Imperial College London

Department of Computing

Speculative Execution

Author:
Jonathon Marks

Supervisor:
Prof. Alexander Wolf

June 17, 2014

Contents

Abstract

Historically chip designers have exploited the ability to pack a larger and larger number of transistors on to chip circuits in order to push clock speed as far as possible. For many years this was sufficient, until 2003 when there was a noticeable plateau due to issues such as heat and power consumption [30]. Since 2003 there has been a pivot towards multicore architectures which support the multithreaded model of software design. However there are many applications which remain single threaded or which do not fully exploit the number of cores provided by the underlying system. Speculative execution looks at exploiting this spare processing capacity by speculating over the future state of an application. Existing solutions have looked at speculatively parallelising sequential code regions, or speculatively prefetching remote data. In this project we look at applying speculation to interactive Java applications where we can couple this inherent spare capacity with the spare capacity that arises when the user is thinking what action to select next.

We present a two phase solution; in the first phase we statically transform the application to support speculation through a mixture of manual code annotations which constrain the speculation space, and automatic code transformations which add control structures used at runtime to guide speculation. For the second phase, we present a speculative engine that dynamically manages speculation at runtime. It includes functionality for isolating speculative side effects, detecting dependency violations and checkpointing speculative state. It also dynamically decides what to speculate based on learning user behaviour.

We have successfully applied speculation to an example Java application, however we found that the overheads from speculation mean that there is a minimum user think time before speculation provides a benefit over the native version of the application.

Acknowledgements

I would like to thank my supervisor, Prof. Alex Wolf, and my second marker, Dr. Cristian Cadar, for their support and guidance throughout this project. I would also like to thank my friends and family for their support and encouragement.

Chapter 1

Introduction

1.1 Context

Speculative execution is concerned with utilising spare processor capacity to carry out work that may be required in a future state of the application. This reduces the latency when the result of this work is requested at a later point. This prediction has historically taken many forms, from specialised eager-execution processors that execute both branches of a conditional before the control decision is taken [19], to speculative prefetching of inputs that the program may require later in its control flow [7,28]. It has also taken the form of thread-level speculation where sequential code regions are speculatively run in parallel and the system is rolled back if any data dependencies are violated [9].

1.2 Motivation

In this project we are interested in speculating over the range of actions that a user can carry out in an interactive Java application. If we can contain this speculation to execute within the user's think time, then actions with inherently long computation times can appear much more responsive to the user. Figures 1.1 and 1.2 illustrate this; we can see that the response time of our action in the speculative system, $\Delta t'$, is significantly smaller than the response time in the non-speculative system Δt.

Whilst most developers desire highly responsive applications, achieving this using a mechanism such as speculative execution should require minimal programmer intervention. A specialised implementation where the speculation logic is tuned to the specific application may increase the accuracy of speculation but it may also obfuscate the structure of the code and requires significant developer intervention. Instead, in this project we have designed the speculative engine to work with a broad range of applications and we allow the engine to expose certain parameters which can be dynamically tuned for different applications.

In this project we explore how we can use Aspect Oriented Programming to instrument

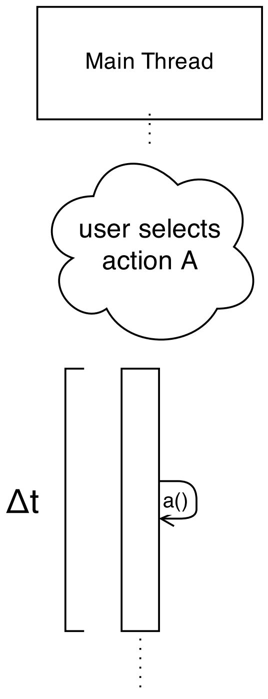

Figure 1.1: Non-speculative system. The user selects *a()* after some think time.

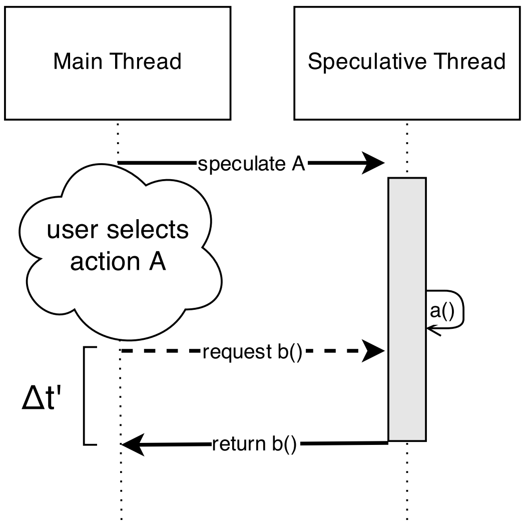

Figure 1.2: Speculative system. While the user is deciding what action to choose the system begins speculating action A. When the user does eventually select A, a large portion of the execution time has been covered by the think time.

the application with speculative capability. Using Aspect Oriented Programming allows us to overcome the problems of instrumentation strategies used in other projects, for example using custom virtual machines [9] which reduces the portability of the system, or custom class loaders [5] which are unable to instrument any classes within the JDK and also incur the instrumentation overhead at runtime rather than at compile time as in this project.

As a motivating example we consider Pixelitor[1], an open-source Java image processing package. This is a useful application to consider as it has a visual representation of its state (the image), which clearly demonstrates whether speculations successfully isolate their side effects. It has a set of image filters which have a variety of execution times for the same input image, allowing us to reason over the benefits of speculation in different scenarios.

1.3 Objectives

We used the following principles to guide the development of the speculative system:

1. Safe and correct speculation
   Speculative threads must not commit any modifications to memory or externalise any output until it has been determined that control should have flowed into this speculative

[1]http://pixelitor.sourceforge.net/

path. A speculatively-enhanced application should produce identical results as its native counterpart.

2. Easy to use
   The speculative system should integrate with the application requiring little effort on the part of the application developer.

3. Silent speculation
   Speculation should appear silent to any non-speculative threads. Non-speculative threads should not contend with speculative threads for CPU resources.

4. Beneficial
   The speculative system should improve the response time of user actions.

1.4 Contributions

1. We have reviewed the state of the art speculative systems, in particular speculative prefetching and speculative parallelisation techniques.

2. We have presented a two phase solution for enhancing an interactive Java application with speculative capability;
   (a) In the first phase we have shown how the source code can be statically transformed to support speculation. This is through a mixture of
      i. manual code annotations to constrain the speculation space and to indicate which methods contain side effects that cannot be isolated
      ii. automatic code transformation to add control structure around loops for checkpointing and to refactor array operations so that they can be intercepted at runtime
      iii. aspect oriented programming for weaving the aspects of the speculative engine which control speculation at runtime into the application's code
   (b) We have presented a speculative engine for the runtime control of speculation (the second phase). This is capable of
      i. isolating the state of speculative threads using copy-on-write
      ii. detecting read after write conflicts
      iii. checkpointing the state of speculative threads and reverting to a suitable checkpoint if a conflict is discovered. We have included two techniques for checkpointing
         A. call stack checkpointing
            Speculative threads recursively move up their call stack until they reach a point in their control flow before a conflict occurred

        B. loop checkpointing

        We checkpoint the speculative state periodically during loops so that a speculative thread that has restarted due to a conflict can skip iterations that were executed before the conflict occurred when it revisits the loop

    iv. dynamically deciding what to speculate based on

        A. the user's current interaction with the graphical interface

        B. learning from how the user has interacted with the application in the past

    v. priority based scheduling. Speculative threads are dynamically created and retired to maintain the maximum amount of speculation without impacting on the performance of non-speculative threads

3. We have presented a modified version of Pixelitor where any of its image filters can be speculated. We have evaluated the effect of speculation on the response time of a subset of these filters and found that

    (a) there is a minimum user think time required before speculation provides a benefit because of the overheads of speculation. Table 1.1 shows these minimum think times

    (b) if developers have some knowledge of these overheads they can restructure their code to achieve a reduction in the minimum wait time, for example of up to 80% with the Unsharp Mask filter

    (c) checkpointing and periodic verification both improve the average response time when we increase the likelihood of a conflict between non-speculative and speculative threads.

4. We have presented areas where future work could extend the functionality of the speculative engine, such as

    (a) speculatively delaying the execution of methods that cannot have their side effects isolated so that the methods that follow them can be speculated (as long as they do not depend on the delayed method)

    (b) using continuations in Java to create a checkpoint at any point that a speculative thread can revert to

    (c) maintaing two versions of each method; a non-speculative version with no copy-on-write logic and a speculative version which does use copy-on-write. This would reduce the overhead for non-speculative threads.

Filter	Native execution time (ms)	Minimum think time (ms)
Unsharp Mask	5003	3500
Glass Tile	3334	12000
Glint	7431	1500
Pointillize	14541	5000
Pixelate	289	1250
Difference of Gaussians	3721	2700

Table 1.1: The time taken to apply the filter to a 5000 x 5000 image of noise in the native, non-speculative version of Pixelitor, and the minimum user think required for speculation to provide a benefit over the native version.

Chapter 2

Background

In this chapter we provide a background on the logistic regression technique used to learn user behaviour, and aspect oriented programming which we use to instrument an application's bytecode to support speculation. We then consider related work in the field of speculative execution.

2.1 Logistic Regression

Logistic regression is a classification technique where we use a set of independent variables to predict a binary dependent variable. Assuming we code the states of this binary variable as 0 or 1, logistic regression returns the probability that, given the values of the independent variables, this dependent variable will take the value 1. More formally we can express logistic regression as follows [4]:

Given a training set of size m, $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ where $x^{(i)} \in \Re^{n+1}$ are the values of the independent variables for sample i and $y^{(i)} \in \{0, 1\}$ represents the binary classification of the dependent variable for sample i.

The hypothesis takes the form $h_\theta(x) = \frac{1}{1+\exp(-\theta^T x)}$,
and the model parameters θ are trained to minimise the cost function
$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right]$

2.1.1 Softmax regression

Logistic regression only works with binary predictor variables, however we can apply the softmax technique to generalise this to multinomial logistic regression where the classification is multiclass. Now we have that $y^{(i)} \in \{1, 2, \ldots, k\}$ since the dependent variable can take one of k values, and the hypothesis estimates the probability that $p(y = j|x)$ for each value of $j = 1, \ldots, k$. The hypothesis for softmax regression takes the form

$$h_\theta(x^{(i)}) = \begin{bmatrix} p(y^{(i)} = 1|x^{(i)};\theta) \\ p(y^{(i)} = 2|x^{(i)};\theta) \\ \vdots \\ p(y^{(i)} = k|x^{(i)};\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix}$$

The $\frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}}$ term ensures that the the elements that the hypothesis returns sum to 1 so that it is a valid probability distribution.

2.2 Aspect Oriented Programming

Aspect Oriented Programming [18] is a programming paradigm that allows the features of an application that are difficult to modularise within an object-oriented setting to be encapsulated within individual *aspects*. For example in Java the natural unit of modularity is a class, but there are features which cannot be contained to a single class such as security or logging. Instead the developer must add the logic for these features to methods within multiple classes, which can obfuscate the existing code and can lead to errors if the logic needs to be changed but is not updated everywhere it appears. Instead Aspect Oriented Programming allows the developer to encapsulate these features, each within a single aspect.

Underlying Aspect Oriented Programming is the notion of a *join point* which represents a well-defined point in the control flow of a program. This can range from the execution of a public method, to the throwing of an exception from a method within a specific class. Every aspect declares one or more *pointcuts* which are statements for describing join points. Aspects attach *advice* to pointcuts which contains the logic that should be executed whenever the join point is intercepted during execution.

2.2.1 AspectJ

AspectJ [17] is a tool hosted by Eclipse which weaves AOP advice into Java applications. It uses Java-like syntax to define pointcuts. It includes a tool which integrates with the Eclipse development environment so that the developer can dynamically view what advice is being applied at any point in their code.

Pointcuts

AspectJ pointcuts allow a variety of join points within the control flow of a Java program to be intercepted. The syntax of each pointcut describes the static context of the join point, including the type of operation (for example method call, field read etc.) as well as the join point's location (for example within class X or package Y). Pointcuts can also expose parameters containing the dynamic context of the join point. These dynamic parameters include the value of *this* - the currently executing object, *target* - the target object of the join point operation, and *args* whose meaning is specific to different join points. When the join point is intercepted at runtime in a static context (for example a static method call, execution of a static method, or accessing/assigning a static field), *this* will return null. Pointcuts are named so that they can be referred to at the beginning of a block of advice to declare which pointcut the advice is attached to.

The join points that we are interested in for this project include

1. method calls
   Pointcut pattern: call(*MethodPattern*)
   Static context: method modifiers, method return type, method name, method parameter types, types of exceptions thrown by this method
   Dynamic context: args - the values of the method parameters

```
pointcut somePointcut()
    : call(@annotation public *(..))
```

Listing 2.1: A pointcut named 'somePointcut' describing join points which are calls to any public method annotated with @annotation and which accept any number of method parameters of any type

2. method executions
   Static context: method modifiers, method return type, method name, method parameter types, types of exceptions thrown by this method

```
pointcut somePointcut(int i)
    : execution(!public static foo(int)) && args(i)
```

Listing 2.2: This pointcut describes join points which are executions of any non-public, static method named 'foo' which takes one parameter of type int. This pointcut exposes one dynamic parameter, *i*, which is dynamically bound to the value of the method parameter.

3. field-reads
   Static context: field name, modifiers, type, enclosing class

```
pointcut somePointcut()
    : get(int X.i)
```

Listing 2.3: This pointcut describes join points which are accesses to the field 'i' of type int, enclosed within class X.

4. field-writes

Static context: field name, modifiers, type, enclosing class

Dynamic context: args - value being assigned to the field

```
pointcut somePointcut()
    : set(set(static * *) && args(newValue)
```

Listing 2.4: This pointcut describes join points which are assignments to any static field. The new value of the field is dynamically bound to 'newValue'.

5. object creation

   Static context: modifiers, type, parameter types, type of exceptions thrown

```
pointcut somePointcut()
    : call(*.new(int, long))
```

Listing 2.5: This pointcut describes join points which are the invocation of the constructor of any object which takes two parameters of type int and long.

Limitations

The main limitation of AspectJ that we encountered in this project is that there are no pointcuts for intercepting index specific array operations. External developers have presented extensions to AspectJ to overcome this limitation [8] however official support for this feature was still under consideration at the time of writing [1].

2.3 Related work

2.3.1 Kernel level vs. user level speculation

Existing solutions in the field of speculative execution can broadly be classified into those that operate at the system-level and those at the user-level. User-level speculative systems transform the application's code array to add speculative capability, whereas system-level solutions rely on the operating system's kernel to control speculation. This provides several advantages because the operating system can

1. spawn a new process for speculation [11, 24, 33] which eliminates the need for ensuring that side effects by a speculative thread do not interfere with a non-speculative thread, because processes live within their own, unshared, memory space.
2. leverage system-wide performance data [11] to improve the scheduling of speculations.
3. enable inter-process communication between a speculative process and a non-speculative process by tracking causal dependencies [25].

4. eliminate the need for the code of each application to be transformed, improving the portability of the speculative system.

However the major disadvantage of system-level speculation compared to user-level speculation is that the operating system has no contextual knowledge about the control flow of the application and where speculation would make the most sense. Woster et al. [33] oppose this black and white classification and they argue that speculation policy, such as deciding what to predict, should be separated from the mechanism of speculation. By making this separation, the policy can be implemented at the user-level and the mechanism at the operating system level. They accomplish this by by creating a new group of system calls that allows a process to direct the speculation mechanism within the operating system. When a thread decides to speculate the code path that follows, it invokes one of these new system calls which triggers the operating system to create a speculative copy of the current process. This speculative copy begins executing at one method ahead of the non-speculative process. When the non-speculative process reaches the point in the control flow when the speculation started, and if there have been no data dependency conflicts, the speculative process is committed. The situation becomes more complicated when the application is multithreaded; now the application must indicate which threads are independent of the speculation and therefore should only run in one of the processes to minimise wasted computation. It was decided that these independent threads should run in the speculative copy because this is more likely to survive.

2.3.2 Speculative prefetching

Speculative prefetching executes code ahead of time in order to cache any data that may be required in a future state of the application. This reduces the latency if this data is eventually requested by a non-speculative thread. Existing work has looked at using the reference history of previous sessions to guide prefetching [20]. Speculative prefetching systems have the advantage that they are not concerned with preventing data conflicts between speculative and non-speculative threads as they are only concerned with buffering remote data rather than producing the result of any execution.

Sprint [28] predicts future accesses by executing a copy of the program in parallel with the original. This parallel copy executes ahead through speculative parallelisation, and shares a cache with the original program so that any remote files that are fetched will are available to both processes. Sprint provided a speedup of between 2.4x and 15.8x, however significant speedup is only noticeable in applications with a large number of read accesses.

SpecHint [7] also approaches the problem of reducing wasted cycles due to a process blocking on I/O. Rather than waste these cycles, it uses a separate thread to continue executing speculatively. SpecHint resulted in between a 29% and 90% reduction in execution time. Although it currently works with single threaded applications only, it could be extended to work with multithreaded applications where only a single thread is blocked and other threads are still executing normally. SpecHint uses a hint log to ensure that speculation is kept on track. When the speculative thread retrieves a file, it notes this in the log. The main thread then checks this log when it needs to retrieve a remote file. If the first entry does not match the file it requires, it resets the speculation thread and resumes.

Fraser and Chang [11] approached prefetching at the system-level by developing an in-kernel pre-fetcher. This allows them to reduce the amount of time spent on handling page faults through the speculative prefetching of pages which reside on disk. It also allows the non-speculative thread to know whether the speculative thread has gone off track by looking at whether a page frame has already been allocated when it makes a request for data. Therefore this has a smaller memory footprint than SpecHint which requires an extra data structure for its hint log.

2.3.3 Thread level speculation

Thread level speculation relies on the out-of-order execution of methods to improve performance. Rather than following the strictly sequential control flow of a sequence of method invocations, a group of sequential method invocations can be executed immediately in parallel. These parallel code regions only commit if they contain no illegal data dependencies. If there are any conflicts then the regions are re-executed sequentially starting from the earliest conflicting region. This is analogous to speculative execution found at the instruction level in out of order processors. This technique has been shown to result in speed ups of at least 1.7 times non-speculative performance [10, 34]. For sequential code regions that contain data dependencies, there has been research into return value prediction [12, 15] so that a code region can speculate using a predicted return value from its preceding method in the control flow.

Kelsey et al. [16] approach thread level speculation differently with their Fast Track technique. The idea is to execute the same code region twice and in parallel, one thread executes the region normally while the other executes speculatively. The speculative thread runs in a modified version of the code region which completes faster but in a potentially unsafe way that may lead to incorrect results, for example using memoisation. As soon as this speculative thread finishes it continues to the next code region where, again, two threads will execute normally and speculatively. When the original non-speculative thread completes the first code region it compares its result with the result of the speculative thread. If the results match then the speculative code region produced the correct result and it can be committed. The non-speculative track continues no further. However if the results do not match, then the speculative version produced an incorrect result and therefore any subsequent computation based on this incorrect result needs to be squashed. Figure 2.1 illustrates an example of this speculative control flow. However Fast Track has no buffering mechanism for I/O within the speculative track of a dual-code region and therefore it will need to block until the non-speculative thread can catch up. A useful technique that Fast Track employs that can be applied to any speculative system is that it limits the amount of memory assigned to the speculative track of the dual-code region. Therefore if this track produces an incorrect result, the extent of memory resources that have been wasted is limited.

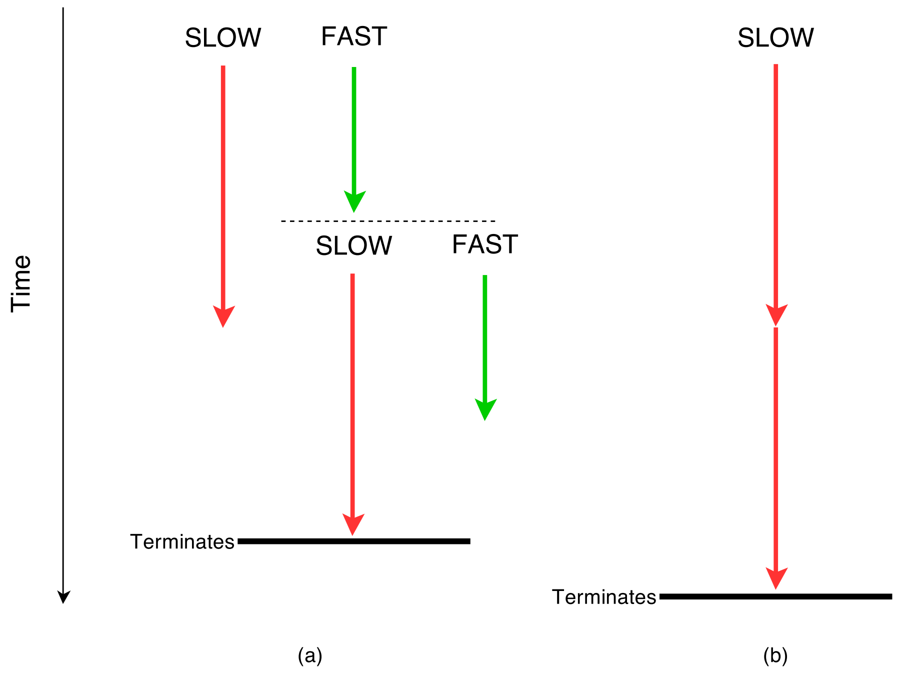

Figure 2.1: Fast Track. In *(a)* we see the execution of two dual-code regions, the dotted line represents a boundary between dual-code regions. At the start of the first dual code region, the safe but slower thread executes in parallel with the faster but potentially unsafe thread. As soon as the fast code region completes, this repeats for the second region. This terminates earlier than in *(b)* which shows the native control flow.

2.3.4 Speculating with interactive applications

Crom [22] is a library which adds speculative functionality to web-based Javascript applications. It is designed to be agnostic of the code that it works with so that it can be applied to a wide range of applications. However this generality means that the developer must manually constrain the speculation space by marking specific event handlers as speculative. A shadow copy is made of each of these handlers which is semantically equivalent but which updates a copy of the underlying DOM tree. This speculative copy is only committed when the current state of the browser is equivalent to the state at the start of the speculation. This equivalence can be defined by the developer so that the two states do not have to be compared bit-by-bit, but rather the developer can use his contextual knowledge of the application to determine if two states are semantically equivalent. Crom allows the developer to assign a speculative mutator function to each handler which defines what the likely outcome states are for the widget based on the current state so that these future states can be speculated. Unlike speculative systems which reduce the amount of speculation when the CPU is busy [11], Crom leaves this decision to the developer to implement.

2.3.5 Speculative development environments

Most integrated development environments highlight compilation errors to the developer as they type. Typically it will also suggest solutions, such as the Quick Fix box in Eclipse. Muşlu et al. [23] have developed a plugin, Quick Fix Scout, which speculatively applies each suggestion so that extra information can be provided, such as the new number of compilation errors if this suggestion is applied. This works by speculating on a hidden copy of the developer's code. It was shown that this leads to a 10% reduction in the average time to fix compilation errors.

2.3.6 Isolating side effects

Various techniques for isolating speculative side effects have been proposed. Historically there has been a focus on hardware solutions [14, 29, 31]. Software solutions usually centre around software-enforced copy-on-write. The area of memory being modified is copied and this is recorded in a data structure which is checked when the speculative thread wishes to load any value from memory. This can be an expensive policy, especially when there is a large number of speculative reads and writes. Chang et al. [7] propose directing speculation to run under a copy of the *.text* section of an executable, which is the section that contains the application code. Only the read and writes in this copy are wrapped within the copy-on-write logic, rather than the reads and writes of non-speculative threads which reduces the overhead for non-speculative threads. Cachopo et al. propose versioned boxes [6] as a technique for software transactional memory. Each object remembers the previous values of its fields so that they can be rolled back in the case of speculative failure.

Woster et al. [33] suggest that not all side effects need to be isolated. For example a conservative speculative system would consider that invoking a call to any third party could mutate some external state and therefore should not be allowed when executing speculatively. In reality this third party call may be a HTTP GET request which only reads rather than mutates any state. However this is not something that can be determined automatically, it requires the developer to manually mark such read-only operations. Lange et al. [21] suggest another reason that it may not always be necessary to suppress side effects; there is a threshold for the amount of inconsistency users are willing to tolerate as long as this is linked to improved performance.

2.3.7 Prioritising speculations

The speculative systems in [26, 28] assign a scheduling priority to speculations based on the amount of useful work the speculation is predicted to carry out.

2.3.8 Conflict detection

Yiapanis et al. [34] discuss the need for conflict detection between non-speculative and speculative threads. Specifically, before a speculative thread can commit we must verify there have been no read after write conflicts; that is all of the fields that the speculative thread has read have not been since modified by a non-speculative thread. Write after read and write after write violations do not need to be considered because speculative threads buffer any writes they make until they are committed. The authors present two types of conflict detection, lazy and eager. Eager conflict detection checks for conflicts on every speculative read in order to catch violations early compared to lazy conflict detection which only detects conflicts when a speculation is committed.

Not all speculative systems require conflict detection, for example speculative pre-fetchers can ignore conflicts because they do not care about the correctness of speculation.

Chapter 3

Overview

The speculative system we present in this project broadly proceeds in two phases. The first phase applies static transformations to the code at compile-time so that it can support speculation. The second phase concerns the runtime of the application when the speculative engine dynamically controls speculation. In this chapter we provide an overview of these two phases. We expand on the implementation detail in the next chapter.

3.1 Phase 1: Static transformation

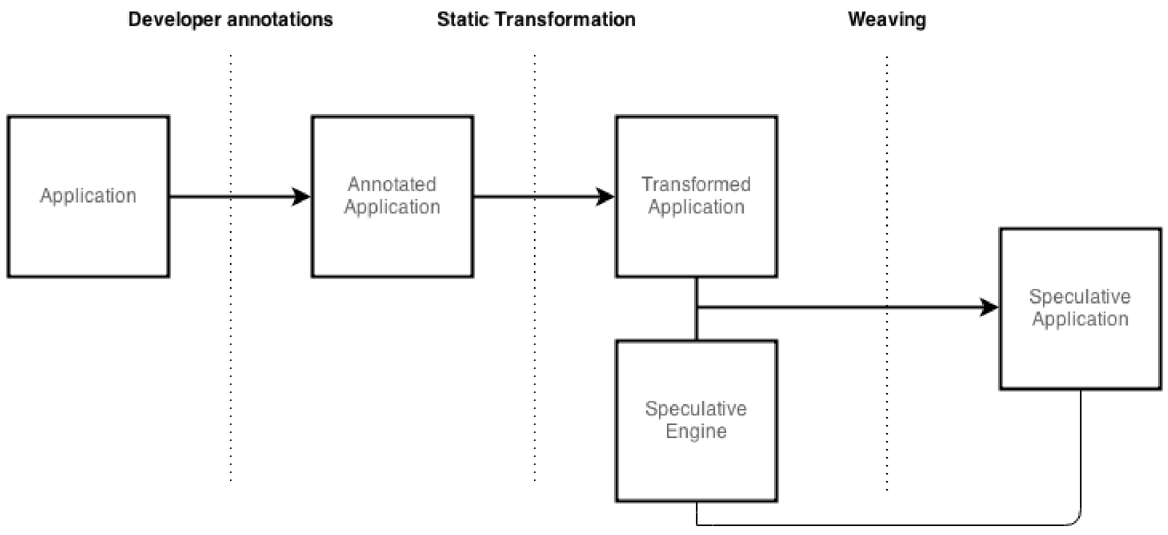

Figure 3.1: Transforming an application to support speculation. The developer manually annotates the code before automatic transformations are applied. Finally it is weaved with the aspects in the speculative engine and compiled.

3.1.1 Input application

The source code of the application must be available. The speculative engine is compatible with any Java application where the entry points to user actions are distinct methods. Figure 3.2 illustrates an example of code that is not valid.

```java
void actionPerformed() {
    if (...) {
        // Code for action A
    } else {
        // Code for action B
    }
}
```

```java
void actionPerformed() {
    if (...) {
        actionA();
    } else {
        actionB();
    }
}

void actionA() {
    // Code for action A
}

void actionB() {
    // Code for action B
}
```

Figure 3.2: The first code sample is not valid for speculation because the two user actions A and B do not have entry points in distinct methods. The second code sample shows the corresponding valid code structure.

3.1.2 Developer annotations

The application must indicate certain information about the structure of the code which the speculative engine can use to guide speculation. Specifically, the developer must classify certain methods by annotating them. We use Java annotations for this because they allow the application to remain fully backwards compatible with the standard Java compiler when the speculative engine is removed. The set of method classifications, and the annotations that represent them are,

1. *Classification*: **Entry method**
   Description: An entry method represents the start point of an action which can be speculated.
   Annotation: @SpeculableAction
   Example:

```
new ActionListener() {
    @SpeculableAction
    void actionPerformed(ActionEvent e) { ... }
}
```

2. *Classification*: **Externalised methods**
   Description: Externalised methods are those that make any native system call or communicate with a third party component. A speculative thread should pause if it reaches an externalised method until it is selected by a non-speculative thread.
   Annotation: @PauseSpeculation
   Example:

```
@PauseSpeculation
void foo() {
    // Native system call
    if (System.currentTimeMillis() == ...) {
        ...
    }
}
```

3. *Classification*: **Ghost methods**
   Description: Ghost methods are a subset of externalised methods whose logic does not contribute to the safety or correctness of the program. Speculative threads can skip ghost methods and can continue on to execute potentially expensive methods that follow.
   Annotation: @SkipWhenSpeculating
   Example:

```
@SkipWhenSpeculating
void showMouseCursorAsBusy() {
    ...
}
```

4. *Classification*: **Trivial**
   Description: Trivial methods are those that contain no nested method calls or loops. When trivial methods are invoked from inside a loop they are not checkpointed as part of the call stack technique we present in section 3.2.4.
   Annotation: @Trivial
   Example:

```
void foo() {
    for (int i = .....) {
        ...
    }
}

@Trivial
int bar(int a) {
    return a << 2; // Some trivial operation
}
```

3.1.3 Static transformation

After the developer has manually annotated the code we apply certain code transformations automatically.

- **Array access/assignment refactoring**
  AspectJ cannot intercept index-specific array operations and therefore these are refactored to use a proxy method which can be intercepted.

```
class A {
    void foo(int[] a) {
        a[1] = a[0];
    }
}
```

Listing 3.1: Before array operation refactoring

```
class A {
    void foo(int[] a) {
        Array.set(a, 1, Array.get(a, 0));
    }
}
```

Listing 3.2: After array operation refactoring

- **Annotating trivial methods**
  Previously we introduced the notion of trivial methods, which are methods that do not contain nested method calls or loops. We automatically annotate these methods to prevent the developer having to manually inspect each method.

- **Loop checkpointing**
  The control flow of loops cannot be intercepted by AspectJ which prevents any loop-level

checkpointing which we present in section 3.2.4. To overcome this we extend the code around loops to inform the speculative engine when the loop is about to begin, before every iteration and after the loop completes.

```
void foo()
{
 for (int i = 0; i < N; ++i)
 {
     ... // Loop body
 }
}
```

Listing 3.3: Before loop transformation

```
void foo()
{
 int i = 0;
 loop_start();
 for (; i < N; ++i)
 {
     loop_iteration();
     ... // Loop body
 }
 loop_finished();
}
```

Listing 3.4: After loop transformation

3.1.4 Weaving the speculative engine into the application

The final stage is to weave the advice contained in the speculative engine into the application. We use the AspectJ compiler for this. Figure 3.3 illustrates the aspects contained in the speculative engine. As far as possible each aspect delegates to its helper class. This reduces the amount of code injected at each join point, significantly reducing the weaving time.

The speculative engine comprises the following components:

1. Speculation Manager
   The Speculation Manager controls the state of speculative threads at runtime.

2. Speculation Generators
   Speculation generator instances decide which actions should be speculated.

3. Side Effect Manager
   The Side Effect Manager isolates speculative side effects so that they are not visible outside of the speculative thread.

4. Scheduler
   The Scheduler prioritises speculations and assigns queued speculations to threads as they become available.

3.2 Phase two: The speculative engine, managing speculation at runtime

3.2.1 Lifetime of a non-speculative thread

In a non-speculative system a thread will invoke a method call, wait for the result and continue. However in a speculatively enhanced application, a call by a non-speculative thread to an entry method[1] is always diverted to the Speculation Manager which returns the result. Figure 3.4 illustrates how, in this way, the Speculation Manager acts as a black box; the non-speculative thread is unaware how the result is retrieved, whether by speculation or normal execution. We considered an alternative to this where the non-speculative thread queries the Speculation Manager to see if a speculative result is available, and if not it would invoke the method directly itself. However this logic would need to appear whenever an entry method is invoked which would increase the size of the application.

3.2.2 Lifetime of a speculative thread

The control flow of a speculative thread follows that of its non-speculative counterpart except for the following operations, illustrated in figs. 3.5 to 3.8,

- **accessing/assigning fields**
  The Side Effect Manager acts as a black box for speculative field read and write operations. For read operations it will return a speculative value if one exists or the actual value otherwise, and for write operations it will save the value for future speculative reads to the same address. The times of these operations are also recorded in order to detect data dependency violations later.

- **constructing objects**
  When a speculative thread constructs a new object or array, and therefore requires heap space, it it will wait until the Scheduler agrees to this request. This is to prevent too much memory being allocated to speculation which can impact the performance of non-speculative threads.

- **invoking externalised methods**[2]
  When a speculative thread invokes an externalised method it will be suspended until a non-speculative thread decides to commit this speculative path. It will then resume execution non-speculatively in the same thread. After it has finished it notifies any non-speculative thread that is waiting for the result that it is available.

[1]In section 3.1.2 we defined an entry method as the start point of an action which can be speculated.
[2]In section 3.1.2 we defined an externalised method as one that contains side effects that cannot be isolated.

- **invoking ghost methods**[3]
  A speculative thread will return immediately when invoking a ghost method.

Figure 3.9 shows the states that a speculative thread moves through during its lifetime. In summary, these states are

- **Live**
  The thread is executing speculatively.
- **Paused**
  The thread has reached an externalised method and is suspended until it is selected by a non-speculative thread.
- **OutOfMemory**
  The thread is waiting until its request for more heap space is accepted by the Scheduler.
- **PeriodicVerify**
  The thread periodically enters this mode to verify that none of the fields it has read since the last periodic verification have been written to by a non-speculative thread.
- **CommitVerify**
  Before a speculation is committed it first enters this state to verify that *all* of the fields it has read during speculation are not out of date.
- **Commit**
  The speculative thread is making all of its side effects visible to other threads. Non-speculative writes are locked when any speculative thread is in this state.
- **Finished**
  The speculative thread has finished speculating and is waiting to be committed by a non-speculative thread.
- **Non-Speculative**
  The speculative thread was committed successfully before the end of the action's control flow was reached. It therefore finishes execution non-speculatively.
- **Terminated** The speculative thread has committed successfully, finished execution, and has returned its result to the non-speculative thread.

[3]In section 3.1.2 we defined a ghost method as one that can be skipped during speculation.

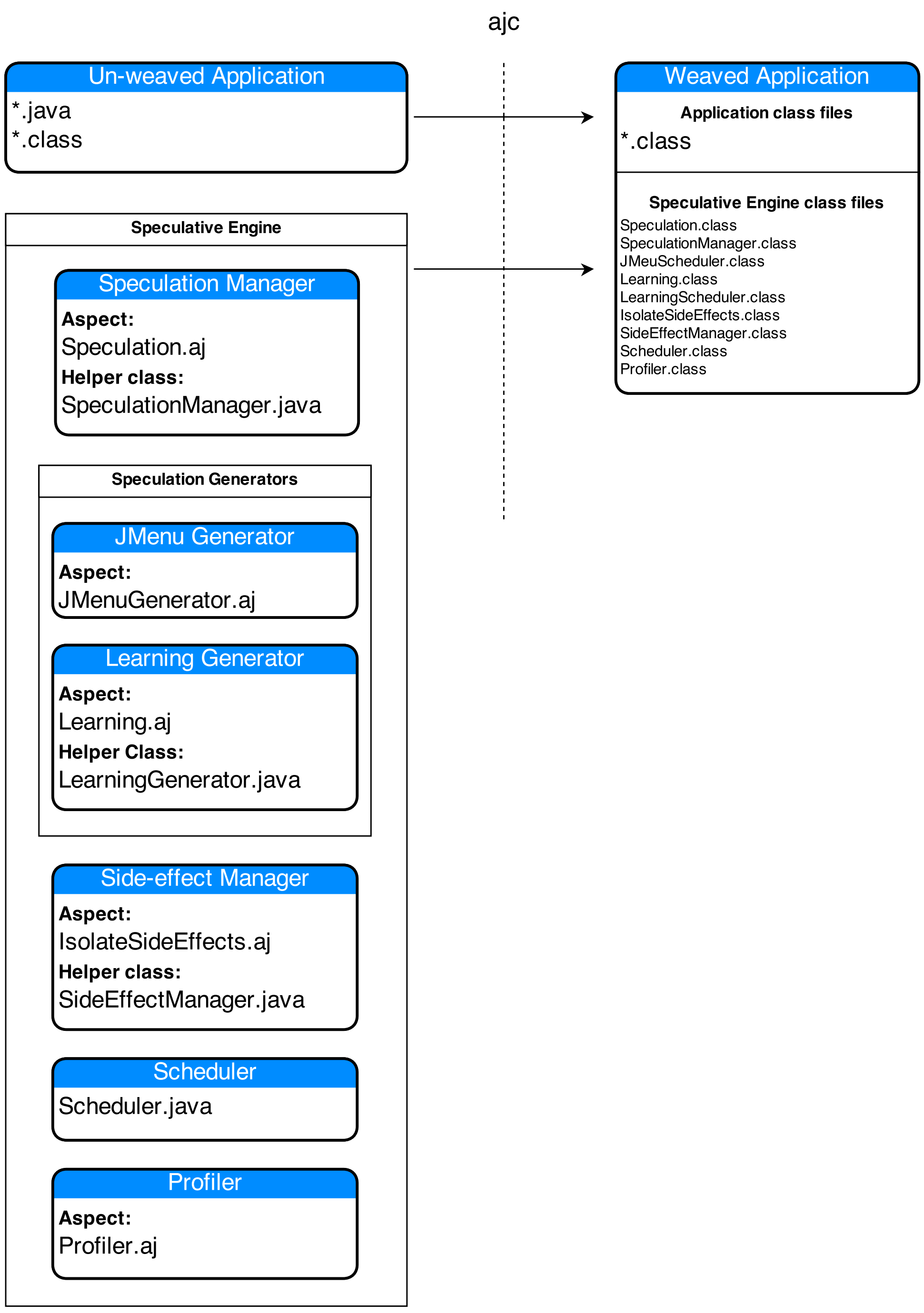

Figure 3.3: The components of the speculative system. The AspectJ compiler, ajc, weaves the advice contained in the speculative engine into the Java application.

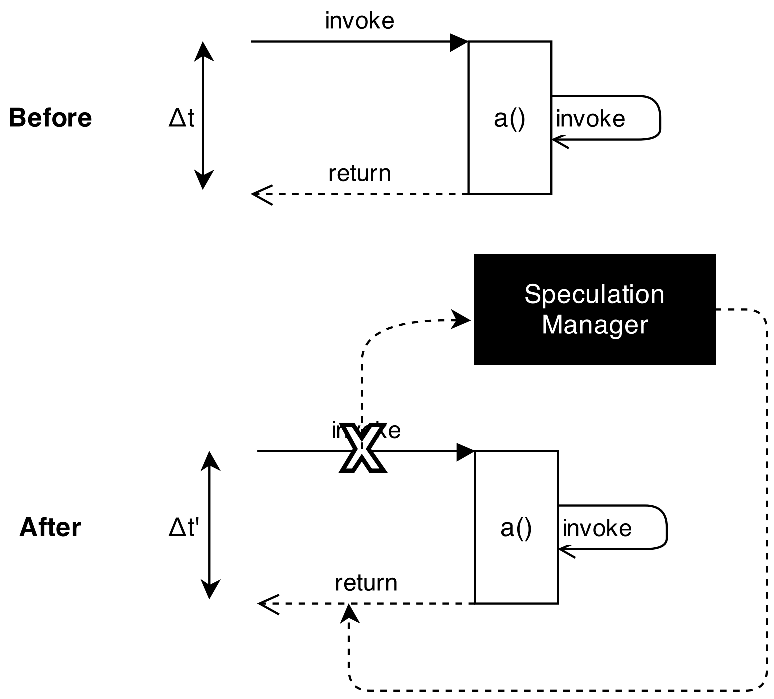

Figure 3.4: How entry methods are invoked before and after the speculative engine is weaved in to the application. Before weaving, invoking an entry method *a()* will synchronously execute the code of *a()*. After weaving, the invocation is redirected to synchronously request the result from the Speculation Manager.

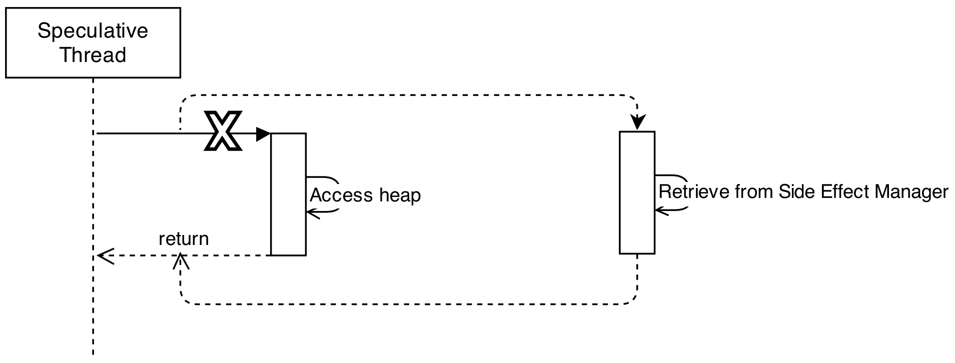

Figure 3.5: Field read and write operations are diverted via the Side Effect Manager so that a speculative value can be retrieved/saved.

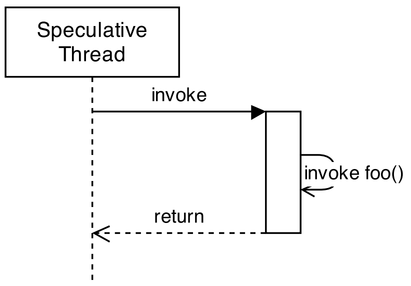

Figure 3.6: Unannotated method invocations are executed normally.

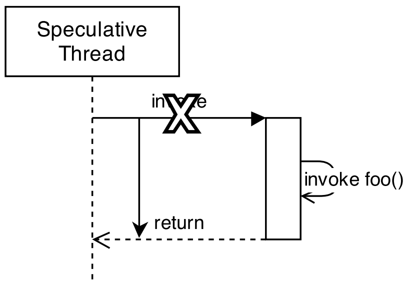

Figure 3.7: Invocations to @SkipWhenSpeculatingannotated methods return immediately.

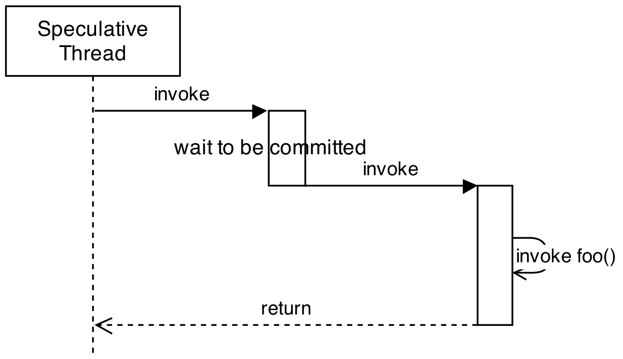

Figure 3.8: Invocations to @PauseSpeculation annotated methods cause the speculative thread to sleep until it is selected by a non-speculative thread. It will then initiate the commit procedure and finish executing non-speculatively.

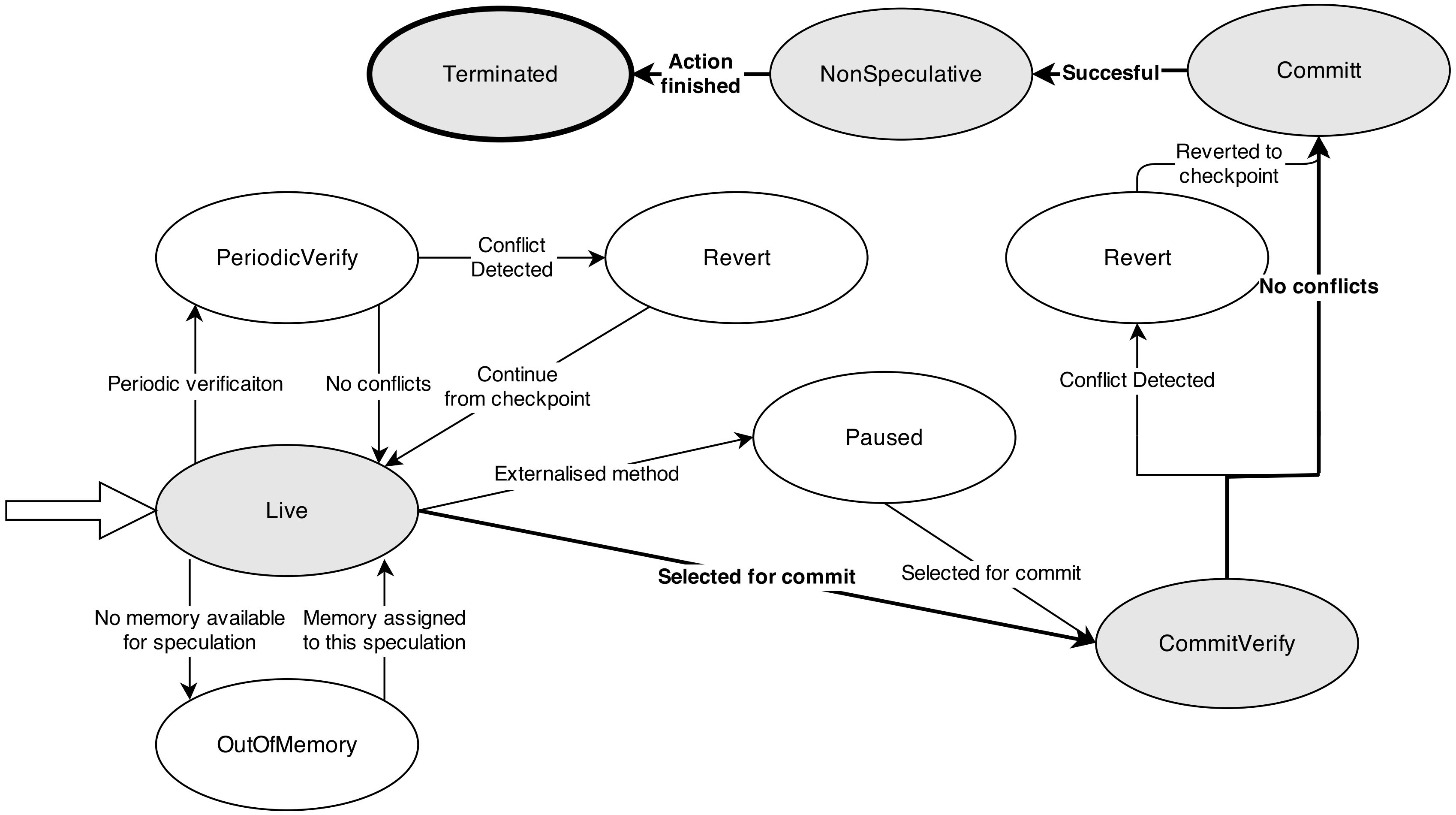

Figure 3.9: State transition diagram for a speculative thread. The shaded states and bold transitions indicate the standard lifecycle of a speculative thread.

3.2.3 Detecting conflicts

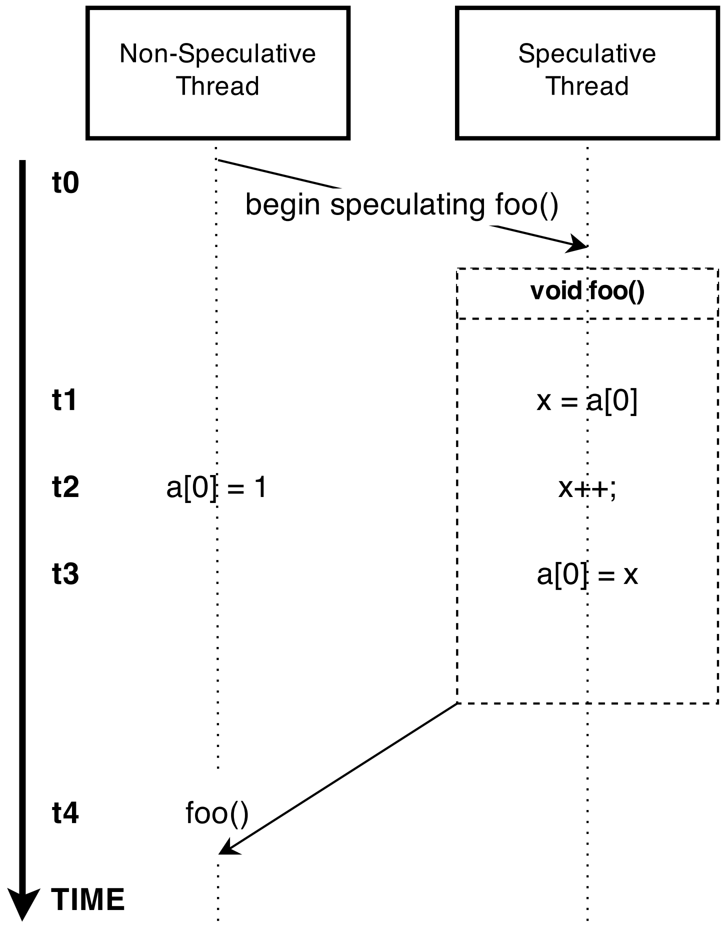

Figure 3.10: An example of a read after conflict occurring between a non-speculative and speculative thread. At t0 the speculative thread begins speculatively executing *foo()*, reading and writing to the address represented by a[0] at t1 and t3 respectively. Between these read and write operations, at t2, the non-speculative thread overwrites the value of this field. Therefore when the speculative result is requested by the non-speculative thread at t4, the speculative thread cannot commit its speculative value of a[0] because its initial read at t1 is now outdated.

Before a speculative thread can be committed we must verify that, for every field read during speculation, none have been written to by a non-speculative thread after the time of the *first* speculative read. Figure 3.10 illustrates an example of a read after write conflict. The speculative engine we present in this project has two modes for verifying that a speculative thread contains no conflicts,

1. Periodic verification
   Periodic verification occurs regularly during the lifetime of a speculative thread. It iterates over the non-speculative writes that have occurred since the last periodic verification, and

for each it looks at the time it first read this field. If the read time is earlier than the write time, a read after write conflict has occurred. Non-speculative writes are not locked during periodic verification so that non-speculative threads are not negatively impacted by speculation. However this does allow race conditions to occur between the verification procedure and a non-speculative write operation, therefore periodic verification is only used as an early-notification technique for conflicts, rather than a complete and accurate check.

2. Commit verification
   Commit verification occurs when a speculative thread is committed. Non-speculative writes are locked to avoid race conditions. We verify *every* read that has occurred since the start of speculation.

3.2.4 Checkpointing

When a conflict is discovered a speculative thread must restart from at least the time when the field was *first* speculatively read. In a system without checkpointing, a speculative thread must completely restart speculation, however this means that any execution until the time of the conflict read is wasted. To overcome this we consider two techniques for checkpointing speculations so that they can restart from fixed points during speculative execution, as illustrated in fig. 3.11. The checkpointing techniques that we consider are:

- Call stack checkpointing
  We recursively move up the call stack to return the control flow to the execution frame before the conflicting read occurred. We also revert the speculative write set to its state at this point. For example fig. 3.12 shows the call stack after reaching line 11 in listing 3.5 during the execution of *foo()*. If a conflict is discovered at this point the speculative thread has two checkpoints it can restart execution from; line 7 in *b()* or line 3 in *foo()*.

- Loop-iteration checkpointing
  We periodically checkpoint the state of speculation during the execution of loops. Therefore if a conflicting read is discovered to have occurred during or after a long running loop then the speculative thread can skip iterations when it revisits the loop after it has restarted.

3.2.5 Scheduling speculations

The Scheduler component, illustrated in fig. 3.13, receives speculation decisions from SpeculationGenerator instances. Each decision comprises information about the action that should be speculated as well as a likelihood that the user will select this action. The Scheduler will decide a priority for each speculation as a combination of this likelihood and a measure of the complexity of the action relative to the other speculation decisions it receives. This complexity can be determined arbitrarily, for example based on the average execution time. The Scheduler queues speculations in priority order and assigns each to a speculative thread when one is available.

The Scheduler also responds to memory requests from speculative threads. A fixed amount of memory is reserved for the Scheduler to allocate to individual speculations. The Scheduler will always agree to memory requests until the limit would be exceeded at which time any speculations with a lower priority than the requesting speculation will be cancelled until enough memory can be freed.

3.2.6 Generating speculations

The speculative engine supports concurrent SpeculationGenerator instances which are responsible for deciding what and when to speculate. This project includes two implementations, each using a different technique for generating speculations within an interactive Java application.

1. Menu-driven speculation
   When the user interacts with the application's menu system we speculate all actions that are contained within the currently selected menu. It requires that the menu stays selected for at least 500ms before beginning speculation in case the user drags their mouse across multiple menus quickly.

2. Learning user behaviour
   The idea here is to use a historical window of the most recently selected actions to predict which action the user will select next. This prediction is based on the order in which the user has historically selected actions. For example it may learn for Pixelitor that the first action when a user loads an image is to apply filter A, or that the user often applies filter C after applying filter B. The advantage of using multinomial logistic regression for this is that it produces a probability for each action that it will be the next action to be selected. We can directly use these probabilities as likelihoods when we forward decisions to the Scheduler.

   In order for the classification to be as accurate as possible it is important that it learns in realtime as the user interacts with the application. This presents a problem however because logistic regression is not an algorithm which works with streaming data; the classifier must be rebuilt every time there is a new data sample. This motivates our evaluation of the cost of building the classifier in section 5.7.1.

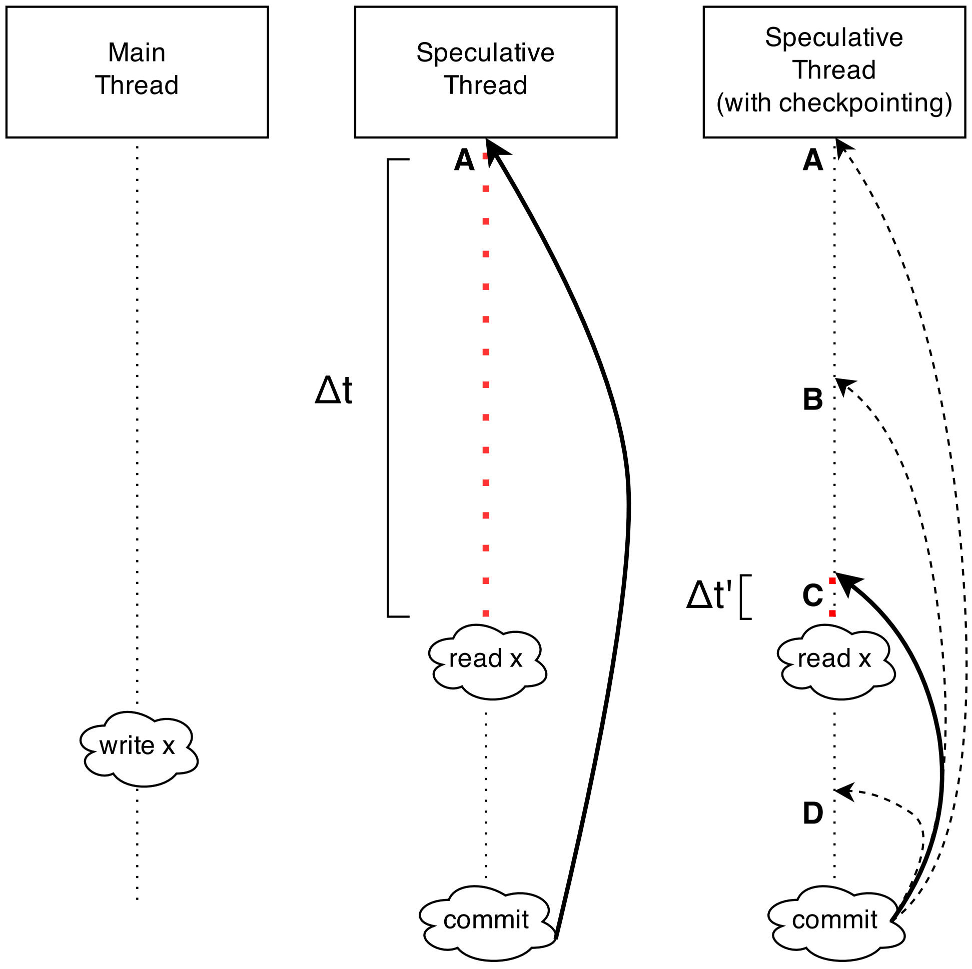

Figure 3.11: Checkpointing a speculative thread. We consider two speculative threads, the first contains only one checkpoint at the start of speculation, the other contains 4 periodic checkpoints (A-D). There is a read after conflict due to the main thread writing to a variable that has been read by the speculative thread. If we assume that this conflict is only detected at commit, then in the single-checkpoint thread we must restart speculation from the beginning, losing Δt worth of useful speculation, compared to the checkpointing thread where we can revert to checkpoint C, losing a much smaller amount of useful speculation $\Delta t'$. We cannot revert to checkpoint D because this is still after the conflicting read occurred.

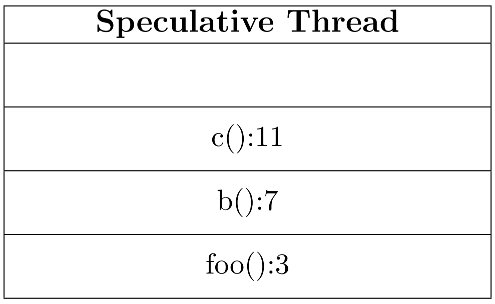

Figure 3.12: The state of the call stack after reaching line 11 of *c()* when executing *foo()*. Call stack checkpointing allows a speculative thread to recursively move up the call stack to return to an execution frame before the conflicting occurred.

```
1  void foo() {
2      a();
3      b();
4  }
5
6  void b() {
7      c();
8  }
9
10 void c() {
11     // Conflict discovered
           here
12 }
```

Listing 3.5: An example of a nested code structure which supports call stack checkpointing.

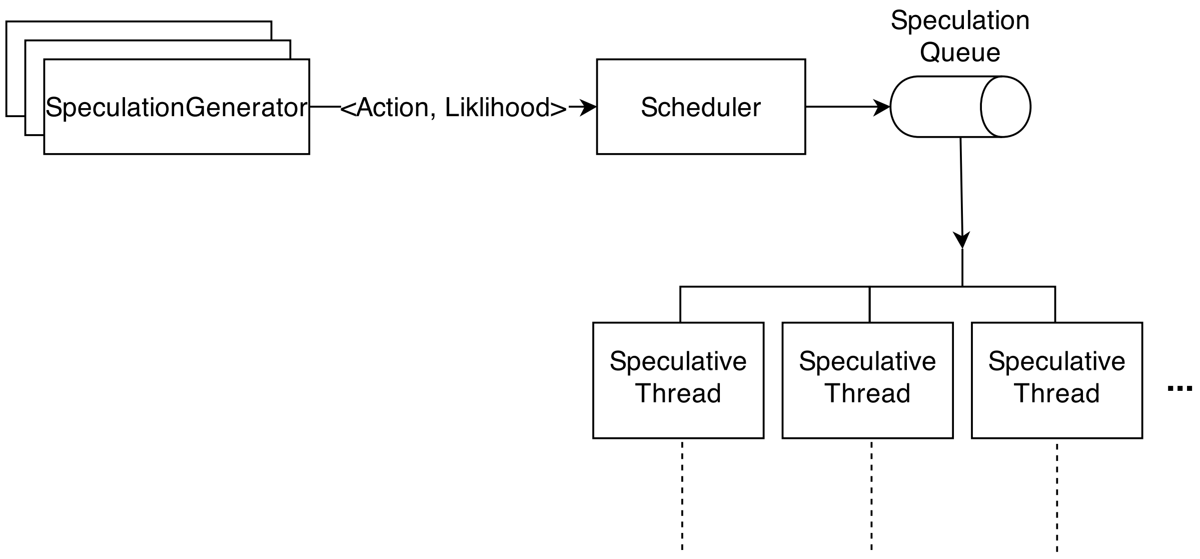

Figure 3.13

Chapter 4

Implementation

4.1 Maintaing speculative state

The SpeculationManager holds a set of *ThreadLocal* fields which are different to standard Java fields in that whenever a thread reads or writes to the field, they are doing so to their own version. These fields are

1. $isSpeculative$: boolean
   This field tracks whether the thread is executing in speculative or non-speculative mode. It is used in many locations, for example by the Side Effect Manager on every field operation to decide whether to read or write on the heap or on the speculative write set.

2. $speculationID$: long
   Every speculation is assigned a unique ID.

3. $target$: SpeculationTarget
   This encapsulates the signature (class, method name) of the entry method[1] of this speculation.

4. $lastRAWVerificationTime$: long
   This records the start time of the last periodic read after write verification. Therefore at the next periodic verification we know that we do not have to consider non-speculative writes that occurred before this time.

5. $startTime$: long
   This records the start time of this speculation. It is reset when a speculation reverts to a checkpoint. It is used during a commit verification so that non-speculative writes before this time are not considered.

[1]In section 3.1.2 we defined an entry method as the start point of an action which can be speculated.

6. *resultHandle* : SpeculationResult
   This *SpeculationResult* instance contains objects used as monitors to synchronise the speculative thread with a non-speculative thread. For example after a non-speculative thread requests the speculative thread's result it will wait on one of these monitors which the speculative thread can notify when it has terminated.

4.2 Retrieving speculative results

In section 3.2.1 we illustrated that the Speculation Manager acts as a black box for retrieving the result of entry methods. The *Speculation* aspect places advice *around* any executions of methods annotated with @SpeculableAction. It was important to use an *execution* pointcut rather than a *call* pointcut, otherwise classes outside the scope of the weaver which contain calls to speculable actions would not be advised.

```
pointcut speculativeActionExecution() :
  execution(@SpeculableAction * *(..))
```

Listing 4.1: The AspectJ pointcut for intercepting executions of *@SpeculableAction* methods

The Scheduler is asked whether the method has been speculated. If it has, it will return a *SpeculationResult* instance which can be used to query the state of the speculation.

- Requesting live speculations
  If the speculation is still live, i.e. speculating, the Speculation Manager will set the 'selected' flag on the *SpeculationResult* instance. When the speculative thread periodically polls this flag, it will see that it has been selected and it will initiate its commit procedure. The non-speculative thread will sleep until the speculative thread notifies it that it has terminated.

- Requesting suspended speculations
  If the speculative thread is suspended, the Speculation Manager uses the *SpeculationResult* instance to notify it to wake up, commit its progress so far and continue executing non-speculatively. The non-speculative thread will sleep until the speculative thread notifies it that it has terminated.

- Requesting finished speculations
  If the speculative thread is sleeping because it has finished, the Speculation Manager will use the *SpeculationResult* instance to notify it to wake up, commit and terminate. The non-speculative thread will sleep until the speculative thread notifies it that it has terminated.

When the non-speculative thread is notified that the speculative thread has terminated, it uses the *SpeculationResult* instance to access the *Future* object which encapsulates the result that the speculation's entry method returned. The non-speculative thread calls *get()* on this object to retrieve the result.

If the Scheduler replies that there is no speculation for the requested action, the Speculation Manager will synchronously invoke the action and return the result.

4.2.1 Rethrowing speculative exceptions

When the non-speculative thread calls *get()* on the *Future* object to retrieve the result, an *ExecutionException* will be thrown if an exception was thrown during the speculation's execution. For example if an *IOException* was thrown during speculative execution this will have been suppressed until now and rethrown as an *ExecutionException.* This is a problem because the caller of the action is expecting to catch exceptions of a specific type, not *ExecutionException.* Therefore we access the *throwable* at the root of the *ExecutionException* and rethrow this.

4.2.2 Example

Figure 4.1 uses a sequence diagram to illustrate the control flow when a non-speculative thread requests the result of two entry methods *a()* and *b().* The diagram omits the actions and components involved in isolating side effecs. To begin, a speculation generator instance decides that an action represented by entry method *a()* should be speculated, and this decision is passed to the Scheduler along with a likelihood that the user will select this action. This communication between the generator instance and Scheduler is asynchronous allowing the generator to immediately return. The Scheduler will decide a priority for this speculation based on the likelihood. When the Scheduler detects that there is a free speculative thread, it will dequeue the highest priority speculation and assign it to the free thread. At some point before, during or after this, the main thread invokes *a()* and this invocation is intercepted by the Speculation aspect. The aspect requests the result from the Speculation Manager which in turn asks the Scheduler whether a speculative result is available. In this example the Scheduler replies positively and returns a result handle. The Speculation Manager uses this handle to notify the speculative thread to commit and continue executing normally. The main thread then sleeps until this is complete. When the speculative thread executing *a()* finishes, it notifies the main thread which wakes up and continues. The main thread then invokes *b()* for which there was no speculation. The Scheduler will respond negatively and therefore the Speculation Manager will synchronously invoke *b()* itself, return the result and allow the thread to continue with execution.

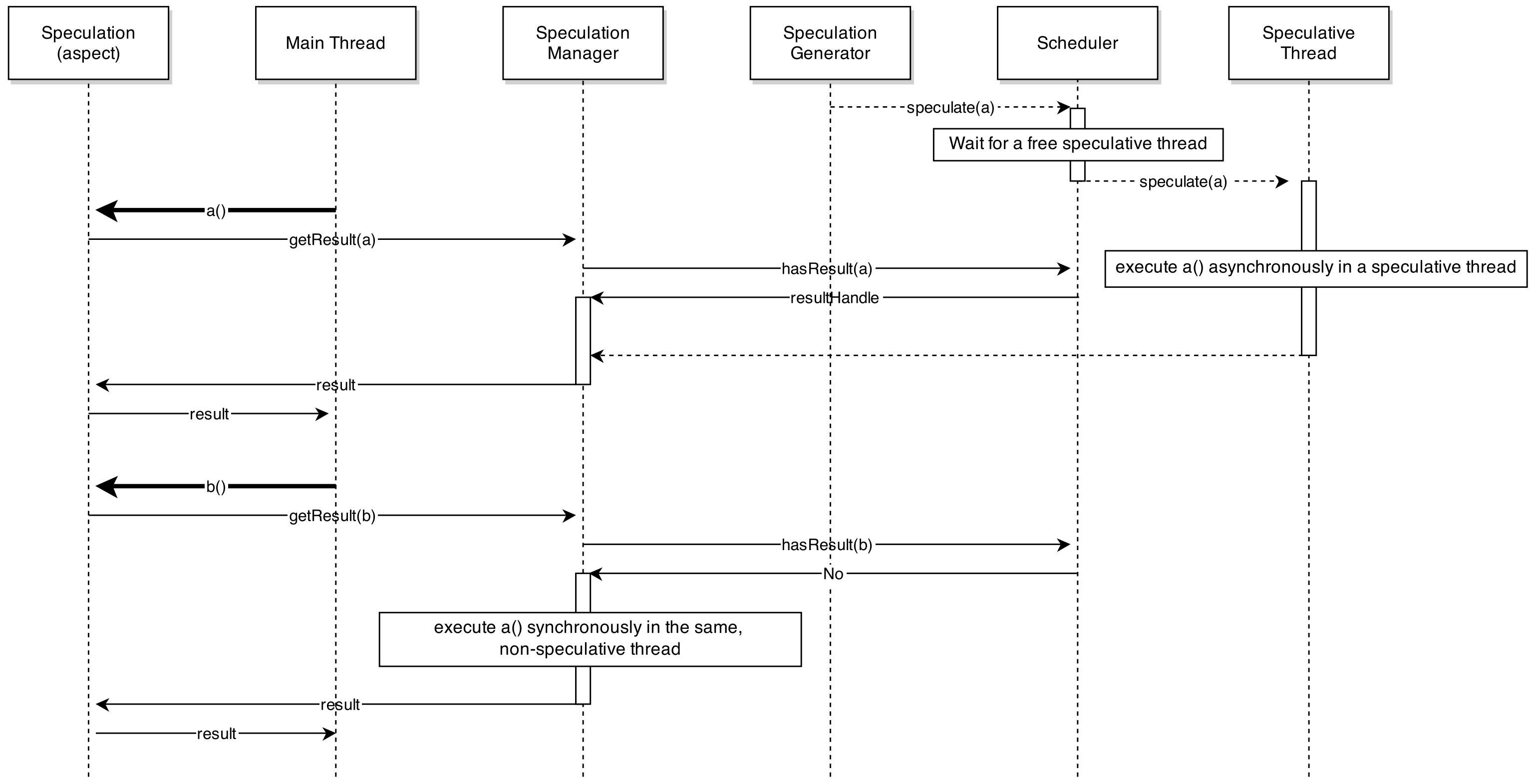

Figure 4.1: Sequence diagram illustrating the speculative control flow of two @SpeculableAction methods, *a()* and *b()*. The thicker arrows denote the pointcut interceptions when a @SpeculableAction method is called and the dashed arrows denote asynchronous method calls.

4.3 Isolating side effects

In Java the heap acts as the shared memory between threads. Therefore we consider any byte code operation which mutates the heap to be a speculative side effect which should be isolated. These operations are:

1. **putfield [objectref] [value]**
   Set the value of a non-static field in the heap
2. **putstatic [value]**
   Set the value of a static field in the heap
3. **new**
   Create a new object on the heap

We use AspectJ to intercept these operations in order to isolate their effects from the heap. We also intercept any operations that *read* from the heap in order to direct the thread to use the speculative value if one exists. These read operations are:

1. **getfield [objectref]**
   Read the value of a non-static object-field in the heap
2. **getstatic [objectref]**
   Read the value of a static object-field in the heap

4.4 Data structures

The speculative engine uses the abstract *FieldTarget* class illustrated in fig. 4.2 to represent the target of a field operation. *NonStaticFieldTarget* is used when the field is not static, and stores a reference to the object the field belongs to. *StaticFieldTarget* is used when the field is static, and stores the name of the class the field belongs to. *FieldTargetOperation* is used to represent a field operation by encapsulating the target field and the time the operation occurred.

The data structures we are about to present store *FieldTarget* objects as keys, therefore we override the *hashcode()* method for *StaticFieldTarget* and *NonStaticFieldTarget.* The hash code of *NonStaticFieldTarget* uses the identity hash code of the object reference it stores (a unique value assigned by the system for every object), rather than a potentially overridden *hashcode()* value. If we did not do this, two semantically equivalent objects, but with different memory locations, would share the same side effects which is undesirable.

The Side Effect Manager relies on four underlying data structures,

1. **Storing speculative values**

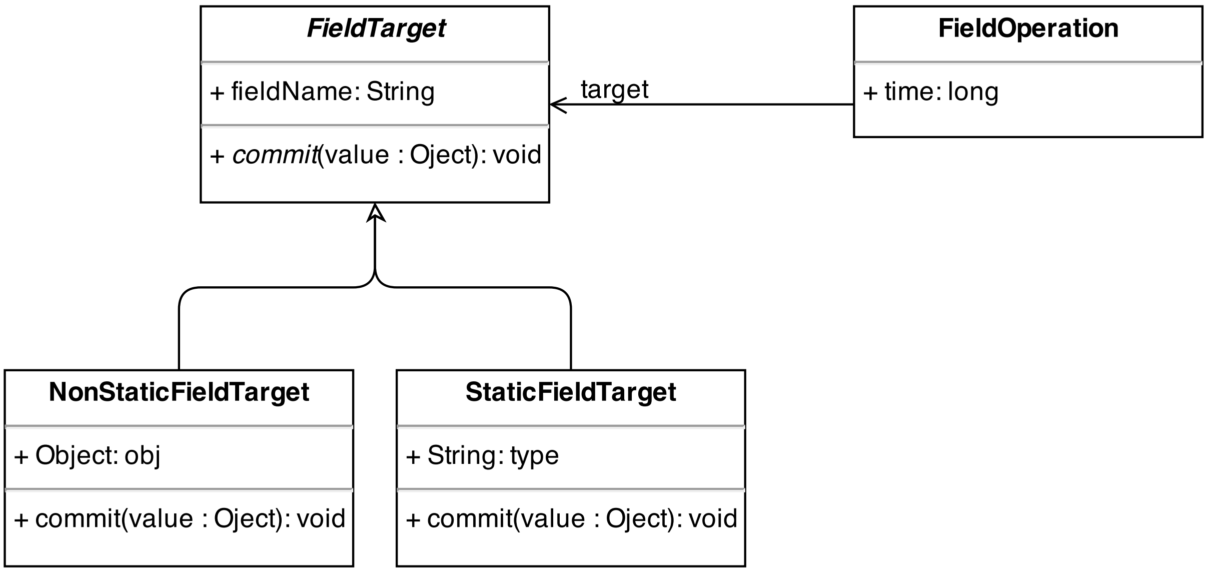

Figure 4.2

```
// Speculation ID ↦ [Field ↦ Speculative Value]
Map<Long, Map<FieldTarget, Object>> sideEffects;
```

This map holds speculative field values using a ⟨*Speculation ID, Field*⟩ key, returning an Object value. We use an underlying *ConcurrentHashMap* implementation, therefore it is thread-safe with multiple speculative threads.

2. **Recording non-speculative writes**
   We record the times of non-speculative writes in two structures, one keeps an ordered list of writes, the other stores only the time of the latest write for each field. These are used during the verification procedures of speculative threads. We do not synchronise access to these two structures because this would introduce an overhead for every non-speculative write. Instead we rely on the application developer to have prevented race conditions over concurrent writes to the same field. We assume that this would be case with any safe multithreaded application.
   (a)
   ```
   List<FieldOperation> nonSpeculativeWriteList;
   ```
   This list records every non-speculative write operation in chronological order.
   (b)
   ```
   // Field ↦ Time
   Map<FieldTarget, Long> nonSpeculativeWrites;
   ```
   This map records the latest write time for any field. It is filled lazily and therefore any field that is not in this map has not been written to by a non-speculative thread.

3. **Recording speculative reads**

```
// Speculation ID ↦ [Field ↦ Time]
Map<Long, Map<FieldTarget, Long>> speculativeReads;
```

This map records the *earliest* time that each speculative thread read a particular field from the heap. It is filled lazily and therefore any field that is not in this map has not been read. This map uses an underlying *ConcurrentHashMap* implementation therefore it is thread-safe with multiple speculative threads.

4.4.1 Intercepting field writes

We use two pointcuts for intercepting writes to static and non-static fields.

```
void around(Object newValue) :
    set(static * *) && args(newValue)
{
    // Field write logic
}
```

Listing 4.2: AspectJ pointcut to intercept static field assignments

```
void around(Object target, Object newValue) :
    set(* *) && target(target) && args(newValue)
{
    // Field write logic
}
```

Listing 4.3: AspectJ pointcut to intercept non-static field assignments

Figure 4.3 illustrates the procedure when a field write is intercepted. We first look up whether the current thread is executing speculatively. If it is, we record the speculative value and the current time in *sideEffects*. If it is executing non-speculatively we wait to obtain the write lock to avoid a race condition with a speculation that is being committed. We then record the time of the non-speculative write in the *nonSpeculativeWriteList* and *nonSpeculativeWrites* structures, update the value on the heap and release the write lock.

We use the *ReentrantReadWriteLock* class to allow multiple non-speculative threads to simultaneously access the write lock so that there is no overhead if concurrent non-speculative threads want to carry out field writes. This is allowed because of our earlier assumption that non-speculative write race conditions are handled by the application's code. However when a speculative thread gains access to the write lock at commit time, it does so exclusively to ensure that no read after write conflict can occur as it is committing.

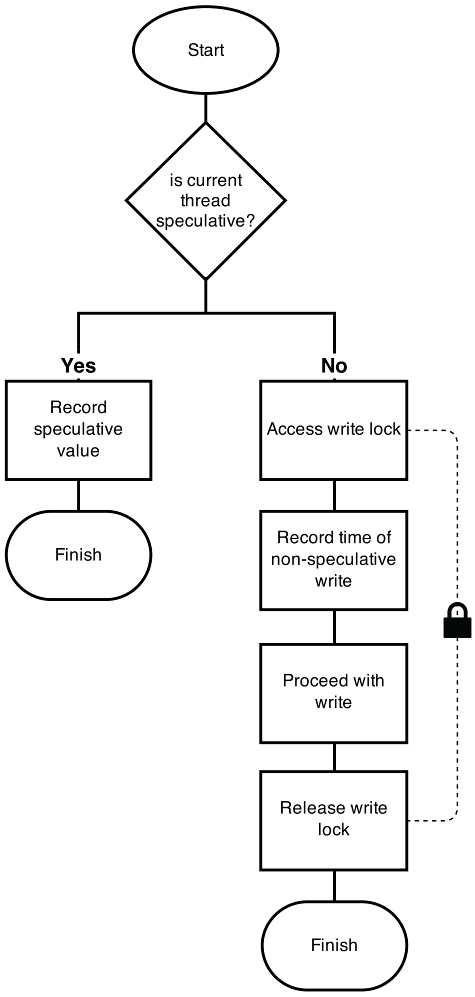

Figure 4.3

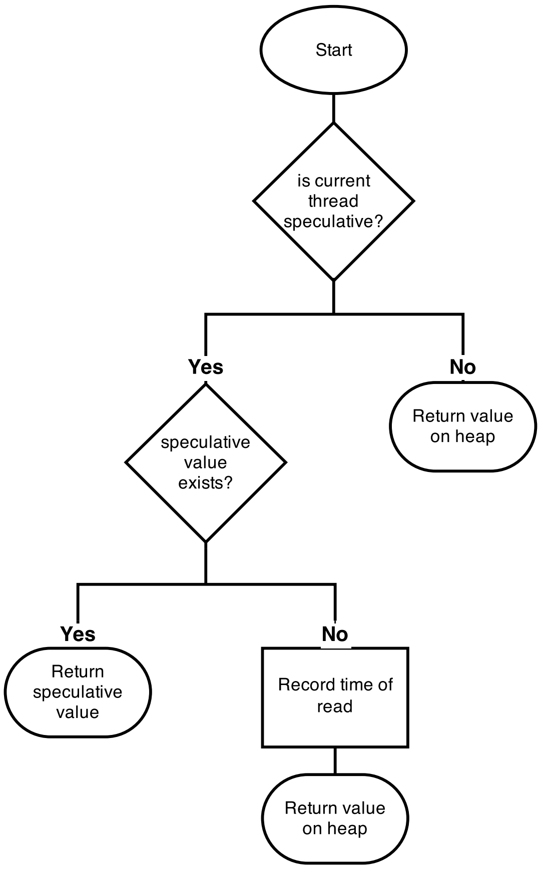

Figure 4.4

In a single-threaded application we can informally argue that the main thread will never have to wait to access the write lock. Let us assume that the non-speculative thread $T1$ is required to wait to access the write lock. Therefore a speculative thread, $T2$, must have gained exclusive access to this lock during its commit procedure. For this to be the case, a non-speculative thread must have requested $T2$'s result. $T1$ is the only non-speculative thread that could have requested this. We have a contradiction because $T1$ is executing a field write operation rather than waiting for a speculative result, therefore our assumption that $T1$ waits is wrong. In multithreaded applications we cannot argue that a non-speculative thread will never have to wait because it may wait on a speculative commit requested by another non-speculative thread.

4.4.2 Intercepting field reads

We use two pointcuts for intercepting accesses to static and non-static fields.

```
Object around() : get(static * *)
```

Listing 4.4: AspectJ pointcut for intercepting static field accesses

```
Object around() : get(* *)
```

Listing 4.5: AspectJ pointcut for intercepting non-static field accesses

Figure 4.4 illustrates the procedure when a field read is intercepted. We first look up whether the current thread is executing speculatively. If it is we retrieve a speculative value if one exists or the heap value otherwise, recording the current time if this is the first read of this field.

4.4.3 Verifying speculative threads

Before a speculation is committed we must verify that there have been no read after write conflicts. In section 3.2.3 we presented two modes of verification. Periodic verification occurs periodically and does not lock non-speculative writes whereas commit verification occurs when a speculation commits and does lock non-speculative writes.

1. Periodic verification
   Periodic verification proceeds by using *nonSpeculativeWriteList* to iterate non-speculative writes in reverse chronological order until we reach the writes verified by the last periodic verification. For each write we use *speculativeReads* to lookup whether the speculative thread has read this field.

2. Commit verification
   In commit verification we use *speculativeReads* to iterate the set of fields that have been read during speculation. For each field we use *nonSpeculativeWrites* to lookup whether the field was written to after the time of the speculative read. We can imagine an alternative where we iterate all the fields contained in *nonSpeculativeWrites* and compare for each if there is a conflicting speculative read. This approach is better if there are a larger number of fields that have been written to compared to the number of fields speculatively read.

4.4.4 Checkpointing

In section 3.2.4 we discussed the motivation for checkpointing speculative threads to reduce the amount of useful speculation that is lost when a read after write conflict is discovered.

In Java, the state of speculative execution at any point is encapsulated by the call stack, the heap and the data structures we presented in section 4.4 that store speculative field values and speculative field-read times. Therefore any checkpointing technique must be capable of reverting these components to their state when the checkpoint was taken. Fortunately we do not have to consider the heap when we checkpoint because speculations do not mutate the heap unil they are committed.

Call stack checkpointing

In call stack checkpointing we create a checkpoint at every method invocation. When a conflict is discovered we recursively move up the call stack until we reach a checkpoint created at a time before the conflicting read occurred.

To accomplish this using AspectJ we add advice *around* each method execution, as illustrated in listing 4.6.

```
Object around() : execution(* *(..)) {
    do {
        try {
            // Create the checkpoint before executing the method
            long timeNow = System.currentTimeMillis;
            int checkPointID = createCheckPoint();
            // Execute the method
            return proceed();
        } catch (ReadAfterWriteException e) {
            // A read after write conflict was discovered at some
                point down the call stack
            if (e.timeOfConflictingWrite() < timeNow) {
                // This checkpoint was created after the conflicting
                    read so rethrow to a checkpoint higher in the call
                    stack
                throw e;
            } else {
                // This checkpoint was created before the conflicting
                    read so restore execution here
                restore(checkpointID);
            }
        }
    } while(true)
}
```

Listing 4.6: Checkpointing advice which surrounds all method executions.

The *createCheckPoint()* procedure clones the *sideEffects* and *speculativeReads* data structures. If a read after write conflict is discovered during the execution of the method a *ReadAfterWriteException* will be thrown which is caught by the advice which surrounds every method execution. This advice rethrows the exception until it reaches the latest checkpoint before the conflicting read was made. The *restore()* procedure restores the structures that were cloned when the checkpoint was created. Speculation then restarts from this point. A future extension to this project could consider lazy checkpointing to reduce memory consumption where only values that have changed since the last checkpoint are saved.

Loop checkpointing

Call stack checkpointing can still lose a substantial amount of useful speculation between the checkpoint and the conflicting read. To reduce this penalty we checkpoint the execution state of loops periodically as they iterate. If a conflict occurs *during* or *after* a loop has finished but the speculative thread reverts to a checkpoint *before* the loop, the speculative thread can restore the execution state to a suitable iteration when it reaches the loop again.

However this presented a technical difficulty because checkpointing loop iterations requires being able to save and restore the state of the local variables of the enclosing method. This is not an issue in call stack checkpointing because local variables are stored within each frame of the call stack, so reverting to a particular frame will automatically restore the correct state of the local variables at that checkpoint. We can use reflection for dynamically changing the values of fields on the heap, however this is not possible for local variables because the Java compiler eliminates the names of these fields in the bytecode. Therefore the automatic code transformation we apply in phase 1 refactors all local variables that are declared before the loop and used within the loop to be encapsulated within a *LocalFields* instance, as in the example in fig. 4.5. Any accesses or assignments to these local variables are redirected through this object. Checkpointing can save the state of this object and it can be restored easily during the revert procedure.

To control loop checkpointing we transform the loop code in the following way:

1. we insert a call to *loop_setup()* between the initialisation of the loop control variable and the loop header, passing it the *LocalFields* object. *loop_setup()* looks up whether the speculative thread is re-executing this loop because of a conflict. We use the time of the read that caused the conflict to find the latest valid checkpoint for this loop. It resets the speculative state, including the *LocalFields* instance to their checkpointed values. The loop will then continue from the iteration of the checkpoint.

2. we insert a call to *loop_iteration()* at the beginning of the loop body which creates a checkpoint every n iterations. This frequency is exposed as a policy which can be overridden.

3. we insert a call to *loop_end()* after the loop has terminated in order to record the number of iterations for this loop. This value can be used to adapt the periodicity of checkpointing for future executions of this loop.

4.4.5 Speculative memory consumption

Constructing objects only has an additive effect on the heap therefore this operation does not need to be isolated from other threads. However it is important that we restrict the amount of memory that speculative threads can consume so that we do not impact the performance of non-speculative threads. We statically allocate a total amount of memory that can be used for speculation, and it is the Scheduler's responsibility to allocate this between running speculations.

We use the pointcuts in listing 4.7 to intercept the construction of an object or array from a speculative thread. The speculative thread will synchronously request permission from the scheduler before proceeding with the construction. It includes in this request an estimate of the amount of memory it requires in bytes. Unlike lower level languages like C, Java has no *sizeof(...)* utility for determining the exact size of an object or array. Therefore we include our own implementation of *sizeof()* which assumes the sizes of the primitive types, as in table 4.1. These sizes must be independently measured for whichever Virtual Machine is used.

The Scheduler queues memory requests in order of the requesting speculation's priority, ensuring that speculations which are likely to be required sooner have their requests serviced first. If there is not enough free speculative memory to service the highest priority request then the Scheduler terminates the lowest priority speculations until there is sufficient space freed. When considering a request, the Scheduler uses the ThreadMxBean to lookup the total amount of heap memory allocated to each speculative thread. If the sum of these amounts, plus the request being considered, is less than the total allocated for speculation then the request is granted.

Primitive Type	**Size (bytes)**
Object shell	8
Object reference	4
Long	8
Int	4
Short	2
Char	2
Byte	1
Boolean	1
Double	8
Float	4

Table 4.1: The assumed sizes of primitive types in Java [3]

4.4.6 Commit Procedure

The commit procedure has two stages; first it initiates a commit verification for detecting read after write conflicts. Next it updates the values of fields on the heap to a speculative value if one exists. The whole of the commit procedure requires exclusive access to the field-write lock to prevent race conditions with a non-speculative thread.

Committing field values

In order to dynamically change the value of a field on the heap we use reflection. One technical challenge that we faced was setting the value of a field on an object whose type extends the type that declares the field. In this case the commit procedure recursively moves up the class hierarchy until it reaches the level where the field is declared and it sets the field's value. It was also necessary to suppress the Java language access checking for dynamically changing fields that are declared private.

4.4.7 Suspending Speculation

For side effects that cannot be isolated using the Side Effect Manager, for example heap modifications by classes on the build path that have not been weaved, or actions that affect a state external to the heap, it is incumbent upon the application developer to add *@PauseSpeculation* annotations to warn the speculative engine to suspend the speculative thread.

Listing 4.8 shows the pointcut and advice that runs *before* the *@PauseSpeculation* method is called.

```
pointcut pauseSpeculation() :
    call(@PauseSpeculation * *(..))

before() : noSpeculateCall() {
    if (speculativeManager.inSpeculativeMode()) {
        speculativeManager.suspendUntilCommit();
    }
}
```

Listing 4.8: The advice that runs *before* a speculative thread calls a *@PauseSpeculation* method

The *suspendUntilCommit()* procedure causes the speculative thread to wait on an object that is notified by a non-speculative thread that requests the result of the speculation.

4.4.8 Notifying a speculative thread to commit or terminate

The Speculation Manager requires a way of communicating with a speculative thread to inform it to commit if it has been selected or to cancel if it is no longer required. To accomplish this using AspectJ we need to select a set of join points where the speculative thread would be advised to look for any notifications from the Speculation Manager. There is a tradeoff here; a larger set of pointcuts would allow speculative threads to receive cancel notifications earlier (freeing up a speculation slot sooner) or commit notifications earlier (reducing the time spent on speculative overheads) but this would carry a higher interception overhead, or a smaller set of pointcuts which carries a smaller overhead but where the speculation receives notifications later.

Cancelling a speculative thread is not trivial because *Thread.stop()* is deprecated and therefore cannot be relied upon in future versions of Java. It is deprecated because stopping a thread this way at an arbitrary point causes a thread to release all the monitors it has acquired, which may leave objects in an unstable state for other threads. This is not in fact a problem for a speculative thread because its side effects are isolated and therefore it would not leave an object in a bad state for other threads. In order to cancel a speculative thread, the advice which polls for notifications will throw a *CancellationException* if it sees that the Speculation Manager has decided it should be cancelled. This is only caught at the top level of the speculative thread's execution stack and it exits immediately. This assumes that the application does not catch anywhere exceptions of the general *Exception* type, or if it does it immediately re-throws the exception.

4.5 Speculation Generators

Speculation generators are components which decide what actions should be speculated. This is encapsulated within a *SpeculationTarget* object which contains the name of the action's method and the reference of the object which the action should be executed on. This *SpeculationTarget* object is passed to the Scheduler with a likelihood value between 0 and 1 that this target will be the next action selected by the user.

4.5.1 JMenu generator

The JMenu generator decides to speculate any actions contained within the menu that the user is currently interacting with. The *JMenuGenerator* aspect adds advice *around* the creation of any *JMenu* object, as illustrated in listing 4.9. The advice adds an anonymous menu listener which is notified when the menu is selected. On this event it will wait 500ms and if the menu is still selected it will submit all the actions contained within the menu to the Scheduler for speculation. The 500ms wait prevents un-necessary speculations if the user scans their mouse quickly down multiple menus.

However in order to communicate with the Scheduler we require a way of converting a *MenuEvent* to a *SpeculationTarget*. To do this we assume that every menu item has been constructed using an implementation of the *Action* interface which contains an *ActionPerformed* method. This is the case in our benchmark application Pixelitor. Therefore we can iterate the menu items of the currently selected menu, and for each create a *SpeculationTarget* using 'ActionPerformed' as the method name and the *Action* instance behind each menu item as the object reference.

```
pointcut newJMenu() :
    call(JMenu+.new(..))

JMenu around() : newJMenu() {
    JMenu menu = proceed();
    menu.addMenuListener(new MenuListener() {
        public void menuSelected(MenuEvent evt) {
            ...
        }
    }
}
```

Listing 4.9: The pointcut, and advice applied, for intercepting the creation of a JMenu.

4.5.2 Learning generator

The learning generator uses a window of the n previously selected actions to produce a probability for each action that it will be the next action selected. We use the Weka framework [13] in order to use multinomial logistic regression for this. The classifier is built using a set of samples where each sample is a historical sequence of $n + 1$ selected actions.

Building sample list

Every time the user selects an action we record a new sample by taking the last window, shifting every element left and replacing the last element with the action the user has just selected. Figure 4.6 illustrates this for a 3-element history window.

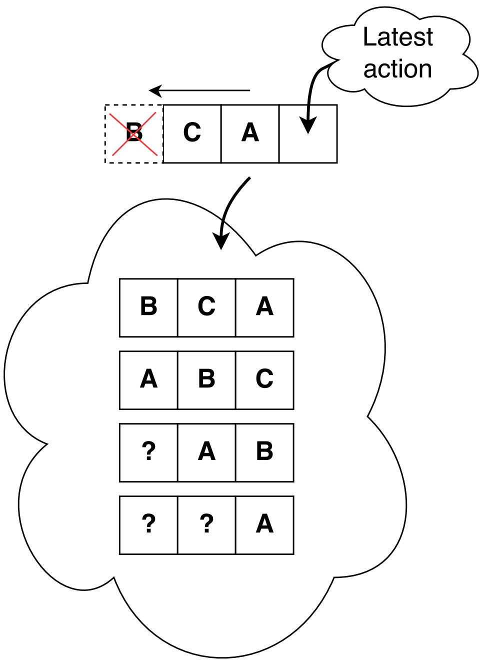

Figure 4.6: Adding a new sample. We take the last history window and shift all actions left one position and add the new action at the end. In this example the user selected actions in the order A → B → C → A.

4.6 Selecting objects to speculate on

We intercept the creation of any objects by a non-speculative thread so that when a Scheduler decides that a *@SpeculableAction* method should be speculated, it has access to all the objects the method can be speculated on.

4.7 Scheduler

The Scheduler uses an instance of the *SpeculationThreadPoolExecutor* class which is an extension of the *ExecutorService* class provided in the JDK, as illustrated in fig. 4.7. This relies on an underlying *PriorityBlockingQueue* of speculations that are waiting for a free thread. We use the notion of *core size* to represent the number of speculative threads that the executor will allow to be created to host speculations. This core size is dynamically adjusted so that the number of speculations does not exceed the number of cores of the underlying processor. It is not necessarily the case that the core size will always be the same as the number of processors because if a speculative thread sleeps, for example having reached an externalised method, it will not vacate the thread until it has woken. However we can immediately consider it as non-speculative because it will only wake after being selected by a non-speculative thread. Therefore we increment the core size to create another thread for speculation. When the sleeping thread eventually terminates we decrement the core size and because the the timeout for speculative threads is set to zero, the thread will be immediately destroyed to stop a queued speculation being assigned. A future extension could consider reducing the overhead of creating speculative threads by only hiding those that retire rather than destroying them.

```
void foo(int w) {
  X x = new X();

  for (int i = 0; i < N; ++i)
  {
    x.f = w + i;
  }
}
```

```
void foo(int w) {
  Locals ls = new Locals();

  // Capture parameter values into Locals
  ls.put('w', w);

  //X x = new X();
  ls.put('x', newX());

  //int i = 0;
  ls.put('i', 0);

  loop_start(ls);

  for (; ls.get('i') < N; ls.put('i', ls.get('i') + 1))
  {
    loop_iteration(ls);
    ls.get('x').f = ls.get('w') + ls.get('i');
  }

  loop_end(ls);
}
```

Figure 4.5: Before and after automatic code transformation of loops. Local variables that are declared before the loop begins and used in the loop body are encapsulated within a *LocalFields* instance.

```
pointcut newObject() :
   call(*.new(..))

pointcut newArray1D(int size) :
   call(*[].new(..)) && args(size)

pointcut newArray2D(int sizeDimension1, int sizeDimension2) :
   call(*[][].new(..)) && args(sizeDimension1, sizeDimension2)
```

Listing 4.7: Pointcuts for intercepting object construction

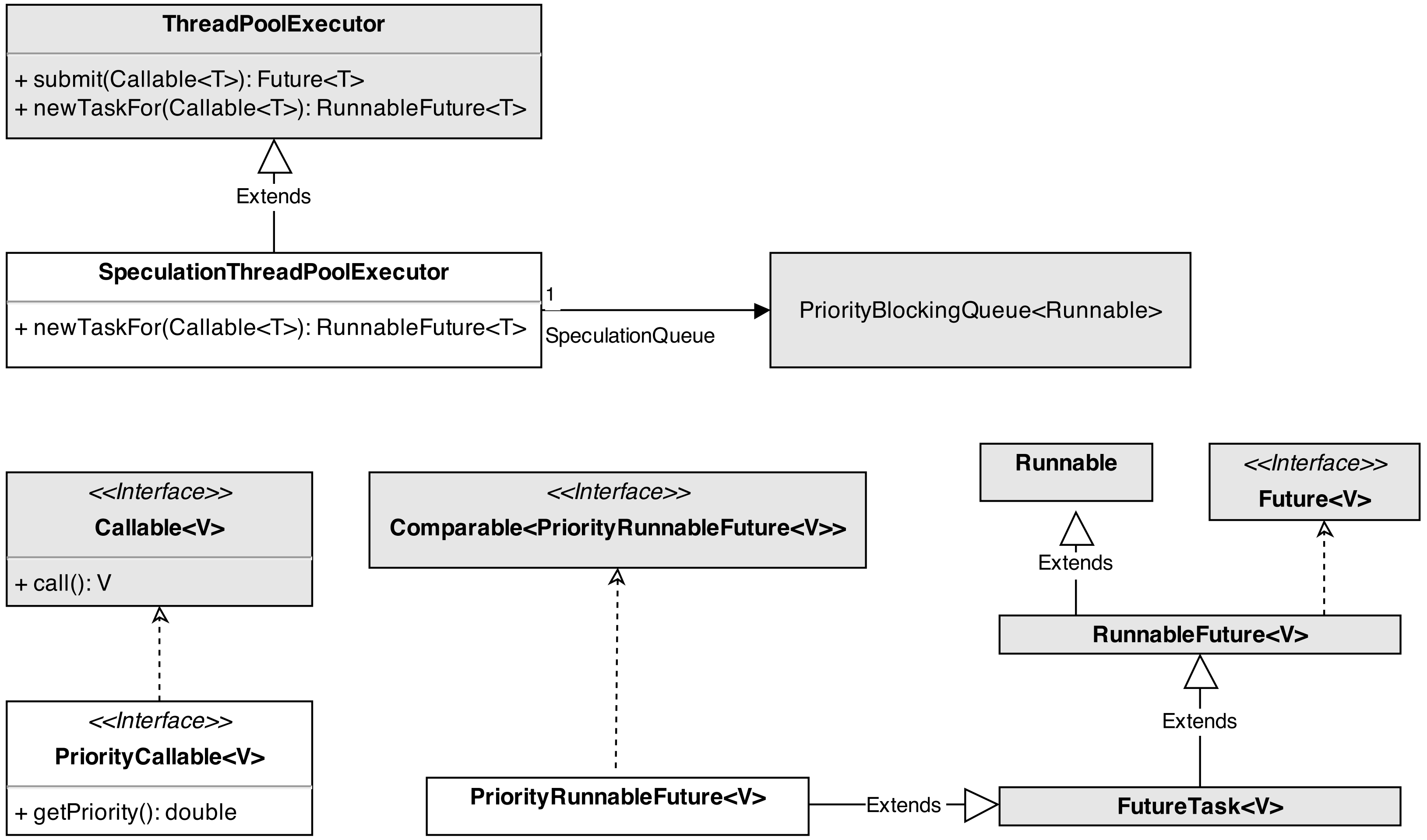
ThreadPoolExecutor
+ submit(Callable<T>): Future<T>
+ newTaskFor(Callable<T>): RunnableFuture<T>
Extends
SpeculationThreadPoolExecutor
+ newTaskFor(Callable<T>): RunnableFuture<T>
1
SpeculationQueue
PriorityBlockingQueue<Runnable>
<<Interface>>
Callable<V>
+ call(): V
<<Interface>>
Comparable<PriorityRunnableFuture<V>>
Runnable
<<Interface>>
Future<V>
Extends
RunnableFuture<V>
Extends
<<Interface>>
PriorityCallable<V>
+ getPriority(): double
PriorityRunnableFuture<V>
Extends
FutureTask<V>

Figure 4.7: A class diagram for the *SpeculationThreadPoolExecutor*. Classes in grey indicate those provided as part of the JDK. The Queuer will *submit* a *PriorityCallable* instance to the executor. The executor overrides *newTaskFor(..)*, used as part of the *submit* procedure, to return a *PriorityRunnableFuture* rather than a *FutureTask*, in order to be compatible with the *PriorityBlockingQueue* instance which requires that the *Runnable* objects that it queues implement the *Comparable* interface.

Chapter 5

Evaluation

5.1 System information

All measurements were performed on a 2.3 GHz Intel Core i7 processor (8 cores) with 16 GB 1600 MHz DDR3 memory. The operating system was OSX version 10.9.3. The Java version was 1.8.0_05 using Java HotSpot(tm) 64-Bit Server VM (build 25.5-b02, mixed mode).

5.2 Warming the JVM

The just-in-time nature of the JVM means that bytecode is optimised at runtime. For example the JVM will compile methods that are invoked more than a thousand times to machine code which significantly improves the performance of subsequent calls to these methods. It is important to ensure that the JVM does not perform these optimisations between different trials in our experiments. This is especially important in this project because our benchmarks mostly comprise loops with many thousands of iterations and each iteration contains nested method calls; therefore the JVM would see these nested methods as candidates for optimisation. We 'warm' the JVM before each experiment by running each action a hundred times to ensure that this optimisation is performed for all actions before we begin experimenting. Warming the JVM also ensures that all the classes used by each action have been lazily loaded, otherwise only the first trial would carry the overhead of loading these class.

5.3 Experimenting with Graphical User Interfaces

In the following experiments we generate speculations by interacting with the menu system of Pixelitor. To ensure that our experiments are fair we must replicate the exact timings of this menu interaction for each trial. To accomplish this we use the *java.awt.Robot* class which can be programmed to simulate native system input events (such as mouse or keyboard movement) at specific times.

The pattern that the Robot followed for each trial is:

1. Execute key sequence to open the menu which contains the action we wish to speculate. This will begin speculation.
2. Wait n seconds, representing the user think time.
3. Execute key sequence to select the action.
4. Wait for the UI event queue to be empty.

When the action is selected in step 3, this invokes a call to the Speculation Manager to request the speculative result. This call will not return until the speculative thread returns the result and because this call is made by the non-speculative Event Queue thread, we can be sure that when the UI event queue is empty the speculation has completed, the result has been returned and the next trial can begin. If we did not wait for this then speculations may have run over multiple trials.

5.4 Timing actions

In order to determine response time of an action, we need to measure the amount of time between the Speculation Manager receiving a request for the result of a speculation, to the time the result is returned. To measure this we use the *ThreadMXBean* to access thread CPU time. It is necessary to verify whether the JVM implementation supports this by calling *isThreadCpuTimeSupported()*. If it is not supported we can also use *System.currentTimeMillis()*, however this will be less fair because the response time will include the time the thread sleeps while the CPU carries out other work, which may vary between trials.

5.5 Evaluating speculation in Pixelitor

In this project we evaluate how speculation can improve the response time of user actions within the Pixelitor application. We consider a user action to be any filter that can be applied to an input image.

Every filter operation that we consider is encapsulated by a class which extends the parent *FilterWithGUI* class, overriding the *filter()* method. This parent class contains an *actionPerformed()* method which is invoked any time the user selects the action. We consider this method to be the entry method to speculation and we annotate it with *@SpeculableAction*. Towards the end of the control flow of each action the *StartDialogSession()* method is invoked which displays a dialogue for modifying the filter parameters. We do not want this dialog to be displayed during speculative execution of the filter and therefore we annotate this method with

Filter	Native execution time (ms)
Unsharp Mask	5003
Glass Tile	3334
Glint	7431
Pointillize	14541
Pixelate	289
Difference of Gaussians	3721

Table 5.1: The time taken to apply each filter to a 5000 x 5000 image of noise in the native version of Pixelitor.

@PauseSpeculation so that the speculative thread pauses at this point until the speculative result is requested by a non-speculative thread.

There are 52 filters within Pixelitor, and all of these are speculable because they all extend from *FilterWithGUI*, which we have annotated. We have randomly selected six of these actions to reduce the experimental burden. In order to appreciate any performance gain from speculation it was first necessary to benchmark the performance of these actions in the native version of Pixelitor. Table 5.1 shows the time taken to execute each filter with a 5000 x 5000 image of noise.

It was also important to investigate whether the execution time of each filter is affected by the size of the input image or the content of the image. Figure 5.1 shows that as we move from a 100 x 100 image size to a 5000 x 5000 image size, each action shows an increase in execution time of at least 9,000%. This had an important impact on the design of the Scheduler; ideally the Scheduler would order speculations using the historical average execution time, but we can see that a historical average would not be reliable if the user executes the same filter over images of different sizes within the same session. In this case the Scheduler would have to resort to static measures to predict the execution time of different actions, for example using the number of nested method calls or loops.

Figure 5.2 shows that varying the image content from a blank image to an image of noise does not significantly impact the execution time regardless of the image size (+/- 5% change). This means that the Scheduler does not need to re-evaluate the priorities of queued speculations if only the content of the image changes during the session.

5.6 Benefit of speculation

The aim of the speculative system is to improve the responsiveness of user actions. We define response time as the period between an action being selected by a non-speculative thread, and the action being successfully executed and its result returned. This is represented by $\Delta t'$ in fig. 3.4. Trivially we could fix some arbitrary time and compare the response time of any action

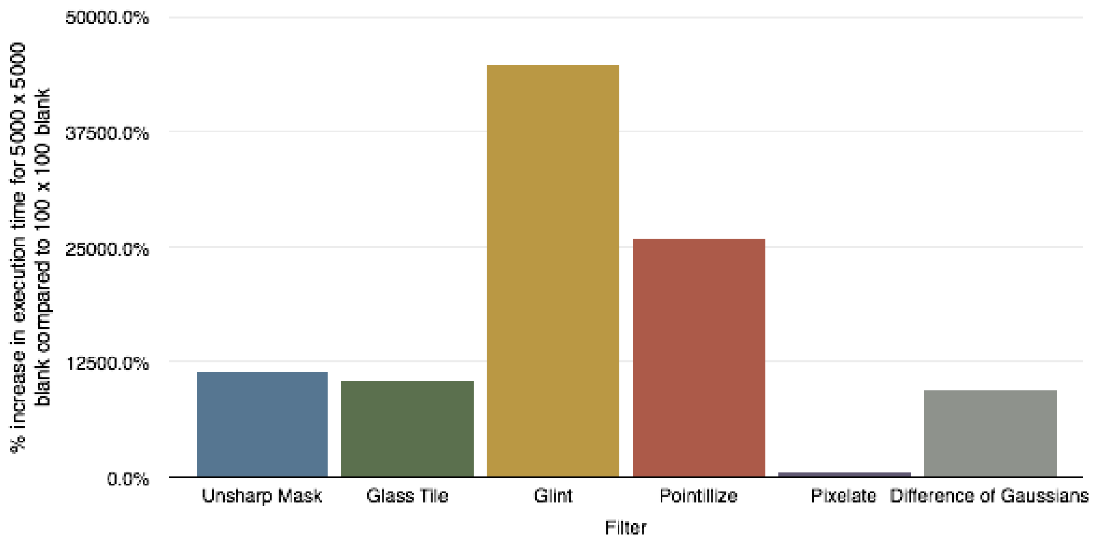

Figure 5.1: The percentage increase in time required to apply each filter to a 5000 x 5000 blank image compared to a 100 x 100 blank image.

between the original and speculative versions of the application. However this would not take into account the amount of time that the action had been speculating before it was selected, and therefore it would not be possible to evaluate how much of an improvement in response time can be attributed to this speculative head start. We would also be unable to evaluate the result separately in terms of the predictive accuracy of the speculation scheduler, and the performance of the code when executing speculatively. In summary we consider the benefit of the speculative system in two parts,

1. relative to the start of speculation

2. accuracy of the scheduler

5.6.1 Measuring response time relative to the start of speculation

We measure the response time of an action at increasing delays from the start of speculation. Ideally we would like to see that selecting an action zero seconds into speculation will provide the same response time as the native version. Selecting it n seconds into speculation would provide an n second decrease in response time. This represents a lower bound on response time unless the speculative thread optimises the speculative code.

For the moment let us only consider the Unsharp filter on a 5000 x 5000 image of noise. In fig. 5.3 we consider how the response time varies when the user selects this filter at increasing delays from the start of speculation. We assume that there are no other actions being speculated. The black line represents the theoretical lower bound on response time in a speculative system. At zero seconds this lower bound would be 5423ms which is the time it takes to execute this

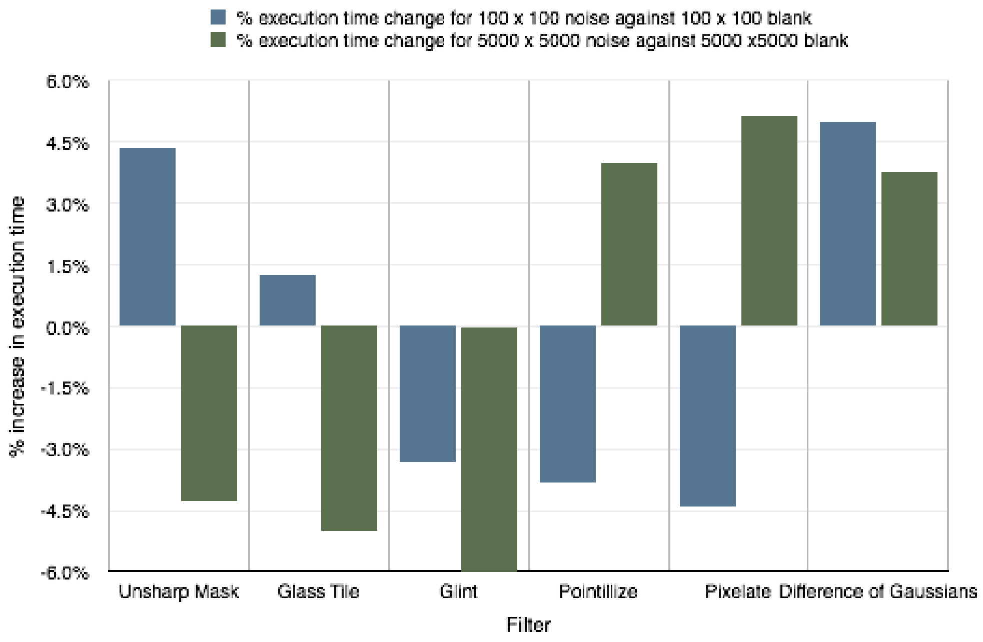

Figure 5.2: The percentage increase in time required to apply each filter to a 100 x 100 image of noise compared to a 100 x 100 blank image, and repeated for an image size of 5000 x 5000.

action in the native version of Pixelitor. This lower bound response time then decreases linearly as user think time increases.

However we see that the blue line which plots the observed response time does not match this. Between zero and six seconds there is a constant overhead compared to the theoretical lower bound response time. Figure 5.4 shows that with up to six seconds of user think time (and hence with up to six seconds of speculation), the vast majority of the total field reads that occur are non-speculative. This means that the first six seconds of the filter's execution do not contain a significant number of field read instructions. The overhead comes from the fact that when these field reads do occur non-speculatively after six seconds, each carries a small penalty for checking whether the thread is speculative or not.

Despite this overhead, the rate at which response time decreases during this period does match the rate at which the theoretical lower bound response time decreases, which shows that the speculative thread is able to fully utilise n seconds of user think time to cover n seconds of the action's execution.

After six seconds we see that this rate slows down which suggests that there is a new overhead. Figure 5.4 shows that after six seconds the number of speculative reads rapidly increases. This suggests that the overhead of a speculative field-read is higher than a non-speculative field-read. This can be seen in table 5.2 where we have micro-benchmarked these two operations. This increased cost is due to the non-speculative field-read needing to lookup if there is a speculative value for the field available, as well as record the time of the first speculative read of any field.

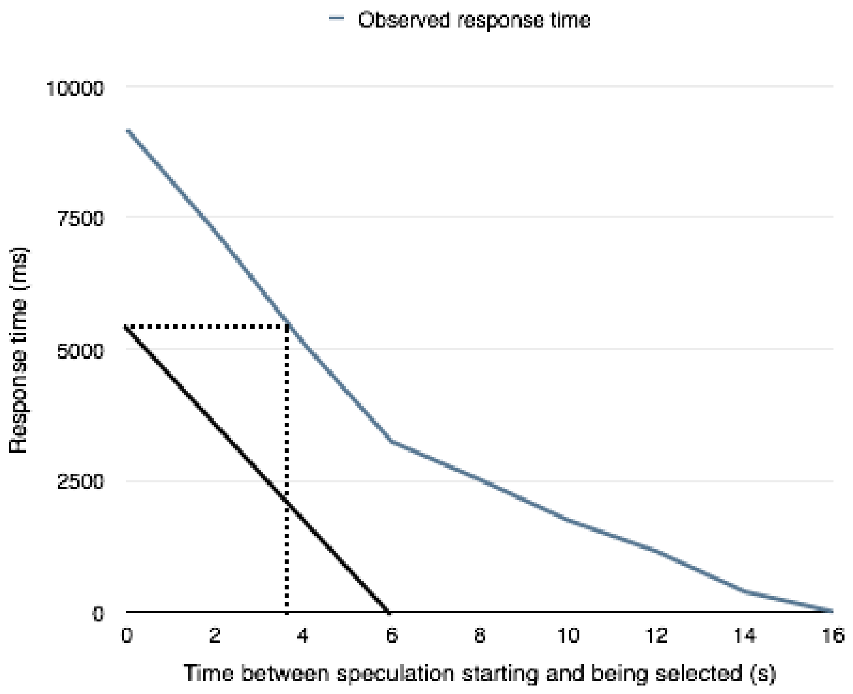

Figure 5.3: The response time for the Unsharp Mask action. The black line represents the theoretical lower bound. The vertical dotted line represents the time after which speculation shows a benefit over the native version.

Operation	Un-weaved (ms)	Weaved, non-speculative (ms)	Weaved, speculative (ms)
Field-read	0.09	0.11	0.15
Field-write	0.05	0.08	0.12

Table 5.2: The results from microbenchmarking a single field-read and field-write operation in the native (unweaved) application, in a non-speculative thread in the weaved application, and in a speculative thread in the weaved application.

The vertical dotted line represents the time after which speculation shows an improvement in response time compared to the native version. We can see that this does not happen until a four second think time, and before four seconds there is in fact a negative effect on response time.

If we now consider the Glass Tile filter, again applied on a 5000 x 5000 image of noise, we achieve the response times illustrated in fig. 5.5. Here we see that we do not achieve a benefit from speculation until after a think time of at least 13 seconds and in fact the response time *increases* as we move from zero to one seconds of user think time. Figure 5.6 shows that this is because the number of speculative field reads increases immediately; there is no lag as we saw with the Unsharp Mask filter. At one second think time we see the worst case response time for the Glass Tile filter which is 80% higher than the native response time. This is worse than

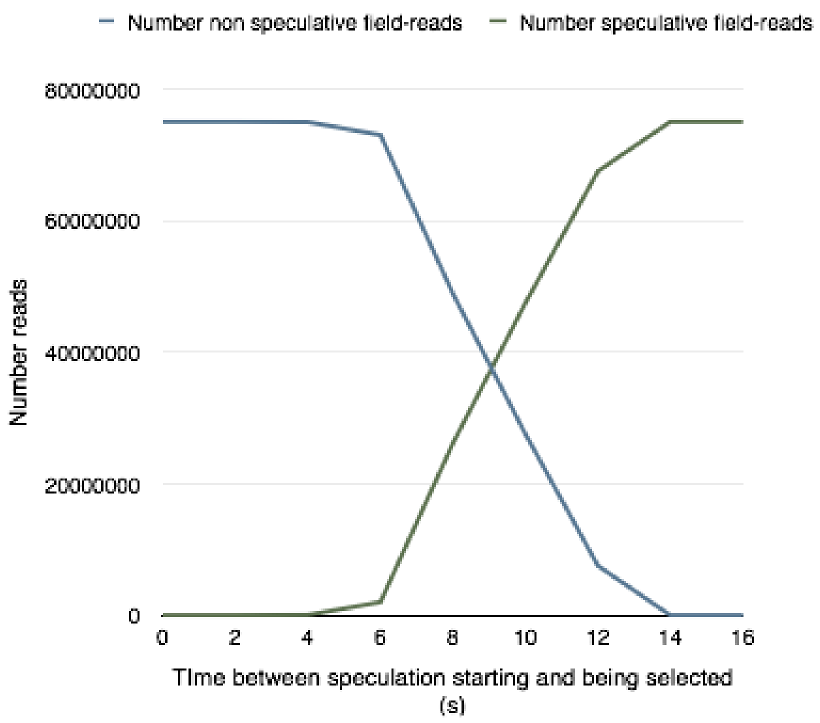

Figure 5.4: The total number of non-speculative field reads and speculative field reads when speculating the Unsharp Mask filter. We repeat this for an increasing time between the speculation starting and it being selected by a non-speculative thread.

the 69% higher response time for the worst case with the Unsharp Mask filter.

Table 5.3 shows for all of the actions we are evaluating; the worst case response time, the minimum think time for speculation to show any benefit compared to the native version, and the total time required to fully speculate the action.

Commit time

Returning to the Unsharp Mask result, another observation is that as the user think time increases beyond 16 seconds, the response time remains static at 16 milliseconds. This is because there is a fixed commit overhead. In the first stage of commit the speculative thread looks up the time of the last non-speculative write for every field it has read during speculation. The second stage updates the value on the heap for every field the speculative thread has written to. Therefore the commit overhead will increase linearly as a function of the number of fields read and written.

Extracting field expressions from loop bodies

When examining the code for the Unsharp Mask action we see that 99% of the total number of field-reads are in fact due to the same field being executed on every iteration of a loop (UnsharpFilter.java:121). This field's value is also a loop invariant and therefore by moving the

Filter	Native execution time (ms)	Worst case (ms)	Minimum think time (ms)
Unsharp Mask	5003	9169	3500
Glass Tile	3334	7220	12000
Glint	7431	7605	1500
Pointillize	14541	21275	5000
Pixelate	289	407	1250
Difference of Gaussians	3721	4414	2700

Table 5.3: The time taken to apply the filter to a 5000 x 5000 image of noise in the native, non-speculative version of Pixelitor, the worst-case response time, and the minimum user think required for speculation to provide a benefit over the native version.

read outside of the loop we can drastically reduce the speculative overhead. Figure 5.7 shows the new response time graph after we have applied this refactoring. We see that speculation now provides a benefit after a delay of only 0.75 seconds, a gain of approximately 80%. This shows that a developer can use knowledge about the overheads of a speculative system when designing the application to allow for optimal speculation.

Read-after-write verification

Thus far we have assumed that speculations have contained no read-after-write conflicts. In reality if a conflict is detected then the whole speculation would need to be aborted. We can imagine a worst case scenario where a long running speculation only detects a conflict when the speculative result is requested by a non-speculative thread. The earlier a conflict can be discovered, the sooner the speculation can be restarted.

In section 4.4.4 we presented a transformation that we apply to loops to add a control statement to the start of every loop iteration. While the idea there was to use this to create a checkpoint, it could also be used as a prompt for carrying out a periodic verification. To evaluate the benefit of periodic verification we conducted an experiment where we created an action which runs as a tight loop, continuously reading a field on the heap. We speculated this action over 100 trials to retrieve the average response time. We repeated this for increasing periods of user think time. During speculation we concurrently executed a non-speculative action which ran as a tight loop, continuously writing to a field on the heap. We repeated this whole experiment three times; one time with zero probability that the field the non-speculative thread was writing to was the field the speculative thread was reading (i.e. no overlap in the read/write set of the two threads), another time with a small probability that the two fields are the same (i.e. a small amount of overlap) and a third time with a large probability (i.e. a large amount of overlap). Most importantly, we repeated all of this twice; once where we verified speculations only when they were committed, and another time with a periodic verification every 1000 milliseconds.

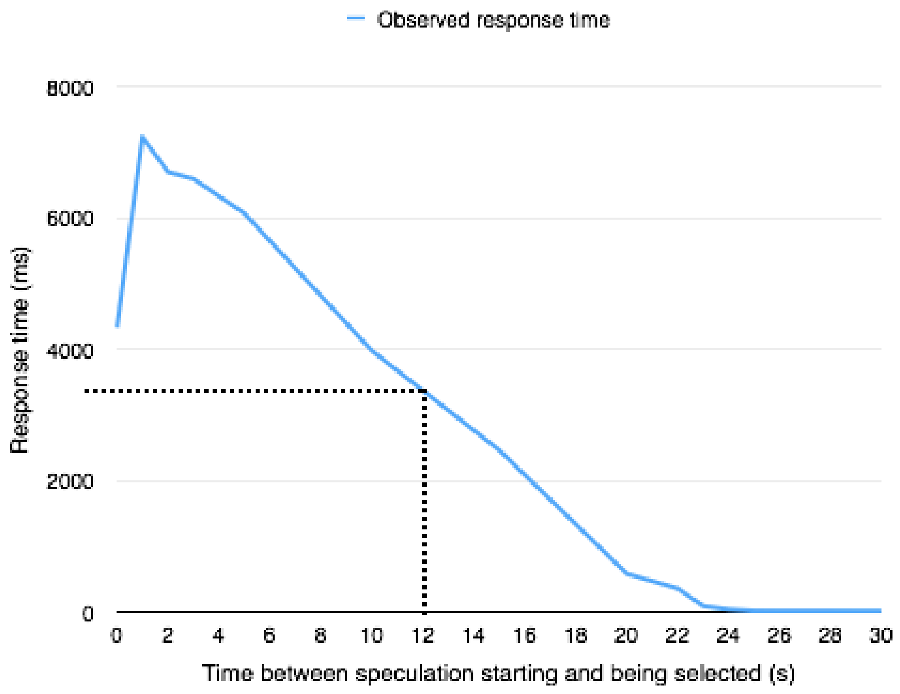

Figure 5.5: The observed response time for the Glass Tile filter applied to a 5000 x 5000 image of noise. The vertical dotted line represents the time after which speculation provides a benefit over the native, non-speculative version.

In fig. 5.8 the three colours red green blue represent no, small and large overlap respectively. The straight lines represent verification only at commit and the dashed lines represent periodic verification. We can see that when there are no conflicts between the non-speculative and speculative threads, the overhead from periodic verification is insignificant. However as we discussed in 5.6.1, this overhead may become more pronounced if the speculative thread reads from a larger set of fields. When we introduce conflict between the threads we can see two things

1. the response time never tends towards the fixed commit overhead.

2. periodic verification results in a better response time than a one-time verification and this is most pronounced with a small probability of conflict.

Checkpointing

Thus far we have assumed that when a read after write conflict is discovered, the speculation must completely restart from the beginning. This means that we cannot extract any benefit from the valid speculation that occurred before the time of the conflicting read. In section 3.2.4 we considered several checkpointing techniques in order to overcome this problem.

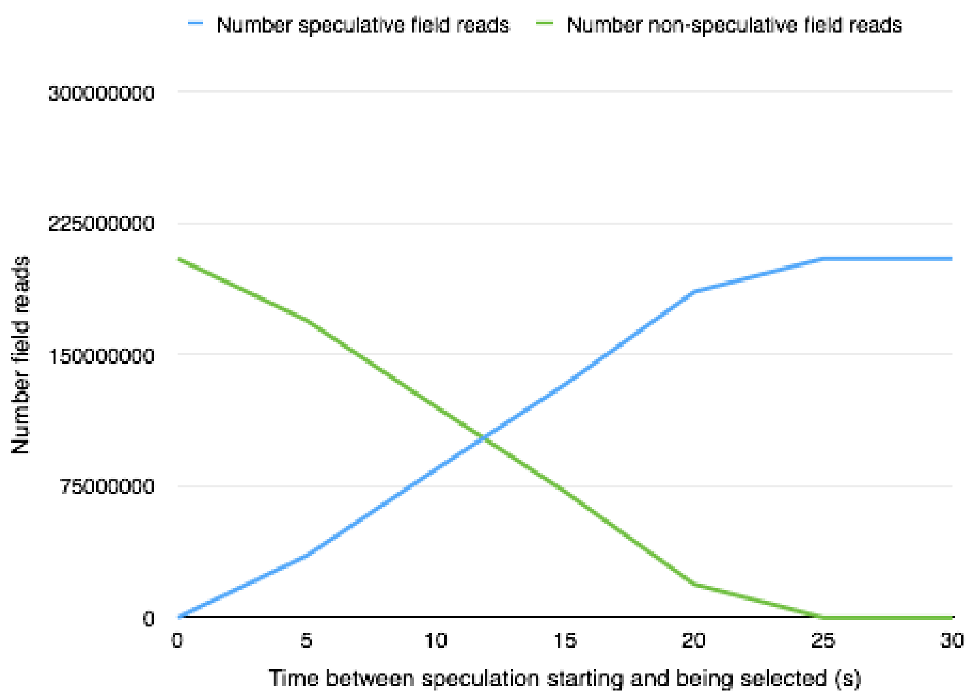

Figure 5.6: The total number of non-speculative field reads and speculative field reads when speculating the Glass Tile filter. We repeat this for an increasing time between the speculation starting and it being selected by a non-speculative thread.

Call stack checkpointing

Call stack checkpointing allows the speculation to revert to the execution frame at any level of the call stack above where the conflict is discovered. This technique is the least intrusive as it does not require any additional source code transformation that cannot be achieved through AspectJ.

Loop checkpointing

Loop checkpointing creates checkpoints at periodic iterations of loops. Therefore when a conflicted speculation revisits this loop it can skip ahead to the last iteration before the conflicting read.

To evaluate the benefits of these techniques we conducted an experiment where we measured the average response time for each action over 100 trials. Each trial the non-speculative thread requested the speculative result at n seconds delay from the start of speculation, where n is the amount of time we measured in table 5.3 that the action takes to fully speculate. We overrode the commit verification procedure to always throw a *ReadAfterWriteException* with the conflicting read time set to a random time between 0 and n.

We repeat this for three checkpointing policies; one where we only checkpoint at the start

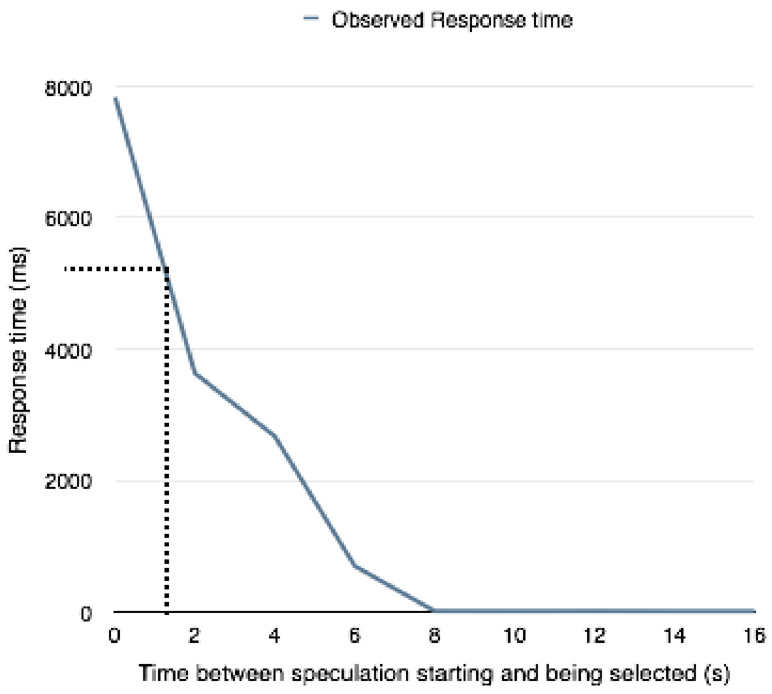

Figure 5.7

of speculation, one where we employ call stack checkpointing, and one where we use loop checkpointing. Table 5.4 compares the results of these policies for each action.

We see that for any action, call stack checkpointing and loop checkpointing shows an improvement over restarting the speculation from the beginning. Let us consider just the Unsharp Mask action. Comparing the the cost of restarting from the checkpoint at the start and the time to execute the action non-speculatively, we can see that there is only a 35ms overhead associated with checkpointing. This overhead comprises:

1. The time taken to propagate the exception to the correct level of the call stack. We would expect this cost to be higher on average for applications that are more recursive or nested in design.

2. The time to restore the data structures behind the speculation to their checkpointed values.

In order to understand why loop checkpointing shows almost a 3000ms improvement over call stack checkpointing we sampled the speculation of the Unsharp Mask action using the VisualVM [32] profiler. It showed that 6698 ms (73%) was spent in *com.jhlabs.image.UnsharpFilter.filter()*, a method which contains just one loop. Therefore when we move up the call stack we jump over over the whole of the execution of this loop, losing all of its work.

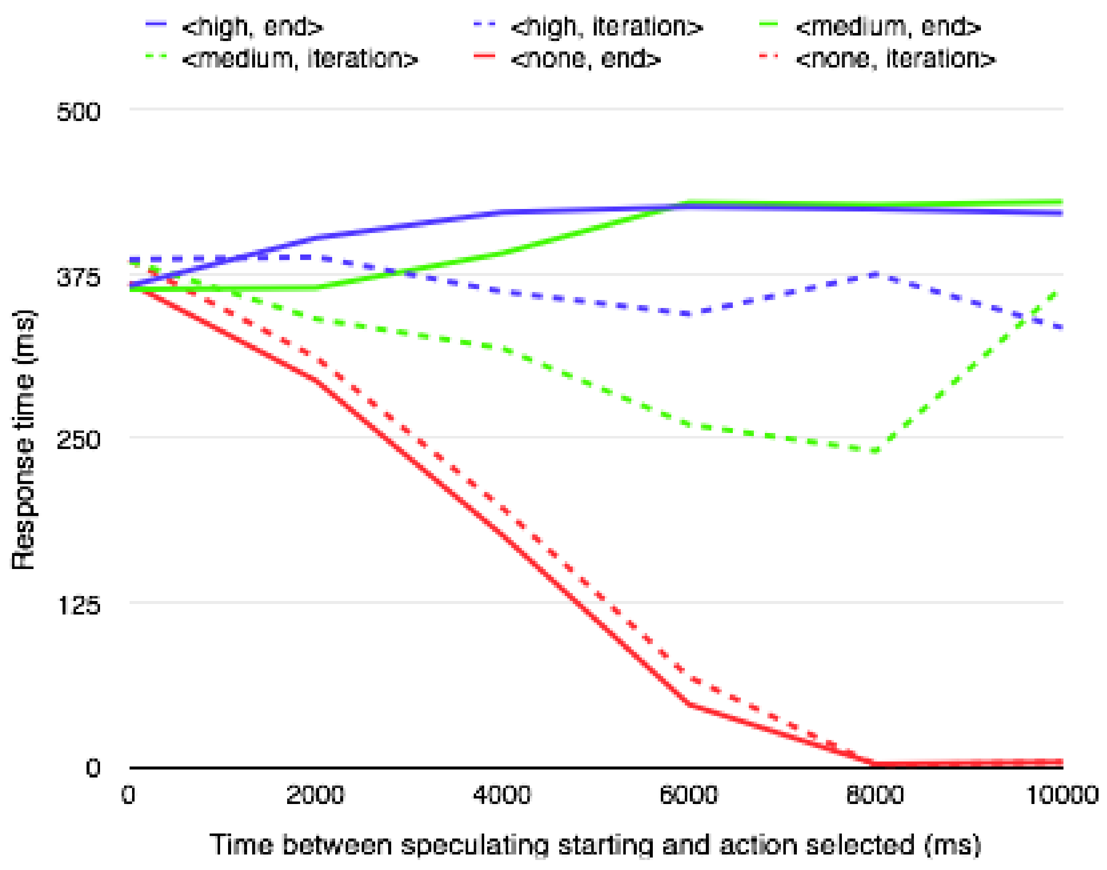

Figure 5.8

5.7 Impact on non-speculative thread performance

One of the key design considerations of the speculative system was that it does not impact the performance of any non-speculative thread. In order to validate this we conducted an experiment in which we timed how long a non-speculative thread takes to apply the Difference of Gaussians filter on a 4000 x 4000 image of noise, averaged over 100 trials. We repeated this, increasing the number of background speculations as well as changing the image size that the speculations were working with. At first the background speculations were speculating over a 100 x 100 image, and then with a 5000 x 5000 image.

Figure 5.9 shows that as we increase the number of background speculations speculating over 100 x 100 images, there is no impact on the performance of the non-speculative thread until the number of speculations exceeds eight, when there is a noticeable impact. This is because there is now contention for CPU resources on an 8 core machine. If we consider the impact when the speculative threads work with a 5000 x 5000 image we can see that only after three background speculations a noticeable impact appears. This is due to the garbage collector which is triggered because the speculations require more memory to apply filters to larger image sizes. The garbage collector then impacts the performance of the main thread. To prevent this

Filter	Time to execute non-speculatively (ms)	Single checkpoint at start (ms)	Call stack check-pointing (ms)	Loop check-pointing (ms)
Unsharp Mask	9169	9204	6307	3817
Glass Tile	24000	24103	22492	15874
Glint	8000	8115	7192	4893
Pointillize	28000	28118	25081	19129
Pixelate	2000	2131	2074	1503
Diffe. of Gaussians	6000	6156	5970	3835

Table 5.4: Average response time using different checkpoint policies, assuming that a conflict is always detected when the speculation commits with a random time for the conflicting read.

we use the technique discussed in section 4.4.5 where the developer statically assigns an amount of memory for speculation.

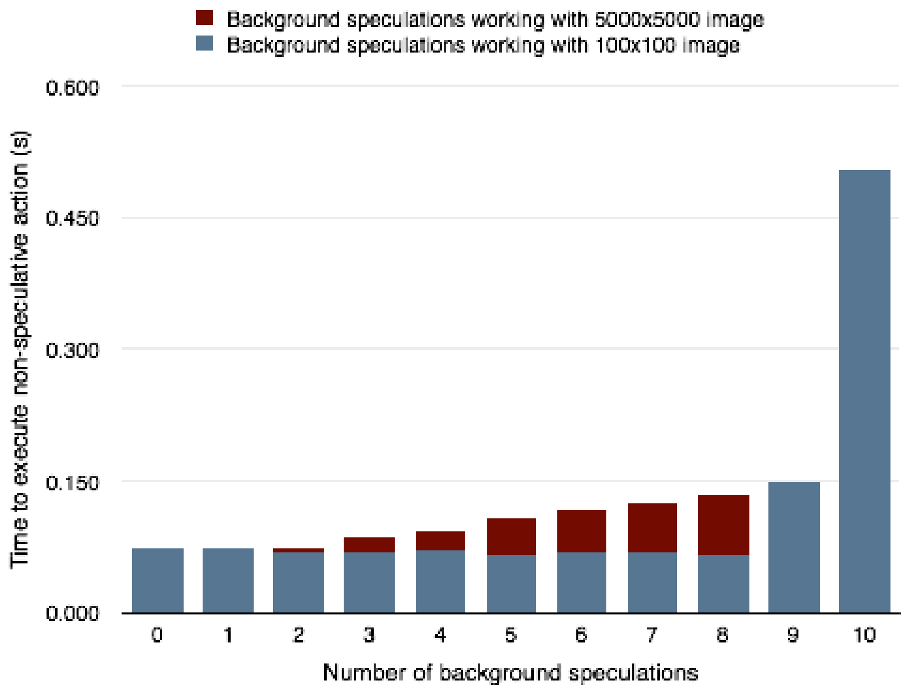

Figure 5.9

5.7.1 Learning cost

In section 4.5.2 we presented a technique for generating speculations based on learning user behaviour. It is not possible to provide a cost-benefit analysis of this technique here because we do not have any real historical data for how users interact with Pixelitor. However we can evaluate the cost of building the classifier on the non-speculative thread.

The dimensions of this classification technique are:

1. window size
   The number of previous actions the classifier uses to classify the next action, i.e. the number of independent variables.

2. number samples
   The number of samples used to build the classifier.

3. number attributes
   The number of distinct actions that can be classified.

When we consider an application with only five attributes (actions) and a window size of 5, fig. 5.11 shows that this only takes 180ms to build the classifier. This seems a small cost but it must be judged relative to the time that the actions are taking to execute. However what is noticeable is that as we increase the number of attributes to 15, there is a dramatic increase in the time required to build the classifier. In fact there is a 4000% increase in the build time for a window size of five and sample size of 4096. This is problematic for an application like Pixelitor, where the number of attributes (actions) we require is 70. However if the set of filters the user works with is much smaller than this, this reduces the severity of the problem.

Figure 5.11 shows that for a sample size of 4096, classifying 5 attributes with a window size of 5 takes 180ms. How noticeable this is depends on what action this build follows. What is striking however is the impact of increasing the number of attributes. By moving from 5 to 15 attributes there is more than a 4000% increase in the time to build the classifier for a window size of 5 and a sample size of 4096. now increases exponentially with window size

5.8 Ensuring accurate speculation

Another key design considerations of the speculative system was that speculation should produce the same result when speculation an action as the native, non-speculative version of the application. To validate this design consideration, we concurrently speculated all the actions that we consider in this section and then requested the speculative result of each in turn, diffing the result with the result we obtained from the non-speculative version of the application. We then repeated this 20 times, each time requesting the speculative results in a different order. There were no differences reported in any trial.

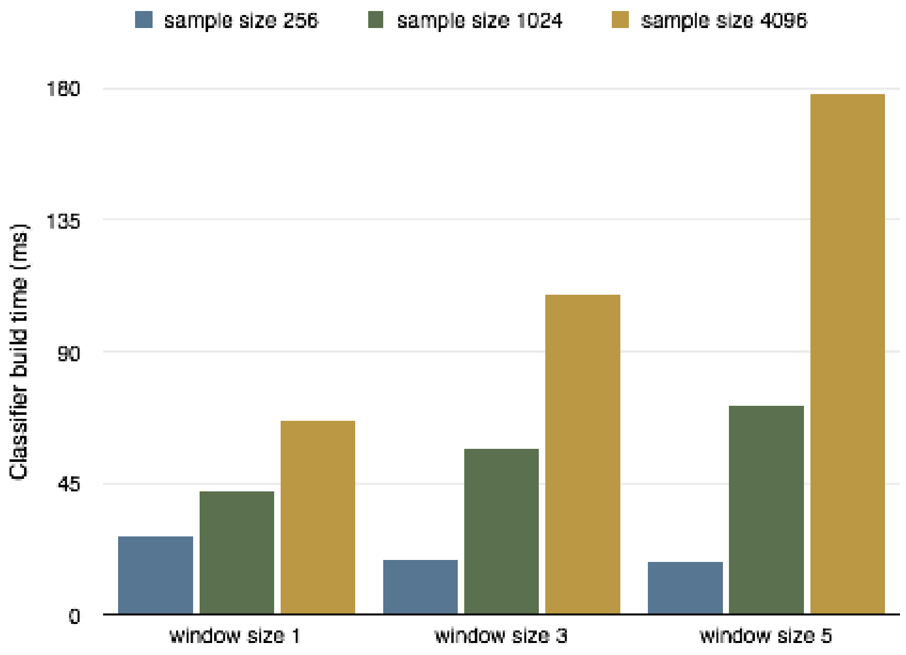

Figure 5.10: The cost of building the classifier that can support 5 attributes (actions).

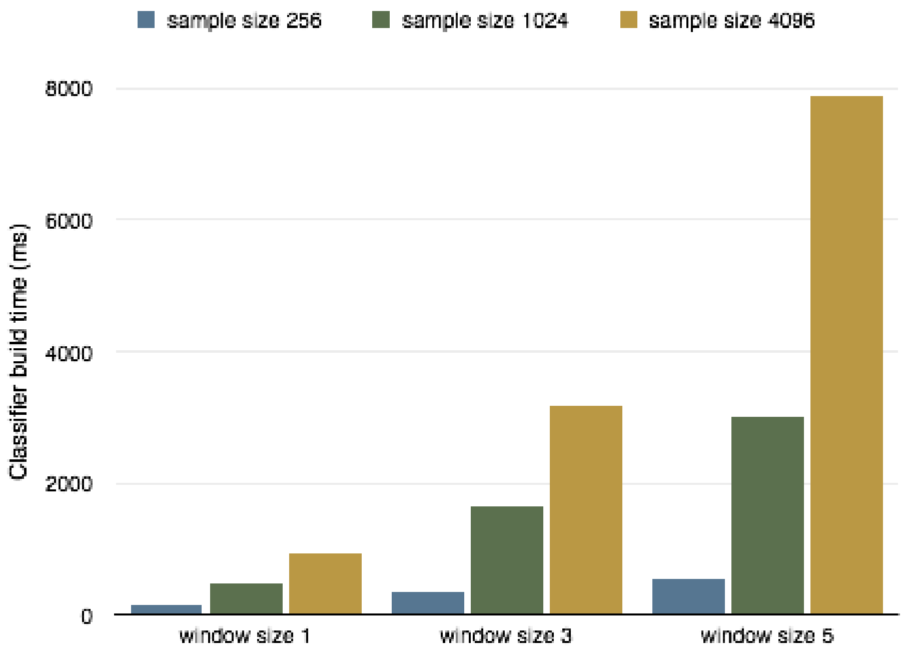

Figure 5.11: The cost of building the classifier that can support 15 attributes (actions).

Chapter 6

Conclusion

6.1 Limitations

1. Speculative synchronisation
   AspectJ provides no way of intercepting the acquisition and release of monitors. Therefore speculative threads will still contend with non-speculative threads over synchronised code regions. This is not desirable; a speculative thread should immediately enter a synchronised region because any field operations which could cause race conditions are isolated. Similarly a non-speculative thread should never be denied access to a monitor because of a speculative thread.

2. Speculative parameters
   We have assumed that the entry methods to speculative actions do not take any parameters. If this assumption was not true we would need to consider some form of parameter prediction.

6.2 Future Work

6.2.1 Delaying externalised methods

When a speculative thread invokes an externalised method, i.e. a method annotated with @PauseSpeculation, it is suspended until it is requested by a non-speculative thread. It will then finish execution non-speculatively. We can imagine a worst case scenario when the methods that followed the externalised method would have benefited from speculation and do not have any data dependencies with the externalised method. In this case there would have been no harm in moving the externalised method to the end of speculation.

We illustrate this idea using the code in fig. 6.1. The speculable action makes a call to three distinct methods but the second, *b()*, is an externalised method. Figure 6.2 shows that in a speculative system without method reordering the speculative thread will speculatively execute *a()* and pause when it invokes b() until it is selected. It will then verify there have been no data conflicts and commit any side effects so far. It will finish by executing *c()* non-speculatively.

In a speculative system with method reordering, when the speculative thread reaches *b()*, instead of pausing it will continue immediately speculatively executing *c()*. When it has finished speculating *c()* it pauses until it is selected. When it is selected it first verifies and commits the methods it speculated in-order, i.e. *a()*. If this commits successfully it next executes *b()* non-speculatively. When it has finished executing *b()* it will verify whether any fields that *b()* read from or set were used in the out-of-order speculated methods i.e. *c()*. If there are no conflicts then it can straight away commit the speculative result of *c()*. If not, it re-executes *c()* non-speculatively.

```
@SpeculableAction
void action() {
    a();
    b();
    c();
}

void a() {
    ...
}

@PauseSpeculation
void b() {
    ...
}

void c() {
    ... // Some expensive operations
}
```

Figure 6.1: A motivating example for for delaying the execution of externalised methods. *b()* has no dependencies with the method that follows, *c()*, therefore its execution can be delayed.

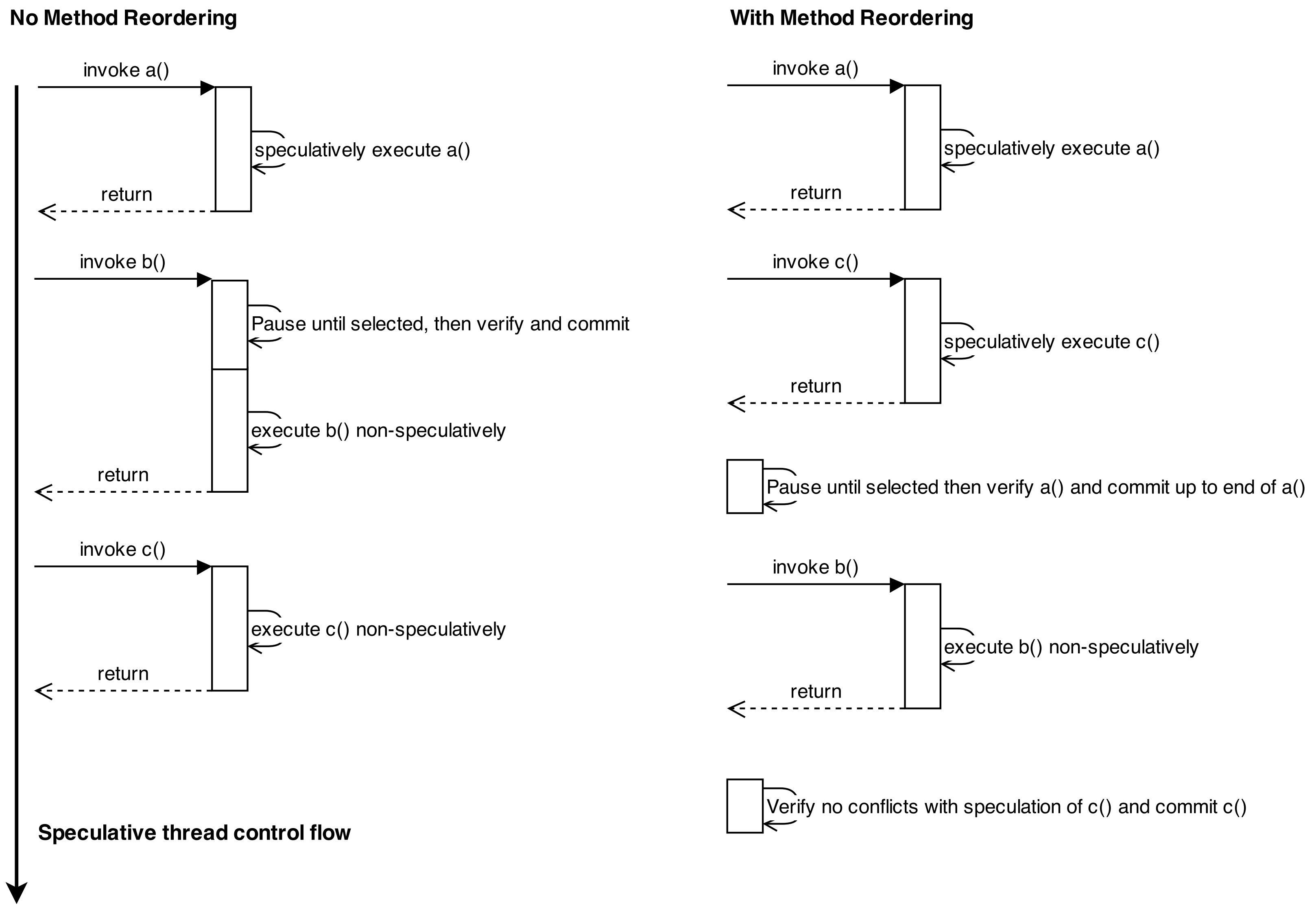
No Method Reordering
invoke a()
speculatively execute a()
return
invoke b()
Pause until selected, then verify and commit
execute b() non-speculatively
return
invoke c()
execute c() non-speculatively
return
Speculative thread control flow
With Method Reordering
invoke a()
speculatively execute a()
return
invoke c()
speculatively execute c()
return
Pause until selected then verify a() and commit up to end of a()
invoke b()
execute b() non-speculatively
return
Verify no conflicts with speculation of c() and commit c()

Figure 6.2: How a speculative thread proceeds with and without method reordering.

6.2.2 Using Java continuations for checkpointing

A continuation is a representation of the full execution state of a program at any point in time, including the call stack, local variables and program counter. Support for continuations is found in a variety of programming languages, for example generators in Python and backtracking in Prolog both rely on continuations. Although there is no official support within Java, there are third party libraries which provide this functionality, for example using bytecde instrumentation [2]. Future extensions of this project could look at using continuations as a technique for checkpointing a speculative thread at any point during speculation.

6.2.3 Reducing overhead for non-speculative threads

Currently, field operations in non-speculative threads carry an overhead because they need to look up whether the thread is executing speculatively. This overhead could be avoided by having two versions of every method. One version would be the native method, the other would be a speculative version where field reads or writes are directed to use the Side Effect Manager and method calls are changed to invoke the speculative version of methods. However this would not be achievable with just AspectJ, and would require bytecode instrumentation.

Bibliography

[1] Array element get/set pointcut. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=157031`. Accessed: 2014-06-13.

[2] Javaflow. `http://commons.apache.org/sandbox/commons-javaflow/`. Accessed: 2014-06-13.

[3] Sizeof for Java. `http://www.javaworld.com/article/2077408/core-java/sizeof-for-java.html`. Accessed: 2014-05-31.

[4] Softmax Regression. `http://ufldl.stanford.edu/wiki/index.php/Softmax_Regression`. Accessed: 2014-06-13.

[5] Ivo Anjo and Joao Cachopo. A software-based method-level speculation framework for the java platform. In *Languages and Compilers for Parallel Computing*, pages 205–219. Springer, 2013.

[6] João Cachopo and António Rito-Silva. Versioned Boxes As the Basis for Memory Transactions. *Sci. Comput. Program.*, 63(2):172–185, December 2006.

[7] Fay Chang and Garth A Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *OSDI*, pages 1–14, 1999.

[8] Kung Chen and Chin-Hung Chien. Extending the field access pointcuts of aspectj to arrays. *Journal of Software Engineering Studies*, 2(2):2–11, 2007.

[9] Michael K Chen and Kunle Olukotun. Exploiting method-level parallelism in single-threaded java programs. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 176–184. IEEE, 1998.

[10] Michael K Chen and Kunle Olukotun. The jrpm system for dynamically parallelizing java programs. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 434–445. IEEE, 2003.

[11] Keir Faser and Fay Chang. Operating system i/o speculation: How two invocations are faster than one. In *USENIX Annual Technical Conference, General Track*, pages 325–338, 2003.

[12] Freddy Gabbay and Freddy Gabbay. Speculative execution based on value prediction. Technical report, EE Department TR 1080, Technion - Israel Institue of Technology, 1996.

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[14] Lance Hammond, Mark Willey, and Kunle Olukotun. *Data speculation support for a chip multiprocessor*, volume 32. ACM, 1998.

[15] Shiwen Hu, Ravi Bhargava, and Lizy Kurian John. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism*, 5(1), 2003.

[16] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast Track: A Software System for Speculative Program Optimization. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 157–168. IEEE Computer Society, 2009.

[17] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001?Object-Oriented Programming*, pages 327–354. Springer, 2001.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented Programming.* Springer, 1997.

[19] Artur Klauser, Abhijit Paithankar, and Dirk Grunwald. Selective Eager Execution on the Polypath Architecture. In *In Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 250–259, 1998.

[20] David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pages 182–189. IEEE, 1991.

[21] John R Lange, Peter A Dinda, and Samuel Rossoff. Experiences with Client-based Speculative Remote Display. In *USENIX Annual Technical Conference*, pages 419–432, 2008.

[22] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *NSDI*, pages 127–142, 2010.

[23] Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Speculative analysis of integrated development environment recommendations. *ACM SIGPLAN Notices*, 47(10):669–682, 2012.

[24] Edmund B. Nightingale. *Speculative Execution Within A Commodity Operating System.* PhD thesis, University of Michigan, 2007.

[25] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative execution in a distributed file system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 191–205. ACM, 2005.

[26] Christopher JF Pickett and Clark Verbrugge. Software thread level speculation for the java language and virtual machine environment. In *Languages and Compilers for Parallel Computing*, pages 304–318. Springer, 2006.

[27] Pixelitor. *version 2.0.0.* lbalazscs, 2014.

[28] Arun Raman, Greta Yorsh, Martin Vechev, and Eran Yahav. Sprint: Speculative Prefetching of Remote Data. *ACM SIGPLAN Notices*, 46(10):259–274, 2011.

[29] Gurindar S Sohi, Scott E Breach, and TN Vijaykumar. Multiscalar processors. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 414–425. ACM, 1995.

[30] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. `http://www.gotw.ca/publications/concurrency-ddj.htm`. Accessed: 2014-06-13.

[31] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *Computers, IEEE Transactions on*, 48(9):881–902, 1999.

[32] VisualVM. *version 1.3.7.* Oracle, 2014.

[33] Benjamin Wester, Peter M Chen, and Jason Flinn. Operating System Support for Application-Specific Speculation. In *Proceedings of the sixth conference on Computer systems*, pages 229–242. ACM, 2011.

[34] Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, and Mikel Luján. Optimizing software runtime systems for speculative parallelization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):39, 2013.