Dynamically Transforming Data Structures

Erik Österlund
Software Technology Group
Linnaeus University
Växjö, Sweden
erik.osterlund@lnu.se

Welf Löwe
Software Technology Group
Linnaeus University
Växjö, Sweden
erik.osterlund@lnu.se

***Abstract*—Fine-tuning which data structure implementation to use for a given problem is sometimes tedious work since the optimum solution depends on the context, i.e., on the operation sequences, actual parameters as well as on the hardware available at run time. Sometimes a data structure with higher asymptotic time complexity performs better in certain contexts because of lower constants. The optimal solution may not even be possible to determine at compile time.**

We introduce transformation data structures that dynamically change their internal representation variant based on a possibly changing context. The most suitable variant is selected at run time rather than at compile time.

We demonstrate the effect on performance with a transformation ArrayList data structure using an array variant and a linked hash bag variant as alternative internal representations. Using our transformation ArrayList, the standard DaCapo benchmark suite shows a performance gain of 5.19% in average.

I. Introduction

Picking the right data structure implementation for a given task can be tedious and time consuming. Programmers tend to optimize for the worst case. However, the data structure performing best in the worst case may, however, perform worse in the actual execution contexts because of higher constants. Also, programmers tend to use the data structure implementations that scales better for large problem sizes. Analyzing the break-even points where one implementation outperforms another is difficult and depends on the target hardware that is usually unknown at program design time. And even with such a break-even point at hand, it is difficult to program dynamic switching between implementations. Since the state of a data structure needs to be copied from one implementation to another, dynamic switching takes time itself, which makes an analysis of break-even points even harder. Hence, programmers would benefit from a library of data structures finding these break-even points and switching between the expectedly best implementation automatically.

The semantics of certain high level languages don't allow programmers to be precise in their choice of data structure [10]. JavaScript for instance has an associative array used both as an array with indices and associative collection with key/value pairs. These more abstract data structures implemented as part of programming languages also benefit from dynamic selection of the internal implementation depending on how it is being used.

Finally, in a multithreaded environment, the choice of data structure implementation may depend on contention from different threads. If contention is very high, it may be beneficial to use a lock-free or wait-free data structure implementation rather than an implementation using semaphores. Likewise, if contention is low or synchronization is not needed, another implementation could be more beneficial. Since contention is a property that can change throughout program executions, so is the optimal data structure implementation.

Context-aware composition [2], [6] approaches these optimization problems dynamically at run-time by profiling the variant's performance in different usage contexts, learning a dynamic dispatcher, and selecting the expectedly best implementation for an actual context at runtime. Context attributes include, e.g., the number of cores available, size and type of input, contention, memory usage etc.

One and the same abstract operation could be implemented with different algorithms depending on the current data structure implementation. Hence, the current data structure implementation is just another context attribute for selecting the appropriate algorithm implementing an operation. Alternatively, the data structure implementation can adapt to the operation to be executed. So in general, both the abstract operation to be invoked and the current data structure implementation are context attributes for the selection of the appropriate algorithm and appropriate data structure implementations. In short, the choice of an algorithm can be delayed to run-time, as well as the choice of the data structure implementation, while preserving the same operation and data-structure semantics to the outside world.

These capabilities of context-aware composition were demonstrated before, varying the data structure operations and their implementation algorithms, as well as data structure state representations, context attributes, and online/offline learning scenarios. This study adds three aspects:

1) We introduce *transformation data structures* that encapsulate variants of its operation implementations (algorithms) and state representations behind a well-defined interface. Based on context-aware composition they switch between different algorithms and representations without changing its functional behavior.
2) Transformation data structures consider operation *sequences* rather than individual operations for selecting the expectedly best-fit data representation variants. These sequences are abstracted with states of finite state machines making the maintenance of the actual usage

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA

context affordable compared to the gain of selecting the appropriate variant for each such context.

3) We implement a transformation data structure derived from a standard API data structure and evaluate its performance using a standard performance benchmark suite. This goes beyond previous evaluations in experiments designed for showing the (potential) benefits of context-aware composition. Here, we do neither control the application usage context nor the importance of the data structure for the overall performance of the benchmark applications. More specifically, an `ArrayList` for Java that transforms between an array-based variant and a linked hash bag variant is used as an example, showing more than 5% (on average) performance gain in an evaluation against the DaCapo benchmark.

The remainder of this paper is organized as follows. Section II explains the design ideas behind efficient transformation data structures and discusses implementation details necessary for high performance. Section III discusses a case study using these ideas. Section IV shows how our implementation performs in practice in a standard benchmark suite. Finally, section V discusses related work and section VI concludes the paper and motivates directions of future work.

II. Transformation of Data Structure Design and Implementation

A. General Design

Let a *data structure* consist of an *abstract data representation* R and a set of *abstract operations* o operating on this data. The abstract data representation captures the state of the data structure. Therefore, R defines data access operations r/w for state construction and for reading and updating the state.

The abstract data representation R allows for different *data representation variants*. Data representation variants are *specializations* of the abstract data representation, i.e., access operations r/w may have stronger pre-conditions in the implementation or not even implement all access operations.

The abstract operations o of a data structure may come in variants too. We refer to these variants to as *algorithm variants*. One and the same abstract operation o can be implemented with different algorithms. Again, the algorithm can be a specialization of the operation. For correctness of the data structure, we must require though that, for each abstract operation, its preconditions imply the disjunction of all implementing algorithms' preconditions. Then, for each legal call to an abstract operation, there exists an algorithm variant that handles the call. This can usually be achieved with a, possibly inefficient, default algorithm for each operation.

The benefit of special data representation variants and algorithm variants is that they can be faster in special contexts (under special pre-conditions) than a general implementation. In order to exploit this potential benefit in practice, we must solve three problems: first, we need a design for changing the data representation variants and algorithm variants, respectively, at runtime. Second, we must be able to find the expectedly best variants for each usage context. Third, we need to *efficiently* switch between the variants and *efficiently* transfer state between them.

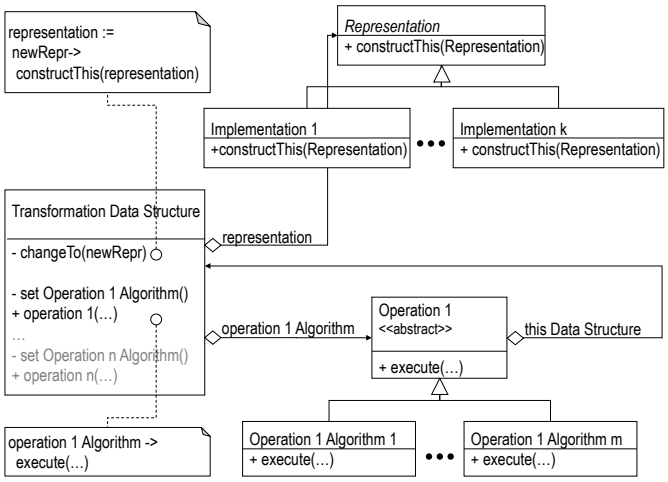

Fig. 1. Transformation Data Structure. UML diagram of the design pattern. The actual implementation design could be different, e.g., for improving access performance, an algorithm could know a data representation variant directly.

The first problem is solved by a general design pattern as proposed in [8] and depicted in Figure 1. It uses a combination of the well-known *bridge* and *strategy* design patterns [5]. The data structure holds a reference to a current representation that is an instance of an abstract representation class. Different data representation variants inherit from the abstract one. Besides the data access operations (readers, writers, iterators, constructors, etc.) a data representation variant implements a `constructThis` operation which constructs an instance of this variant from the general one and, hence, from any other variant. Therefore, some core data access functions (e.g., a read-iterator) must be available in any data representation variant. The data representation variant can be changed by a `changeTo` operation which creates a new empty instance of the desired data representation variant and reconstructs the state using its `constructThis` operation. Also the data structure holds references, one for each operation, to the current algorithm variants implementing an operation. Algorithm variants are implemented as algorithm classes specializing an abstract operation class. This abstract class of an operation provides an abstract `execute` operation implemented by all its algorithm variant classes. Calls to an operation are delegated to the `execute` operation of the corresponding current algorithm class. Algorithms have access and can modify state via a back reference to the data structure, which, in turn, knows the current data representation variant. One can change the algorithms by (re-) assigning an algorithm class instance as the current algorithm object for an operation. In what follows, data structures following this design pattern are referred to as transformation data structures. In short:

Definition 1: A *Transformation data structure* is a data structure $DS = (R, V, F, O, A, v_i)$. R denotes the abstract

data representation, V its data representation variants with $v_i \in V$ an initial variant, and F a set of transformation functions f_t changing state from a source variant v_s to a new target variant v_t with $f_t : v_s \rightarrow v_t$. O denotes a set of abstract operations, where each operation $o \in O$ is realized by a set $A_o \subset A$ of algorithm variants with a default implementation $d_o \in A_o$.
Note, that the transformation functions are specific to the *target* variants v_t. In order to read the current state, a general public reader or read iterator of A can be used.

B. Context-aware Composition

The second problem and, in principle, even the third problem are solved by *context-aware composition* as proposed in [2], [6]. It dynamically composes and selects between different algorithm and data representation variants based on the actual usage context.

Finding the expectedly best (algorithm and data representation) variant, requires the designer to define an optimization goal, e.g., minimize run time execution time, and formal context attributes, e.g., problem size or number of processors and amount of memory available, which are expected to have an impact on the goal. Formal context attributes, i.e., the actual context, must be evaluable before each call to an operation o; the property to optimize must be evaluable after each call. Context-aware composition finds the expectedly best variant automatically using profiling, analysis, and machine learning. Using profiling or analysis or a combination thereof, the fitness of variants can be assessed for different *selected* actual contexts. Using machine learning, dispatchers can be trained selecting the expectedly best fitting variants of operations and representation implementation for *each* actual context.

Note that best-fit-learning can be performed offline at design or deployment time, or even online using feedback from online monitoring during program execution [1].

C. Efficient Dynamic Transformations

The third problem, switching between variants and, especially, transferring the state between the data representation variants, uses these dispatchers and the transfer functions F. Each dispatcher works like a generalized dynamic dispatch in object-oriented programming: for an operation called in an actual context, it decides the best-fit algorithm and data representation variant. Switching to an algorithm is trivially implemented with an indirect method invocation. Switching state to a data representation variant v_t requires the invocation of the corresponding transformation function f_t before.

Each algorithm and data representation variant comes with certain trade-offs. Transforming from any data representation variant v_s to another v_t takes time, depending on the complexity of the general read iterator and the specific constructor of v_t. If we for instance convert from an array implementation with read complexity $O(n)$ to an ordered, balanced tree implementation with constructor complexity $O(nlog(n))$, the transformation would cost $O(nlog(n))$. This transformation cost is traded off against the complexity of operations on trees, e.g., $O(log(n))$ for search instead of $O(n)$.

In general, transformation between data representation variants has a complexity of $\Omega(n)$; many data structures, such as trees, actually require $O(nlog(n))$. This is quite expensive compared to the costs of an individual operation; let alone the benefits of one of its algorithm variants compared to another. If we could amortize the transformation costs with a sequence of such operations, i.e., with a sequence of algorithms performing better on the target data representation variant, then a transformation would more likely pay off. An *operation sequence* is here a sequence of method invocations at run time to one and the same data structure *instance*.

Context-aware composition cannot be used directly to trade an operation sequence off against a single transformation: before the first operation (first method invocation) it cannot be guaranteed, in general, that a certain operation sequence will follow. For instance, a loop may terminate after the first iteration; a statement sequence with call operations may at runtime target different data structure instances.

The basic idea to overcome this problem is to use the observed sequence of operations called in the past and the operation that is about to be called as a context attribute to decide whether to transform to another representation implementation or to stay with the current one. We use a rather simple heuristic: the observed operation sequence is likely to remain stable in the future. This heuristic is somewhat justified by the success of just-in-time compilers and dynamic load balancers and leads to quite a performance gain in our experiments, cf. Section IV.

In short, data representation variant transformation would ideally be defined on a context attribute abstracting future operation sequences. While static analysis could provide such an (often imprecise) approximation, we prefer to use the observed operation sequence as a context attribute for controlling the transformation decision. This decision can still be influenced by additional context attributes such as, the size of the data structure, the number of processors available, lock contention, memory usage etc.

The time for evaluating the actual context attributes is an overhead of context-aware composition. Hence, we should bias between precision of a context attribute, here encoding the past operation sequence, and the attribute's evaluation time and storage effort. A state machine is a powerful yet simple way of encoding operation sequences observed during execution. Each transformation data structure implements such a state machine; each of its instance objects holds its current state.

In the following subsections we will show how to derive such a state machine. We start by defining an initial state machine, later improved by edge expansion, leading to a final state machine. As a running example accompanying the theory we will use an abstract list data representation with two realized data representation variants; a linked hash bag and an array variant. The list supports the operations `add` (faster on array variant by a constant) and `contains` (lower complexity class on linked hash bag variant).

1) Initial State Machine: To define a state machine for the transformation data structure, we assume for a first try that the operation sequence is the only context attribute regarded. We create an initial state machine with n states, one for each data representation variant. Reaching the state corresponding to a variant will cause the transformation to or will let the data structure stay in this variant. Hence, an operation o ought to trigger a transition from state s_s to s_t, and cause a transformation from v_s to v_t if the best algorithm variant a_s realizing o on v_s is more expensive than the best algorithm a_t realizing o on v_t.

The necessary information is derived in the learning phase of context-aware composition: for each operation, each algorithm variant realizing this operation and each data representation variant admissible for this algorithm, we assess the performance by profiling sample executions. If other context attributes should be regarded, e.g., size or number of processors available, this should be reflected in sample executions covering different actual contexts, i.e., different problem sizes. Using the sample data set of operations, actual contexts and best performing algorithm and data representation variants, machine learning extrapolates a dispatcher $disp : O \times C \rightarrow A \times V$ mapping each call to operation $o \in O$ in an actual context $c \in C$ to the best fit algorithm variant $a \in A$ and the data representation variant $v \in V$.

This dispatcher constructs an initial state machine: as mentioned, each data representation variant v corresponds to a state s. For each operation o called in actual context c, let $a(o, c)$ be the best fit algorithm variant, $v(o, c)$ be its best fit data representation variant *disregarding transformation costs to switch to these* and $s(o, c)$ the corresponding state. There is a transition to $s(o, c)$ from any other state labeled with $(o, c/a(o, c))$. It is interpreted as follows: on a call of an operation o in the actual call context c, use algorithm $a(o, c)$ on the data representation $v(o, c)$. If the current variant $v_s \neq v(o, c)$, transform to $v(o, c)$ before.

In the list example, this results in a greedy state machine with a transformation to whichever data representation variant has the fastest algorithm for the next operation, i.e. if the current data representation variant is array-based and the next operation is a `contains` operation, this would trigger a transformation to the hash bag variant. Likewise, if the current data representation variant is based on a linked hash bag and the next operation is `add`, it will eagerly transform to an array based variant instead. Note that eager transformation happens even though they may not necessarily be beneficial for a single operation. The initial state machine does not consider transformation costs vs maybe only constant gain in execution time for a single operation.

Note, that there are only finitely many different operations $o \in O$, algorithm variants $a \in A$ and representation variants $v \in V$. From the latter, it follows directly that we only have finitely many states. Also, we only need to distinguish finitely many actual contexts. More formally, we can partition the infinitely many actual contexts $c \in C$ into finitely many partitions $\hat{C}_i \subseteq C$, one for each different triple (o, a, v), and we consider state transitions

$$(s_s, (o, \hat{C}/a(o, c)), s(o, c)) \tag{1}$$

with s_s any source state, $\hat{C} \subseteq C$ a context partition and $c \in \hat{C}$. Hence, our state machine has finitely many transitions.

In the discussion below, we deliberately skip the (now obvious) algorithm variant selection and focus on representation transformation. Also, we omit a discussion of other contexts but the call sequences for the sake of presentation clarity. Hence, the general state transitions form of Equation 1 simplifies to (s_s, o, s_t) with s_s and s_t source and target states, resp.

The initial state machine would eagerly transform to the variant $v(o)$ that is advantagous for the next operation o to be called. This behavior disregards the transition costs and is suboptimal. Instead, the eager data representation variant transformation to $v(o)$ ought to be delayed until we can (heuristically) assume that more calls to o will follow and amortized compensate for the transformation costs. Expanding the initial state machine in a controlled way finds us a better final state machine reflecting this desired behavior.

For expanding the initial state machine, we first expand individual edges to paths. Then we will put together these paths to the final state machine.

2) Edge Expansion: An edge (s_s, o, s_t) is replaced with two edges involving an intermediate or *biased* state: (s_s, o, s_t^{bias}) and (s_t^{bias}, o, s_t). Transition to the biased state s_t^{bias} does not trigger a transformation to v_t, only the second call to operation o in a row does.

The new state machine delays transformation and requires a sequence of two operations before it triggers a transformation to the target variant. Similarly, more biased states can be introduced replacing a single transition in in the initial state machine with a path over biased states requiring an operation call sequence to trigger transformation.

Expanding edges should stop if transformation is expected to pay off. Let $trans(v_s, v_t)$ be the transformation time from v_s to v_t and $exec(a_s)$ and $exec(a_t)$ be the execution times of the expectedly fastest algorithms a_s and a_t implementing o on v_s and v_t, respectively. Expansion stops, if a sequence of $n/2$ operations o is required to trigger transformation from v_s to v_t and

$$trans(v_s, v_t) + n \times exec(a_t) < n \times exec(a_s), \tag{2}$$

again, based on the heuristic assumptions that past operation sequences observed are likely to continue in the future.

This expansion of state transitions is done for any transition in the initial state machine until the inequation (2) holds for the transition and execution costs of the respective algorithm and representation variants. As a result, we get paths $s_s, s_t^{bias_1}, \ldots, s_t^{bias_{\lceil \frac{n}{2} \rceil - 1}}, s_t$ for edges (s_s, o, s_t) in the initial state machine.

3) Final State Machine: Putting these paths together to our final state machine is done in the three steps below:

(i) For any set of paths with the same source state s_s and the same target state s_t, we compute the product state machine

with all states of the product containing s_t being aggregated to one single accepting state. Assume operations o and o' with edges (s_s, o, s_t) and (s_s, o', s_t) in the initial state machine. The effect of computing the product of the paths expanding these edges is that an interleaving of operation sequences of o and o' leads eventually to the target s_t and triggers a transformation to the representation variant v_t.

(ii) All edges $(s_s, \_, s_t)$ in the initial state machine are removed and replaced by the product state machines computed for their expanded paths $s_s, \ldots, s_t$ in step (i).

(iii) There are operations $\bar{o}$ which are best performed on the source variant v_s or on another variants $v_{\bar{t}}$. With respect to these operations $\bar{o}$, all biased states towards s_t (the product state machine contains a state s_t^{bias} and $s_t \neq s_s, s_t \neq s_{\bar{t}}$) should have the same behavior as s_s. Let $(s_s, \bar{o}, s_{\bar{t}}^{bias})$ be the behavior of s_s in the state machine after step (ii). The desired behavior of the final state machine is enforced by adding edges from $(s_t^{bias}, \bar{o}, s_{\bar{t}})$ for all biased states towards s_t. The effect is that, after observing a sequence of operations o biasing towards s_t, an operation $\bar{o}$ stops this biasing towards s_t and starts biasing towards $s_{\bar{t}}$ instead.

In our running example, this implies that transforming from the linked hash bag to the array variant on an `add` operation would go through a number of biased states until inequation (2) holds, i.e., until it is likely enough that we see a sequence of `add` operations faster implemented on arrays than on linked hash bags. Conversely, transforming from the array variant to the linked hash bag variant happens immediately when we see a `contains` operation. It pays off for the first `contains` already as it is a complexity class faster on the target variant.

The construction of the final state machine is obviously costly as it involves the construction of several product state machines in step (i) each taking $O(n^m)$ with n the maximum length of the paths expanded from an edge in the initial state machine and m the number of these paths. In practice, it works in cases with only few states and edges in the initial state machine. The former is a realistic assumption as there are usually only few representation variants available. Aggregating individual operations to operation classes can enforce the latter. Heuristically, we suggest, just two such classes for any pair of source and target variants: one class O for all operations with a lower complexity class on the target, one class O' for all operations that are only cheaper by constants on the target. The termination of state machine expansion needs to be adapted to the latter case: expansion terminates if inequation (2) holds for all $o \in O'$. Step (i) requires then only the construction of product state machines from (at most) two paths each.

D. Replacing Existing Variants with a Transformation Data Structure

When integrating a set of *existing* variants implementing the same functional behavior to a transformation data structure and replacing all these variants with this transformation data structure in an *existing* environment, certain potential pitfalls ought to be regarded.

One issue is concurrency in a multi-threaded execution environment. All operations may change the state of the state machine of the transformation data structure and potentially even trigger a transformation between variants. This means that even a read operation could cause a mutation while the corresponding read in any of the existing variants does not. The usage context of a variant might assume that reading operations do not need to be synchronized because there is no mutation. In order for transformation data structures to work correctly in existing usage contexts this assumption ought to be regarded.

The solution to this problem is to allow data races in the state machine and hold more than one copy of the internal state in different variants. In the worst scenario, the transformation data structure would make a transformation decision to a suboptimal variant. This is fine as long as it does not happen too often and the amortized time over all operations is shorter. Eventually, a (synchronized) write operation guarantees a consistent state again.

If the internal data representation variant was exposed outside its data structure there could be abandoned representations referenced in the heap. Normally this can never happen as the internal representation is not exposed but all accesses go through the data structure object. Iterators, however, are an exception here which needs to be explained more in detail. Iterators have a reference to the data structure's internal representation. A transformation triggered during (read) iteration interleaved with any write could lead to inconsistencies. Data structures usually (for good reasons) forbid arbitrary writes during iteration. However, the iterator itself may allow writes to the underlying internal representation. Care must then be taken to handle this case by, e.g., (1) distinguishing an external iterator object referencing an internal one, and abandoning the internal iterator when its internal data representation is abandoned (similar to the data structure facade object itself) and then constructing a new internal iterator after a transformation, (2) not allowing the iterator access to the internal representation but only go through the facade object's, or (3) deferring transformation until after iteration.

Another issue arises from using a variant as a monitor object to synchronize with. The monitor object used has to be the transformation data structure object encapsulating the internal variants, as this is what is exposed to the outside world. This potentially requires code transformations in both the usage environment and the variants where references to `this` object need to be redirected to the encapsulating transformation data structure object.

Finally, it may happen that a variant has side effects either directly or by invoking callback methods in the environment outside the transformation data structure. It must be assured that all side effects and side effecting callbacks and exactly those happen in the right order in all variants even if not needed in all of them. This can be tricky since, e.g.

- a side effecting constructor might never be executed since the variant is not used or, due to transformation, it might

TABLE I
TIME COMPLEXITY OF ARRAYLIST OPERATIONS IN DIFFERENT VARIANTS.

Operation	Array	LinkedHashBag
Array operations on par with LinkedHashBag		
add(E)	$O(1)$	$O(1)$ *
add(int,E)	$O(n)$	$O(n)$ *
addAll(Collection<E>)	$O(n)$	$O(n)$ *
clear	$O(1)$	$O(1)$
removeAt	$O(n)$	$O(n)$
indexOf(E)	$O(n)$	$O(n)$
isEmpty	$O(1)$	$O(1)$
lastIndexOf(E)	$O(n)$	$O(n)$
remove(int)	$O(n)$	$O(n)$
removeRange(int,int)	$O(n)$	$O(n)$
size	$O(1)$	$O(1)$
toArray	$O(n)$	$O(n)$
Array operations slower than LinkedHashBag		
remove(E)	$O(n)$	$O(1)$ **
contains(E)	$O(n)$	$O(1)$ **
ensureCapacity	$O(n)$	$O(1)$
trimToSize	$O(n)$	$O(1)$
Array operations faster than LinkedHashBag		
get(int)	$O(1)$	$O(n)$ ***
set(int,E)	$O(1)$	$O(n)$ ***

* v_a gain proportional to $O(1)$
** v_h gain proportional to $O(n)$
*** v_a gain proportional to $O(n)$

possibly be executed several times, or it is executed in a different order with other side effecting methods,

- a side effecting callback might not occur in a variant but it does in another, or two side effecting callbacks might occur in different orders in different variants.

Especially, in the presence of callbacks, e.g., to `equals` or `hashCode` methods in Java, this cannot be guaranteed without the knowledge of the usage environment. Then, side effect free callbacks ought to be explicitly required in preconditions; they can be enforced by conservative static analysis.

III. CASE STUDY: ARRAY LIST

As a case study, we developed an `ArrayList` transformation data structure implemented in Java. It dynamically switches between an array-based data representation variant and a data representation variant using a linked hash bag.

The linked hash bag variant `LinkedHashBag` is constructed as a circular double ended linked list, where each node in the linked list is additionally added into a bucket in a hash table. This means that the order and semantics of the list are preserved, while the nodes in the linked list are hashed into buckets, making lookup time fast. The array of buckets is doubled when a certain load limit is reached.

A. Efficient Dynamic Transformations

The array variant uses native Java arrays in the underlying list implementation. The array gets an initial size. When more space is needed, it allocates a new array and copies the old elements over. The growth of size is exponential, keeping the amortized time complexity constant. The array is not circular, so manipulating the beginning of the array is expensive.

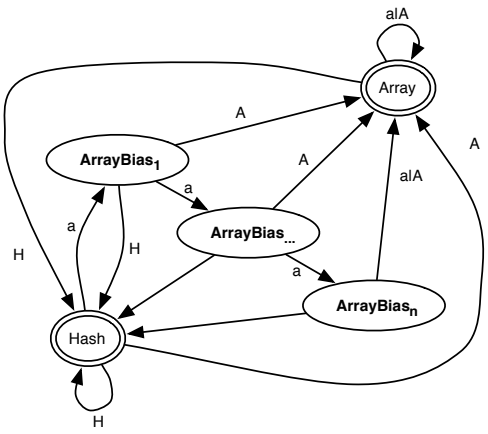

Fig. 2. Transformation `ArrayList` state machine. A: operations performing $O(1)$ on arrays instead of $O(n)$ on hash bags, a: operations performing better on arrays than on hash bags by a constant margin, H: operations performing $O(1)$ on hash bags instead of $O(n)$ on arrays.

Table III-A displays the different time complexities for the two `ArrayList` variants. The rightmost column classifies the gain of the variants.

As discussed, there is a tradeoff that the more accurately the best variant is predicted, the more time is spent on predicting, and not on doing actual work. It is therefore beneficial to construct a smaller state machine sacrificing precision for lower dispatch time constants. Especially in the array variant, the operation costs are so low that even the smallest prediction logic has a few percent of overhead.

For keeping the state machines small, transitions are made directly when the best-fit operation on one representation is a complexity class better. There is no operation where the linked hash bag has lower constants than the array. Hence, the array variant v_a does not have any biased states towards the linked hash bag variant v_h. In the other direction, there are a few operations with better constants on v_a compared to v_h. Consequently, the linked hash bag variant v_h has biased states towards the array variant v_a. The complete final state machine for the transitions between the array variant v_a to the hash variant v_h can be found in Figure 2.

The interpretation of this state machine is that we have some operations that really benefit from the array variant, e.g., where we would have an $O(1)$ complexity on one variant instead of an $O(n)$ complexity on the other. In these cases, we greedily transform immediately. In other cases, we have the same time complexity in both variants, but the array variant has lower constants. Then we have intermediate states biasing towards but not eagerly transforming to the array variant. When we are convinced enough that there is a sequence of operations to follow that would benefit from an array representation variant, the transformation occurs.

Figure 2 shows the full state machine determining the transitions between different variants of `ArrayList`.

Despite the theoretical complexity of the final state machine, for a transformation data structure transforming between an

array variant and a linked hash bag variant, the final state machine remains quite simple. There are several reasons for this. The foremost and most obvious is that there are only two states in the initial state machine corresponding to the two variants. Second, we applied the classification of operation heuristic discussed at the end of Section II. This reduced the number of edges in the initial state machine to a maximum of four (two for each state corresponding to a transformation target). Finally, there was no single operation where, according to empirical data during profiling and learning, the best-fit algorithm on the linked hash bag variant would have a better performance constant than the best fit on the array variant. The transformations are only advised when operations admit an algorithm on the linked hash bag variant with constant complexity and an algorithm on the array variant with linear complexity. This reduced the number of edges in the initial state machine to three (one towards the linked hash bag and two towards the array).

Comparing a linked hash bag variant to an array variant has operations where linear time complexity reduces to a constant time complexity, which triggers immediate transformation. Additionally, we also observe more subtle cases where time complexity is lower by constants in the array variant. In these cases biasing is introduced in the final state machine. In practice, this is implemented as a counter being increased.

B. Replacing Existing Variants with a Transformation Data Structure

The observable behavior of the linked hash bag variant is expected to be exactly the same as the array variant. The array has a notion of order, which has to be maintained in the linked hash bag variant. In the latter, ordering is ensured using a doubly linked list between the nodes. The linked hash bag variant uses a hash function h for faster access, the array variant does not. To ensure the same observable behavior, the objects added to the collection have to have the same equality semantics ($\equiv$). This can only be guaranteed if $o_1 \equiv o_2 \Rightarrow h(o_1) = h(o_2)$. If this does not hold, which is a programming mistake violating the pre-conditions of the h, it could be that an operation using the hashing such as `contains` will find no matching object in the bucket—even though the object is actually in the list—because it was hashed to the wrong bucket. In fact, it does not matter if objects with inconsistent equals and hash semantics are added to the transformation data structure unless operations are called that use hashing in the linked hash bag variant.

When using the linked hash bag, the usage environment has an impact on observable behavior due to callbacks to the `equals` and `hashCode` methods. This can be found out conservatively by static analysis checking if the two methods are not overridden in user-defined classes of objects added to the `ArrayList`. This is beyond the scope of this paper. We validated that our assumption holds in all of the benchmarks discussed in the next section. There we check and exclude these inconsistencies during runtime using Java reflection before the actual performance measurements. No inconsistencies affecting the benchmark were found.

IV. Evaluation

This section will first show our own micro benchmarks how well different standard data structures from the Java utility API as well as our transforming data structure perform for some operations. The second part of this section shows the result of using our transforming data structure in the standard DaCapo benchmark suite.

Results were evaluated by building two versions of OpenJDK - one with the original `ArrayList` and one with our own transforming `ArrayList` implementation, i.e. all `ArrayList` were exchanged uniformly.

A. Data Structure Operations

To construct the state machine triggering the transformation from one variant to another, a number of small micro benchmarks were run to determine how fast certain existing data structures perform operations in certain contexts. In profiling we assessed the performance of all operations. In this section, however, only the most relevant findings are discussed.

We assess the performance of up to five container data structures: `ArrayList`, `LinkedList`, `HashSet`, and `TreeSet` are the well-known implementations from the Java utility API. We compare (a subset of) them with our own transformation `ArrayList` data structure as introduced in the previous section. The micro benchmarks are performed using Integer objects stored in the different containers.

Each micro benchmark first initializes data structures of different sizes ranging from 100 to 100.000. Then we run the actual operations 10 times without measurement for warming up the caches, followed by 100 times with measurements. The whole procedure is performed in three iterations in an external loop to increase the accuracy of the numbers. We finally calculate the average of all measured times in each context (problem size, data structure). In the performance comparison figures below, the x-axis displays problem size, the y-axis running time. As we are only interested in relative performance, we omit absolute problem sizes and time values.

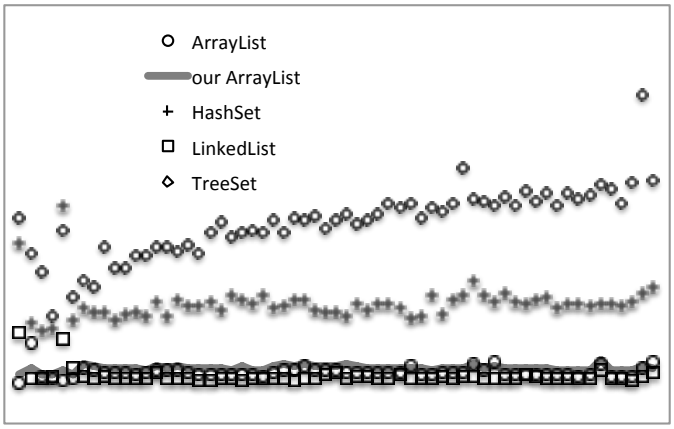

Fig. 3. Performance for the `add` Operation using Different Variants

Figure IV-A shows that `ArrayList`, `LinkedList`, and our transforming `ArrayList` all seem to perform roughly equally when adding elements while `HashSet` has a higher constant execution time and the execution time of `TreeSet` grows logarithmically in the problem size. Here, the transforming data structure managed to keep the performance on par with the fastest data structure.

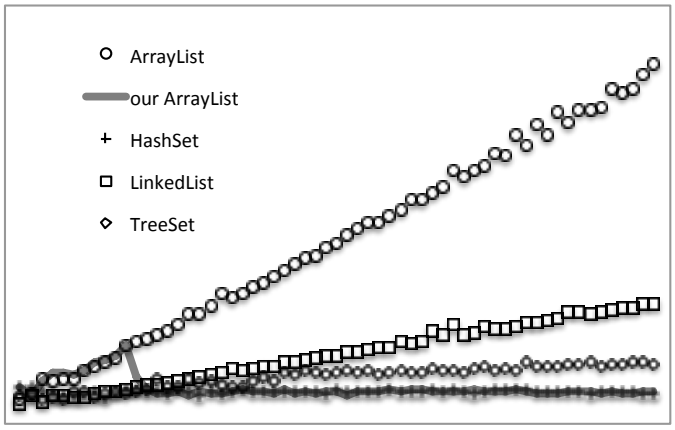

Fig. 4. Performance for the `contains` operation using different variants

Figure IV-A shows that `contains` grows linearly in the size of the data structure for the `ArrayList` and `HashSet` variants. Our transforming `ArrayList` follows the line of the original `ArrayList` until the threshold to transform to a hash based solution is reached. It then performs as well as the `HashSet`, which outperforms the other variants in this micro benchmark. The `TreeSet` is relatively cheap with its logarithmic time complexity, but does not match the $O(1)$ complexity of the hash based variants.

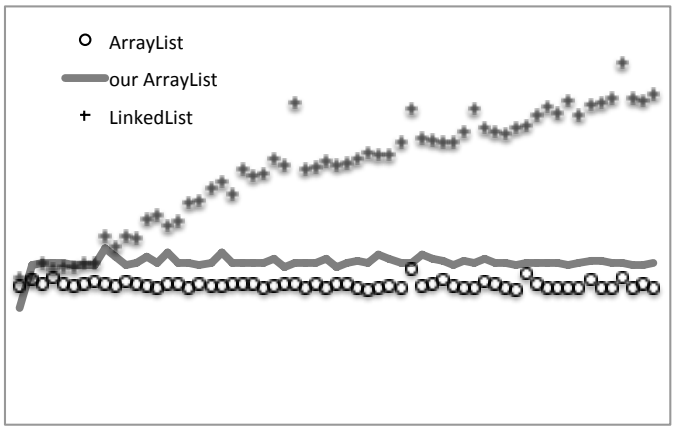

Fig. 5. Performance for the `get` operation using different variants

Figure IV-A shows how the different data structures perform when the `get` operation is called, fetching the element at a certain index. Not surprisingly, the `LinkedList` implementation suffers from the lack of indexing. The `ArrayList` is superior because of its array based implementation where elements at an index can be fetched in $O(1)$ time. Once again, our `ArrayList` comes close to the optimal, because it automatically picks an array based variant. The composition overhead makes it perform slightly worse than the original `ArrayList`.

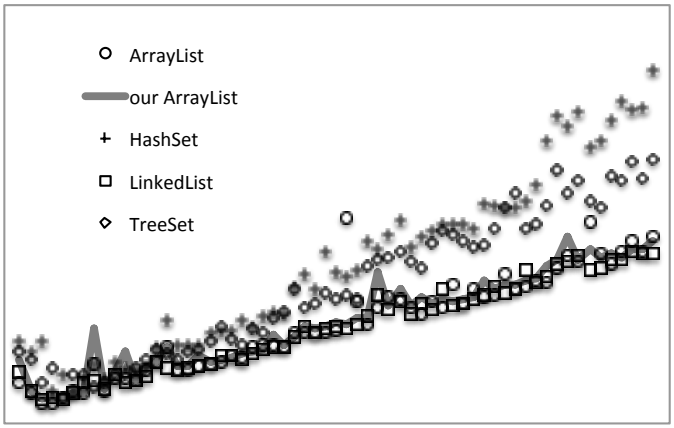

Fig. 6. Performance for iterating through the elements of different variants

Figure IV-A shows how the variants perform when iterating through all elements of data structures. The `HashSet` is slower as it has to iterate through buckets and elements and the `TreeSet` is also slightly slower. The `LinkedList` and `ArrayList` seem to perform equally as well and we see that our transforming `ArrayList` performs on par with them.

The micro benchmarks are performed using Integer objects stored in the different containers. Of course, container data structure execution times depend on the type of objects captured in the data structures. For example, sometimes hashing is very expensive and comparing elements is rather cheap. In such cases, it is expected that the `TreeSet` data structure outperforms the others. An even more clever implementation of a transforming data structure may identify the class of objects captured as an additional context attribute and transform to a tree based data representation variant when applicable.

Conclusively, if operations are used in a predictable way, the transforming `ArrayList` gets close the optimal execution times regardless of the operation, making it a good candidate for many applications. For operations with low constants, the overhead of the context aware composition logic becomes slightly more visible, and has to be traded off against the possible performance gain very carefully.

B. Benchmarks

For benchmarking in a realistic context where we do not know the operation sequences in advance, we use the standard DaCapo 9.12 benchmark suite [3].

We instrumented the benchmark in a pre-evaluation, looking for seemingly important data structures that could be optimized. We logged and analyzed method calls to data structures in the java.util package and found that among them the calls to `ArrayList` methods are the most frequent calls occurring in DaCapo. Therefore, we decided to take `ArrayList`s as a case study. No other assumptions from DaCapo influenced our

TABLE II
RESULT OF RUNNING THE DACAPO BENCHMARK SUITE 10 TIMES PER BENCHMARK, EXCLUDING 3 WARMING UP RUNS.

Benchmark	Old average	New average	Gain
avrora	6530	6570	-0.61%
h2	12094	10033	20.54%
jython	16291	16210	0.50%
lusearch	9561	9410	1.60%
pmd	11234	11724	-4.17%
sunflow	7399	6442	14.85%
tomcat	11667	11334	2.93%
xalan	10463	9648	8.44%
eclipse	86341	85274	1.24%
luindex	3467	3253	6.58%
tradebeans *	-	-	-%
		Average speedup: 5.19%	

* Didn't work at all on neither of the two openjdk versions.

implementation. Especially, profiling and final state machine construction was solely based on the micro benchmarks and not of observed operation call sequences in DaCapo.

In the actual evaluation, we ran all multithreaded benchmarks in DaCapo on a MacBook Pro 2.53 GHz Intel Core i5 with 4 GB 1067 MHz DDR3 memory running OS X 10.8.2. First we ran each benchmark 3 times to warm up caches; then we averaged over 10 actual measurements.

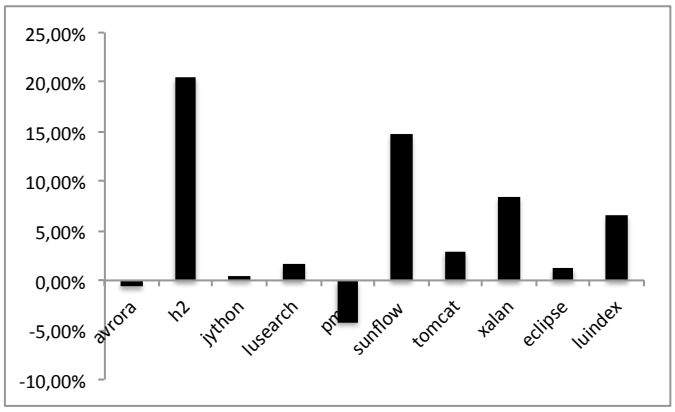

Fig. 7. Average speedup of DaCapo benchmarks using our transformation `ArrayList` instead of the default `java.util.ArrayList`.

Table IV-B shows the result of the benchmarks. The first column shows which specific DaCapo benchmark was run; the second column shows how long time (in ms) the benchmark executed in average over the 10 runs using the default `java.util.ArrayList`. The third column shows how long time the same benchmark required in average using our new transformation `ArrayList` instead. Finally, the last column shows how high the speedup is in percent. Figure IV-B shows what the speedup was for each benchmark.

As we can see, most benchmarks improve performance. However, some minor performance losses are worth noting.

Especially, in the pmd benchmark, the execution time was 4% worse with the transformation `ArrayList` compared to the default implementation. Because of the operation sequence used in this benchmark, the internal representation changes too much, oscillating between the two variants, never deciding which one is the better. Recognizing oscillation and avoiding transformations in these cases or maintaining both an array representation and a hash table at the same time could potentially avoid these situations.

V. RELATED WORK

Automatic component specialization has been a great concern in the composition community for many years. While general components are reusable in many contexts, the cost of generality is sometimes unacceptable. Hence, specializing the components to specific usage contexts is desirable.

For object-oriented languages, Schultz *et al.* [11] show different approaches using advices from the developer to automatically specialize applications. Löwe *et al.* [8] assume that the programmer will specialize algorithms and data representation variants dynamically. Their design of such a data structure uses bride and strategy design patterns for exchanging the representation implementation and the algorithms, repectively. However, both specialization operations are assumed to be public at the data structures' interface.

Svahnberg et al. present a taxonomy of techniques for variability realization and specialization [12]. Within this framework, our work in fact constitutes a variability realization technique for variant binding during run time.

Also, our technique may be considered as a generalization of the dispatch mechanism in object-oriented languages. Context Oriented Programming (COP) [13] offers generic, language level, mechanisms suitable for implementing context-aware optimizations at run time.

Autotuning in domain-specific library generators achieve adaptive optimizations. Profiling data gathered during off-line training processes is used to tune key parameters, such as loop blocking factors to adapt to, e.g., cache sizes. Examples include ATLAS [14] for linear algebra computations, and SPIRAL [9] and FFTW [4] for Fast Fourier Transforms (FFT) and signal processing computations. Li *et al.* [7] implement a library generator using dynamic tuning to adapt to the target machine. A number of context attributes such as the size and the distribution of the input and hardware environment properties such as cache size are input to a machine-learning algorithm; the resulting dispatcher is able to select the most appropriate algorithm for a context. In contrast to our approach, auto-tuning is domain specific. Data representations are not changed dynamically.

There are several approaches to automatic optimization of algorithm selection, resource allocation, or scheduling at runtime aiming at parallel target machines. Again, these are often for specific domains, and dynamic selection of the data representation is hardly considered. We refer to [6] for a detailed discussion of approaches for parallel machines.

Context-aware composition [2], [6] is the approach most closely related to transformation data structures. Using online or offline profiling and different machine-learning approaches, the implementation of operations as well as data structures

could be adapted to meet different usage contexts. However, thus far context-aware composition did not regard call sequences of operations as context attributes and only optimizes individual operation calls. Hence, it could not decide for changing a data representation, which only amortizes over a number of operations.

Guoqing Xu addresses a very similar problem [15] but focuses more on the automated generation of code for transforming data structures, while we focus more on how to improve the context awareness to avoid oscillations.

VI. Conclusions and Future Work

Dynamically transforming data structures are introduced to relieve programmers from deciding which data structure is the optimal in a usage context. Instead, a good representation variant is automatically selected behind the visible interface of the data structure. It can even out-perform carefully picked data structure implementations, as it is often impossible to know at compile time which data structure implementation becomes optimal at run time. The optimal data structure implementation may even change throughout the execution of a program such that any static decision gets suboptimal. In the present paper we introduced the design of transformation data structures, we exemplified this design in a case study on array lists, and supported our performance claims with measurements using the DaCapo benchmark suite. The latter showed that by replacing the default array list data structure with our transformation array list implementation leads to more than 5% performance improvement (on average) in the set of, otherwise unmodified, benchmark programs.

Detailed conclusions and open issues targeted by future work are sketched below.

a) Deferred transformation: The cost of transforming from one internal representation to another is typically $\Omega(n)$, which is expensive. If an operation with the current variant is $O(n)$ and the operation in the potential target variant is $O(1)$, then it may be suitable to perform the transformation immediately. When the difference between the operations on different variants is just a constant, eager transformation would be counter-productive. Instead transformation should amortize over a sequence of several operations and, hence, be deferred if such a sequence can be expected. Currently, we use observed sequences to heuristically predict future sequences.

Knowing when it is worth to transform is crucial and future work will improve deferred transformation: we could flag that a data structure benefits from changing internal representation but, not perform the transformation immediately. Instead, it could be done when it is more suitable, piggy-backing on a copying garbage collector. Since it has to go through all objects, copying them from from-space to to-space, it would be possible to let the transformation happen during this transition. A pure copying garbage collector performs an identity transformation that replicates the object graph as it is, without changing it. It could as well perform a semantic identity transformation of sub-graphs that are representations of transformation data structures. This way, transformation would become cheap as part of garbage collection happening anyway, and tone down the current problem of oscillating transformations.

The GC could also help with keeping the constant time required by the state machine down. More specifically, there is a need for a level of indirection and dynamic binding to get to the internal data representation. This costs was measured to in the worst case (Array variant add) 10% overhead, which is bad if the programmer already knows this is what he wants. A GC could forward pointers to the latest versions as part of garbage collection, and hence reduce even this cost, by letting references point directly to the real representation.

b) Adding off-line intel to the on-line dispatcher: The current solution optimizes the sequence of operations that are observed on-line. However, sometimes other sequences of operations could be performed giving the same results, and which may be accelerated with a different data structure.

Consider for example if the usage pattern `sort`, `get(0)`, `get(1)` ... `get(n-1)` is seen many times. When trying to optimize this sequence as it is observed on-line, operation by operation, an array-based variant would probably perform best. If we, however, recognize that this pattern is an iteration through the sorted elements, it could potentially be achieved more efficiently by maintaining a tree representation instead, using a sorted iterator.

A solution would use a compiler to automatically reverse engineer operation sequences, abstracting their semantic meaning, and then transform it to more suitable, abstract, operation sequences that can vary dynamically at runtime with a dynamic dispatcher using context aware composition.

c) Experimental evaluation: Measurements show a 5.19% increase in performance (on average) on the DaCapo multi processor benchmark suite when just replacing a single data structure with a corresponding transformation data structure. DaCapo is a standard benchmark; for the applications contained, data structure implementations were probably statically selected with care but they cannot adapt to dynamically changing usage contexts.

Still, more work needs to be invested in evaluation. More transformation data structures need to be implemented and performance evaluated. Also, the connection of transforming data structures and context-aware composition for other context attributes should be evaluated more in detail, especially for controlling the size of the transformation state machine.

d) Other open issues: The current solution looks mainly at one single policy - optimizing runtime execution time. Maybe there could be other conflicting policies to optimize such as memory usage. Future work could describe how to manage these potentially conflicting goals.

Acknowledgements

This research was supported by the Swedish Research Council under grant 2011-6185. We also thank the anonymous reviewers who helped making this paper better and added some interesting topics for future work.

REFERENCES

[1] Nadeem Abbas, Jesper Andersson, and Welf Löwe. Autonomic software product lines (aspl). In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 324–331, New York, NY, USA, 2010. ACM.

[2] Jesper Andersson, Morgan Ericsson, Christoph W. Keßler, and Welf Löwe. Profile-guided composition. In Cesare Pautasso and Éric Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 157–164. Springer, 2008.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovi'c, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, oct 2006. ACM Press.

[4] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue "Program Generation, Optimization, and Platform Adaptation".

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] C. Kessler and W. Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498, 2012.

[7] Xiaoming Li, María Jesús Garzarán, and David Padua. A dynamically tuned sorting library. In *Proc. Int. Symposium on Code Generation and Optimization (CGO'04)*, page 111ff. IEEE Computer Society, 2004.

[8] Welf Löwe, Rainer Neumann, Martin Trapp, and Wolf Zimmermann. Robust dynamic exchange of implementation aspects. In *TOOLS (29)*, pages 351–360. IEEE Computer Society, 1999.

[9] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, and Manuela Veloso. SPIRAL: Automatic implementation of signal processing algorithms. In *High Performance Embedded Computing (HPEC)*, 2000.

[10] E. Schonberg, J.T. Schwartz, and M. Sharir. Automatic data structure selection in setl. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 197–210. ACM, 1979.

[11] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. In *Proc. 13th European Conf. on Object-Oriented Programming (ECOOP'99)*, pages 367–390. Springer, 1999.

[12] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software–Practice and Experience*, 35(8):705–754, July 2005.

[13] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proc. Int. Conf. on Dynamic Languages (ICDL'07)*, pages 143–156. ACM, 2007.

[14] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[15] Guoqing Xu. Coco: Sound and adaptive replacement of java collections. In *27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, pages 1–26. Springer Berlin Heidelberg.