, since they receive arguments and returns results. The syntax is:

```
<Block> ::= [ <expressions list> <result type> ] |
      | <parameter declarations>
      | <expression list> <result type> ]
<parameter declarations> ::=
      <parameter> |
      <parameter> , <parameter declarations>
<parameter> ::=
      : <identifier> |
      : <identifier> : <type expression>
<result type> ::=
<result type> ::= : <type expression>
```

3.2. Pseudo-variables

There are two variables, "self" and "super," which have special meanings in Smalltalk-80. "Self" points to the receiver, when the method is executed. For example, if we use recursion to implement multiplication, it can be written as:

```
multiply: n ||
  (n = 0) ifTrue: [↑self].
  ↑(self + (self multiply: (n-1)))
```

in Smalltalk-80. Then, we execute 5 multiply: 3 to obtain 15. The same method is written in Kiku as:

```
multiply(n: sub Integer) ||
  (n = 0) ! ifTrue([↑self]).
  ↑(self + (self ! multiply(n - 1)))
```

The method to be invoked for the message sent to self cannot be determined statically if the method is redefined in the subclasses because of the reasons explained in section 2; thus, we have still to use slow method look up. We introduced a pseudo-variable "here" to force the search from the class where the message send occurs. Thus, the static binding is always possible.

3.3. Multiple Value Returns from Methods

It is important for any programming languages to have multiple-value-returning functions to simplify programs and to make programs efficient. PASCAL allows results to be returned through parameters, thus multiple value returns are possible.

It is difficult to introduce this feature into Smalltalk-80, since we cannot tell whether a particular message returns a single result or multiple results because messages are overloaded.

What we did was to introduce a different concept and solved this problem. We introduced array constructors and array extractors. When we want to return multiple values, we write

```
↑.(x, y)
```

which creates a two-element array with x and y as elements and returns it. We can obtain multiple results from a message by using an array extractor as follows:

```
(x, y) ← point coordinates
```

The expression point coordinates returns a two-element array, and the first element is assigned to x and the second element is assigned to y. Semantically this solution is different from the multiple-value-returning functions, but the syntax is very similar.

This feature has created a nice side-product. We often nest the calls such that multiple values have to returned up several levels. If we have not treated results as one object, we may have to extract results and construct results repeatedly. For example, we can compute the remainder and quotient simultaneously by the following method:

```
divMod: n || ↑self divMod: 0 by: n
divMod: q by: n ||
  (self < n) ifTrue: [↑.(q, self)].
  ↑(self - n) divMod: (sum + 1) by: n
```

The last call returns multiple results as an array, so they are returned through all the levels of divMod:by: without disassembling and assembling.

4. CODE GENERATION AND RUN-TIME TECHNIQUES

In this section we describe the code generation and run-time system implementation techniques we are using to solve the system bottlenecks described in section 2.

4.1. Avoiding the Dynamic Search of Methods

In Smalltalk-80 a message and a method are bound dynamically. The basic mechanism for binding a message and a method is defined in the book [5]. The expression is of the form

```
r sel: e
```

where r is a receiver, and sel: e is a message. First, sel: is looked up in the method dictionary of the class of r. If it exists then the method is executed with the supplied arguments. If it does not exist, then the dictionary of the superclass of the class of r is searched. This process of looking up in the successive superclasses is repeated until we come to a class which does not have a superclass, and then the message produces a dynamic error. Our code generation technique for the message is to speed up by avoiding the search whenever possible.

4.1.1. Arithmetic and Other Primitive Methods

In Smalltalk-80 arithmetic operators such as + are overloaded just as all the other methods. However, it would be very inefficient to use dynamic binding for common arithmetic operators. The probability that the receivers of the arithmetic operators are integers is higher than 95% [6]. Smalltalk-80 bytecodes for arithmetic operators exploit this fact; they first check the classes of the receivers. If they are class SmallInteger, arithmetic operations are executed. We use this technique to create open compiled code. We dedicate one register receiverClassReg to hold the class of the receiver, and the code is

```
if receiverClassReg = Class SmallInteger
      then branch(GeneralSearch)
      else <in-line code for integer addition>.
```

4.1.2. Techniques Useful for General Messages

For all the other messages we use the following statistical facts. More than 90% of the messages invoke methods that are in the class where the messages are written. We can exploit this fact to produce the following code.

```
if receiverClassReg = methodClassReg
      then branch(GeneralSearch)
      else <jump to the place where (sel:) is defined>.
```

where receiverClassReg is a dedicated register which always holds the class of the receiver and methodClassReg holds the class of the method, that is the class where the message is written. Then the statistics tells us that the last leg of the conditional is taken more than 90% of the cases; thus, we can eliminate most of the expensive method look up.

4.1.3. Messages Sent to Pseudo-variables

In Kiku there are three pseudo-variables "self", "super", and "here". As explained in section 2, messages sent to "super" and "here" can be bound statically. Messages sent to "self" can be statically bound, if no methods with the same selector are defined in the subclasses.

4.1.4. Messages Sent to Typed Expressions

The effects of type designations are that we will guarantee that the objects that are pointed to by the typed variables are instances of the classes that the type expressions denote. When a message is sent to a typed variable, we know exactly which method we should invoke. If a variable is declared to be of type Integer, the method to be invoked must be declared in the class Integer or in its superclasses. If a variable is declared as a union of Integer and Float, the object code created is

```
if receiverClassReg = Class Integer
        then <directly call a function in Class Integer>
        else <directly call a function in Class Float>,
```

where receiverClassReg is a register dedicated to hold the class of the receiver.

4.2. Linearization of Contexts

Procedure activation environments, which are called contexts in Smalltalk-80, are also objects. Therefore, contexts can be assigned to variables and be sent messages. Smalltalk-80 gains much power and flexibility because of this feature. Various control mechanisms like coroutines and processes are implemented using this mechanism; various important system software such as debuggers can be very easily implemented. However, this mechanism disallows the use of a simple linear stack to allocate contexts. Therefore, most of the existing Smalltalk-80 implementations allocate contexts from the heap. However, contexts are reference-counted and garbage-collected, thus, substantial part of the execution time is spent in contexts allocation and deallocation. In Smalltalk-80 Virtual Machine 83% of objects allocated are contexts (MethodContext) [6].

One candidate data structure for contexts is a spaghetti stack [1]. It is used in Interlisp to realize processes and coroutines. However, this mechanism is not flexible enough for Smalltalk-80 which treats contexts as objects.

We invented a mechanism called a "delayed retention" of contexts. We have a fixed size area to be used as a stack. This area may be able to hold at most 10 contexts. When a new context is necessary, it is always allocated on this stack. As long as a context is on the stack, it is not an object; it is not reference-counted, and it cannot be sent messages. Furthermore, all the contexts in the stack are linearly ordered so that the callee is always on top of the caller. Therefore, when a method returns and there is a context in the stack, context deallocation is simply a stack pointer change.

The underlying assumptions for this mechanism to work much more efficiently than ordinary heap allocation are that message sends and returns alternate frequently, so that it is rare that many contexts are allocated without any deallocation in the middle. Furthermore, operations that require context retention such as creation of Block contexts and message sends to contexts occur quite infrequently so that if we allocate contexts in a stack, we rarely have to reorganize the stack.

Therefore, contexts on the stack behave just like frames in the Pascal stack. In order to maintain this property we have to perform some operations. When the stack becomes full, contexts have to be swapped out in FIFO manner to make room for new contexts. Space are allocated in the heap to hold the contexts being swapped out. However, the contexts are not swapped into the stack from the heap even though the stack becomes empty by the message return from the last context in the stack. Therefore, the current context can be either on the stack or in the heap.

When a context is needed to be retained, all the contexts in the stack are swapped out from the stack. Then the current context is pushed on the stack. This always occurs with the execution of the bytecode pushActiveContext, we only need to modify this bytecode. This is the reason why we limit the size of the stack. We can put an upper bound on the pushActiveContext execution time.

When there is a process switch all the contexts in the stack have to be swapped out also, since a process switch may violate the property that the callee is on top of the caller in the stack.

Whether this technique is practical or not depends on the percentage of contexts that must be retained among all the contexts. This can be estimated by the number of contexts allocated on the stack (MethodContext) versus the number of BlockContexts, which have pointers to them. This ratio is 20 to 1 [6], so we predict that retention is required rarely.

4.3. Improving Reference Counting Garbage Collection

Smalltalk-80 uses reference counting garbage collection in order to improve the real-time response. However, it slows down the average operations significantly. Consider, as an example, the most frequently used bytecode "PushTemporaryVariables," which obtains a local variable and pushes it onto the stack. This simple operation requires two reference counting operations: first, the reference count of the object pushed on the stack must be incremented, then the reference count of the object replaced is decremented. In our implementation on MC68000 reference count up and down take 10 instructions each. Since obtaining the value and pushing it on the stack takes 5 instructions, 80 % of the time is spent in reference counting.

The key observations are that most of the objects are short lived, and that the references to them are only from stacks or from some system defined variables. Therefore, we try to eliminate reference counting if the objects are pointed from the stack. The statistics on four major stack bytecodes are:

	Relative dynamic frequencies	Number of Ref counts
PushTemporaryVariables	20	2
PushReceiverVariables	7	2
PopAndStoreTemporaryVariables	4	2
PopAndStoreReceiverVariables	2	2

If no reference counting is done from the stack

	Number of Ref counts
PushTemporaryVariables	0
PushReceiverVariables	0
PopAndStoreTemporaryVariables	0
PopAndStoreReceiverVariables	2

Simply not counting does not work, because an object may be pointed from some other object, too. Even if the reference count of an object is decremented to 0 it may not be deallocated, since the pointer from the stack still exists.

So what we will use is a variation of a transaction-based garbage collection by Deutsch and Bobrow [3]. If a reference count becomes 0, the pointer is stored in a ZCT (zero count table), and the correct reference counting is done at the end of a transaction. The entire algorithm is:

(1) The transaction begins: ZCT contains all the objects whose reference counts are 0 but are pointed from the stack. These are the only objects contained in ZCT.

(2) If the reference count becomes 0, the pointer is stored in ZCT.

(3) Pointer operations to push and pop or copy from the stack are not counted.

(4) We complete a transaction at some point, either because it runs too long since the beginning of the transaction, or ZCT becomes full.

At the completion of the transaction, we scan the stack and put all the pointers in a hash table. For each pointer in ZCT, check whether the reference count is zero. If it is not zero, just remove it from ZCT. If it is still zero, check whether it is in the hash table. If so, just leave it in ZCT. Otherwise deallocate.

Now let us consider the performance. According to our implementation, push and pop take 5 steps in all the cases, the reference counting 10 steps, and ZCT store 5 steps. Then,

	Steps for the New Method	Steps for the Old Method
PushTemporaryVariables	5	25
PushReceiverVariables	5	25
PopAndStoreTemporaryVariables	5	25
PopAndStoreReceiverVariables	$25 + \alpha*5$ (2 ref count, 1 possible ZCT store)	25

where α is the ratio of decrements which result in zero reference count. Empirical data is obtained for this ratio is $\alpha = 0.04$ [6]. Therefore, the overall improvements excluding the transaction close procedure can be computed by multiplying the number of steps in the above table by the relative frequencies of operations in the previous table

$$\frac{20*5 + 7*5 + 4*5 + 2*(25 + 0.2)}{(20 + 7 + 4 + 2)*25} = 0.25$$

We can estimate the time for creating the hash table and scanning ZCT at the end of a transaction from the ratio of the dynamic frequencies of pushes and pops against the storage creation which is the upper bound of the number of contexts created. From the book [6], pushes and pops are 55% of the bytecode, whereas the message send bytecode for "new" is 0.35%. So the transaction close operation overhead is negligible compared with the reference counting operations eliminated from pushes and pops.

5. CONCLUSION

We extended Smalltalk-80 to create Kiku; we described features of Kiku which help to generate efficient object code. We are creating an optimizing compiler and a run-time system for Kiku. Three major performance bottlenecks of Smalltalk-80 systems are: method search, context allocation and deallocation, and reference counting garbage collection. We invented compiling and run-time system construction techniques to solve these problems. We analyzed the performance of these techniques using statistics available on the current Smalltalk-80 implementation and concluded that each of the technique improves the system performance significantly.

We learned from Peter Deutsch [4] that he has created a native code compiler for MC68000, which runs very fast. Alan Borning told me that it is very efficient and uses techniques similar to ours. However, we did not get a copy of Deutsch's paper in time to compare with our techniques.

We are grateful to the comments by Ichiro Ogata who is another implementer or the compiler, and Alan Borning.

BIBLIOGRAPHY

[1] Bobrow, D. and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments," CACM 16, 10 (October 1973), pp.591-603.

[2] Borning, A., and Ingalls, D., "A Type Declaration and Inference System for Smalltalk," Proceedings of POPL, ACM, 1982.

[3] Deutsch, P., and Bobrow, D., "An Efficient Incremental Automatic Garbage Collector," CACM, September 1976.

[4] Deutsch, P., Proc. of POPL, ACM, 1984.

[5] Goldberg, A. and Robson, D. "Smalltalk-80: The Language and its Implementation," Addison-Wesley, Reading, Massachusetts, 1983.

[6] Krasner, G., "Smalltalk-80: Bits of History, Words of Advice." Addison-Wesley, Reading, Massachusetts, 1983.

[7] Suzuki, N., "Inferring Types in Smalltalk-76." Proceedings of POPL, ACM, 1981.