Dynamic code optimization for statically typed OO languages in an integrated incremental system

Anders Dellien

LU-CS-TR:94-131

Also published in Proceedings of NWPER'94, Nordic Workshop on Programming Environment Research, Lund, Sweden, June 1994.

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

Dynamic code optimization for statically typed OO languages in an integrated incremental system

Anders Dellien

Dept of Computer Science, Lund University
Box 1 18, S-221 00 Lund, Sweden
e-mail: Anders.Dellien@dna.lth.se

Abstract. This paper describes the implementation of code optimization algorithms in an interactive programming environment. Program execution information is used during the code optimization to enhance the code quality .

1 Introduction

Object-oriented language implementations have traditionally had a performance disadvantage compared to their procedure-oriented counterparts. This performance difference is primarily due to the greater frequency of procedure calls in object-oriented programs, and greater cost of procedure calls in some object-oriented language implementations.

This paper describes the application of dynamic code optimization techniques (i.e. code optimization based on information about the program execution profile) to Orm [Mag90], an interactive programming environment for statically-typed object-oriented languages.

The execution profile is primarily used for making better inlining decisions, but it can also be used to guide other traditional optimization techniques.

2 An example

An example of the application of profile information to code generation is to guide the compiler in making inlining decisions.

The increased number of procedure calls is a consequence of the object-oriented programming paradigm, where an object is viewed as an encapsulated entity , thus all access to internal state from other objects should be done by procedure calls, rather than directly accessing internal state variables.

The greater cost of executing a procedure call in an object-oriented language is a negative side-effect of dynamic binding, i.e. the binding of procedure names to implementation code during program execution. Implementations of dynamic binding require extra indirection steps or runtime type tests, which causes the greater cost of procedure calls. In statically typed languages this is, however , small.

Another effect of dynamic binding is that inlining becomes more difficult to implement, because the code invoked by a procedure call is not known at compile time. In statically typed languages, a set of procedure implementations invoked by a call is known at compile time, but that is not the case in dynamically typed languages.

The characteristics of object-oriented program executions described above suggests that procedure inlining, i.e. the insertion of the implementation code of a procedure at its call site, would increase the execution speed. Previous research [JGZ88], [A TK86] has verified this hypothesis. Unfortunately , there is a price to pay for the increased efficiency: procedure inlining increases the memory consumption and compile time of a program.

It is the responsibility of the compiler implementor to devise an algorithm for deciding which procedure calls to inline that achieves a reasonable compromise between execution speed and memory- and compile-time consumption. The behavior of this algorithm has a large effect on the overall performance of the programming environment, due to the following effects:

- A too pessimistic algorithm (not enough inlining) causes the program to run slower than necessary
- A too optimistic algorithm (too much inlining) causes unnecessary memory- and time consumption.

Traditional compiler implementations use a static analysis of the source code to determine the procedure calls that are suitable for inlining. A static program analysis is necessarily approximate, which sometimes makes the compiler take the wrong decision, causing time to be spent on optimizing infrequently executed code, or missing optimization possibilities.

As an example, consider the following code fragment, where a procedure is called from within a nested loop:

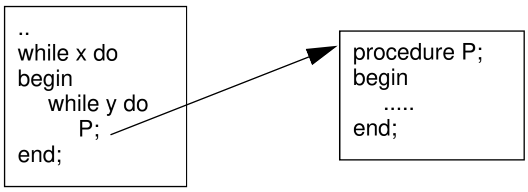

Figure 1: Procedure call from a loop

A static program analysis might decide to inline the call of P , due to the call being placed inside a nested loop and therefore expected to execute often. This might not be correct, e.g. if one of the conditions happens to be false in this particular program execution, the decision to inline the call gains nothing, and costs a certain amount of compile time.

A code generator with access to profile information would discover that, in this case, P is never called, thereby saving an unnecessary procedure inlining.

This illustrates the advantage of using profile information during code generation: because the code generator has more information about the program than a static analysis would discover , it can generate efficient code where it counts, i.e. the frequently executed parts of the program. This approach saves compilation time and potentially leads to better runtime performance.

3 Related work

Code optimization research for object-oriented languages has been concentrated on dynamically typed languages, e.g. Smalltalk and Self, which differ from statically typed languages in that:

- Reference variables are not typed, i.e. they may potentially reference any object in the program execution.
- Type checking (which includes the binding of procedure names to implementation code) of the program is done at run time.

The implication of these differences is that it is not always statically possible to determine which code (if any , a procedure call may not have a matching definition in the called object) should be invoked as the result of a procedure call.

Statically typed languages typically allow both statically and dynamically bound procedures. The static type checking guarantees that every procedure call has a corresponding implementation, but which code is executed as the result of a dynamically bound call is still determined during runtime.

3.1 Smalltalk implementations

Smalltalk [GR83] is a dynamically typed object-oriented language, intended to support an exploratory programming style. Several efficient Smalltalk systems have been implemented:

The Deutsch-Shiffman Smalltalk-80 system.
The Smalltalk-80 definition specifies that source code is translated into a sequence of primitive operations called *byte codes* . Smalltalk-80 was originally run on machines which implemented the byte codes in microcode [Deu83]. Early implementations of Smalltalk-80 on stock hardware [Kra83] interpreted the byte codes in software, which led to poor performance. The Deutsch-Shiffman system [DS84] is a Smalltalk-80 implementation that achieves high performance without specialized hardware support.

The cost of interpreting byte codes is eliminated by so-called dynamic compilation, i.e. source code is translated into machine code on demand at run-time. The dynamic compilation increases performance by executing machine code instead of interpreting byte codes, another advantage is that execution can begin immediately after modifications of the source code.

The system also uses so-called inline caching, a technique for reducing the overhead for dynamic binding. Inline caching is based on the observation that although a given call site is potentially able to call several procedure definitions, many call sites exhibit a “static” behavior in the sense that only one procedure definition is actually executed.

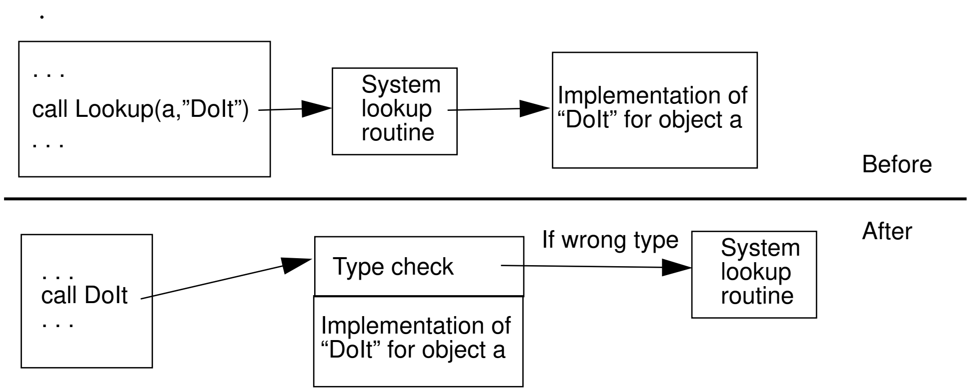

Figure 2: Inline caching

To take advantage of this observation, the call instruction that originally invoked a runtime routine for procedure lookup is replaced by a call to the result of the lookup, thus replacing a dynamic call with a static call, as shown in figure 2.

This is incorrect if the receiver of the call changes, thereby invalidating the previous procedure lookup. The solution is to include a type test as a prologue of the machine code for a procedure, and if the type test fails, the runtime lookup routine is called and the call site is re-patched to the new procedure definition.

Typed Smalltalk (TS)

TS [JGZ88] is an extension of Smalltalk where variables are declared as belonging to a specific type, i.e. a set of classes. The type information is used to increase performance by decreasing the overall cost of runtime procedure lookup.

- If the type of a variable contains exactly one class, all procedure calls through the variable can be statically bound or inlined.
- If a variable has a small number of classes in its type, the compiler generates a sequence of runtime type tests, which branch to the appropriate code.
- If a type contains a large number of classes, the inline caching technique of Deutsch-Shiffman is used for calls to objects referenced by variables of the type.

3.2 SELF

SELF [Cha92] is similar to Smalltalk, but it is a pure object-oriented language in the sense that all operations, including variable accesses and basic arithmetic operations, are implemented by dynamically bound procedure calls.

This language design makes the efficiency of the procedure call mechanism crucial for the overall system performance. Some of the techniques used to implement efficient procedure calls in the Self system are the following:

Inline caching

The Self system uses a variant of the inline caching technique of Deutsch-Schiffman called polymorphic inline caching [HCU91]. Polymorphic inline caching handles the case when a procedure call has a few receiver types. This is implemented by replacing the call to the system lookup routine by a call to a stub routine that contains a sequence of type tests which branch to the correct code.

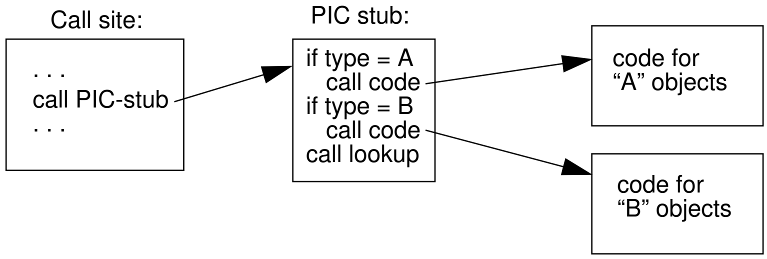

Figure 3: Polymorphic inline caching

Procedure inlining

Procedure inlining is difficult to implement in dynamically typed languages, because there is no static type information that tells which code gets invoked as the result of a procedure call. The Self compiler obtains the missing type information by a static analysis of the source code. Some sources of type information can be e.g. primitive operations that are known to return objects of a particular type, or customization (described below). If the compiler can use this information to infer the receiving type of a procedure call, that call can be inlined or statically bound.

Customization

The term *customization* means that multiple machine code versions of a given procedure are created. The procedures that are candidates for customization are those that are inherited by many classes. Each version of a customized procedure is specialized for a given class, which makes it possible for the compiler to statically look up all procedure calls from the object to itself. This in turn creates possibilities for procedure inlining.

3.3 Code generation in Orm

The proposed code generation algorithms in the Orm system differ from the above mentioned techniques for two reasons: the languages supported are statically typed, and the code generator is supplied with measurements of the program execution.

The static typing ensures that all procedure calls during runtime have a definition in the receiving object, and also determines a set of possible destination code segments for each dynamically-bound procedure call. If this set is small, the call may be inlined.

The availability of profile information during code generation should result in more efficient code, simply because the code generator has more information on which to base decisions regarding e.g. inlining.

4 The architecture of the Orm programming environment

Orm is an interactive, integrated programming environment for statically typed object-oriented languages. It is partitioned into three processes: the runtime system, the development system, and the binary code generator . The source code is maintained by the development system, which also generates intermediate code for the program. The runtime system controls the program execution and observation.

The code for a program is generated incrementally when the program is run, with the granularity of a block (i.e. procedure or class). The runtime system executes the user program until control reaches a block which has no corresponding binary code in the runtime system.

The runtime system then sends a code request to the development system, which then responds by generating the requested code, and sending it to the runtime system, after which the execution continues.

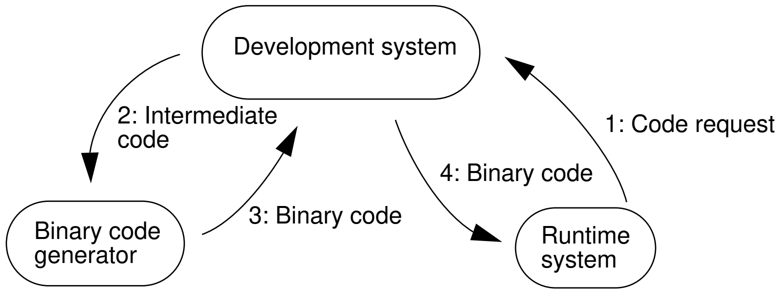

Figure 4: Orm process architecture

This architecture, with the code generator and the runtime system concurrently executing, enables the use of program execution statistics to improve the code generated for the program.

5 Execution profiling

The idea behind dynamic code generation is to identify the most heavily executed parts of the program, and to generate efficient code for these parts. In a procedure-oriented program, this may be achieved by counting the execution frequencies of the procedures in the program, or , at a finer level of granularity , the execution frequencies of the basic blocks. In addition to the execution counts, it is useful to know the distribution of the execution count over the call sites of the procedure.

In an object-oriented language, this kind of execution profiling does however not tell the whole story of the execution behavior , due to the late binding of procedures. The above mentioned execution profiling may thus be complemented with information about the runtime types of reference variables, to determine execution frequencies in the presence of late binding.

5.1 Implementation of execution profilers

Execution profilers can be implemented using two techniques:

Counter -based execution profiling uses a counter variable for each location in the source program that is monitored. This counter is incremented every time the location is visited. This method demands minimal support of the runtime system, but is somewhat inefficient

Sample-based execution profiling is implemented with the execution profiler executing as a separate process. The profiler randomly samples the program counter of the user program, and when a sufficiently large number of samplings has been made, the execution profile can be determined. To get the corresponding location in the source code, the runtime system must provide a mapping from PC value to source code location.

5.2 Execution counts

The execution count of a procedure records how often it is executed. This information can be represented with a weighted call graph, where the weight of an edge records the number of executions for the call site associated with the edge. In addition, every node in the graph records the total number of invocations. The basic-block level information is analogous to the procedure-level information described here.

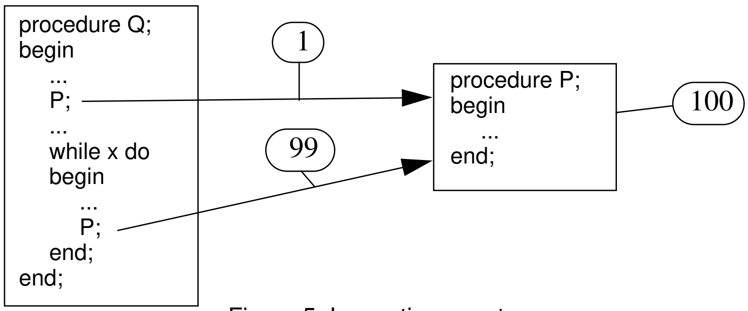

Figure 5: Invocation counts

The figure above shows a program fragment with a procedure P that is called from two places in procedure Q. The execution count of P (100) is the sum of the two execution counts inside Q. In this example, the execution information will probably cause the second call of P in Q to be inlined.

5.3 Dynamic qualification of reference variables

In order to determine the execution frequencies of dynamically bound procedures, it is necessary to measure the dynamic qualification of the reference variables that the procedures are invoked through.

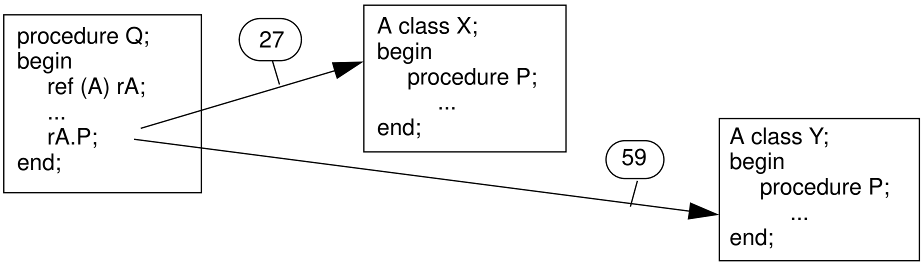

Figure 6: Dynamic qualification information

This kind of measurement may , for example, lead to the code of P in Y being inlined, provided that the call frequencies are sufficiently skewed in favor of Y .

6 How is the profile information used?

This section describes how the code generator uses the profile information to generate efficient code for the source program. Many classic code optimizations can benefit from runtime information, as demonstrated in [CMH91]. Briefly , the profile information is used as follows:

Procedure invocation counts are used to determine the most heavily executed procedures in the program. This enables the code generator to spend more time generating efficient code for the most frequently executed parts of the program, and generating simple code for the other parts.

Call-site counts are used to determine which procedure calls to inline. The payoff when inlining a procedure call is obviously greater when it is frequently executed, and this information is an important factor when deciding if a call should be inlined or not.

Dynamic qualification counts for reference variables are needed when inlining dynamically bound procedures. A dynamically bound procedure call may invoke several different code segments, depending on the dynamic qualification of the reference variable used, and inlining all possible code segments would be costly in terms of storage. This information makes it possible to inline the most frequently executed code segments, and calling the rest normally .

Basic block execution counts are used to find frequently executed paths through a procedure. These basic blocks in these paths are then globally optimized, at the expense of slower code in the less frequently executed paths, which gives an increased total performance.

7 Applying dynamic code generation techniques to Orm

The architecture of the Orm system, with the devopment system and the executing program as parallel processes, possibly on different machines, allows communication from the program execution to the compiler . This communication enables the code generator to generate code based on information about the program execution, as described in the previous sections. This section describes the implementation decision when applying adaptive code generation to Orm.

The issues that need to be considered are:

- Measurements, which information is needed and how is it used?
- Execution profiling, how is it implemented efficiently?
- How does the debugger handle an optimized program?
- How does an incremental compiler handle the compilation dependencies (i.e. the modification of a procedure may cause other procedures to be recompiled) introduced by some optimization techniques?

The expected benefits of using dynamic code optimization are:

- No static program analysis is needed, making the code generation fast
- The most frequently executed parts of the program are optimized using profile information, making the program execution fast

The results of the dynamic code optimization should be an improvement over static code optimization techniques, preferably in both code generation speed and program execution speed.

7.1 Measurements

The execution profile of the program can be used as described in section 3, briefly:

- Procedure execution counts are used for identifying frequently executed parts of the program, and can be used for deciding which procedures to optimize using conventional global algorithms, e.g. register allocation.
- Procedure execution counts per call site are used for accurately deciding the benefit of inlining a procedure.
- Dynamic types are needed when inlining dynamically bound procedures.
- Basic-block execution counts are used for optimizing frequently executed paths inside a procedure.

An interesting possibility is to use the execution profile to determine which parts of the program that should be monitored, and how accurately . This dynamic profiling should decrease the profiling cost, because the less frequently executed parts of the program can be monitored with simple procedure invocation counts, whereas the hot spots could be more accurately profiled with call-site information and basic block execution counts.

7.2 Implementation of execution profiling in Orm

The measurements are made at the basic block level or the procedure level. We must find a reasonable balance between the cost of the execution profiling and the execution speed gained by utilizing this information. In the Orm system, the execution profiling is done at the (final) program execution, i. e. no separate profiling step is used. This means that time taken by execution profiling must be less than the time gained by the code optimization. In addition, the user should not notice any increase in the response time of the system.

The implementation of the execution profiling will have to be fast enough not to disturb the user interaction. In addition, the profile information should be accurate even for optimized programs, for example inlining a procedure P inside procedure A should correctly attribute executions of the code of P to P, and not A.

7.3 Debugging

A useful programming environment should contain a source-level debugger. The debugger allows the programmer to single-step the code, and also to examine and modify variable values. Many optimization techniques involve code transformation that makes source-level debugging difficult. Procedure inlining, for example, introduces object code that does not have a counterpart in the source program. Other optimizations may rearrange and delete parts of the original program.

These code transformations cause problems when e.g. single-stepping the source code (the object code may be reordered so that the instructions for one statement may be mixed with the instructions for the following statement) , and displaying variable values (some variables in the source code may not be present in the object code).

A solution to these problems could be to avoid optimizing the parts of the program that are being examined by the debugger [HCU92], thus maintaining a simple mapping from object code to source code.

7.4 Compilation dependencies

The inlining of a procedure body causes a compilation dependency between the call site and the procedure body, i. e. when a procedure body is modified and recompiled, the block containing the call site also needs to be recompiled. This is not a problem in a batch compiler, but when incrementally generating code for a program, the originally modified procedure causes all dependent call sites to be recompiled.

The dependency information is a directed graph, with a node for every block in the source program, and an edge from node A to node B site iff the procedure represented by node A is inlined at node B.

Figure 7 shows a program fragment with the procedures P, Q and Z. Procedure P is nested inside class A. The grey arrows show the compilation dependencies introduced by procedure inlining (all call sites are assumed to be inlined). When a procedure is modified, the dependency graph is traversed with origin in the node represented by the procedure, and all nodes that can be reached must be recompiled. In the example above, Z and Q must (in addition to P itself) be recompiled if P is modified.

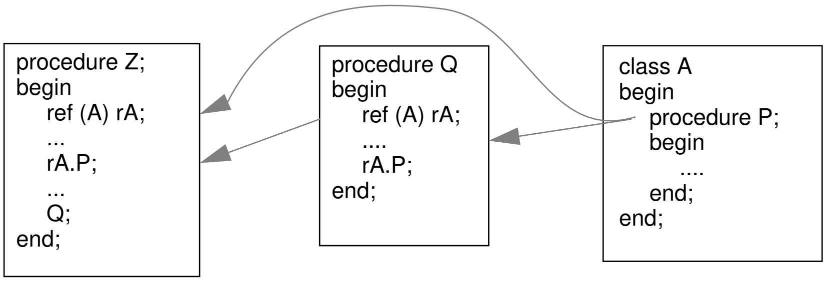

Figure 7: Compilation dependencies

8 Conclusions

This paper has presented an application of dynamic code optimization techniques to statically-typed object-oriented languages in Orm, an integrated programming environment. The code generator uses the program execution profile to e.g make better inlining decisions, and to optimize frequently executed parts of the program. The expected benefits of using dynamic code optimization are:

- No static program analysis is needed, making the code generation fast
- The most frequently executed parts of the program are optimized using profile information, making the program execution fast

References

[ATK86] Robert G. Atkinson. *Hurricane: An Optimizing Compiler for Smalltalk*. In OOPSLA-86 Conference Proceedings.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Ph.D. Thesis, Stanford University, 1992.

[CMH91] Pohua P. Chang, Scott A. Mahlke and Wen-Mei Hwu. *Using Profile Information to Assist Classic Code Optimizations*. Software - Practice and Experience 21(12).

[CU92] Craig Chambers and David Ungar. *Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language.* Proceedings of SIGPLAN-89 Conference on Programming Language Design and Implementation.

[Deu83] Peter Deutsch. *The Dorado Smalltalk-80 implementation: Hardware's Impact on Software Architecture*. In Krasner. ed. *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, Reading, MA, 1983.

[DS84] Peter Deutsch and Alan Shiffman. *Efficient implementation of the Smalltalk-80 system*. In Proceedings of the 11th Symposium on the Principles of Programming Languages, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[HCU91] Urs Hölzle, Craig Chambers and David Ungar. *Optimizing Dynamically-*

Typed Object-Oriented Languages With Polymorphic Inline Caches. Proceedings of ECOOP-91, Springer Verlag Lecture Notes in Computer Science 512, July 1991.

[HCU92] Urs Hölzle, Craig Chambers and David Ungar. *Debugging optimized code with dynamic deoptimization*. In Proceedings of SIGPLAN-92 conference on Programming Language Design and Implementation.

[JGZ88] Ralph E. Johnson, Justin O. Graver and Lawrence W. Zurawski. *TS: An Optimizing Compiler for Smalltalk*. In OOPSLA-88 Conference Proceedings, SIGPLAN Notices 21(11).

[Kra83] Glenn Krasner, ed. *Smalltalk-80: Bits of History, Words of advice*. Addison-Wesley, Reading, MA, 1983

[Mag90] Boris Magnusson et al. *An overview of the Mj ¢lner/Orm Environment*. Proceedings of TOOLS-90.