Kent Academic Repository

Full text document (pdf)

Citation for published version

De Wael, Mattias and Marr, Stefan and De Meuter, Wolfgang (2014) Data Interface + Algorithms = Efficient Programs: Separating Logic from Representation to Improve Performance. In: Proceedings of the 9th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems.

DOI

https://doi.org/10.1145/2633301.2633303

Link to record in KAR

http://kar.kent.ac.uk/63830/

Document Version

Author's Accepted Manuscript

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

Data Interface + Algorithms = Efficient Programs

Separating Logic from Representation to Improve Performance

Mattias De Wael
Software Languages Lab
Vrije Universisteit Brussel
madewael@vub.ac.be

Stefan Marr
RMoD, INRIA Lille Nord Europe
Lille, France
stefan.marr@inria.fr

Wolfgang De Meuter
Software Languages Lab
Vrije Universisteit Brussel
wdmeuter@vub.ac.be

Abstract

Finding the right algorithm–data structure combination is easy, but finding the right data structure for a set of algorithms is much less trivial. Moreover, using the same data representation throughout the whole program might be sub-optimal. Depending on several factors, often only known at runtime, some programs benefit from changing the data representation during execution. In this position paper we introduce the idea of Just-In-Time data structures, a combination of a data interface and a set of concrete data representations with different performance characteristics. These Just-In-Time data structures can dynamically swap their internal data representation when the cost of swapping is payed back many times in the remainder of the computation. To make Just-In-Time data structures work, research is needed at three fronts: 1. We need to better understand the synergy between different data representations and algorithms; 2. We need a structured approach to handle the transitions between data representations; 3. We need descriptive programming constructs to express which representation fits a program fragment best. Combined, this research will result in a structured programming approach where separating data interface from data representation, not only improves understandability and maintainability, but also improves performance through automated transitions of data representation.

Categories and Subject Descriptors D.2.8 [*SOFTWARE ENGINEERING*]: Metrics– Performance measures

Keywords algorithms, data structures, performance

1. Context

Finding the right combination of a data structure and an algorithm is easy: it either requires knowledge (i.e., courses and books on Data Structures and Algorithms), or it requires experience. But in most cases, an optimal solution requires performance engineering on the actual program, because programs seldomly rely on a single algorithm to do their job. Finding the right combination of a data structure and a sequence of algorithms (i.e., program) is not so trivial and might not even be possible without taking the dynamic characteristics such as the amount of data and properties like sortedness, into account. In these situations it is beneficial to consider multiple data representations for the same data interface and change representation between phases of the program, i.e., when the cost of changing from one data representation to another, is smaller than the cost introduced by using the "wrong" (i.e., sub-optimal) data structure . In our research we target this kind of *phased programs* that can benefit from a changing data representation.

Consider an email client application as an example: most of the time you want your emails sorted by date, but at the same time you also like to browse through a history of emails from a single correspondent, while new emails keep on arriving in your mailbox. A trained computer scientist or programmer can find the best algorithm for sorting a mostly sorted sequence (i.e., sort by date), find the best data structure for iterating over a sub-set of elements (i.e., iterate by author), and find the best data structure for cheap insertion at-the-front (i.e., new emails arrive by definition in chronological order). The question which data representation fits all needs will likely be left unanswered. Moreover, the programmer responsible for combining the different functionalities probably does not care *how* the data interface is implemented. Does he want to program with a *linked-list-of-emails*, an *array-of-emails*, or would he prefer to program against the interface of a *mailbox*?

It is well accepted that programming with abstract data types (interfaces to data) instead of with concrete implementations (interfaces to data representation) is better for the sake of maintainability, understandability, and modularity. We assume that separating the choice of data representation from algorithm design and other program logic can also have a significant beneficial impact on the performance of programs by changing the representation at runtime. In recent research (see Section 2) this assumption has been successfully verified in specific contexts, e.g., higher-level collections. We want to develop a structured programming model that generalises this technique and brings it to the level of the application programmer. In this model separation of interface from implementations is encouraged, such that changing data representation can be automated, with improved performance as a result.

2. Related Work

We are not the first to think about finding the right algorithm–data structure combination for the sake of performance. It is a known and studied problem, but hitherto, we are lacking a general solution. In this section we give an overview of what research has been done in the past.

Foremost, there has been done a large body of groundwork on the performance of algorithms in combination with data structures in theoretical computer science. Measuring and/or estimating the time complexity of algorithms is part of every Bachelor level course on this subject and is part of every book on this subject, e.g., Cor-

[Copyright notice will appear here once 'preprint' option is removed.]

men et al. [3] who popularised the master theorem for estimating asymptotic bounds on the time complexity of recursive programs.

A second body of work (i.e., in the area of compiler and programming language research) focusses on a specific data structure: multi-dimensional arrays. In Fortran, for example, arrays are stored column-major order, in C arrays are stored row-major order, and Java's 2D arrays are implemented with an extra level of indirection, i.e., an array of references to arrays. Compilers for these languages take these specifications into account when generating code for loops over arrays, where they i.a. rely on the polytope model [8]. This mathematical model can be used to reason about nested iterations over multi-dimensional arrays and can help to improve performance by enumerating "allowed" transformation of the iteration code. Besides the idea of improving general computations (e.g., bounded loops in Fortran), other approaches focus on highly specific algorithms. The Pochoir stencil compiler, for example, focusses on stencil computations [12].

Yet another body of work focusses on higher-level collections as are offered by e.g., STL or java.util.Collection. These approaches improve the performance of a program by selecting or proposing the "right" data collection implementation. Both online and offline approaches exist. In an offline approach the best collection implementation is selected during development time. The online approaches come into play at runtime and make dynamic decisions on when to switch to a different data representation. Brainy [5] and Chameleon [11] are examples of the first approach, they collect data during test runs which is then used to make an informed decision on which collection implementation to use in production code. CoCo [13] and Storage Strategies [1] are examples of the online, dynamic approach.

Other dynamic approaches can be found the in realm of "Self Adjusting Data Structures". These data structures are developed with a certain use case in mind and readjust themselves according to the concrete usage at runtime. Examples are *Move-To-Front-Lists*, *AVL-trees*, object-maps in SELF [2], hybrid representation in Databases [6]. These data structures are optimised for a single use case and are therefore expected to outperform more general approaches, i.e., where the use cases are not necessarily know in advance.

Finally, there exists a lot of software engineering knowledge in the area of object-oriented design patterns to model changing state and changing behaviour [4]. As well as on data (representation) hiding through inheritance, interfacing and the like.

All techniques, approaches, and tools mentioned above try to improve either performance or understandability of a program by separating data representation from data usage. The approaches differ in whether they change the computation to match the data representation or vice versa. They differ in the level of abstraction they target, i.e., the algorithmic level or the hardware level. They also differ in the level of generality they support, e.g., a lot of work focusses solely on collection types. Moreover, these techniques play their role at different times in the development process, ranging from analysis phase, trough development aid and automatic tuning at install time, to full dynamic performance optimisation.

We envision a technique that allows programmers to implement a data structure that can change its representation at runtime, but where the burden on the programmer of changing the representation during execution is eventually eliminated. Conversely, performance specific knowledge, for instance expressed in a DSL, is used to automatically change between representations.

3. Research Vision

Choosing the right data structure for a program (sequence of various algorithms) is not as trivial as choosing the right data structure for a single algorithm. Moreover, in some programs it is beneficial for performance to change the structure of the data between the different algorithms, i.e., when the transformation cost is much lower than the the performance improvement of using the other representation. The decision of when to change representation is crosscutting with the conceptual design of a program. We argue that the data representation should be able to change during execution in order to achieve optimal performance, while the control of these changes remains separated from the program's core logic. To this end, we envision a structured programming approach where separating data interface from data representation is not only beneficial for the understandability and maintainability of the software, but also improves performance. Separating interface and implementation is nothing new, the novelty in this approach is the *Just-In-Time data structure*, an object that can transition between semantically equivalent representations without imposing any technical burden onto the programmer. To this end, we want to automate the changes in representations based on domain-specific knowledge about the performance characteristics of the different implementations. The following paragraphs focus on the research activities we foresee in developing this structured programming approach.

Cost Model for Switching In a program where subsequent algorithms have affinity for different data representations, changing the representations in between algorithms can introduce performance benefits. This, however, is not necessarily true and can even prove to be counter productive. In order to make the envisioned approach viable, it is a necessity to better understand the characteristics of the data and its representation that play a role in the efficiency of a program. These characteristics are often only known at runtime, e.g., the density of a graph or the sparsity of a matrix, and depend on the hardware on which the program is executed.

In *matrix multiplication* $A \times B = C$, for instance, it is a well known technique to swap the rows and columns (transpose) of the second matrix B. Here, the transposition is an example of changing the data representation, i.e., from row-major-order to column-major-order. Even though the transposition comes at a cost (there exists a transposition algorithm in $O(n)$), the overall execution time is better if B has been transposed, and if the matrixes A, B, and C do not fit in (cache) memory. Thus, the decision of transposing B depends on the "size" of the matrixes in relation to the executing hardware.

To understand the synergy between algorithms and data representations we want to build a cost model that takes the properties of a data sets (e.g., size, density) into account. To this end we need to study a significant body of data structures and algorithms to understand which characteristics of the data structure and interface design play a role in performance. A significant part of the cost model will have to deal with *transition algorithms*, i.e., algorithms that transform data from one representation to the other. Ideally, the expertise of building the cost model for one data type can be consolidated into a structured approach to extend the cost model for more data types.

Just-In-Time Data Structures Based on the cost model, as discussed above, it becomes possible to make an informed decision on when it is beneficial to change the data representation. We call a data structure that is open for changing its representation at runtime a *Just-In-Time data structure*. The bridge design pattern [4], for instance, can be used to implement Just-In-Time data structures because they allow an object to change their representation and behaviour without changing their identity. A similar pattern will be the bases of the implementation of our approach and therefore we want to embed the highly specialised pattern *into* a language or framework that allows to define these Just-In-Time data structures. The bases of such a language is the classic *single interface multiple implementation* construct found in many object oriented lan-

guages (e.g., Java). Pseudo code for the "Matrix" example is shown in fig. 1. To make changes between the implementations possible *transition functions* are needed.

A transition function is a function that takes an instance A of one implementation as argument and returns an instance B of another implementation while the data represented by A and B are equivalent. In theory, n implementations requires $n \times (n-1)$ transition functions. In practice, however, we think that implementing $n \times (n-1)$ different functions will not be necessary. First, some transitions can be unnecessary in practice. Second, in the case of dual implementations (e.g., row major order and column major order) A and B, the transition function $A \rightarrow B$ can be commutative. Third, transition functions can be composed, i.e., if there exists a transition function f from implementation A to B and there exists a transition function g from implementation B to C then $g \circ f$ can transform an instance of implementation A into an an instance of B. Finally, we assume that for many interfaces there exists one transition function that can map from any implementation to any other, we call such a function a *canonical transition function*.

```
1  data-interface Matrix {
2    construct(rows, cols)
3    int  rows()
4    int  cols()
5    int  get(row, col)
6    void set(row, col, value)
7  }
8
9  data-representation RowMajor of Matrix {
10   cols, rows, data[]
11
12   construct(rows, cols){
13     this.rows = rows
14     this.cols = cols
15     data      = array-of-size( rows*cols )
16   }
17
18   get(row, col) := data[ row*cols + col ]
19
20 }
21
22 data-representation ColMajor of Matrix {
23   cols, rows, data[]
24
25   construct(rows, cols){
26     this.rows = rows
27     this.cols = cols
28     data      = array-of-size( rows*cols )
29   }
30
31   get(row, col) := data[ row + col*rows ]
32 }
```

Figure 1. Proving an interface and two implementations of Matrix.

A *canonical transition function* is a transition function that describes the transition from any implementation to any other using only the proposed interface. For the Matrix example such a function would set the value of a new Matrix on row r and column c to the value of the original Matrix row r and column c. This is shown in fig. 2. This function works regardless of the chosen data representation. The down side of a canonical transition functions is that, due to the generality, the performance will be suboptimal.

For the sake of performance, it is possible to define *specialised transition functions* that implement a more performant transition between two (or more) representations. In the Matrix example, changing from row-major-order to column-major-order is know as

```
1  Matrix transition(Matrix m) {
2    Matrix c = Matrix(m.rows(), m.cols() )
3    for (int r : m.rows() ) {
4      for (int c : m.cols() ) {
5        c.set(r, c, m.get(r, c) )
6      }
7    }
8  }
```

Figure 2. A canonical transition function for Matrix.

"transposition of a matrix" (shown in fig. 3 for square matrixes) for which more efficient algorithms exist than a naive transformation.

```
1  Matrix transpose(Matrix m) {
2    for (int r : m.rows() ) {
3      for (int c : m.cols() ) {
4        if (c > r) {
5          temp = m.get(r, c)
6          m.set(r, c m.get(c,r))
7          m.set(c, r, temp)
8        }
9      }
10   }
11 }
```

Figure 3. Transposition is a dedicated transition function for row-major-order to column-major-order (or vice versa).

From a set of implementations and a set transition functions it is then possible to construct a data structure that can swap between its different representations.

When a read intensive program phase ideally uses representation B, while both the preceding and successive phases have affinity for representation A, two representation swaps are needed ($A \rightarrow B \rightarrow A$). Introducing the overhead of swapping twice might be to much w.r.t. the performance benefits from using representation B. This is one of the cases where the best solution is to (temporarily) keep multiple copies with different representations in memory [6]. Therefore, when permitted by memory constraints, it should be possible to obtain a *working copy* of a Just-In-Time data structure. Ideally, in the presence of writes, working copies can be merged back into a single the Just-In-Time data structure. For multi-threaded applications concurrency has to be taken into account, e.g. many data structures have a threat-safe variant. However, research on concurrency and thread-safe data structures as such, fall outside the scope of this work.

Structural Coercion With a cost model for switching in the one hand and a language to implement Just-In-Time data structures in the other hand, it is a matter of combining these two efforts to develop efficient programs. In a first step, we think about augmenting sections of the program's logic with the knowledge of the cost model in a declarative way using *Structural Coercion Hints*. It is then the responsibility of the language or framework to coerce the representation of the data into another representation based on these hints. In fig. 4 the classic matrix multiplication algorithm is augmented with the knowledge that when the matrixes are big (e.g., much larger than the L1 cache), it is beneficial to execute the program with matrix A stored in row-major-order and matrix B in column-major-order. Lines 2-6 in fig. 4 show what the structural coercion hint could look like, with the actual *condition* on line 5, and the *ideal representation* expressed on line 6. This extra information (e.g., the magical constant 5) is expressed in a DSL by a performance expert.

Depending on the type of information, structural coercion hints can be compiled into statically enforced coercions or compiled

```
1  function matrix_multiply(A, B) {
2    let A-size = (A.rows() * A.cols())
3        B-size = (A.rows() * A.cols())
4        C-size = (A.rows() * B.cols())
5      A-size + B-size + C-size > 5 * L1-cache-size
6      => RowMajor(A) and ColMajor(B)
7
8    C = Matrix( A.rows(),  B.cols())
9    for (i : C.rows()){
10     for (i : C.cols()) {
11       C.set(i, j, 0)
12       for (k : A.cols()) {
13         let c = C.get(i, j)
14             a = A.get(i, k)
15             b = B.get(k, j)
16           C.set(i, j, c + (a * b) )
17       }
18     }
19   }
20 }
```

Figure 4. In some cases matrix multiplication can benefit from a specific data representation.

into dynamic checks that will do a potential coercion at runtime. The overall result is that the programmer is relieved from writing explicit conversions ("how") as statements in the code ("when"), but instead is allowed to express the actual knowledge ("what").

Automating Just-In-Time Data Structures Programming with Just-In-Time data structures allows programmers to concentrate on the program's core logic and on the program's performance separately. However, when a program is written to target multiple hardware architectures, a single set of structural coercion hints to coordinate the transitions between data representations will not suffice. Second, when programs grow and when the number of implemented representations grow, the number of possible combinations explodes. Then, even an expert programmer or performance engineer will have a hard time expressing an optimal set of structural coercion hints. When the solution space for Just-In-Time data structures becomes unmanageable for humans, it is possible to resort to automated techniques such as static analysis, machine learning, and auto-tuning. These techniques have proven to be successful in related work. E.g., Jung et al. [5] use machine learning to effectively select the best implementation of data collection. SPIRAL [10], a DSL for linear digital signal processing, generates auto-tuned code optimised for a given platform. Similar, but generalised, efforts can automate the dynamic changes in representation, and render manual insertion of structural coercion hints obsolete.

4. Conclusion

Writing programs is about modelling concepts of the real world and representing them as symbols in a computer [9]. Over time, the evolution of software engineering has proposed various levels and techniques of abstraction to cope with the discrepancy of both worlds (e.g., ADTs [7]). Object-oriented programming introduced the programming technique of polymorphism, where objects with the same interface can behave differently depending on the implementation. This technique is frequently used in object-oriented software to improve maintainability and understandability. We propose to use polymorphism for the sake of performance. For instance, when semantically identical implementations, that only differ in performance characteristics (i.e., both space and time), can change their representation dynamically when the cost of changing is payed back by the remainder of the computation.

We envision a structured programming approach where the programmer is (eventually) relieved from the burden of changing the data representation during the execution of a program. Our idea towards this goal is "Just-In-Time Data Structures", a data interface with multiple implementations that allows to change its representation at runtime. Together with "structural coercion hints" or automated techniques e.g., based on machine learning, it is then possible to dynamically swap the data representation to obtain better performance.

References

[1] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. *SIGPLAN Not.*, 48(10):167–182, Oct. 2013.

[2] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, Sept. 1989.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[5] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. *SIGPLAN Not.*, 46(6):86–97, June 2011.

[6] J. Krueger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, and F. Faerber. Optimizing write performance for read optimized databases. In *Proceedings of the 15th International Conference on Database Systems for Advanced Applications - Volume Part II*, DASFAA'10, pages 291–305, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, Mar. 1974.

[8] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

[9] G. H. Mealy. Another look at data. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 525–534, New York, NY, USA, 1967. ACM.

[10] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation*, 93(2):232–275, 2005.

[11] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. *SIGPLAN Not.*, 44(6):408–418, June 2009.

[12] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[13] G. H. Xu. Coco: Sound and adaptive replacement of java collections. In *ECOOP*, pages 1–26, 2013.