Impact of Code Refactoring using Object-Oriented Methodology on a Scientific Computing Application

Malin Källén
Department of Information Technology
Uppsala University, Sweden
Malin.Kallen@it.uu.se

Sverker Holmgren
Department of Information Technology
Uppsala University, Sweden
Sverker.Holmgren@it.uu.se

Ebba Póra Hvannberg
Computer Science
University of Iceland, Iceland
ebba@hi.is

Abstract—Methods and tools for refactoring of software have been extensively studied during the last decades, and we argue that there is now a need for additional studies of the effects of refactoring on code quality and external code attributes such as computational performance. To study these effects, we have refactored the central parts of a code base developed in academia for a class of computationally demanding scientific computing problems. We made design choices on the basis of the SOLID principles and we used object-oriented techniques, such as the Gang of Four patterns, in the implementation. In this paper, we discuss the effect on maintainability quantitatively and also analyze it qualitatively using a set of software metrics extending the Chidamber-Kemerer suite. Not surprisingly, we find that maintainability has increased as an effect of the refactoring. We also study performance and find that dynamic binding, which inhibits inlining by the compiler, in the most frequently executed parts of the code makes the execution times increase by over 700%. By exploiting static polymorphism, we have been able to reduce the relative increase in execution times to less than 100%. We argue that the code version implementing static polymorphism is less maintainable than the one using dynamic polymorphism, although both versions are considerably more maintainable than the original code.

I. INTRODUCTION

A. Refactoring

Refactoring is defined by Martin Fowler [1] as ‘the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure’. William Opdyke [2] uses the following definition of unchanged behavior: ‘if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same’.

Several research efforts focusing on methods and tools for refactoring are described in a survey by Mens and Tourwé [3]. However, less emphasis has been put on studying the effect of refactoring on different qualities of the code, eg maintainability [4] and computational performance. Mishbaudin and Alshayeb [5] remark that out of 49 papers examined by them, only ten comprise experimentally validated results. Also, only six of the 94 articles referred to address the quality of models of code in a structured way. Hence, we argue that there is a need for studies evaluating how refactoring affects the quality of the code. This argument is also brought forward in eg Zhang et al. [6].

B. Refactoring of Academic Software

We see a specific need for more research on refactoring of scientific applications from academia. Development and usage of scientific software becomes more and more important for scientists [7]. In academia, software is generally not developed and maintained by software professionals, but by scientists in the scientific application discipline [8]. Scientists are more knowledgeable in constructing software than in designing, testing and maintaining it [7]. Kelly discusses three case studies of evolution of scientific applications [9]. She notes that changes suggested by developers who are not skilled professionals do not necessarily result in better software. Instead, they can result in increased cognitive load for developers and the modified code can even be less maintainable than the original version.

C. Maintainability vs. Performance

In this study, we evaluate which effect refactoring may have on code quality, using a set of internal object-oriented software metrics, and also discuss the code quantitatively. We use the ISO definition of maintainability [4] to define code quality. We perform a case study on a piece of academic software, which is described in Section II.

As will be explained, an execution of the software can take several weeks. Such execution times are not uncommon for scientific software. For reasons of computational costs, performance is an important aspect of the software discussed in this study, and of many other scientific code bases. Consequently, improvements on the maintainability of the code should not be allowed to substantially deteriorate computational performance. Demeyer [10] shows that replacing conditionals with polymorphic method calls often results in faster code, but we are not aware of any earlier research papers where both maintainability and performance is studied for scientific computing software. In this study, we will do so and focus on finding a balance between the two qualities.

II. HAParaNDA

We have refactored a code base called HAParaNDA [11], which is developed in an academic setting. HAParaNDA is an iterative solver of time-dependent, high-dimensional linear partial differential equations (PDEs) on structured computational grids. In an iterative solver, values of a function are
repeatedly operated on until they are sufficiently close to the solution of the equation. For a description of structured computational grids, see e.g. Bruaset [12].

The class of equations addressed by HAParaNDA arises in a wide span of application areas, ranging from quantum chemistry over systems biology to computational finance. Solving an equation that is of interest in the application area can take days or weeks of continuous computing. Therefore, HAParaNDA is designed for use on large-scale parallel computers and much effort has been put on achieving close-to-optimal performance. However, less consideration has been given to maintainability. In this section, we provide a brief description of the parts of HAParaNDA that we have refactored. In the remainder of this paper, we will use the term the original code when referring to the code base as it was before refactoring.

HAParaNDA consists of 12 000 lines of C++ code, parallelized using a hybrid parallelization model. OpenMP [13] is used for parallelization over threads within a compute node, that is a number of cores organized as one or a few CPUs, with shared memory. MPI [14] is used for parallelization over such nodes.

A. Spatial Domains and Function Data

When a PDE is solved numerically, the solution is represented as a grid function with values given only at a discrete set of grid points (more generally, degrees of freedom) representing the spatial domain. The spatial domain in HAParaNDA is an orthotope ("hyper-rectangle") of arbitrary dimensionality, $d$. The dimensionality may be higher than 3 and a typical use case of the code is solving six-dimensional problems. The domain is intended to be discretized using a block-structured grid, consisting of $d$-dimensional orthotopes, called domain blocks as described by Gustafsson et al. [15]. The idea is that the domain blocks will be stored in a tree, where the domain is the root and the successively smaller blocks form the branches. The function data is stored in the leaf blocks. Currently, the code base only provides limited support for block-structured grids with equal-sized domain blocks. In the future, support for blocks of different size, and with different ways of storing the data, will be needed.

B. Operators

There are two types of operators in HAParaNDA. One is linear operators, which can be applied on an entire grid function. The other one is block operators, which are applied on one block at a time. Currently, there is one type of block operator implemented, namely a finite difference stencil. A finite difference stencil is basically a weighted sum and is used to approximate spatial derivatives of grid functions, which is a very frequently executed operation in the PDE solver. In Figure 1, an approximative value of the derivative at the center point of a stencil is computed as a weighted sum of the function values at the points covered by the stencil. In the future, there will be a need for other block operators as well, particularly for other types of derivative approximations.

C. Ghost Regions

When a stencil is applied close to the boundaries of a block, data from neighboring blocks are needed. This is implemented using a halo exchange technique as described e.g. by Bruaset [12]. The data received from the neighboring blocks is stored in data buffers referred to as ghost regions. A block with ghost regions is illustrated in Figure 1.

III. REFACTORING HAPARAANDA

The parts of HAParaNDA that are refactored in this study are the ones handling the spatial domain, including ghost regions, and the block operators. The refactored code constitutes about 40% of the total code base, but contains the parts in which the largest fraction of the execution time is spent. We plan to refactor the remaining 60% of the code in the future.

A. Preparatory Work

1) Identification of Needs for Refactoring: As a first step in the refactoring process, we identified a number of code smells [1] [16]. Three smells that had a large impact on maintainability was:

- Divergent change -in several classes
- Shotgun surgery -in several classes
- Inappropriate intimacy -mainly when the grid function class, which exposed several member variables publicly, is used

Thereafter, based on these smells and on the SOLID principles [17], we identified a number of refactoring actions, which we documented before starting changing the code. As an example of a refactoring action, we eliminated the occurrences of both divergent change and shotgun surgery by moving methods to classes where they belong. In some cases, this required extraction of new classes, for example the iterators described below. Our strategy for eliminating
the inappropriate intimacy was mainly to hide the instance variables, in some cases after moving them to other classes. During the refactoring process, we made considerable changes in the structure of the code and we describe the resulting design in Section III-B. We have taken performance into consideration in several ways. We describe some of them below, but we have also taken actions on a level that is too detailed for this paper.

2) Tests: For every unit of code that has been a target for the refactoring described in this paper, or that has been added or considerably changed as an effect of the refactoring, we wrote at least one test before starting changing the code. We wrote unit tests for each concrete class, with a few exceptions: When all classes used by a certain class were tested elsewhere and it was impractical to write a unit test for it, we employed an integration test instead. The results of the testing revealed several defects in the original code. Before starting the refactoring of a unit of code, we corrected all defects that we had discovered.

B. New Code Structure

1) Multi-Dimensional Fields: In order to apply operators (eg stencils) in HAParaNDA, one needs to iterate over the domain blocks. In the following description, we use the more general term field when referring to a block or to the entire domain. The dimensionality of the fields can vary from one run to another.

In the original code, two different solutions were used to implement iterations over fields. Both solutions resulted in code that was difficult to maintain, and also required that the dimensionality of the fields was known at compile-time. In the refactored code, we have specified an iterator interface, c.f. the Iterator pattern [18]. As we were not sure of the performance effects of defining the dimensionality at run-time, the refactored code also requires it to be known at compile time. However, we have written the code such that binding the dimensionality dynamically in the future will require a small amount of work. We have implemented iterators for both pure fields and composed fields. A pure field consists of a single field while a composed field also contains information about other fields, eg ghost regions. We wrote the pure field iterator and the composed field iterator as abstract subclasses of the iterator interface. These subclasses implement all the iterator operations. In the implementation of the composed field iterator, we used the Decorator pattern [18]: The composed iterator decorates the iterator for a pure field with iterators for the other fields.

The pure field iterators use strategy classes for stepping through the field and for accessing elements, c.f. the Strategy pattern [18]. We applied the pattern Factory Method [18] for creating the strategies in the pure field realizations and for creating the iterators needed in the composed field iterator realizations.

During an application of a block operator, a large number of calls to dynamically bound methods in the iterators, and the corresponding strategy classes, is needed. This means that there is a big potential performance impact of this solution. In order to assess the performance effect, we created an alternative version of the code, where we applied static polymorphism on these classes using the Curiously Recurring Template Pattern (CRTP) [19]. This solution has also been used eg for iterators in Boost [20]. We implemented CRTP as described by Lutz et al. [21].

In the following, we will refer to the refactored code version that applies dynamic polymorphism as the dynamic code. We will use the term the static code when referring to the alternative version. When we write the refactored code, we mean both the dynamic code and the static code.

We also introduced an interface Iterable which guarantees that iterators for the inner regions and for the boundaries can be retrieved for a data structure. All classes in which elements need to be iterated over implement this interface. However, for performance reasons, when domain block structures need to be iterated over, we integrated the iterators into the classes that hold the blocks.

2) Blocks: We implemented the tree structure described in Section II-A using the Composite pattern [18]. In order to make it easy to extend the code base with different numerical methods, requiring different ways to represent the data, we made the leaf class abstract. This class keeps track of the grid function indices stored in the block that it represents. We also introduced an additional type of block, computational blocks, to provide a temporary container for grid function values from a block that is being used in the computations. We expect these blocks to facilitate future implementation of different numerical methods. We represented this type of block by an abstract class, and realized it with pure and composited blocks respectively. The former realization can be used eg for the output from a block operator, as there is no need to use ghost regions for output data. Composed computational blocks, on the other hand, are suitable for input data of the block operators. Following the Dependency Inversion Principle (DIP) [17], we used an abstract factory [18] for creating the different types of blocks.

3) Grid Functions: In the original code, the grid function class keeps both function data and detailed information about the structure of the grid on which the function resides. In the refactored code, instead, we use domain blocks to store the information about the grid. The grid function class contains an iterator over the domain blocks. The iterator differs from the ones described above in the sense that instead of returning a block, the current method takes a computational block as its argument and initializes it.

We use the Strategy pattern for initialization of the grid functions. The abstract base class employs the pattern Template Method [18] and delegates computation of the actual function values to its subclasses. The refactored code also provides the possibility to initialize a grid function using a composition of functions that depend on one variable each.

4) Block Operators: From the finite difference stencil in the original code, we extracted two levels of abstract base classes, one that represents block operators in general, and one that
represents multuncial\(^1\) stencils. Naturally, the block operator class is the base class of the multuncial stencil class. We implemented template methods on both levels. The concrete subclass, which represents a multuncial stencil with constant weights, uses a Strategy pattern to initialize its weights.

Like the iterators, the block operators contain polymorphic methods that are repeatedly called in the most frequently executed part of HAParaNDA. Consequently, in the static code we also applied static polymorphism in the block operator class hierarchy.

C. Qualitative Evaluation

We find that the dynamic code adheres well to all five SOLID principles and that the static code adheres well to four of them. As the classes in the refactored code have limited and well-defined responsibilities, we consider the design to agree well with the \textit{Single Responsibility Principle} (SRP) [17]. The two main differences compared to the original code are:

- Thanks to the iterators, the index handling is performed separately from the operations on the field elements. An effect of this is that the readability has increased, as the index operations are not hiding the actual computations on the field elements.
- We have separated the blocks from the grid function class instead of mixing grid structure with function data.

Our efforts to follow SRP have made all interfaces small. Consequently, with the exception of the interface \texttt{Iterable}, we have not seen any need to split the interfaces further in order to follow the \textit{Interface Segregation Principle} (ISP) [17].

We have introduced several abstract base classes, which implement behavior that we expect to be common for all derived classes. An example is the use of Template Methods in the grid function initiators and in the block operator classes. The pattern makes it possible to add new subclasses, which alters the current behavior, with a small coding effort. This agrees well with the \textit{Open-Closed Principle} (OCP) [17].

Moreover, we made use of the Strategy pattern in the iterators, grid functions and block operator classes. New strategies can easily be added, and the strategies can be reused in different classes. This solution limits the efforts needed to add new iterators, grid functions or block operators. All in all, the strategies contribute to a better adherence to OCP. Before refactoring, adding functionality to the code was time-consuming and error-prone, and we are of the opinion that we have made considerable improvements on this point.

In the refactored code, we cannot find any subclass that is not substitutable for its base class, and therefore conclude that all classes follow the \textit{Liskov Substitution Principle} (LSP) [17].

Thanks to the usages of the patterns Abstract Factory and Factory Method, we have been able to keep the number of dependencies to concrete classes low in the dynamic code, which agrees with DIP. However, at this point, CRTP comes at a price. In all places where a class that implements static polymorphism is used, the concrete type of this class must be known. In other words, we break DIP in the static code. This has a considerable effect on the changeability, and most of the smoothness brought by the dynamic binding is lost.

There are also other maintainability issues of using CRTP. The fact that the pattern does not allow virtual methods makes it harder to add a class in a hierarchy where CRTP is applied, since it is not possible to see from the interface which methods can, or must, be overridden. This is a severe limitation. It also brings the consequence that it is not possible to implement hooks, ie methods implementing default behavior, which can be overridden when needed, in the base class. In addition, the fact that virtual inheritance cannot be used at the same time as CRTP complicates our iterator class hierarchy.

In addition to better following the SOLID principles, we argue that we have improved data hiding by being able to hide several of the member variables that are public in the original code.

Finally, we note that the refactoring is likely to having improved the load balancing of the code. In the original code, a number of nested loops were used when a field was iterated over. The parallelization, ie partitioning of loop indices using OpenMP, was done only at the outermost loop level. In the refactored code, the iterators internally view the field as a one-dimensional data structure, which can more easily be partitioned in equal chunks when the code is parallelized.

To conclude, our qualitative evaluation of the refactored code has convinced us that the maintainability has improved as an effect of the refactoring. However, we also find that there is a considerable risk of the refactoring having degraded the performance of the code. In the two following sections, we perform quantitative evaluations of the maintainability and of the computational performance.

IV. SOFTWARE METRICS

A. Background

A widely used object-oriented metrics suit is presented by Chidamber and Kemerer [22]. Many alternative internal metrics have been developed and used in other studies, eg fan-in and fan-out [23], CAMC [24] and SCC [25]. Both code cohesion and code coupling are included in the suite presented by Chidamber and Kemerer [22], and these metrics are further discussed eg by Moghadam and Cinneide [26] and by Akiyama et al. [27]. Hopkins [28] collects software metrics such as cyclomatic complexity and number of knots for some numerical libraries, but we are not aware of any efforts to collect object-oriented software metrics for scientific computing software.

Using internal design metrics is not the only way to measure maintainability. Other metrics are related to the stability of the software, ie the number of changes done, or the number of bugs. However, we find internal metrics being most relevant to apply in this study, as the time passed since the refactoring is short. Moreover, the measures of eg changes done may be misleading for HAParaNDA, as the current main developer

---

\(^1\)\textit{Multuncial} is an adjective describing an object with the shape of a multunx, which in turn is our dimensionality-independent generalization of the term quinuncx.
has different routines for changing and checking in code than her predecessor. We have chosen to use the above-mentioned Chidamber and Kemerer suite to evaluate the refactoring, as the metrics are well motivated, and related to maintainability. Moreover, the fact that the suite is commonly used makes future comparisons to other studies easier. We have also collected two additional metrics which we find relevant, namely number of Lines Of Code (LOC) [29] and Average Method Size (AMS) [29] for each class. These metrics can give an indication of the ease of understanding each class or method.

**B. Data Collection**

We collected the metrics values for the original code and the dynamic code using the tool *Understand* [29], Build 731. Unfortunately, we have not been able to make *Understand* identify the classes on which we have applied CRTP as classes. This means that we have not been able to collect the metrics values for the static code. Instead, we discuss the expected effect on the metrics values at the end of this section. In cases where *Understand* lacks support for collecting a separate metric, we complemented the results by measurements that we performed by direct inspection of the code.

We measured at class or class template level for both the original and the dynamic code and treated each class template as one class. Likewise, we treated structs as classes. For simplicity, we hereinafter use the term ‘class’ when referring to a class as well as to a struct or a class template. In the measurements, we included all classes that have been added, removed or considerably affected by the refactoring process, but we excluded test code. We also noted the number of classes analyzed.

Parts of the original code consist of functions implemented outside classes. In order to be able to compare the different versions of the code, we treated each file that consists of such functions as a class. Our motivation for this is that these files can be seen as a manifestation of how earlier HAParaNDA developers structured the code. If these developers would have used a language that forces use of classes, they would probably have divided the code into classes in the same way as they have now divided it into files.

One class template in the original code lacked some includes and a declaration of a local variable. Here, we added the missing includes and declaration in order to make *Understand* able to analyze the template.

It should be noted that one of the classes (Potential) that we included in the measurements has not been an explicit target for refactoring, but it has nevertheless been considerably modified: As an effect of refactoring of other parts of the code, about 70% of this particular class has been moved to other classes.

We measured LOC, AMS, Depth of Inheritance Tree (DIT) [22] and Number of Children (NOC) [22] using the *Understand* metrics CountLineCode, AvgLineCode, MaxInheritanceTree and CountClassDerived respectively for each class. For practical reasons, we used the metric PercentLackOfCohesion in *Understand* for collecting the Lack of Cohesion in Methods (LCOM) [22] values. However, the definition is different from the one proposed by Chidamber and Kemerer. The Coupling Between Object classes (CBO) metric [22] includes both afferent (incoming) and efferent (outgoing) coupling. We used the *Understand* metric CountClassCoupled to collect the efferent coupling for each class, and manually inspected the code in order to collect the afferent coupling values. To calculate the Weighted Methods per Class (WMC) [22] values, we used the metric SumCyclomaticComplexity. This corresponds to using the cyclomatic complexity as the weight (ci in the original definition). In order to collect the Response For a Class (RFC) [22] values, first, we used the metric CountDeclMethodAll in *Understand* to count the number of methods in each class, including the inherited ones. Thereafter, we inspected the code manually in order to find the number of external methods called directly from each class and its base classes, and added these values to the ones from the first step. Following Chidamber and Kemerer [22], we defined an external method to be a method or macro that is declared outside the class and is part of the framework. We included explicit constructor and destructor calls, but no implicit ones.

**C. Results**

The number of classes for which we collected metrics values is 25 in the original code and 47 in the dynamic one. We summarize the results from the metrics collection in Table 1. We also provide histograms for the average CBO, WMC and AMS values before and after refactoring (see Figures 2, 3 and 4 respectively).

As can be seen from Table 1, the values of DIT and NOC were close to zero before the start of the refactoring. Consequently, the relative increases of these values are huge. The decrease of the average value of CBO is small, but it can be seen from Figure 2 that a larger portion of the classes has a relatively low coupling after refactoring, while a few classes with high coupling keep the average value close to that in the original code. The average LCOM value has increased by 49% and the average RFC value has more than doubled, while the average WMC, LOC and AMS values have about halved. The single class with an average AMS value of more than 50 (see Figure 4) is Potential.
The fact that both DIT and NOC were approximately zero in the original code base means that a low coupling is an indication of a modular way) [16] [1], which in turn facilitates changing the code. A high WMC value indicates that the classes are complex and that encapsulation is used to a higher extent. The increase in the values of DIT and NOC indicate that we make use of the possibilities of reuse brought by object-orientation better in the dynamic code [22]. As discussed in Section III-C, this facilitates extending the code. Despite the large increase, we consider DIT being low also for the dynamic code and suppose that this is an effect of us favoring object composition over inheritance in the refactoring as argued for by Gamma et al. [18]. A high value of DIT would have indicated a complex design [22].

2) LOC and LCOM: We attribute the large decrease in LOC to the dynamic code following SRP to a higher degree than the original one (see Section III-C). The better adherence to this principle should also cause the LCOM values to decrease, which is not the case in this study. We argue that this is an effect of the PercentLackOfCohesion being measured in a deceptive way. Firstly, as the metric considers the number of methods using a certain variable, the usage of helper methods increases the lack of cohesion. However, this type of methods can significantly increase the readability of the code [1] [16]. Secondly, abstract methods are included in the metric, although they naturally never use any member variable. Therefore, the introduction of abstract base classes, which we claim increase maintainability, causes an increase in the LCOM values. Finally, the metric is defined such that its value is zero for classes without member variables, and for classes without methods. As a majority of the classes in the original code lack member variables, we find that comparing the values before and after refactoring is somewhat misleading.

3) CBO: A low coupling is an indication of a modular design, which is important for maintainability [22]. However, we find that extraction of classes is likely to cause a higher amount of coupling, although it increases modularity. During the refactoring, we have extracted several classes from the original ones. If no other refactoring actions were performed, we are convinced that this would have caused a considerable increase in the coupling values. Hence, the fact that the coupling has actually decreased as an effect of the refactoring indicates that we have improved the modularity of the code base. We expect the average value of the coupling of the classes covered by this refactoring to decrease even more later in the refactoring process. We motivate this by noting that one of the classes that has a high number of couplings from parts of the code that are not yet refactored.

4) AMS and WMC: Short methods increase readability of a code base (as long as the methods are defined in an adequate way) [16] [1], which in turn facilitates changing the code. A high WMC value indicates that the classes are complex and that they are likely to be application specific, which limits the potential for reuse. Another class has lots of incoming couplings from the start of refactoring. We argue that the increases of these values show that the code has gone from procedural to object-oriented. We further support this statement by noting that the portion of instance variables that are public has decreased from 44% to 6% in the refactored parts of the code, which indicates that encapsulation is used to a higher extent. The increase in the values of DIT and NOC indicate that we make use of the possibilities of reuse brought by object-orientation better in the dynamic code [22]. As discussed in Section III-C, this facilitates extending the code. Despite the large increase, we consider DIT being low also for the dynamic code and suppose that this is an effect of us favoring object composition over inheritance in the refactoring as argued for by Gamma et al. [18]. A high value of DIT would have indicated a complex design [22].
A major reason for the increase of the RFC values is the usage of the helper methods mentioned above. Another reason is that two of the most frequently inherited classes have a high RFC value compared to the rest of the framework. This propagates to the derived classes and the base class methods contribute to the average value multiple times. However, as argued in Section III-C, usage of inheritance has improved adherence to SRP, OCP, ISP and DIP. Hence, in this particular case, we argue that changes that actually improve maintainability have increased the RFC metric value.

5) RFC: A major reason for the increase of the RFC values is the usage of the helper methods mentioned above. Another reason is that two of the most frequently inherited classes have a high RFC value compared to the rest of the framework. This propagates to the derived classes and the base class methods contribute to the average value multiple times. However, as argued in Section III-C, usage of inheritance has improved adherence to SRP, OCP, ISP and DIP. Hence, in this particular case, we argue that changes that actually improve maintainability have increased the RFC metric value.

6) Effects of Static Polymorphism: As mentioned in Section III-C, CRTP prevents usage of virtual inheritance. This forced us to remove one class from the iterator hierarchy and merge its behavior (basically one method) into other iterator classes. As a consequence of the removal of the class, we expect small decreases in the NOC and DIT in the static code, compared to the dynamic code. We also expect a small increase in the average LCOM value as an effect of integrating the behavior from the removed iterator into other classes. This increase reflects a real, albeit small, reduction of the adherence to SRP.

When implementing CRTP in a class hierarchy, we were forced to implement some methods, which are abstract in the dynamic code, in the base class. As an effect of this, we expect the WMC and RFC metrics values to be larger than those of the dynamic code. However, we expect the increase to be small, as the number of new methods and method calls is small compared to the total number. As an effect of the new methods, and of the fact that the pattern forces us to keep track of the concrete type of some classes, we also expect small increases in the AMS and LOC values.

The basic idea of the CRTP pattern is to let the concrete type of a class be known by every class that uses it. As the abstract type must also be known, and since the iterators are used by several classes in the framework, we therefore expect the application of CRTP to considerably increase the CBO values. As discussed above, this is an important metric and the increases are a sign of decreased maintainability. This is in accordance with the qualitative evaluation in Section III-C, which showed that CRTP decreases maintainability.

V. PERFORMANCE

A. Data Collection

To assess the performance impact of the refactoring, we performed a number of runs of the different versions of the code and reported the elapsed times. We used a test code that mimics the typical use of the refactored part of HAParaNDA: A finite difference stencil is repeatedly applied to a block of a grid function. We ran a test program using two-dimensional as well as six-dimensional stencils, which were applied ten times in each execution. For each application, the output of the previous one was used as input, which mimics the behavior of an iterative solver (see Section II).

We expect future users of HAParaNDA to make tens of thousands of stencil applications during an execution. The time for set-up and tear down is comparable to just a few stencil applications, which is negligible in that context. Consequently, we did not include set-up and tear down in the time measurements. We executed the test code on a computing cluster which comprises 2560 computational cores in the form of 160 dual CPU (AMD Opteron 6220 "Bulldozer") Supermicro nodes with 64 or 128 GB RAM per node. The nodes are interconnected with QDR Infiniband. The cluster runs Scientific Linux. More information about the system is available online [30]. We ran all experiments on the 64 GB nodes, and reserved full nodes for all runs.

We compiled the code using gcc version 4.8.2 with the flags \
-ffic -O3 -DNDEBUG. When compiling and running the parallel code, we used Open MPI version 1.7.5. To assess the effect of inlining, we also compiled the two refactored versions of the code with inlining disabled. Following Haveraaen and Hundvebakke [31], we ran the test program five times for each experimental set-up and reported the shortest execution time.

First, we ran all the five above-mentioned versions serially on different domain sizes. The two-dimensional domain sizes ranged from 1024² to 32768² elements while the six-dimensional ones ranged from 8⁶ to 32⁶ elements. The largest problem sizes made use of at least 50% of the memory available on a compute node.

As a next step, we ran the three code versions in which inlining was enabled on a varying number of threads on a single cluster node. A node has a fixed amount of RAM and stencil applications are memory bound, which makes it relevant to find how many threads the code scales to. Therefore, we performed a strong scaling study [32], which means that we used fixed problem sizes and varied the number of threads. We used the problem sizes 32768² and 28⁶ elements, which filled up around 50% of the memory available on a node.

Finally, we evaluated the parallel performance using the code versions compiled with MPI, and inlining enabled. We used multiple cluster nodes and employed hybrid (inter-node and intra-node) parallelization. In this setting, the amount of available memory depends linearly on the number of nodes used. Hence, we performed a weak scaling study [32], which means that the problem size is proportional to the number of cluster nodes used. As we expect future users of HAParaNDA to use all resources available, we ran 16 threads on each node.

B. Results

In Figures 5 and 6, we present the serial execution times for different domain sizes and set-ups. We see that the execution times increase linearly with the domain size, which is consistent with the increase in computational work. We also note that the execution times for the dynamic code are 3.4 – 5.7 times those of the original one. For the static code, the difference is only about a factor 1.6 in the two-dimensional case and 1.7 – 2.0 in the six-dimensional case. The difference in execution times for the dynamic code and the static code is at most 4% when inlining is disabled. In Figures 5 and
Fig. 5. Serial execution time as a function of domain size for two-dimensional problems.

Fig. 6. Serial execution time as a function of domain size for six-dimensional problems.

6, the two curves showing these execution times are hardly distinguishable.

Figures 7 and 8 show the execution times from the strong scaling experiments for $32768^2$ and $2^6$ elements respectively. The execution times for the dynamic code are $4.0 - 8.3$ times those of the original code in the two-dimensional case and $3.6 - 6.5$ times those of the original code in the six-dimensional case. The corresponding numbers for the static code are $1.3 - 1.7$ times and $1.1 - 2.0$ times respectively.

In the two-dimensional case, we observed similar scaling properties for the original code and the static code up to 8 threads. The execution times decreased with almost 50% when the number of threads doubled. However, when we increased the number of threads from 8 to 16, the execution time for the static code decreased with 31%, while the execution time for the original code decreased with only 9%. As a result, the difference in the execution time of the static code and the original code was only 26% when we ran the code on 16 threads. The execution time for the dynamic code also scaled more or less linearly up to 8 threads. However, the execution times did only decrease with $30\% - 40\%$ when the number of threads doubled. When we increased the number of threads from 8 to 16, on the other hand, the execution times decreased with 54%.

In the six-dimensional case, on the other hand, the execution time for the dynamic code decreased by about 50% when the number of threads doubled, up to 8 threads. The static code scaled in a similar way, but the increase from 1 to 2 threads yielded a decrease of only 38%. When we increased the number of threads from 8 to 16, the execution times decreased by only a little bit more than $20\%$ for both refactored versions of the code. The relative decreases in execution time of the original code were about $50\%$ when we increased from 1 to 2 and from 2 to 4 threads respectively. However, when increasing the number threads further, we gained less performance and when we ran the test code on 16 threads, the static code was only 13% slower than the original one.

Figure 9 shows the results from the weak scaling experiments in six dimensions. Here, the execution times for the dynamic code are $2.8 - 3.3$ times those of the original code.
code. The execution times for the static code are 0.9 – 1.1 those of the original code. In the two-dimensional case, the corresponding numbers are 3.8 – 4.3 and 1.1 – 1.5 respectively. We see from Figure 9 that in six dimensions, the execution times vary only slightly with the number of nodes for all three versions of the code. In two dimensions, the behavior is similar, but the execution times for the static code increase with 40% when the number of nodes increases from 4 to 8.

C. Discussion

The execution times for the dynamic code are considerably larger than for the original code. The experiments suggest that an important reason for the performance loss of our application is dynamic binding of methods called from the most frequently executed parts of the code. This may seem contradictory to the results obtained by Demeyer [10]. However, an important difference is that there were no conditionals where we introduced polymorphism, as the original code only had support for one type of blocks, stencils etc. If new functionality was to be added to the original code, conditionals (or a considerable amount of code duplication) would be needed where we have introduced polymorphism. Hence, without the refactoring, this performance reduction could be hard to avoid when HAParaNDA, in the future, is extended with more functionality.

From the execution times of the versions with inlining disabled, we conclude that the performance culprit is not the mechanics of the dynamic binding per se (eg vtable lookups), but mainly the fact that it prevents inlining at compile time. This is in accordance with the results obtained by Demeyer [10] since if the mechanics of the dynamic binding were to blame, replacing conditionals by polymorphism would have resulted in slower code in that study.

We do not think that an increase of up to more than 8 times in the execution times is acceptable for users of HAParaNDA. By implementing static polymorphism, we have been able to limit the increase in execution times to up to 2 times. This is still not negligible for an application that can take days to run, but it is a clear improvement compared to the dynamic code. In Sections III-C and IV-D, we argued that application of static polymorphism decreased maintainability, but the static code is still more maintainable than the original one. Consequently, static polymorphism can be a compromise which gives a reasonably maintainable object-oriented code without losing too much performance compared to a procedural code written for optimal performance.

From the strong scaling experiments, we saw that the refactored code scales more or less linearly up to 8 threads. However, the scaling is worse for the two-dimensional runs of the dynamic code than for the other runs. Further analysis is needed in order to explain this. The original code also scales linearly up to 8 threads in the two-dimensional experiments, but only up to four threads in the six-dimensional experiments. A reason for this can be the imperfect load balancing in the original code (see Section III-C). 28 domain slices can be evenly distributed between 1, 2 or 4 threads, but not between 8 threads. If so, the effect of the improved load balancing in the refactored code can be seen in this experiment.

However, when increasing from 8 to 16 threads, we saw only small decreases in the execution times for the original code in the two-dimensional as well as in the six-dimensional case. This cannot be explained by inadequate load balancing, as 32768 domain slices can be evenly distributed between 16 threads. Instead, there seems to be other factors limiting the performance.

In the experiments performed, the execution times of the refactored code decreased also when the number of threads increased from 8 to 16. Consequently, on 16 threads, the differences in the execution times of the original code and the static code are small. In one of the parallel experiments, the static code was even faster than the original one. As HAParaNDA is aimed for runs in parallel environments, we find this an encouraging result.

When the runs were parallelized over nodes, we saw small variations in execution time. Constant execution times would correspond to linear speedup (perfect scaling). Hence, we conclude that all versions of HAParaNDA can efficiently make use of at least up to 16 nodes (256 threads) on the computer system used.

VI. CONCLUSIONS

In this paper, we have refactored a computationally demanding code base using object-oriented methodology. We have evaluated the effects on maintainability as well as on performance and discussed how a balance between the two factors can be achieved. This is a case study on a performance critical code base for scientific computing and the results are not generalizable to all software, but can be worth considering for developers of code bases similar to ours.

Expectedly, the qualitative and quantitative results indicated that the refactoring has turned a procedurally written code into a considerably more object-oriented one. From our qualitative evaluation, we concluded that the refactored code adheres well to the SOLID principles and is more maintainable than the original one. Quantitatively, we also found that the refactored code has less dependencies and complexity, and smaller units of code.

On the other hand, the values of two collected metrics, RFC and LCOM, have increased. We claim that an important reason for the increase of the values of these is the fact that we have introduced numeral helper methods in the code, which is considered good coding practice (see Section IV-D). This suggests that when interpreting internal metrics values, some care must be taken and the quantitative analysis must be combined with a qualitative discussion. Also note that some aspects of maintainability are not reflected by the metrics used in this paper. An example is variable and method names, which are essential for the readability of the code [11] [16].

We see a need for more appropriate ways to measure the cohesion and indeed, considerable research has been carried out since the LCOM metrics was presented by Chidamber and Kemerer [22]. Notably, cohesion metrics are still being
developed [33]. Comparing different cohesion metrics by applying them on a scientific computing application could be a direction of further research.

Our refactoring has reduced the performance of the code considerably, although we see a risk that this performance reduction would appear as an effect of future extensions of the code base if the refactoring was not done on this stage. We attribute the performance loss to the extensive number of calls to dynamically polymorphic methods in the most frequently executed part of the code. We have also performed experiments that suggest that it is not the mechanics of the dynamic binding per se that causes the slowdown, but mainly the fact that the dynamic binding prevents inlining at compile time. By using static polymorphism for some classes, we were able to decrease the performance loss to a more reasonable level. However, even after we implemented static polymorphism, the refactored code was in most cases slower than the original one. An important reason may be the large number of memory indirections brought by the encapsulation. In a future study, we will evaluate if software prefetchers can limit this effect.

We note that in our code, static polymorphism comes with a considerable maintainability penalty compared to dynamic polymorphism. However, the code employing static polymorphism is more maintainable than the original, procedural one. We conclude that for performance-critical kernels, static polymorphism can be a reasonable compromise, but we would not use it for anything but very frequently called methods.

Finally, we want to raise the question whether there are internal metrics that can also give some guidance when it comes to performance. If so, internal metrics could give designers indications about both aspects covered in this paper: maintainability and performance. As the main culprit for performance loss in our application is dynamic binding, the amount of dynamic polymorphism may indicate performance bottlenecks. The metrics DIT and NOC are connected to polymorphism but a complementary addition of metrics would be needed in order to reflect the amount of dynamic binding.

ACKNOWLEDGMENT

We would like to thank Michael Thuné, Magnus Grandin and Kurt Otto for valuable comments regarding this study. The performance measurements were performed on resources provided by SNIC-UPPMAX.

REFERENCES

[29] [Online]. Available: http://www.uppmax.uu.se/the-tintin-cluster