SUBMITTED version 0.4 — Monday, September 9, 2002
Submitted to ICSE'2003

Maintainability versus Performance: What's the Effect of Introducing Polymorphism ?

Serge Demeyer
Lab on Reengineering (LORE)
University of Antwerp, Universiteitsplein 1, B —2610 Wilrijk (Belgium)
http://win-www.uia.ac.be/u/sdemey/

Abstract

The notion of refactoring —transforming the source-code of an object-oriented program without changing its external behavior— has been embraced by many object-oriented software developers as a way to accommodate changing requirements. If applied well, refactoring improves the maintainability of the software, however it is believed that it does so at the sake of performance. To investigate this trade-off, we compared the performance of a program which contains large conditionals against one where the conditionals were replaced by polymorphic method calls. We discovered that C++ programs refactored this way perform faster than their non-refactored counterparts, hence advise programmers not to obfuscate their programs with conditional logic in order to obtain good performance.

1. Introduction

> "*Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning.*" *[Fowl99a]*

Several scientific studies concerning large scale software systems have shown that more than 80% of the total budget of a software project is spent during system maintenance. What may seem surprising at first is that this percentage is increasing: "the more modern methods you use in building software, the more time you spend maintaining the resulting product" [Glass98a]. The explanation for this observation is that modern software —more than their traditional counterparts— "undergoes continual change or becomes progressively less useful" [Lehm85a]. Not surprisingly, the recent trend with agile software development processes recognize change as the only constant factor in software development [Beck99a], [High99a]. Consequently, successful software systems are those that gracefully evolve and adapt to changing requirements, hence maintainability is considered a crucial quality factor.

One of the accepted techniques for improving the maintainability of object-oriented software systems, is called refactoring [Fowl99a]. The key idea is to redistribute instance variables and methods across the class hierarchy in order to prepare the software for future extensions. Refactoring has been studied for quite a while in academic circles (see [Berg97a], [Opdy92a], [Casa92a], [Gris93a], [Moo96a]), resulting in an industrial-strength refactoring tool (the popular refactoring browser [Rober97a]), which is currently being mimicked by many tool vendors (see http://www.refactoring.com/ for a list of tools supporting refactoring).

Unfortunately, refactoring will almost certainly have a negative impact on the performance. Most refactorings introduce indirection by means of procedure calls which cost extra processor instructions. Worse, due to their object-oriented context, many of these procedure calls must be polymorphic (i.e., the C++ `virtual` declaration) which may induce a performance penalty of 15% compared to a normal procedure call [Copl92a].

Therefore, programmers often argue that they cannot afford the cost of refactoring. This argument is usually countered with the 80/20 rule [Meye96a]: since 80 percent of a program's resources are used by approximately 20% of the code, one should only sacrifice the maintainability of the code in the 20% that is executed frequently. Refactoring people emphasize this rule, claiming that well refactored programs make it easier to identify the critical 20% and focus optimization efforts thereon [Fowl99a]. While this may be true, the 80/20 rule is not always applicable. Especially with the current generation of battery-operated devices, extra processor cycles waste precious battery power which is deemed unacceptable.

However, is it really true that refactored programs run slower? At least one report indicates that a refactored program runs *faster* than its non-refactored counterpart, and that conditional logic is better replaced by polymorphism [Russ88a]. Moreover, lots of research has been directed at improving compiler and processor technology, especially with respect to the resolution of polymorphism [Drie96a]. Hence it is possible that the performance cost of refactoring —especially replacing conditional logic by polymorphism— is negligible.

To investigate this hypothesis, this paper compares the performance of a C++ program containing large conditionals against one where this conditional logic is implemented using polymorphism. To assess the impact of compiler and processor technology, we compile and execute the program on different platforms and see whether the results have a common trend.

The paper is structured like a comparative study. It starts with motivating the choice for the "Replace Conditional with Polymorphism" refactoring, which serves as the basis for our experiment (section 2). Next we discuss the performance implications of conditionals and polymorphism (section 3), to continue with the experimental set-up (section 4) and the results (section 5). After giving an overview of the related work (section 6), we summarize our findings (section 7).

2. Eliminating Conditional Logic

For a number of years we have been involved in projects concerning the reengineering of object-oriented legacy systems. Our experience is summarized in the form of reengineering patterns, which cover best-practices applicable during various stages of the reengineering life-cycle [Deme02a]. One of the recurring issues we experienced during all of our projects is how to deal with misplaced responsibilities, that is functionality which is defined inside a given class but which more logically belongs into another class. Typical symptoms for misplaced responsibilities are duplicated code, navigation code, god classes and complex conditional logic. To make the structure more logical, we advise maintainers how to refactor such code by creating new classes, extracting new methods and then moving the extracted methods to the appropriate classes.

However, while discussing with C++ programmers maintaining such object-oriented legacy systems, we often had trouble convincing them to remove the conditional logic. Maintainers typically see clear benefits from eliminating duplicated code (duplicated code implies duplicated bugs), navigation code (faster compilation times), god classes (smaller classes are easier to understand, debug and modify), and are easily convinced to refactor such code. However, when explaining how to remove conditional logic, we inevitably get the reply "Oh, but this will introduce virtual methods … and then our program will become slower". Arguing with the 80/20 rule is rarely convincing, and since no hard numbers on the performance penalties of virtual methods exist, the conditional logic often remains as it is.

Nevertheless, complex conditional logic represents the most important opportunity to refactor legacy systems because it is so typical for software that is maintained over a long period of time. Indeed, while a system is functioning in a changing business context, its users will inevitably come up with special cases that should also be handled. Such cases appear as extra conditionals in the code and there is nothing inherently wrong with that. However, if it happens repeatedly, it is a clear sign that the program is missing an important business abstraction and removing the conditional logic will gracefully unveil the abstraction in the form of a polymorphic method, sometimes even a new inheritance hierarchy.

During our projects, we have identified three variants of complex conditional logic and ways to remove it.

- *Transform Client Type Checks*. The first variant has to do with a client testing the type of a certain provider object, before performing a series of operations. This can be refactored by moving code from the client to the provider. A special case of this variant is a client which tests whether a provider is null or empty, which may be refactored by introducing a special Null Object [Wool98a].
- *Transform Self Type Checks*. The second variant has to do with an object testing an attribute serving as some kind of type-tag, which may be refactored by creating a new subclass for each leg of the conditional and moving the code as a polymorphic method in the subclass. There is one special case concerning an attribute which represents dynamically changing state, in which case it should be refactored by introducing a State object [Gamm95a]. Another special case is about an attribute representing different possibilities to compute a certain result, in which case it should be refactored by introducing a Strategy [Gamm95a].
- *Transform Conditionals into Registration*. The third variant is about clients which test the type of a series of objects before performing a certain action. This can be refactored by a central registration mechanism, which acts as a mediator between objects providing services and clients requesting services.

Note that it is not always possible —nor desirable— to replace conditional logic by polymorphism. For instance, to Transform Self Type Checks, one must introduce subclasses, hence should be able to change the code where the class constructors are invoked. Also, if conditional logic is encapsulated in a single piece of code, it is a clear and concise way to show the intent of the code. Only when the some conditional logic reappears in other pieces of code it becomes a maintenance problem, because one risks modifying one condition without adapting the other part as well.

Of the three variants sketched above, the "Transform Self Type Checks" is the most interesting for further examination because it introduces both a new polymorphic method and a new class hierarchy. All other variants restrict themselves to the introduction of new

polymorphic method only, sometimes accompanied with a single new class. Therefore, we use the "Transform Self Type Checks" as representative for the other variants and use it as a basis for our experiment.

Of course, removing conditional logic is but one way to improve the logical structure —hence the maintainability— of a program. Other refactoring sequences are necessary to remove duplicated code, navigation code and god classes and our experiment does not investigate the performance penalties thereof. However, we have experienced that few programmers resist such refactorings, possible because the performance trade-off's are well-known as they mainly concern procedural constructs. Therefore, these refactorings were omitted from the experiment.

3. Performance Cost of Polymorphism

To understand the motivation of our hypothesis "the performance cost of replacing conditional logic by polymorphism is negligible" its necessary to understand how typical C++ compilers generate code for virtual functions and compare it with code generation strategies for conditionals. This is only a very general overview and we refer the reader to other sources for a more detailed treatment of the subject [Drie96a], [Drie99a].

The preferred C++ compiler strategy for implementing virtual functions is by means of *virtual function tables* (VFT). Each class maintains a table containing target function addresses for all functions it must respond to (thus including the ones declared in a superclass) and which are declared virtual. Since each class has its own VFT, overriding a method corresponds with overwriting the target function address in the VFT. At compile-time, all virtual functions are numbered consecutively, hence performing a look-up in the VFT table corresponds to indexing an array. Each object maintains a pointer to its class, via which it accesses the VFT table. Therefore, the performance overhead of a virtual function compared to a non virtual one corresponds to following two indirections: one for going from the object to the class, and one from the class to the index in the VFT. (Note that multiple inheritance makes the issue slightly more complicated but here as well some optimization strategies exist).

With nested if-statements it appears that no run-time overhead is involved. However, each nesting level must retest the type of an object, thus the cost varies depending on the nesting level and the object types most frequently accessed: on average this is half of the nesting level. Compared to polymorphism this varying cost is replaced by the constant cost of following two pointer indirections.

Case-statements allow a compiler to avoid this varying cost. In C++, the case labels must be constant and of an integral type, hence an optimizing compiler maintains a table with an entry for each of the case-labels. The performance cost then corresponds to indexing an array, which is the same as for the VFT-implementation of a virtual method. Of course, the switch expression must also be evaluated and the speed there depends on complexity of the expression. Compared to polymorphism, evaluating the case-expression may be faster or slower than following the pointer from the object to the class, but the indexing in the table should be equal.

Given the above, we expect that case-statements and polymorphic methods have the same performance overhead, while if-statements fare slightly better when the nesting level is low, but should perform slower if the nesting level exceeds a certain size. However, current hardware technology plays a role as well, because today's processors scan the instruction stream trying to perform as much as possible in parallel and trying to predict targets of branches to avoid idle cycles. Hence, to verify whether the performance cost of polymorphism is indeed negligible, one should experiment with different compilers targeting different processors and compare the actual performance results.

4. Experimental Set-up

This section describes the experiment so that other researchers may replicate the results. We first specify the benchmark program itself and afterwards give some details concerning the compilers and optimization options we use and the processors that execute the benchmark programs.

4.1. The Benchmark Program

To benchmark program we used to assess the performance cost is a C++ program inspired by the reengineering pattern "Transform Self Type Checks" [Deme02a]. The central data structure is a simple record, containing two data members which are both initialized using the same value.

```
class MessageConditional {
private:
   short type_;
   int data_;

public:
   MessageConditional(short type);
   ~MessageConditional() {};
   int actionSwitch ();
   int actionIf ();
};
```

```
MessageConditional::
     MessageConditional(short type)
{
   type_ = type;
   data_ = type;
};
```

On this data structure we define an `action` function which subtracts a constant value from the `data_` member. However, the subtraction operation is embedded within a large conditional, which tests the value of the `type_` member, serving as a type discriminator. Therefore, the result is always 1, yet this cannot be inferred by compilers using peep-hole optimization.

```
int MessageConditional::
       actionSwitch () {
   switch (type_) {
      case 0:
        return data_ + 1;
      case 1:
        return data_ + 0;
      case 2:
        return data_ - 1;
```

… similar code goes here …

```
      case 18:
        return data_ - 17;
      case 19:
        return data_ - 18;
      default:
        return -1;
   };
};
```

To test the difference between conditional logic using case-statements and if-statements, the program includes the same function implemented with the two variants.

```
int MessageConditional::
       actionIf () {
   if (type_ == 0)
      return data_ + 1;
   else if (type_ == 1)
      return data_ + 0;
   else if (type_ == 2)
      return data_ - 1;
```

… similar code goes here …

```
   else if (type_ == 18)
      return data_ - 17;
   else if (type_ == 19)
      return data_ - 18;
   else
      return -1;
};
```

Next, the same functionality is implemented but now using polymorphic method calls. Thus, we first define a base class with a single data-member; absorbing the type-discriminator into the inheritance structure.

```
class MessagePolymorf {
protected:
   int data_;

public:
   MessagePolymorf(int data);
   ~MessagePolymorf() {};
   virtual int action ();
};

MessagePolymorf::
     MessagePolymorf(int data) {
   data_ = data;
};
```

Then, the leg of the conditional corresponding with the default value is put in the root of the inheritance hierarchy as a virtual function. For each other leg we introduce a polymorphic method, by creating a new subclass and overriding the default virtual function

```
// default case
int MessagePolymorf:: action () {
   return -1;
};

// case 00
class MessagePolymorf00 :
   public MessagePolymorf {
public:
   MessagePolymorf00();
   ~MessagePolymorf00() {};
   virtual int action ();
};

int MessagePolymorf00:: action () {
   return data_ + 1;
};
```

… similar code goes here …

```
// case 19
class MessagePolymorf00 :
   public MessagePolymorf {
public:
   MessagePolymorf19();
   ~MessagePolymorf19() {};
   virtual int action ();
};

int MessagePolymorf19:: action () {
   return data_ - 18;
};
```

Finally, we create two arrays holding 20 objects, one for those objects implemented with conditionals and one for the objects using polymorphism. Each of the objects is initialized with a separate data-value between 0 and 19. Via a loop we call the various `action` functions on each of the objects a million times, counting the amount of clock-ticks such an operation takes. To account for variation, we take 10 samples and average the results. Also, we verify the effect of the size of the conditional, by trying all condition sizes between 1 and 20.

4.1. Compilers under Study

To analyze the differences between several compilers, we perform the experiment with three different compilers using various optimization settings. The resulting benchmark program is then executed on the target processor. Due to the variation in compilers and platforms, we are confident that the results are representative for other C++ compilers and processors.

The benchmark program uses very little memory, hence we do not list the memory available on the machine running the benchmark. Also, since we are only interested in the performance differences between conditionals and polymorphism, the frequency of the processor's internal clock does not really influence the results. Note that this experiment is not set-up as a comparative study investigating which of the three compilers is the best. Rather its meant to see whether there is a common trend in how current compiler technology deal with polymorphism.

CodeWarrior, IDE 4.0, using compiler settings (a) normal: optimizations off; (b) fast: level 4 optimization. The code executes on a Macintosh iBook, running MacOS 9.0.4 and equipped with a PowerPC G3 processor.

VisualC++, version 6.0, using compiler settings (a) normal: optimizations default; (b) fast: optimizations Maximize Speed. The code executes on a Dell latitude C600, running Windows98SE and equipped with a Pentium III processor.

g++, using compiler settings (a) normal: compiled with -o option not set; (b) fast: compiled with -o option set. Since the developers mentioned that they worked hard on optimizations recently, we run experiments with versions 2.95 and 3.01. The code executes on a Sun Blade 2000, running Solaris 2.9 and equipped with two UltraSparc III processors.

5. Experimental Results

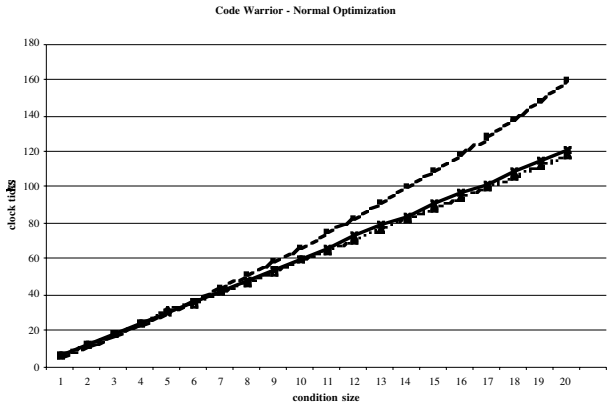
Below you find the results of the benchmark program compiled by four different compilers, running on three different platforms. Since we are mainly interested in a relative performance of the conditional logic (if-statement and case-statement) versus polymorphism, we showed the results visually and left the absolute numbers for the appendix.

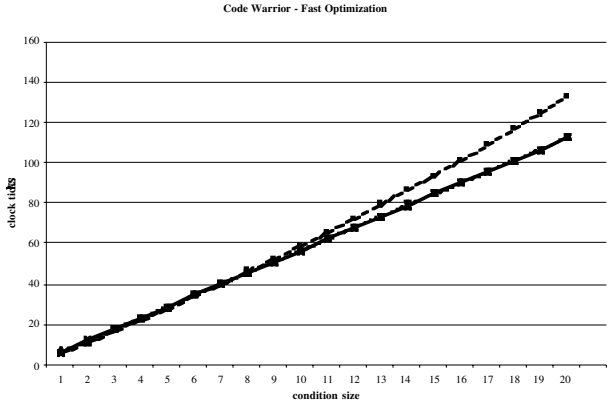
In the charts below, the horizontal axis represents the condition size (nesting level of the if-statement) while the vertical axis represents the number of clock ticks for executing the benchmark. The straight line with crosses represents the implementation using polymorphism, the short dashed line with the triangles represent the case-statement, and the long dashed line with the squares represent the if-statement.

if-statement
switch-statement
polymorfism

5.1. CodeWarrior

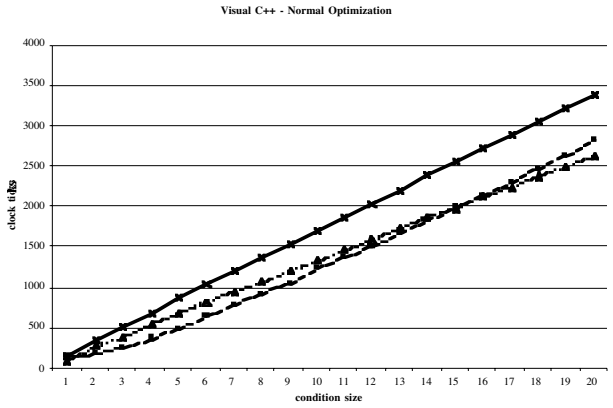
For the CodeWarrior compilers, we discover the expected trend with both normal and fast optimization. An if-statement is slightly faster than both the case-statement and polymorphism, but after a nesting level of five (in normal mode) and eight (in fast mode) the if-statement quickly performs worse. Both the case-statement and polymorphism run approximately as fast.

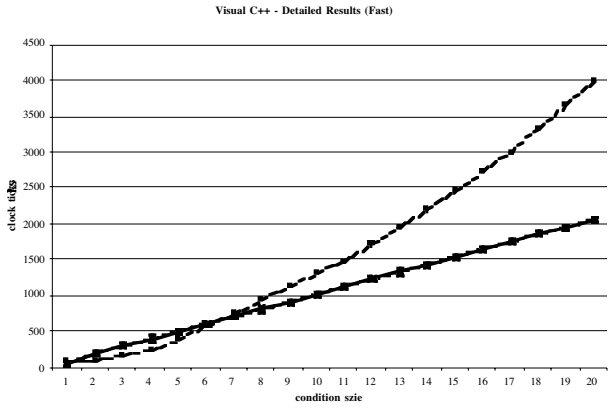




5.2. VisualC++

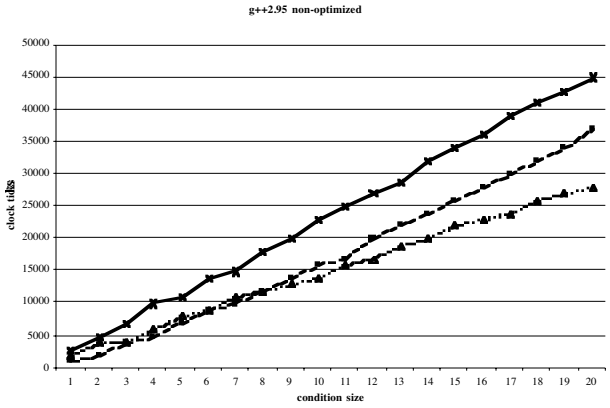
For the VisualC++ compilers, we notice a big difference between the normal and the fast optimization level. In the normal case, the polymorphic implementation is notably slower than the conditional one; while smaller if-statements are faster than the same case-statement. For the fast optimization case, we discover the expected trend; now an if-statement becomes slower than polymorphism after a nesting level of six.

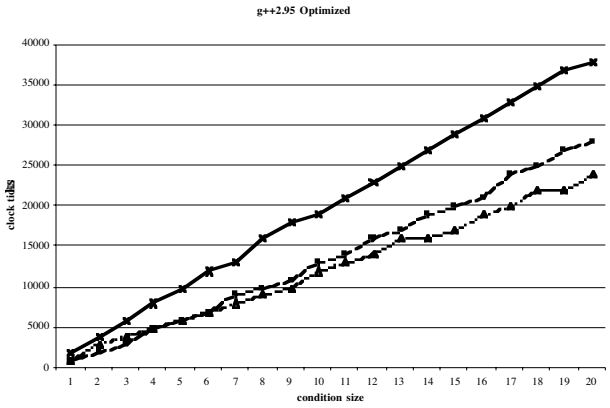




5.3. g++ 2.95

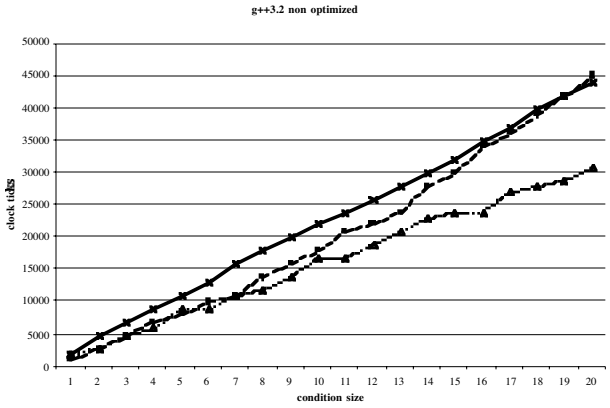
The g++ 2.95 compiler, was the one exception where we did not observe the expected trend. However, consulting the internet we found out that developers had been working on compiler optimizations, hence did the same experiment with the brand new 3.01 release.

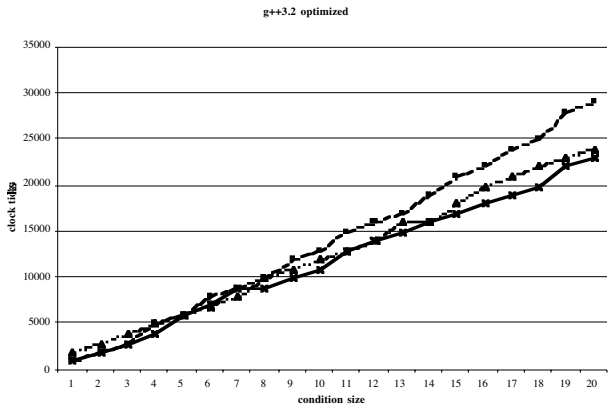




5.3. g++ 3.01

And indeed, the g++ 3.01 compiler in optimized mode showed the expected results, with a case-statement approximately having the same performance as the polymorphic implementation and with an if-statement that performs gradually worse when the nesting-level increases. However, here an if-statement is never faster than the polymorphic implementation.





6. Related and Future Work

Program transformations that preserve behavior but improve structure have been studied for as long as their exist programming languages [Gris93a]. Refactoring —the specific variant transforming object-oriented constructs— has been investigated in the context of schema-transformations for object-oriented database systems [Berg97a] and in the context of incremental program redesign performed by humans, yet supported by tools [Opdy92b], [Opdy93a], [John93b]. This line of work resulted in the Refactoring Browser, a tool that represents the state of the art in the field [Robe97a]. In contrast, both Casais and Moore report on tools that optimize class hierarchies without human intervention [Casa92a], [Moor96a]. More recently, Schulz et al. report on refactoring to introduce design patterns in C++ systems [Schu98a], while refactoring has also been studied in the context of UML [Suny01a].

Refactoring is now considered a mature technology and has among others been adopted by software development processes like eXtreme Programming [Beck99a]. Nevertheless, few empirical studies exist that verify the effect of refactoring on program quality. Tokuda has shown that a typical system evolution can be reproduced significantly faster and cheaper via refactoring [Toku99a]. A recent report by TogetherSoft indicates plans to measure the effect of refactoring on code quality attributes such as code complexity, coupling/cohesion, but the results are preliminary [Pitt00a]. Except for generic guidelines concerning performance (like for instance in [Meye96a]) the only hard data concerning the performance effect of refactorings we found was an old paper comparing two C++ implementations of a Scheme interpreter [Russ88a].

Future Work. We ourselves have participated in refactoring research as well. On the one hand, we investigate meta-models that could help us specify the result of a refactoring in a language-independent way [Deme99a], [Tich00a]. We continue this line of research investigating formal models to specify the behavior that should be preserved [Mens02a]. On the other hand, we catalogue reverse- and reengineering techniques in the form of *reengineering patterns* [Deme02a]. While putting together the catalogue, we discovered that there is little emperical evidence concerning the quality effects of reengineering techniques in general and refactoring in particular. Therefore, we plan further experiments with other languages (Java, Smalltalk), other cases (in particular the LAN-simulation [Mens02a]) and other quality factors (besides performance we are interested in the effect on maintainability and repairability).

7. Conclusion

This paper reports the results of a comparative study, investigating whether refactorings really make a program slower, given today's compiler and processor technology. During the study we compile a C++ benchmark program containing a large conditional (once implemented by means of a nested if-statement, and once using a large case-statement) and compare the performance against a program where this conditional is replaced by a polymorphic method call. The benchmark is performed with different compilers, optimization options and processors, to see whether the same trend can be observed across compilers and processors.

We discovered that for three of the four C++ compilers under study, the maximum optimization level produces code which makes polymorphic method calls run faster than nested if-statements and as fast as case-statements. Only a small if-statement will run faster than the corresponding polymorphic method call, but how small depends on the compiler/processor combination — it varies between zero and five. The one exception to the rule is the g++ 2.95 compiler, however this was an older version generating slower code. The more recent 3.0.1 version exploited better code-optimization algorithms and the results complied with the other compilers.

Therefore, we conclude that C++ programmers should not try to avoid virtual method declarations for the sake of performance. This study thus confirms the following observation:

> "*From the point of view of efficiency, however, you are unlikely to do better than the compiler-generated implementations by coding these features yourself*". *[Meye96a]*

Of course we did not try every possible C++ compiler, nor did we perform tests with other languages (such as Java and Smalltalk), so the results are somewhat inconclusive. We plan to do more tests in the future, but in the meantime every programmer can verify for himself

how well his particular compiler is dealing with polymorphic method calls.

The results give a fair indication with respect to the grander question: Does a refactored program run slower ? Our results show that transforming conditionals into polymorphism will make a program faster; however there are many other refactorings and it is unclear what their effect will be. Here as well, we plan further experiments, however based on the results reported in this paper, we advice that programmers should not overestimate the performance penalties since they are likely to be negligible.

Acknowledgments.

Thanks goes to Stéphane Ducasse and Oscar Nierstrasz for proofreading an early draft of this paper. Also, to our system administrator Pieter Donche for setting up the necessary infrastructure.

References

[Beck99a] Beck, Kent Extreme Programming Explained, Addison-Wesley,1999.

[Berg97a] Bergstein, Paul, "Maintenance of Object-oriented Systems during Structural Evolution", *Theory and Practice of Object Systems*, 3(3), pp. 185-212, Wiley & Sons, 1997.

[Casa92a] Casais, Eduardo, "An Incremental Class Reorganization Approach," *Proceedings ECOOP'92*, O. Lehrmann Madsen (Ed.), LNCS 615, Springer-Verlag, 1992.

[Copl92a] Coplien, Jim *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

[Deme99a] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. "Why unified is not universal. UML shortcomings for coping with round-trip engineering." Proceedings UML'99, volume 1723 of Lecture Notes in Computer Science, pages 630–644. Springer-Verlag, October 1999.

[Deme02a] Demeyer, Serge, Stéphane Ducasse and Oscar Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan-Kaufmann, 2002.

[Drie96a] Driesen, Karel, "The Direct Cost of Virtual Function Calls in C++", *Proceedings OOPSLA '96*, ACM Press, 1996.

[Drie99a] Driesen, Karel, "Software and Hardware Techniques for Efficient Polymorphic Calls", Ph.D. Thesis, University of California, Santa Barbara, June 1999

[Fowl99a] Fowler, Martin, Kent Beck, John Brant, William Opdyke and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Gamm95a] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns,* Addison-Wesley, 1995.

[Glass98a] R.Glass,"Maintenance:Less is not More", IEEE Software July/August 1998.

[Gris93a] Griswold, William, and David Notkin, "Automated Assistance for Program Restructuring", ACM Transactions on Software Engineering and Methodology, 2(3), July 1993.

[High99a] Highsmith, J., *Adaptive Software Development*, Dorset House Publishing,1999.

[John93b] Ralph E. Johnson and William F. Opdyke, "Refactoring and Aggregation," Object Technologies for Advanced Software - First JSSST International Symposium, Lecture Notes in Computer Science, Vol. 742, Springer-Verlag, 1993.

[Lehm85a] Lehman, M and L. Belady, *Program Evolution:Processes of Software Change*, Academic Press, 1985.

[Mens02a] Tom Mens, Serge Demeyer, and Dirk Janssens. "Formalising Behaviour Preserving Program Transformations". Proc. Int'l Conf. Graph Transformation, LNCS ???, pp. ??-??, Springer-Verlag, 2002 (to appear).

[Meye96a] Meyers, Scott, *More Effective C++,* Addison-Wesley, 1996.

[Moor96a] Moore, Ivan "Automatic Inheritance Hierarchy Restructuring and Method Refactoring," *Proceedings OOPSLA '96*, ACM Press, 1996.

[Opdy92a] Opdyke, William, *Refactoring object-oriented frameworks*. Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Technical Report UIUC-DCS-R-92-1759, 1992.

[Opdy93a] William F. Opdyke and Ralph E. Johnson, "Creating Abstract Superclasses by Refactoring," Proceedings CSC'93, ACM Press, 1993.

[Robe97a] Roberts, Don, John Brant and Ralph E. Johnson. "A refactoring tool for Smalltalk", *Theory and Practice of Object Systems* 3(4): 253-263, 1997.

[Russ88a] Russo, V. F. and S. M. Kaplan, "A C++ Interprter for Scheme", Proceedings of the USENIX C++ Conference, Usenix Association, 1988, pp.95-108.

[Schu98a] Benedikt Schulz, Thomas Genssler, Berthold Mohr and Walter Zimmer, "On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems," *Proceedings TOOLS 27 - Asia '98*, IEEE Computer Society Press, 1998.

[Suny01a] Gerson Sunyé, Damien Pollet, Yves LeTraon and Jean-Marc Jézéquel, "Refactoring UML models",

Proceedings of UML2001, LNCS 2185, Springer Verlag 2001.

[Tich00a] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz, "A meta-model for language-independent refactoring", In Proceedings ISPSE'2000 (International Symposium on Principles of Software Evolution), IEEE Press, November 2000.

[Toku01a] Lance Tokuda and Don Batory, "Evolving Object-Oriented Designs with Refactorings," *Automated Software Engineering*, pp. 89-120, Kluwer Academic Publishers, 2001.

[Pitt00a] Pitt, R, A.R. Carmichael, "Measuring the Effect of Refactoring", Proceedings of OOIS'2000 (6 th International Conference on Object Oriented Information Systems)

[Wool98a] Woolf, Bobby, "Null Object," in *Pattern Languages of Program Design 3*, Robert Martin, Dirk Riehle and Frank Buschmann (Eds.), pp. 5-18, Addison-Wesley, 1998.