Master Thesis

Adaptive Class Data Layouts for the Synchronized-by-Default Approach

Author(s):
Koradi, Roger

Publication Date:
2018

Permanent Link:
https://doi.org/10.3929/ethz-b-000267999

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Adaptive Class Data Layouts for the Synchronized-by-Default Approach

Master Thesis
Roger Koradi
May 3, 2018

Advisors: Prof. Thomas Gross
and Martin Bättig
Department of Computer Science, ETH Zürich
Contents

1 Introduction
   1.1 Motivation .............................................. 4
   1.2 Goals .................................................. 4

2 Preliminary Survey ............................................. 7

3 Concepts
   3.1 Adaptive Class Data Layouts .................................. 11
   3.2 Micro-Benchmarks ........................................ 14
      3.2.1 Indirection Overhead .................................. 14
      3.2.2 If-Else Vs Cast ....................................... 14
      3.2.3 Viability of Intel TSX ................................ 15

4 Implementation .................................................. 19
   4.1 Implementation Details .................................... 19
      4.1.1 Background: Handling the Different Access Types .... 19
      4.1.2 Background: Lock Layout .............................. 20
      4.1.3 Split and Why It Matters .............................. 20
      4.1.4 Version Check ........................................ 21
      4.1.5 Migration ............................................. 22
      4.1.6 Detecting Inlining Opportunities ...................... 24
      4.1.7 Creating a new Layout ................................. 26
   4.2 Example ................................................ 27
   4.3 Limitations & Future Work .................................. 29

5 Evaluation ...................................................... 31
   5.1 Test Environment & Procedure ............................... 31
   5.2 General Viability ........................................ 32
## Contents

5.3 Heuristics .................................................. 33

6 Conclusion .................................................. 37

7 Acknowledgements ......................................... 39

Bibliography .................................................. 41
Abstract

Adaptive class data layouts are not determined at compile-time. Instead, the system relies on information learned at runtime to generate and employ new data layouts on demand.

This thesis provides a brief overview over related works and then outlines the hows and whys of an adaptive class data layouts implementation for use in a Synchronized-by-Default [6] prototype written in Java.

Although the implementation is still in an unoptimised state and suffers from teething troubles, the thesis is able to show that it is possible for the adaptive class data layouts to yield a speedup with a fairly aggressive inlining strategy for the benchmarks it was tested on.

After dealing with aforementioned teething troubles, further improvements can be made by reducing the number of checks inserted, cutting down on the amount of unnecessary transaction abort-and-retries or by reducing the bookkeeping overhead.
Chapter 1

Introduction

The Synchronized-by-Default (SBD) approach [6] aims at unburdening the programmer from having to explicitly synchronize interacting parts of a concurrent program by making use, instead, of implicit checks and locks inserted by a software-transactional memory (STM) system automatising the processes that need to take place to ensure correct program execution and on-demand exclusive access to shared memory regions of competing concurrent actors (transactions). The approach is at the time of this writing implemented as a prototype for the managed and object-oriented language Java.

The SBD approach executes all instructions within atomic sections – even if they happen to have irreversible side effects (e.g. I/O, starting a thread, ...). To increase concurrency, these atomic sections must be explicitly split. When an atomic section is split, the currently running transaction is committed and a new transaction is started for the next atomic section.

Prior to the thesis, these locks were collected exclusively in per-instance lock structures. Section 1.1 outlines why and how we changed that with what we refer to as the concept of adaptive class data layouts, which allows locks to become inlined at runtime, meaning they move from the locking structure to a dedicated field in the instance.

The transactional nature of the approach and the concurrent environment make it hard to rely on insights gained at compile-time. On the other hand, the same transactional nature also means that any string of execution can simply abort-and-retry if something is out of the ordinary. As such, it enables the approach to make runtime-decisions about the layout, that would require extensive checks and safeguards outside of a transactional system.

In Chapter 2, we provide a brief overview over some of the already existing attempts on class layout modification.
1. Introduction

The Chapter 3 explains the ideas behind our implementation and provides a few measurements that were done to help finding an implementation.

Whereas Chapter 4 explains what was done to the prototype and how our migration works.

Chapter 5, evaluates the implementations and Chapter 6 provides the conclusions.

1.1 Motivation

Key motivation is a potential performance improvement by making use of spatial locality.

We classify memory accesses into three types:

T1: accesses to instances created within the currently active atomic section
T2: accesses to locations whose locks are already held
T3: accesses to locations whose locks are not held

Type 1 accesses require no locking because no concurrently running transaction knows about the accessed instances until the creating transaction is committed.

Type 2 accesses require a check and re-entrant locks.

Type 3 accesses require the full locking process.

An evaluation [6] showed that five out of six programs most frequently use Type 1 accesses. As such, the algorithm is optimised for these. However, a program can have more accesses of Type 2 than of Type 1, by – for example – doing repeated calculations on the same model. For these, we hope to reduce the overhead of accessing the lock by moving the lock closer to the data it protects, making it more likely for both to reside in the same cache line. This lock inlining should happen at runtime.

Evidently, any solution that reduces the overhead for Type 2 accesses by supporting lock-inlining will increase the overhead for the other types. The question this project tries to answer is whether the trade-off is worthwhile.

1.2 Goals

The following goals have been set for the thesis:

- conduct a survey of previous work regarding class and data layout modifications
- implement a class modification using the provided prototype
1.2. Goals

- determine a suitable heuristic for lock inlining
- evaluate the findings
Chapter 2

Preliminary Survey

This chapter is a brief overview over some works related to inlining or dynamic object layouts in general.

Lhotak and Hendren [13] argue that inlining objects can have significant benefits but also that the benefits are highly dependent on the use case.

Dolby and Chien [9] agree that the benefits can be substantial and evaluate three compiler analyses, finding that on average, object inlining can eliminate 40% of all object accesses and allocations.

Judging by these two papers, this project can have a chance at succeeding.

Kandemir and Kadayif [12] rely heavily on static analysis to estimate the number of cache misses if the data is arranged in a matrix that is iterated over, then determine places at which they insert layout transformation points, where they change the layout of the data by copying it to a new array that is then used in place of the old. Their use case further differs from ours inasfar as they attempt to optimise frequent iteration over all data, rather than frequent access to the same value within the data.

Dolby [8] uses an inter-procedural data flow analysis to determine which objects would most benefit from having other objects inlined into them. It also uses Method Contours [5]. Method contours are context-sensitive representations of a method’s execution environment and “can discriminate arbitrary data-flow properties”. As such, they use them for alias analysis and whenever the flow analysis “needs to distinguish some property” (e.g. for polymorphic fields). The actual inlining step is then a global transformation that requires specialisation of methods that access the inlinees. Such an approach cannot be done at runtime, of course.

Object Combining [14] acknowledges that having a lot of small objects in a program produces strain on both, the object allocator as well as the garbage
2. Preliminary Survey

collector, and aims to reduce it by appending certain objects’ fields to another object, where it believes the objects share about the same life time. They say their approach differs from object inlining in that the latter most often requires the object into which another object is inlined to have a unique pointer to the inlinee while their approach still works without such a relation because it does not immediately get rid of the pointer indirection from the container object to the inlinee. By hooking into object allocation, they can therefore not only combine objects that are linked to one another but also objects that are frequently used together. It relies on heap approximations [10] to learn which objects can be combined and where.

This project, on the other hand, is not primarily concerned with the number of objects available.

Wimmer and Mössenböck[15] present an array inlining optimisation where objects and arrays referencing each other are placed in consecutive memory, fixing their relative offsets, thereby allowing field accesses to be replaced by pointer arithmetic. The just-in-time compiler inserts read barriers that increment a per-class-and-field counter to detect what they call hot fields, i.e., fields accessed so frequently they are worth optimising for. The Java HotSpot garbage collector – for which they implemented their approach – is a copying collector, meaning on each pass it copies reachable objects from a from-space to a to-space. If the garbage collector now moves objects linked by hot fields, it places them in consecutive space (what they call object colocation). If an array is colocated with an object and grows in size, the optimisation is temporarily suspended until the new array can be placed in consecutive space again on the next garbage collector pass.

Unfortunately, the Garbage Collector cannot be manipulated at runtime because it is implemented in C, although editing that C code remains an option. As will be shown in Subsection 3.2.3, care would need to be taken to limit the Java-C interactivity.

Cohen, Tal and Petrank [7] introduce the notion of a layout lock. In a nutshell, a layout lock is a reader writer (RW) lock, where data writers acquire a read lock, layout changers acquire a write lock and data readers do not acquire a lock. Instead, the layout lock maintains a per-thread dirty flag that the readers check after their optimistic read. If a layout changer is active, they discard what they read and try again. It is important to point out that if the data structure a layout lock is added to cannot handle optimistic reads by itself, this layout lock is meant to be used in addition to existing synchronisation between data readers and data writers.

In our case, a different kind of layout lock is already integrated into the approach: it simply uses the locking structures already in place to force the readers and writers to enqueue themselves.
OPTIK Locks [11] are another type of lock that guards access to some kind of versioned data structure. It acquires the lock then immediately unlocks and retries if the version number it expected does not match the locked object’s actual version number – which is increased after a successful locking attempt.

Finally, there’s versioned programming [16]. Although we eventually went a different way because the lock-free approach does not meet our requirements and the additional synchronisation layer is not needed, versioned programming shares a key idea with our approach: allowing multiple versions of something to exist at the same time and switching between them as needed.
Chapter 3

Concepts

3.1 Adaptive Class Data Layouts

Each object instance with lockable fields has its own *locks* array; an array of longs. These locks are used exclusively for instance-level locking. In particular, static members have their own dedicated locks separate from that locks array and final fields are not locked. If and when the prototype detects that one such field gets used very often, we want to move its lock from the locks array into the class. Ideally, of course, that lock will end up either right above or right below its corresponding field (see Figure 3.1), where the chances of their residing in the same cacheline is maximal. In our implementation we offer no such guarantee because although the order in which Java aligns the fields of an instance can be calculated, it cannot be arbitrarily

![Diagram](image)

Figure 3.1: Spatial locality is desirable because it enhances the chance of both the field and its lock to be in the same cache line.
changed.

The idea is also impossible to implement insofar as fields cannot be added to classes at runtime, let alone to their instances. It makes sense, therefore, to merely pretend it were possible and implement the prototype with a small layer of indirection added: all lockable fields are pulled into a separate dedicated data class to an instance of which the locks array is attached. (see Figure 3.2).

Now, each object instance has a data instance that can be swapped out at pleasure. In particular, it is now possible to create a new data class in which the hot lock has been inlined and give the object instance an instance of that class to use as its data instance. This does, of course, require migrating all data from the old data instance to the new in a way that maintains the guarantees provided by the transactional system in which the prototype operates.

There are two directions in which this train of thought can be taken.

- decide for each instance on its own whether or not to use the new layout
- force all instances to migrate to the new layout

Deciding for each instance on its own requires per-instance access tracking, per-instance decision-making and an additional, suitably sophisticated locking structure that harmonises with the SBD prototype’s transactional system to temporarily suspend access to the data object, not to mention a way for clients to handle every existing variant of the data class ever produced. It also makes keeping track of what combination of inlined fields there already was a layout created and for which a new one needs to be created a bit trickier. In return, the prototype only inlines locks that are truly frequently used while leaving the instances alone that belong to the same class but that aren’t accessed as often.
3.1. Adaptive Class Data Layouts

Figure 3.3: Tracking accesses to a field is as simple as unsynchronizedly decreasing a dedicated per-class-and-field counter contained in an array. Each Transaction writes into its own region and to its own region only, whereas the layout switcher periodically scans all of it.

Here, we felt that the demerits outweigh the benefits. The additional locking overhead in particular was something we wanted to avoid, and supporting multiple versions at the same time likely meant having to make all field accesses method calls – i.e., yet another source of unnecessary overhead.

Instead, each class has exactly one active data layout, which forces all instances to migrate to that the next time they are accessed. While resulting in more migrations overall because all instances of a class need to migrate rather than just the truly frequently used ones, making class-based decisions means it is possible to use the locking structure already in place. Because classes have a limited number of lockable fields, tracking accesses to these no longer requires any kind of synchronisation as a fixed-sized array of counters can be allocated, where each transaction writes into a dedicated section of that array. A dedicated Layout Switcher thread can then periodically scan the array and make decisions based on the sum of all transactions’ entries.

The layout switcher first decides which fields need be inlined, then creates a new layout for each class one of the newly inlined fields belongs to. It also creates new layouts for all their subclasses. Finally, it adjusts all clients such that they are aware of there being a new layout and that they migrate the instances they use when necessary.
3. Concepts

3.2 Micro-Benchmarks

After confirming that hot-swapping method body bytecodes multiple times during a JVM execution was at all possible, we performed a short series of microbenchmarks to help guide the design of what would eventually get implemented.

3.2.1 Indirection Overhead

We wanted to know how much of an overhead we can expect just from introducing some indirection.

(a) a direct memory access, can be executed by a simple `getfield`

![Diagram](a.png)

(b) an indirect memory access, the correct object must first be retrieved itself.

![Diagram](b.png)

Figure 3.4: Indirect accesses can be expected to be slower than direct accesses, on average.

Our measurements (Table 3.1) clearly confirm that indirect accesses are always slower than direct accesses. They also show, however, that the differences for the sequential cases are insignificant in comparison to the differences in the randomised access pattern.

For randomised accesses, we can expect reads to be somewhere around 15% slower for the indirect case and writes about 66%.

3.2.2 If-Else Vs Cast

In the beginning stages of the project, we were also considering working with version numbers, in particular as they were used in some locking schemes found while doing the survey (Chapter 2) and as the no-additional-synchronisation policy that the design would later embrace was not yet established.
3.2. Micro-Benchmarks

An alternative to version numbers was using class casts and exceptions. Basically, instead of checking whether a field was of an expected value, the system would simply cast the object to the class it expected, catching the `ClassCastException` that would be thrown if not to branch in the migration path.

We wanted to find out whether or not either of these options was going to be significantly more expensive than the other. And as Table 3.2 shows, that is not the case. Although the exception-handling way appears to be ever-so-slightly more expensive, the difference is not significant enough to tip the scales in favour of one approach or the other.

In the end, the exceptions made it into the design as the system would eventually have to cast the data object anyway for further use.

3.2.3 Viability of Intel TSX

The Intel Transactional Synchronisation Extensions [2] provides cacheline-level transactions. Supporting processors are equipped with the necessary hardware to execute code in atomic regions optimistically – i.e., when not contested, no synchronisation would be required, otherwise the transaction would be rolled back.

It was intriguing enough that we wanted to test whether we could potentially make use of the RTM, the `Restricted Transactional Memory` interface to the TSX, to guard parts or all of the migration process.

Unfortunately, Figure 3.5 shows that using RTM for even as little as an uncontested field read or write is already much more expensive than an optimistically executed direct or unsafe memory access. `Unsafe` here means using the `sun.misc.Unsafe` class to access the memory location with an object and an offset rather than by means of a `getfield` instruction.

It is important to note that the rather large overhead shown in Figure 3.5 is unlikely to stem from the Intel TSX itself. Rather, the reader needs to understand that in order to make use of the RTM interface, our Java implementation needs to call native C code. That C code in turn cannot directly interact with Java objects due to the JVM’s memory handling and the fact that the Java Garbage Collector in particular keeps moving the objects around. (The Java Garbage Collector is a copying collector.) Even if it were possible to get a meaningful object pointer and offset for a Java field in C, the system could at no time after receiving it guarantee that they are still valid. So the only way to interact with Java is for the native C code to make a callback into Java and execute Java code. These repeated switches from Java to C and back are rather expensive, rendering the intel TSX unsuitable to our purposes.
Figure 3.5: Measured uncontested access to a memory location either direct (using getfield instructions, unsafe (using sun.misc.Unsafe and using the intel RTM interface. The RTM is much more expensive because it needs to repeatedly switch between executing Java and C code.)
### 3.2. Micro-Benchmarks

#### Table 3.1: Measurements of read and write access times in nanoseconds, both random and sequential for either the direct or the indirect case. Column overhead shows the overhead of indirect compared to direct accesses.

<table>
<thead>
<tr>
<th>option</th>
<th>subcase</th>
<th>count</th>
<th>mean</th>
<th>std</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECT</td>
<td>READRND</td>
<td>100</td>
<td>175.576401</td>
<td>1.080074</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READSEQ</td>
<td>100</td>
<td>3.894915</td>
<td>0.03099</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITERND</td>
<td>100</td>
<td>109.580224</td>
<td>0.527775</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITESEQ</td>
<td>100</td>
<td>5.094607</td>
<td>0.010426</td>
<td></td>
</tr>
<tr>
<td>INDIRECT</td>
<td>READRND</td>
<td>100</td>
<td>202.535847</td>
<td>1.049041</td>
<td>+15.4%</td>
</tr>
<tr>
<td></td>
<td>READSEQ</td>
<td>100</td>
<td>5.859691</td>
<td>0.035949</td>
<td>+50.4%</td>
</tr>
<tr>
<td></td>
<td>WRITERND</td>
<td>100</td>
<td>182.367565</td>
<td>0.489854</td>
<td>+66.4%</td>
</tr>
<tr>
<td></td>
<td>WRITESEQ</td>
<td>100</td>
<td>6.459346</td>
<td>0.044907</td>
<td>+26.8%</td>
</tr>
</tbody>
</table>

#### Table 3.2: Measurements of the branching times for a basic if-else condition and for an exception-driven branching in nanoseconds, for both, the failing (IFFALSE) and the succeeding (IFTRUE) cases. Column overhead shows the overhead of exception-driven branching compared to if-else branching.

<table>
<thead>
<tr>
<th>option</th>
<th>subcase</th>
<th>count</th>
<th>mean</th>
<th>std</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC_IFELSE</td>
<td>IFFALSE</td>
<td>100</td>
<td>8.110549</td>
<td>0.47679</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFTRUE</td>
<td>100</td>
<td>7.855983</td>
<td>1.117597</td>
<td></td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>IFFALSE</td>
<td>100</td>
<td>9.556694</td>
<td>0.521078</td>
<td>+17.8%</td>
</tr>
<tr>
<td></td>
<td>IFTRUE</td>
<td>100</td>
<td>8.307265</td>
<td>0.503861</td>
<td>+5.7%</td>
</tr>
</tbody>
</table>
Chapter 4

Implementation

4.1 Implementation Details

4.1.1 Background: Handling the Different Access Types

This subsection provides details about the handling of the access types mentioned in Section 1.1.

Specifically, Type 1 accesses require no locking because no concurrently running transaction knows about the accessed instances until the creating transaction is committed. To optimise for them, the locking structure is set to a special transaction-specific “tag” value. The locking algorithm then checks for that value and forgoes locking if the tag matches the currently active transaction’s. When a transaction is committed, the next accessing transaction will replace the special value by the actual locks.

Each transaction is given a unique ID. To make the locks re-entrant, a lock contains a bitmask keeping track of all transactions that have acquired a lock. As such, there cannot be more transactions than there are bits available to be used for the bitmask.

Type 2 accesses are optimised for by comparing the current lock with a transaction-specific bitmask consisting of its ID and whether or not it tries to acquire a write lock.

Type 3 accesses aren’t optimised for. Since the transaction neither has exclusive access to the region nor the required lock, it needs to acquire it – and potentially enqueue itself, if the lock is currently held by a write – before it can continue execution.
4. Implementation

4.1.2 Background: Lock Layout

The SBD prototype uses 64 bit locks because that is the largest value that can still be compare-and-swapped using Java’s Unsafe functionality. For the scope of this thesis, it is not necessary to understand every detail of the locking algorithm used by the SBD prototype. It is useful, though, to know about the regions marked in Figure 4.1 so they do not need to be explained every time they are mentioned.

Queue ID: if a transaction tries to acquire a lock that is currently blocked, it will enqueue itself in a FIFO-queue, the ID of which is stored here for later transactions to reuse.

Writer Lock: encodes whether or not a transaction holds a write lock to the location or whether a transaction holds a read lock that it wants to upgrade to a write lock.

Invalid Flag: newly introduced in this thesis. If a transaction tries to acquire a lock whose invalid flag is set, the transaction is aborted and restarted.

Bitmask: shows all transactions that hold any kind of lock to the location this lock belongs to. As a consequence, at most 54 transactions can be concurrently active at any one time. The bitmask is used to make the locks re-entrant as well as to efficiently check whether a given transaction holds a certain lock.

4.1.3 Split and Why It Matters

As mentioned in Chapter 1, the SBD approach runs all code in atomic sections. To increase concurrency, a programmer has to explicitly split the section. A split commits the currently active transaction – thereby releasing all locks it held up to that point – and starts a new one.

So what happens when a transaction is not committed but aborted instead? The short answer is “a rollback and retry”, but that retry, especially, deserves a closer look because it is quite crucial to the functioning of the adaptive class data layouts.

Internally, every split and every call to a method that can split also saves the state of the stack frame. When after a rollback a transaction needs to be
4.1. Implementation Details

restarted, the method that saved the last committed stack frame is invoked. At the beginning of that method, the saved stackframe is used to restore the situation at that point in the execution after which execution continues either

- with the method call, if the last save was before a call to a method that can split, or
- with the statement following the split, if the last save was before a split instruction.

and that the method gets re-invoked is essential. Why? Because as mentioned in Section 3.1, when the layout switcher creates a new data layout for a class, it needs to let all clients know that there is a new version available. It does so by re-instrumenting these classes’ methods (as will be covered in more detail in Subsection 4.1.7). However, hot-swapped bytecode never affects running methods. The new bytecode is only loaded for new invocations (c.f. [4]).

When the system rolls a transaction back because another transaction that modified a value the first transaction read was committed, going to the last committed frame is sufficient. However, when the system rolls back because an instance needed to migrate to a new layout version (as explained in more detail in Subsection 4.1.5), it can quite simply take the very first method invoked and rebuild the stack from there on, ensuring that all active methods operate with their new bytecode when execution resumes. This would be impossible to guarantee if rollback-recovery happened within the called method.

4.1.4 Version Check

When a new data layout has been made available, clients of the corresponding class need to know about it and transition towards using it, rather than keep using whatever layout instances they may be given. As detailed in Figure 4.2, that version check is done by simply casting the received data instance to the expected version. Subsequent data layouts for the same class do not inherit from one another, although they may have a common ancestor, such that this check always fails when a new data layout has been made known to the executing class.

Strictly speaking, not every call to migration needs be followed up with a MigrateException. In particular, when the bytecode of the method calling is already up-to-date but the layout version of an instance is not, it is sufficient to simply migrate the data layout of the object instance to the newest version and continue with that. For now, however, all migration attempts lead to an abort to reload all called methods’ bytecode.
4. **Implementation**

```java
A a;
TrxData data;
A$Data$0 castData;
int value;

data = a.data;
try{
    castData = (A$Data$0)data;
    catch(ClassCastException e){
        data.migrate(a)
        throw new MigrateException(); //triggers transaction abort and retry
    }
    value = castData.x;
}
```

Figure 4.2: Example code corresponding to the read access value = a.x.

### 4.1.5 Migration

The `migrate(Object base)` function every data layout class has does exactly one thing: call a static migration method on the original layout class. This extra level of indirection was done to deal with upcast values; let `a` in Figure 4.2 be an instance of `B`, whereas class `B` inherits from class `A`. Then it can still be cast to an instance of `A`, but its `migrate`-invocation will still call the static migration method of `B$Data$0` rather than `A$Data$0`. As a consequence, the static migration methods do not need to know about the class hierarchy at all, they must simply be able to handle migration from an arbitrary version of itself to the most recently created version of itself. The layout switcher ensures that the static migration methods are always up-to-date before any other class knows about there being a new layout at all.

There are two cases in Figure 4.3, that are of interest, here.

- the one labelled `C1`, where the passed version is up-to-date, and
- the one labelled `C2`, which corresponds to the actual migration step, where the passed version is older

The one labelled `C3`, where execution falls through all version checks, is an error that must not occur.

Keep in mind that after both cases, the currently running transaction is aborted and restarted (Figure 4.2).

Although the versions differ in where the values and locks are situated, the actual migration step always uses the following algorithm. The algorithm requires an additional `invalid` flag within the lock structure (see Subsection 4.1.2). If a transaction tries to acquire a lock whose `invalid` flag is set, the transaction is aborted and restarted.
4.1. Implementation Details

class B$Data$0 extends A$Data$0{
    public static void migrateFrom(B b){
        TrxData data = b.data;
        try{
            B$Data$3 b3 = (B$Data$3)data;
            return; //case C1: the passed version already is up-to-date
        }catch(ClassCastException e){}
        //case C2: version is older and data must be migrated
        try{
            B$Data$2 b2 = (B$Data$2)data;
            b.data = migrateFrom2To3(b2);
            return;
        }catch(ClassCastException e){}
        try{
            B$Data$1 b1 = (B$Data$1)data;
            b.data = migrateFrom1To3(b1);
            return;
        }catch(ClassCastException e){}
        try{
            B$Data$0 b0 = (B$Data$0)data;
            b.data = migrateFrom0To3(b0);
            return;
        }catch(ClassCastException e){}
        //case C3: Error
        throw new IllegalStateException("unknown version of B: " + data.getClass());
    }
}

Figure 4.3: A skeleton of the static migration method for class B when there are four layouts available, B$Data$3 being the most recent.

(1) if the data instance was created in this transaction (which the system can tell because then the locks array will correspond to the special value mentioned in Subsection 4.1.1), simply return. Eventually, a new data instance of the correct type will be created after an abort and retry.

(2) otherwise acquire write locks for all fields (including inherited and/or shadowed)

(3) invalidate the acquired locks (set their invalid bit [see Subsection 4.1.2]), such as to deny further use of the old data instance

(4) create a new instance of the correct data layout type

(5) copy all fields’ values from the old data instance to the new, again including inherited and shadowed fields

(6) copy all acquired locks from the old data instance to their location in the new, unsetting the invalid bit as well as the queue ID (see Subsection 4.1.2)
4. Implementation

(7) create a map from the old locks to the new, telling the rollback procedure to unlock both, the old locks and the new (the old locks remain invalid, but by unlocking them, transactions placed in the queues corresponding to them are awakened such that they can now read the new invalid bit and abort)

(8) finally, return, triggering the abort and retry.

The main benefit of this algorithm is that it integrates into the existing synchronisation layer and thus does not add synchronisation overhead to accesses that do not require migration.

The fact that both, inherited and shadowed fields, must (of course) be handled too means by extension that `getfield` / `putfield` instructions are not sufficient as these cannot access the shadowed fields without complete knowledge of the inheritance tree. Instead, each version must maintain a map from a unique identifier to a structure that holds the offsets required to handle the fields with `sun.misc.Unsafe`. That map does not change and is constructed in a data class’ static initialiser.

Note further that even in the C1 case, where we already have a data Object of the correct type, we must still follow up with an abort-and-retry. There are two reasons why we end up in C1 at all.

• in the brief period between the `ClassCastException` and our invocation of the static migration method, a concurrently running transaction already completed migration. While this is not in and of itself a reason to abort, the system cannot distinguish it from the other, which is:

• the migration was completed earlier but the code still tries to cast to a wrong type. This is much more common and indicates that the method that triggers the migration call is not yet operating with the correctly updated bytecode. This can happen, for example, when the layout switcher has to update a lot of classes after creating a new layout (Subsection 4.1.7) and it simply has not updated this class yet. In such a case it is possible that the object already was migrated by a different class and its data field set to an instance of the yet-unknown (for the class attempting the migrate) layout to exist

4.1.6 Detecting Inlining Opportunities

One preparatory step in the transformation of available classes by the SBD prototype includes assigning each field a unique identifier. In our implementation, that identifier is simply a non-negative integer with no further meaning attached to it. Negative integers have been reserved for special fields, e.g. the locks array itself, that need to be handled differently from the normal fields.
Let $N$ here denote the amount of fields that received a unique ID. When a transaction accesses a field, it will also increase a counter in its dedicated region in the log array (as illustrated in Figure 3.3). More specifically, if the $t$-th transaction accesses a field with ID $i$, it will increment $\text{accessLog}[t \times N + i]$.

The layout switcher, on the other hand, does not differentiate between the transactions and simply adds each field’s entries up (see Figure 4.4).

```java
private int[] createSnapshot() {
    int[] snapshot = new int[N];
    System.arraycopy(accessLog, 0, snapshot, 0, N);
    for(int i = N; i < accessLog.length; ++i){
        snapshot[i % N] += accessLog[i];
    }
    return snapshot;
}
```

Figure 4.4: Creating a snapshot of the accessLog.

The system also does not differentiate between read and write accesses.

It is obvious that this is a very simple way to keep track of the accesses and that it could have easily been replaced with something much more sophisticated. However, this bookkeeping is an overhead that must be paid with every data access. As such, we thought it provident to keep track only of the very basics and forgo additional complexity. And as the system does not un-inline fields, there is no point in logging the accesses to inlined fields. Therefore, inlining a field’s lock also turns off the logging for that field.

**Heuristics**

The layout switcher periodically creates a snapshot (Figure 4.4) and then passes the two most recent snapshots through a heuristic to decide what classes to change and how. That process is explained in more detail in Subsection 4.1.7. Here, we want to focus on one aspect only: when to inline a field.

There are two types of heuristics that we can apply to our access count, both of which are simple such as to not take a lot of time to compute.

**Frequency**: inline a field if

$$F = \frac{\text{total No f Accesses}}{\text{elapsedTime}} > Y$$

where $Y$ can be chosen arbitrarily.
4. Implementation

*Always*: the simplest heuristic and a special case where $Y = 0$. Inline a field if there has been any access to it at all.

### 4.1.7 Creating a new Layout

In a preparatory step, all parts that must be adapted when a class’ data layout updates are outsourced to generated bytecode. The arguments needed to generate the respective functionalities are encoded into a static dummy method name. That method does not exist but is called nonetheless. The resulting *dummy class* is stored to disk for later reuse. Using ASM[3], these static method calls are first read and processed to yield the arguments, then replaced by the generated functionality. Arguments that will only exist at runtime are passed by their containing local variable slot’s number. To make sure that the generated code does not corrupt the rest of the method it is generated in, the amount of local variable slots used is tracked and passed to the generating methods.

When a heuristic (Section 4.1.6) is applied, there are two types of classes that need be considered:

- classes that need to be recreated
- classes that need to be re-instrumented

*Re-instrumenting* a class means, in this context, to update the arguments encoded in the dummy method names in all the class’ methods and run the dummy class by the dummy replacer to receive the finished class’ bytecode that can be loaded. The bytecode is then hot-swapped for the class’ currently operating bytecode.

*Recreating* a class data layout, on the other hand, means generating a completely new data class version for it. The resulting class is then injected into the JVM.

In either case, the injection of the new bytecode into the running JVM is done with ByteBuddy [1].

- All data classes for which the heuristic says a lock is going to be inlined, need be recreated, of course.
- All data subclasses of classes that need to be recreated need to be recreated as well.
- Clients of any class that needs to be recreated need to be re-instrumented.

And all classes must be handled in an order that makes sure that a new data class is available before one of its clients gets re-instrumented. The layout switcher first creates all new layouts to ensure that and only then starts re-instrumenting classes.
Whenever a class is recreated, the corresponding zero-Layout, i.e., the data layout with which execution starts, is re-instrumented such that its static migration method (Subsection 4.1.5) is capable of handling the new layout version. Only then are other classes re-instrumented. It can happen, of course, that some already re-instrumented classes migrate an object’s data instance to a version another class cannot yet handle because it was not yet re-instrumented. In such a case – as outlined in Subsection 4.1.5, that second class will simply be caught in a migration loop until it is re-instrumented, at which point it can then make progress.

4.2 Example

Consider the sample benchmark in Figure 4.5. Note that the System.out uses are not transactional because they are meant to illustrate the control flow, instead. A typical output will look similar to the one provided in Figure 4.6.

Why does SECOND TEST get printed twice?

The method execute2CanSplit performs a lot of accesses to f.x. So sometimes during that execution, the prototype will decide that the lock for f.x should be inlined. It will create a new data layout for the class Foo, Foo$Data$1, load this newly created class and re-instrument first the zero-layout, Foo$Data$0, such that its static migration method knows how to migrate from the Foo$Data$0 class to the Foo$Data$1 class. It then re-instruments the Test class because it is a client of Foo.

However, as execution does not leave the execute2CanSplit just yet, it keeps using the old bytecode, not migrating yet.

Eventually, the method returns to execute – that also still uses the old bytecode – and prints SECOND TEST.

Execution then calls execute2CanSplit a second time. Because it is a new invocation, it is using the new bytecode. And therefore, it recognises that f still has an instance of the zero-layout attached to it. This will trigger an abort. (Note that if we had passed the second execution of execute2CanSplit a new instance of Foo, it would have been created with a Data$1 instance, already, and no migration would take place.)

Execution resumes at the last Keyword.split encountered. In this case, that is the split just after the assignment to timeFirst. And so, SECOND TEST gets printed a second time.

It is important to notice that although we say execution resumes at that split, it is in fact a completely new call to the execute method that jumps to that point in the very beginning. Because as outlined in Subsection 4.1.5, that
4. Implementation

```java
package parc.benchmark;

import parc.lang.Keyword;

public class Test {
    public static final int NOF_INC = 1500;
    public static final long WAIT = 1000000L;
    
    public static void execute() {
        Foo f = new Foo();
        Test t = new Test();
        System.out.println("FIRST test");
        System.out.flush();
        long start = System.nanoTime();
        t.execute2CanSplit(f); //will run with unmigrated Data$0
        long timeFirst = System.nanoTime() - start;
        Keyword.split(); //prevents rollbacks from altering measured time
        System.out.println("SECOND test");
        System.out.flush();
        start = System.nanoTime();
        t.execute2CanSplit(f); //will migrate and then run with migrated Data$1
            //logging also turned off
        long timeSecond = System.nanoTime() - start;
        Keyword.split(); //prevents rollbacks from altering measured time
        System.out.println(String.format(
            "1st test:\t %d ns
" +
            "2nd test:\t %d ns",
            timeFirst, timeSecond
        ));
        System.out.println("f = " + f);
        System.out.println(String.format("f.x = %d", f.x)); //no migration because bytecode already up-to-date
    }
    
    public void execute2CanSplit(Foo f){
        int i = NOF_INC;
        while(i --> 0){
            f.x += 1;
            long k = WAIT;
            while(k --> 0){ //time waster to give the TrxLayoutSwitcher a chance
                Keyword.split(); //I’m done with f.x for this iteration
            }
        }
    }
    
    class Foo {
        public int x = 0;
    }
}
```

Figure 4.5: an example benchmark
4.3 Limitations & Future Work

The current implementation still suffers from some teething troubles. Unfortunately, these could not all be identified and eliminated in time for the project’s deadline. Getting rid of them is definitely on the to-do list.

As mentioned in Subsection 4.1.4, it is also not necessary to follow up every single migration with an abort-and-retry circle. When the method’s bytecode is up-to-date and an outdated data layout is migrated to the newest, the method could continue execution with that newly created migration target. Adaptations in that regard should be straightforward.

Currently, the version check outlined in Subsection 4.1.4 is done every single time a data instance is accessed. That is excessive because no other thread can migrate the data instance so long as any of the data instance’s locks are held by a different thread. A flow analysis could, therefore, reduce the amount of checks performed to once per instance and transaction.

Future iterations of the implementation may want to turn off the access logging for certain non-inlined fields, either because a certain time limit has passed, after which no more logging takes place or because it’s become impossible to reach the inlining threshold level. Analysis would need to take place to determine when exactly that decision is taken.

Figure 4.6: an output of the example benchmark

means that the execute method now also operates with the new bytecode and not just execute2CanSplit. If that were not the case, there would be another abort-and-retry to fetch the new bytecode at the very end of execute, where \( f.x \) is read.
Chapter 5

Evaluation

Evaluating the implementation proved harder than anticipated. The implementation still suffers from teething troubles that prevent testing it on the full scale benchmarks mentioned in [6]. Unfortunately, they could not all be identified and eliminated in time for the project’s deadline, so we decided to instead write some shorter, simpler benchmarks and measure with these. The following benchmarks have been implemented:

- a random access benchmark where we have a large array of objects, random elements of whose fields are accessed.
- a multi-threaded kmeans implementation
- a linked list implementation, where the threads randomly execute one of addLast, addFirst, remove and contains.
- an AVL-tree implementation, where the threads randomly execute one of insert, remove and contains.

5.1 Test Environment & Procedure

All test were done on the ETHZ Mafushi machine. It has four Intel Xeon E5520 processors with (each)

- 4 cores
- 8 threads (hyperthreading is disabled on this machine)
- 8 MB L3 Cache (shared)
- 2.26 GHz base frequency (turbo mode disabled)
- 5.86 GT/s QPI

All experiments were repeated 10 times, each of which reported an average over 40 runs (a run consisting of one complete execution of the benchmark).
The prototype is implemented for Java 1.7.0.65 and we compare three states of it:

- the original state without any modifications
- an indirect-only state where the system already has the indirections micro-benchmarked in Subsection 3.2.1 but does not migrate
- a migrating state where the system performs the migrations outlined in Subsection 4.1.5

Out of the four smaller benchmarks mentioned in the beginning of the chapter, only the random access benchmark ran reliably on all prototype states and in a multi-threaded setting. We use it, therefore, to measure the general viability.

The linked list and AVL-tree benchmarks ran single-threadedly on all prototype states. We use them to measure differences in our heuristic.

The kmeans benchmark did not run on the original state and was dropped out of time concerns.

### 5.2 General Viability

We used the random access benchmark to measure the general viability of a migrating prototype.

The benchmark is fairly simple:

There is an array of 5'000'000 object references, to be filled with instances of Foo, where Foo has exactly one int-typed field, x.

In the migrating prototype state, the benchmark forces the system to inline the lock for Foo.x before it fills the array.

The benchmark allocates the array and fills it with new instances of Foo. It then performs 4'000'000 accesses to that array in total, split in equal parts amongst the available threads, after which the benchmark ends.

As a baseline for comparison, we take the single-threaded original case. Figure 5.2 summarises our findings.

The indirect-only case is just overhead and in all instances slower than the original implementation. That overhead is particularly noticeable for the single-threaded case.

With increasing parallelism, the migrating prototype performs better than either the original or the indirect-only prototype, indicating that when multiple accesses can be performed concurrently, the savings for each access – which Figure 4.6 already showed – outweigh the overhead imposed by the indirection.
5.3. Heuristics

```java
public void doWorkCanSplit(Random rnd, ArrayDim1<Foo> data){
    int value = 1;
    for(int i = 0; i < NOF_ACCESSSES_PER_THREAD; ++i) {
        int pos = rnd.nextInt(NOF_ENTRIES);
        boolean write = rnd.nextBoolean();
        if(write) data.get(pos).x += value;
        else value += data.get(pos).x;
        Keyword.split();
    }
}
```

Figure 5.1: The access pattern for the worker threads in the RandomAccess benchmark. NOF_ACCESSSES_PER_THREAD depends on the amount of threads available and is pre-computed.

Of course the differences between the cases are small, here, due to the limited nature of our testing benchmark. But it stands to reason that an effect already measured at such a small scale will likely propagate to more complex use cases as well.

5.3 Heuristics

The Linked List benchmark consists of a simple linked list implementation that supports operations addFirst, addLast, remove and contains. The benchmark performs 2’000’000 random operations and then ends (see Figure 5.3).

Figure 5.5 shows the measured speedups for the single-threaded execution across different heuristic settings.

Similarly, the AVL benchmark consists of an AVL tree implementation that supports operations insert, remove and contains. The benchmark performs 2’000’000 randomised operations and then ends (see Figure 5.4).

Figure 5.6 shows the measured speedups for the single-threaded execution across different heuristic settings.

Although the benchmarks are too small to say much in a more general sense, both graphs show that in these cases, at least, a fairly aggressive inlining policy yields the highest speedup as opposed to a more relaxed one.
5. Evaluation

Figure 5.2: Although small, the benchmark does show an improvement in the speedup when comparing the migrating prototype to the unmigrating ones.

public void doWorkCanSplit(){
    Random rnd = new Random(Thread.currentThread().getId());
    int a = NOF_ACCESSSES_PER_THREAD;
    while(a --> 0) {
        int item = rnd.nextInt(MAX_ITEM);
        int op = rnd.nextInt(4);
        switch(op){
            case 0: list.addLastCanSplit(item); break;
            case 1: list.addFirstCanSplit(item); break;
            case 2: list.removeCanSplit(item); break;
            case 3: list.contains(item); break;
            default: throw new RuntimeException("unexpected op: " + op);
        }
        Keyword.split();
    }
}

Figure 5.3: the work of a worker thread in the *LinkedList* benchmark.
public void doWorkCanSplit(){
    Random rnd = new Random(Thread.currentThread().getId());
    int a = NOF_ACCESSSES_PER_THREAD;
    while(a --> 0) {
        int item = rnd.nextInt(MAX_ITEM);
        int op = rnd.nextInt(4);
        switch(op){
            case 0:
                case 1: tree.insertCanSplit(item); break;
            case 2: tree.removeCanSplit(item); break;
            case 3: tree.contains(item); break;
            default: throw new RuntimeException("unexpected op: "+ op);
        }
        Keyword.split();
    }
}

Figure 5.4: the work of a worker thread in the AVL Tree benchmark

![Graph of Speedups for the Linked List benchmark](image)

Figure 5.5: Single-threaded speedups of the LinkedList benchmark across various heuristic settings. Remember from Section 4.1.6 that a heuristic $Y$ is interpreted as the minimum number of accesses per millisecond.
5. Evaluation

Figure 5.6: Single-threaded speedups of the AVL tree benchmark across various heuristic settings. Remember from Section 4.1.6 that a heuristic $Y$ is interpreted as the minimum number of accesses per millisecond.
Chapter 6

Conclusion

Adaptive Class Data Layouts as explained in this thesis can result in a speedup when enough accesses are performed, even in the unoptimised state its implementation is in at the time of this writing. For the benchmarks that we tested with, best results were reached for an aggressive inlining strategy, although the benchmarks were not large enough to make that statement for more general use cases.

The teething troubles the implementation still has need be identified and dealt with.

Further improvements may then be made by limiting the amount of checks executed per method and by making sure that migration only results in an abort-and-retry when it is strictly necessary.

It may also be beneficial to stop logging field accesses when either a certain amount of time has passed or it becomes clear that the field whose accesses are logged cannot under any reasonable program execution reach the threshold set by the heuristic.
Chapter 7

Acknowledgements

Thanks go to Martin Bättig for his patience and invaluable support, and to Prof. Thomas Gross for enabling the thesis.
Bibliography


Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Adaptive Class Data Layouts for the Synchronized-by-Default Approach

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):
Koradi

First name(s):
Roger

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zürich, 3.5.2018

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.