A Value Profiler for Assisting Object-Oriented Program Specialization

Takahiro Kamio*
Graduate School of Arts and Sciences
University of Tokyo

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo
masuhara@acm.org

ABSTRACT

We present a value profiler for object-oriented programs that counts frequencies parameters to method calls. It is aimed at identifying methods that can be optimized by program specialization techniques. By adding timestamps to objects, the profiler accurately tests equality over mutable objects on a per-method basis. Our experiments with a 64602 lines of Java program showed that the profile report can reduce effort at manually finding the target methods of optimization, which speeded the overall execution time up more than 10%.

1. INTRODUCTION

This paper proposes a tool to help improving runtime performance of generic programs. Modern programming languages and programming techniques encourange to build generic programs for faster development and eaiser maintenance. Such languages and techniques include object-oriented languages, design patterns, component frameworks, and so forth. The basic idea of those languages and techniques is to extract common parts from similar systems, and to parameterize them so that they can be used in specific contexts.

Such generic programs are, however, less efficient in terms of runtime performance since they usually have to check parameters at runtime. For example, a generic GUI component (e.g., a button on a screen) can take many parameters to control its appearance (e.g., the background color and the thickness of the border). Even such parameters are not changed during executions of some programs, they have to be checked for each time the component is drawn on a screen.

In order to improve the efficiency of generic programs, several program specialization techniques are proposed, including partial evaluation[5, 10, 13, 18], data specialization[4, 12], and memoization[14]. A common idea behind those techniques is to replace a code fragment that frequently runs with the same parameters with a specialized code fragment that directly yields the result.

One of the difficulties of those techniques is to accurately specify a target code fragment in a program. This is because those techniques are effective to only the code taking the same parameters every time or sufficiently many times, and could even degrade performance when applied to inappropriate code. However, the values of parameters and their variations are runtime properties, and thus difficult to be reasoned about from a program text.

This paper presents a *value profiler* for object-oriented programs. Value profiling[1, 2, 20] is a technique to find out instructions that are frequently executed with the same values in an execution of a program. It is useful for applying instruction-level optimizations[16] and for specializing in a C-like programming language[15]. Our value profiler is different from the existing ones in that it profiles information at method call level, and it targets to object-oriented programs.

Our value profiler monitors an execution of a Java program, and shows a list of methods that are frequently executed with the same parameters. The report from our profiler is intended to be used with the report from the time profiler, which shows elapsed times of methods. Since the parameter values to a method mostly determines the behavior of the method[1], the profiler reports are useful for finding methods that can be optimized by specialization techniques. From our experiments to profile an execution of a Java program consisting of 1513 methods, the profiling report is estimated to be useful to find out the methods that can be optimized by memoization techniques. In fact, optimizing the methods improved performance of the program more than ten percent. Currently, we merely used the profiling report for manual optimizations, but it could be a good basis for building an automated system.

One of the main contributions of our profiler is that its ability to accurately handle object values. In object-oriented languages like Java, a value in an object field can be changed by an assignment operation. As a result, even when a single object is used as a parameter to method calls, the method could behave differently if the object had different values in its field. Our proposed profiling technique can distinguish objects that have different values in their fields by using timestamps.

The rest of the paper is organized as follows. Section 2 introduces an example program to be profiled and presents an optimization technique to the program. Section 3 presents a basic framework of our profiler and the definitions of equal-

*The first author is currently with Fuji Research Institute Corporation, reachable at takahiro_kamio@fuji-ric.co.jp.

In Proceedings of Workshop on New Approaches to Software Construction (WNASC 2004), Tokyo, September 2004.

[1]Values in global variables can also determine the behavior. Although our current profiler ignores global variables, it can straightforwardly profile global variables by regarding them as implicit parameters to every method.

```
class ASTObject{
  ASTObject parent;
  SourceLocation srcLoc;
  TypeDec getBytecodeTypeDec(){
    if(getParent() instanceof TypeDec)
         return (TypeDec)getParent();
    else return getParent().getBytecodeTypeDec();
  }
  int getSourceLine(){ return srcLoc.getLine(); }
  ASTObject getParent(){ return parent; }
  ...
}
class TypeDec extends ASTObject{ ... }
class SourceLocation {
  int lineNumber;
  int getLine() { return lineNumber; }
  ...
}
```

Figure 1: An Excerpt of AspectJ Compiler

ity over objects, which play crucial roles in the profiler. Section 4 presents implementation of the profiler. Section 5 evaluates the effectiveness and efficiency of the profiler by applying it to a practical software system. Section 6 discusses related work. Section 7 concludes the paper.

2. AN EXAMPLE TARGET PROGRAM AND OPTIMIZATION

2.1 Target Program

We chose the source code of the AspectJ compiler version 1.0.6 as a target of profiling and optimization. The compiler, consisting of 64602 lines of Java code, is widely used for developing various applications including commercial ones, is written in a reusable manner by exploiting the design patterns. Figure 1 is an excerpt of `ASTObject` class in the compiler, which is the common superclass for all classes of abstract syntax tree nodes. Each `ASTObject` object has a link to its parent node in an abstract syntax tree in `parent` field, and has information about the source code location in `srcLoc` field[2] The `getBytecodeTypeDec` method returns an enclosing type (e.g., class or interface) declaration of a node by recursively following the `parent` field until it finds a node of type `TypeDec`. The `getSourceLine` method returns the line number in the source program that corresponds to the `ASTObject` node.

2.2 Memoization of a Method

Method `getBytecodeTypeDec` can be optimized by the memoization technique[14]. Figure 2 shows the memoizing version of the `ASTObject`. It defines a hash table in class variable `cache` for recording a pair of the parameter and the result of the method[3]. When the method is called, it

```
class ASTObject{
  ASTObject parent;
  SourceLocation srcLoc;
  // a map to record the parameter-result pairs
  static HashMap cache=new HashMap();
  TypeDec getBytecodeTypeDec(){
    if(cache.containsKey(this))
      // return the recorded result if the cache
      // contains the value for this.
      return (TypeDec)cache.get(this);
    else {
      // perform original computation otherwise
      TypeDec r;
      if(getParent() instanceof TypeDec)
           r = (TypeDec)getParent();
      else r = getParent().getBytecodeTypeDec();
      cache.put(this,r); // record the pair
      return r;
    }
  }
  ...
}
```

Figure 2: `getBytecodeTypeDec` with Memoization

looks in `cache` for the return value of the parameter (i.e., the receiver object), and returns the result if there is. If there is not, it performs original computation, records the parameter and the result, and returns the result.

When the compiler runs with the memoizing version of `getBytecodeTypeDec` in the figure, it is 12.9% faster than one with the original version as we will see in Section 5.2.

The method `getBytecodeTypeDec` satisfies conditions that make memoization safe and effective. The method always returns the same result for the same parameter (i.e., the same receiver) because there is no code in the program that modifies the `parent` field after constructed an `ASTObject` node. The method takes the same parameters for sufficiently many times as will see in Section 5. As we discussed earlier, the latter condition can not be confirmed without running the program.

3. PROFILER

3.1 Framework of Value Profiler

We propose a value profiler for object-oriented programs, which monitors a program execution to show how frequently methods[4] in the program are called with the same set of parameters.

Figure 3 shows the framework of the profiler. The following three steps in the profiler generate the report:

1. It monitors an execution of a target program. When the program calls a method, it records the parameter

[2]The `srcLoc` field and related methods are added by the authors for the explanation purposes.

[3]Since the return value of the method depends only on the receiver object, we can use an instance variable instead of a global hash table in this case.

[4]Precisely, the profiler also reports on constructor executions. As they can be treated as in the same manner as method calls, we do not explicitly mention about constructors in the rest of the paper.

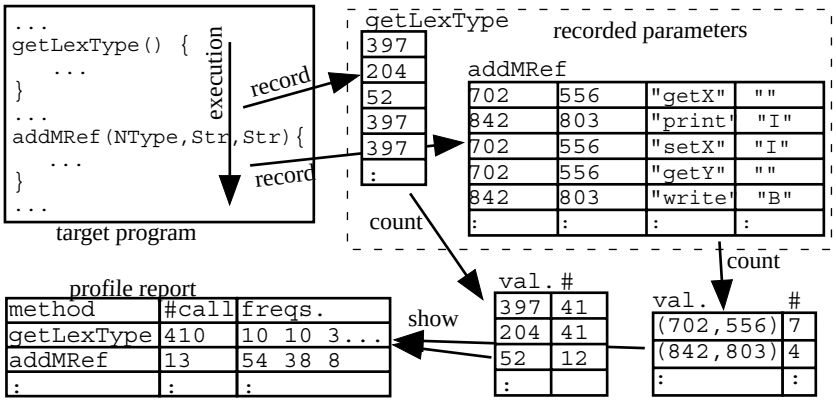

Figure 3: Framework of the Profiler

values into a parameter table of the method. The values include the value of the receiver object when it is a call to an instance method.

2. After finished execution of the target program, it counts the number of the same parameter values for each method. In other words, for each method, it forms all equivalence classes of parameter values, and then counts the sizes of the classes. This process may ignore parameter values at some positions, which is explained below.

3. It shows the methods with the frequencies of the same parameters. The frequencies are the sorted percents of the number of same parameter values counted in the previous step.

Figure 4 shows a part of the profile report on an execution of AspectJ compiler. Each row corresponds to a method in the target program, consisting of the method signature, the number of calls, and the frequencies of the same parameters. The underlined types in the signature denote that the parameters in those positions are used for counting the frequencies of the same parameter sets. A parameter frequency is the relative number of calls to the method that take the same parameter values with respect to the underlined parameter positions. For example, the first row shows that, among 410 calls to `ASTObject.getLexType`, 10 percent of the calls were performed on one receiver object, another 10 percent were on another object, another 3 percent were on yet another object, and so on.

The profiler ignores values at some parameter positions when a method does not have sufficiently high parameter frequencies. In other words, when it finds that the values at some parameter positions to a method that are always different, it tries to count parameter frequencies by excluding the values at those parameter positions. Since some specialization techniques can improve performance even with some stable parameters, this feature will extend optimization opportunities. In an execution of the target program, the parameter values to `CP.addMRef` are always different when we compare all four parameter positions. If we ignore the last two parameter positions, it turns out that there are only three pairs of values for the first parameter positions.

3.2 Equality between Object Values

We proposed a technique to test equality between object values so that the profiler can count the parameter frequencies even when a method takes mutable objects as its parameter. As mentioned above, a parameter frequency is the number of calls that take the *same* set of parameters to a method. It is thus important to have an adequate equality definition for profilers to generate useful reports.

Here we discuss four definitions to test equality between object values, namely (**R**) *referential equality*, (**S**) *structural equality*, (**MT+R**) *modification time and referential equality*, and (**MT+R/M**) *per-method modification time and referential equality*, which is our proposed one. Table 1 shows the properties of those definitions. There are two efficiency-related properties: how much memory space is required for recording one parameter at a call, and how much additional memory space is required for each object. There are also two precision-related properties: whether it can detect modifications on object fields, and whether it can ignore modifications on fields that are not accessed by a method. To summarize, our proposed definition realizes more precise equality with reasonable amount of memory requirements.

The properties of those definitions are discussed by using the following scenario.

> **An Example Scenario:** Assume that a code fragment runs with an object graph shown in Figure 5. The code calls methods `getSourceLine` and `getBytecodeTypeDec` to object `ast` before and after assignment to field `lineNumber` in object `srcLoc`, which is referenced from `ast`.

This example suggests an intuitive definition of equality; object values should be equal each other when fields that are accessed through the objects have equal values. For method `getSourceLine`, values of object `ast` are different before and after the assignment even though the fields of `ast` are not changed. This is because the field `lineNumber` of object `srcLoc`, which is accessed through object `ast`, gives different values. Conversely, for method `getBytecodeTypeDec`, values of object `ast` are equal because the fields accessed by `getBytecodeTypeDec` give the same value.

Below, we present the four equality definitions and how they distinguish the parameters in the example scenario. The definitions are given as the conditions to make values of o_1 and o_2 to be equal in a situation when a program calls method m with a parameter o_1, and then calls m with a

method signature	# calls	param. freqs.
ASTObject.getLexType()	410	10, 10, 3, ...
ASTObject.getBytecodeTypeDec()	275	25, 4, 3, ...
JComp.beginSection(String,boolean)	32	75, 25
CP.addMRef(NType,String,String)	13	54, 38, 8
NType.getLegalString()	10	40, 20, ...

Figure 4: Excerpted Profiler Report on an Execution of AspectJ Compiler (some method signatures are abbreviated)

Table 1: Properties of Equality Definitions (M denotes the number of the methods in a program. K denotes size of an object graph)

property \ equality definition	R	S	MT+R	MT+R/M
required memory for recording one parameter value (words)	1	K	2	2
additional memory to each object (words)	0	0	1	$\leq M$
can detect field updates	no	yes	yes	yes
can ignore unaccessed fields	no	no	no	yes

```
l1 = ast.getSourceLine();
t1 = ast.getBytecodeTypeDec();
ast.srcLoc.lineNumber = -1;
l2 = ast.getSourceLine();
t2 = ast.getBytecodeTypeDec();
```

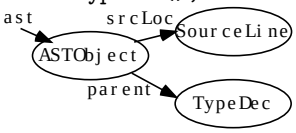

Figure 5: An Example Scenario that Manipulates an `ASTObject` Graph

parameter o_2. Note that o_1 and o_2 can be the same object.

3.2.1 Referential Equality

Values of objects o_1 and o_2 are *referentially equal* if and only if o_1 and o_2 have the same reference. It can be easily implemented by merely recording a reference to each parameter at method calls, and by comparing recorded references when counting frequencies.

An obvious problem of the referential equality is that it does not distinguish the values in object fields. In the example scenario, the receiver objects of the calls to `getSourceLine` have referentially equal values though the calls return different values.

3.2.2 Structural Equality

Values of o_1 and o_2 are structurally equal if and only if either of the following holds: (1) o_1 and o_2 are equal primitive values, or (2) o_1 and o_2 belong to the same class and have structurally equal values in each field.

Though it is powerful, this definition has two problems. First, a naive implementation for structural equality is not memory efficient. Since the profiler needs to compare values that are used at different time in an execution, it has to make a copy of all reachable objects from a parameter value at a method call, and has to compare recorded object graphs when counting frequencies. Since an object graph can be large, making a copy of an object graph is not acceptable in many applications.

Second, structural equality is too sensitive to the changes in object fields. In the example scenario, the receiver objects of the calls to `getBytecodeTypeDec` are not structurally equal due to the change in a field that is not accessed during `getBytecodeTypeDec`. The problem can be generalized as follows. Suppose an object has two kinds of fields, namely, unstable (i.e., frequently modified) fields and stable (i.e., rarely modified) fields. A method that accesses only the stable fields can be a good candidate for program specialization. However, the profiler that uses structural equality can not detect such a method because it regards that the method always takes different parameter values due to the modifications to the unstable fields. As practical programs tend to allocate various kinds of information into one object, this problem could frequently happen.

3.2.3 Modification Time and Referential Equality

By improving referential equality definitions, we can implement approximation of structural equality more efficiently.

We first define the *modification time of an object.* The modification time of an object o is the latest time when an assignment operation to a field of o is executed. The *modification time of the object graph from* o is the latest modification time in objects reachable from o.

Values of objects o_1 and o_2 are modification time and referentially equal if and only if o_1 and o_2 have the same reference and the object graphs reachable from o_1 and o_2, respectively, have the same modification time.

The modification time and reference equality approximates the structural equality. When all objects that are reachable from an object are not changed from time t_1 to t_2, the values of the object at t_1 and t_2 are modification time and referentially equal and also structurally equal. When two values are not structurally equal, they are not modification time and referentially equal as well. (On the other hand, values that are not modification time and referentially equal can be structurally equal.) In the example scenario, object `ast` does not give modification time and referentially equal val-

ues before and after the assignment because the assignment changes the modification time of the object graph from object `ast`.

The modification time and referential equality can be implemented (1) by adding a timestamp field to every object, (2) by updating the timestamp of an object when a field of the object is modified, and (3) by recording a reference to a parameter value and a modification time of the object graph for each parameter of a method call. When counting frequencies, the profiler only needs to compare the pairs of reference and timestamp to test the equality.

Note that the modification time and referential equality does not solve the second problem in structural equality, which is too sensitive to field assignments. The next definition solves this.

3.2.4 *Per-Method Modification Time and Reference Equality*

We propose an equality definition called *per-method modification time and reference equality*, which compares different set of fields in object graphs on a per method basis. Intuitively, two parameters o_1 and o_2 to method m are equal when they have the same values in the fields that are accessed during execution of m and methods called from m.

First, we define the *reference field set* of a method m as a minimal set of field signatures that contains all fields that can be referenced during any execution of m. For example, the reference field set of `getBytecodeTypeDec` in Figure 1 is {`ASTObject.parent`}, and the reference field set of `getSourceLine` is {`ASTObject.srcLoc`, `SourceLocation.lineNumber`}. Note that the reference field set can contain a field that are not directly referenced by the method itself.

Next, we extend the notion of modification time to each method. The modification time of an object with respect to m is the latest modification time when an assignment operation is executed to a field in the reference field set of m. The modification time of an object graph from o with respect to m is the latest modification time with respect to m among objects that are reachable from o by only following the fields in the reference field set of m.

The values of objects o_1 and o_2 are modification time and referentially equal *with respect to* m when o_1 and o_2 have the same reference, and the object graphs from o_1 and o_2 have the same the modification time with respect to m.

This definition effectively ignores the changes in the fields that are not accessed by a method. In the example scenario, the values of object `ast` before and after the assignment are modification time and referentially equal with respect to `getBytecodeTypeDec` but not with respect to `getSourceLine`. This is because the assignment to the field `lineNumber` of the object `srcLoc` updates the modification time with respect to `getSourceLine`, but not with respect to `getBytecodeTypeDec`. In other words, the equality gives more accurate results by ignoring the assignments to field `SourceLocation.lineNumber` that are not accessed by `getBytecodeTypeDec`.

The equality for a particular method can be straightforwardly implemented by merely restricting the timestamp manipulations to the fields in the reference field set of the method. The next section presents a more efficient implementation that can test equality of values with respect to any method in a target program.

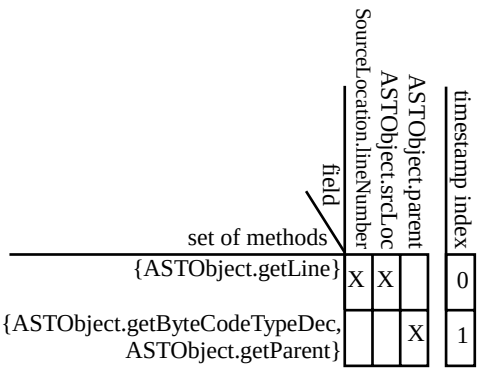

Figure 6: Compressed Timestamp Indices

4. IMPLEMENTATION

4.1 Overview

The profiler is implemented as an aspect in an aspect-oriented programming language AspectJ[11]. The AspectJ compiler, or weaver, takes a target program and the profiler definition as source code, and generates Java bytecode in which profiling code is inserted. Thanks to the AspectJ compiler, the profiler definition is compact (merely 1102 lines) and the compiled code runs on standard Java virtual machines.

The profiler runs a target program twice. The first run is for constructing the reference field sets of the methods. The profiler records the following two kinds of events in the target program:

- a call to method m from method m', and
- a reference to field f in method m.

After finished execution of the target program, the profiler can build a reference field set for each method. We chose to collect information by dynamic monitoring mainly for the ease of implementation. Future work is to employ static analysis here, which could give more robust information.

The second run is for recording the parameters of method calls. The profiler monitors the following two kinds of events in the target program, whose details are explained later.

- When the target program calls method m with parameters $o_1, \ldots, o_n$, the profiler records the references to o_i and the modification time of an object graph from o_i with respect to m for all $i = 1 \ldots n$.
- When the target program assigns a value to a field f of an object o, the profiler advances the current time t, and updates the timestamps of o with t.

4.2 Implementation of Per-Method Modification Time and Reference Equality

We implemented the profiler so that it will gather information on all the methods by two runs, and it will require a reasonable amount of memory.

As we have seen, an object may have a different modification time with respect to an interested method. A naive profiler implementation would execute a target program for each method in the program in order to calculate parameter

frequencies of the method. Another naive profiler implementation would associate a vector of time-stamps to each object, so that the profiler can access modification times with respect to all the methods in one execution of the program. For a target program with M methods, the former implementation should run the program for M times, and the latter should add M words to each object. Both are unacceptable for programs with large number of methods.

Our implementation runs the target program only once[5] by associating M' words to each object where M' is the number of different reference field sets, which is significantly smaller than M. This implementation can be considered as an improvement of the latter naive implementation by sharing a timestamp by the methods that have the same reference field set. In the case of AspectJ compiler, this reduces the length of time-stamp vectors by more than the factor of 90%.

The implementation can be explained by the table shown in Figure 6. Each column except for the rightmost one corresponds to a field declaration in the program. Each row corresponds to a reference field set. On the left side of each row, the methods that share the reference field set are written. The rightmost column has the indices numbers for timestamps. For example, the second row denotes that `ASTObject.getBytecodeTypeDec` and `ASTObject.getParent` have the same reference field set {`ASTObject.parent`}, and shares the timestamp index for the reference field set.

The profiler manipulates timestamps with the table in the following ways. To each object, it associates a vector of M' timestamps where M' is the height of the table. When a value is assigned to a field f of an object o, it loops over the column for f. It then updates ith timestamp entry of o for all i that ith row of the column is checked. When a method m is called with an object o as a parameter, it first identifies the row in the table that has m on its left side to get the timestamp index for m. It then computes the modification time of the object graph from o with respect to m by using the timestamp entries at the obtained index.

The vector of timestamps is realized by using the intertype declaration mechanism in AspectJ. The table is realized by (1) a hash table that maps a method signature to an index, and (2) a hash table that maps a field signature to a bit vector.

5. EVALUATION

The efficiency and effectiveness of the profiler is evaluated by profiling an execution of a practical application program. In the following evaluation, we executed AspectJ 1.0.6 compiler consisting of 64602 lines of Java code to compile a 75 lines Java program[6].

5.1 Profiler Efficiency

Two of the implementations of the profiler with per-method modification time and referential equality discussed in Section 4.2 are compared by measuring profiler execution time. The two implementations are (A) to associate one timestamp field to each object and repeatedly execute the target program for all the methods in the target program, and (B) to associate a vector of time-stamps to each object and execute the target program twice (once for gathering the reference field sets and once for recording parameters). As a reference, we also measured execution time of unprofiled (i.e., original) program.

[5]Excluding the run for building the reference field sets.

[6]The AspectJ compiler can compile Java programs as the language is a superset of Java. By profiling compilation of a Java program, we measured only the Java compiler part of the compiler; i.e., the part for the AspectJ specific features are practically excluded from the experiments.

Table 2: Execution Times of Profilers(sec.)

implementation	user + sys
A: 1 timestamp/object, M runs	2.07×10^5
B: M' timestamps/object, 1 run	3.46×10^2
unprofiled	1.82×10^0

The benchmark tests are executed on a Vine Linux 2.6 machine system an 1.9GHz Pentium 4 processor and 512MB memory. The profiler and the target programs are compiled and executed by AspectJ 1.1.1 and Java2 SDK 1.4.1. The execution times are measured by summing the system and the user times reported by the `time` command.

Table 2 shows the results. The implementation B is faster than A by the three orders of magnitude, even though it requires more memory to keep a vector of timestamps for each object. Compared to the execution time of the unprofiled program, it is slower by the the two orders of magnitude, but it remains within the practical range, compared to the implementation A, which took more than two days for profiling.

5.2 Profiler Effectiveness

We estimated effectiveness of the profiler compiler at applying the memoization technique to the methods in the AspectJ compiler.

5.2.1 Manual Application of Memoization

We first manually optimized AspectJ compiler by applying memoization without using the value profiling. The optimized methods are selected in the following steps:

1. First, the method execution times in the target program were measured by running the program on a Java virtual machine with `-Xrunhprof` option in the "samples" mode.

2. We then examined each method in the program from the one has longest execution time.

3. We rejected the method when (1) it is too trivial (like getter methods), or when (2) it is too complicated to memoize (like the ones relying on many state modifications).

4. Otherwise, we modified the method into a memoizing version. We then measure the overall execution time of the program to see how much the memoization speeded it up. If the execution time was not improved, the method was rejected.

5. We repeat the above two steps until we found a sufficiently many methods.

The result of the manual optimization is summarized in Table 3. The leftmost column shows the signatures of the selected methods. The middle two columns show the relative

execution times of the methods, and the speedup factors of the memoized version. The right two columns are explained in Section 5.2.2.

All execution times are measured by averaging the execution time of a loop that repeats the the body of the `main` method for 1000 times. Those times are measured by calling `System.currentTimeMillis` method in Java. Some methods show the greater speedup factor than its relative execution time. This is because (1) the speedup factor is computed by the execution time of the `main` method whereas the relative execution times accounts for the startup times of the virtual machine, and (2) the memoization can prevent the method executions invoked by the optimized method, which is not included in the relative execution time. The programs are executed on a Windows XP Professional system[7] with a 2.8GHz Pentium4 and 2GB memory running Java2 SDK 1.4.1.

5.2.2 Estimation of Effectiveness

Based on the results of optimizations, we estimated the effectiveness of the profiler by calculating how much efforts on manual optimization can be reduced when we performed the optimization procedure with a report from the value profiler.

The optimization procedure with a report from the value profiler adds the following step before step 3 in Section 5.1.

before 3. When the sum of the top n parameter frequencies of the method is less than threshold T, the method is rejected.

This will effectively rejects the methods that do not take the same parameters for sufficient number of times. The rightmost column of Table3 shows the sum of the top 3 parameter frequencies. As we can see, there are no methods whose numbers are close to zero.

For the top 56 methods, 13 methods will be rejected by the value profiler report when we set $n = 3$, $T = 20\%$. In other words, the value profiler reduced the efforts of manual optimization more than 20%. When we plot the methods by their relative execution time and top 3 parameter frequencies in a graph in Figure 7, we can see many rejected methods (plotted by the crosses) locate lower or left part of the graph, compared to the optimized methods (plotted by the circles).

It is not possible to derive a general conclusion from this estimation. The results of our profiler were also useful when optimizing methods because they give us more information how methods are called.

6. RELATED WORK

Time profiling, which measures execution times of individual methods in a program, is a common technique for optimizing large programs. It gives hotspots in a program that could be further optimized manually or automatically. Just-In-Time and HotSpot compilers for Java are well-known commercial examples that use runtime time profilers.

Relatively fewer attempts have done for profiling further information than execution times. Profiling on types of values is useful for optimizing method dispatching in object-oriented programs[3, 7]. Value profiling[1, 2, 16, 20] counts

[7]Although the benchmark is executed on a different platform from the one used in Section 5.1, both two platforms yielded the similar results. We used Windows XP platform here for merely availability reasons in our environment.

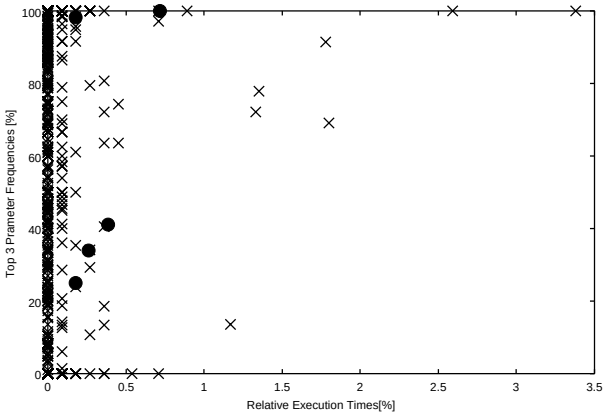

Figure 7: Execution Time and Parameter Frequencies of All Methods

frequencies of parameter values to machine instructions in a program execution. The value profilers are mainly used for instruction-level optimizations. Calpa[15] has a value profiler for automatically selecting targets of runtime specialization in DyC[9]. Though it targets a C-like language, the profiler can gather information on reference values with the aid of the points-to analysis, which shall be compared with our approach in future. Shaham showed a profiler on access times of objects is useful for optimizing memory usage in object-oriented programming languages with garbage collection[19]. The profiler records access time of individual objects by associating timestamps to each object, and reports the objects that are not used but not yet reclaimed by the garbage collector. The profiler manipulates timestamps through a customized Java virtual machine.

Programming techniques are also useful to find out the targets of program specialization. Schultz presented the specialization patterns in object-oriented programs that suggest the targets of partial evaluation[17]. As this technique relies solely on the structure of the program, it does not need to run programs for gathering information. The approach is shown to be useful to small scale programs.

7. CONCLUSION

We proposed a value profiler for object-oriented programs in order to assist in applying specialization techniques, such as memoization and partial evaluation. The profiler records parameter values of all method calls in a program execution, and shows the methods that are frequently called with the same parameter values. The per-method modification time and referential equality and its implementation accurately compares values of mutable objects which reasonable amount of memory.

Application of our profiler to a real-world application program showed that the execution time of the profiler is acceptable, and that the profile report can be estimated to reduce the effort of manual optimization by more than the factor of 20%.

Our future plan is to apply the profiler to many programs so as to have better criteria for measuring methods in target programs. In particular, combining results of static analysis would be promising to predict usefulness of methods in

Table 3: Optimized Methods and Their Profile Results

method signature	%exec. time	% speedup	#calls	param. freqs.
ASTConnection.makeTypeD(String)	0.71	0.0	184	98.5
JarClassMgr.makeSubPkgMgr(String)	0.36	2.0	121	40.5
ASTObject.getBytecodeTypeDec()	0.27	12.9	275	34.2
ASTObject.getLexType()	0.18	13.3	410	23.9
CP.addUtf8(String)	0.18	2.1	124	98.4

terms of specialization techniques.

8. REFERENCES

[1] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, March 1999.

[2] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *International Symposium on Microarchitecture*, pages 259–269, 1997.

[3] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-type object-oriented programs. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 1990.

[4] Sandrine Chirokoff and Charles Consel. Combining program and data specialization. In Olivier Danvy, editor, *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, volume NS-99-1 of *BRICS Notes Series*, pages 45–59, San Antonio, Texas, January 1999. ACM SIGPLAN.

[5] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of Symposium on Principles of Programming Languages (POPL96)*, pages 145–170, St. Petersburg Beach, Florida, January 1996. ACM SIGPLAN-SIGACT.

[6] Olivier Danvy and Andrzej Filinski, editors. *Second Symposium on Programs as Data Objects (PADO II)*, volume 2053 of *Lecture Notes in Computer Science*, Aarhus, Denmark, May 2001. Springer-Verlag.

[7] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. pages 93–102.

[8] Erich Gamma, Richrad Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[9] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimization in DyC. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[10] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.

[12] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)*, volume 31(5) of *ACM SIGPLAN Notices*, pages 215–225, Philadelphia, PA, May 1996. ACM.

[13] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization: A portable approach to generating optimized specialized code. In Danvy and Filinski [6], pages 138–154.

[14] David Michie. Memo functions and machine learning. *Nature*, 218(1):19–22, April 1968.

[15] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: A tool for automating selective dynamic compilation. In *33rd Annual Symposium on Microarchitecture*, December 2000.

[16] Robert Muth, Scott A. Watterson, and Saumya K. Debray. Code specialization based on value profiles. In *Static Analysis Symposium*, pages 340–359, 2000.

[17] U. P. Schultz, Julia L. Lawall, and Charles Consel. Specialization patterns. In *ASE'00*, 2000.

[18] Ulrik Schultz. Partial evaluation for class-based object-oriented languages. In Danvy and Filinski [6], pages 173–197.

[19] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient java. In *Programming Languages Design and Implementation (PLDI '01)*, ACM SIGPLAN Notices, Snowbird, Utah, USA, June 2001. ACM.

[20] Scott Watterson and Saumya Debray. Goal-directed value profiling. In *Proc. 10th International Conference on Compiler Construction (CC2001)*, April 2001.