ROYAL INSTITUTE
OF TECHNOLOGY

Flow Java: Declarative Concurrency for Java

Frej Drejhammar

A Dissertation submitted to
the Royal Institute of Technology
in partial fulfillment of the requirements for
the degree of Licentiate of Philosophy

the Royal Institute of Technology
Department of Microelectronics and Information Technology
Stockholm

March 2005

TRITA-IT LECS AVH 04:15
ISSN 1651-4079
ISRN KTH/IMIT/LECS/AVH-04/15–SE

Printed by Universitetsservice US-AB 2005

Abstract

This thesis presents the design, implementation, and evaluation of Flow Java, a programming language for the implementation of concurrent programs. Flow Java adds powerful programming abstractions for automatic synchronization of concurrent programs to Java. The abstractions added are single assignment variables (logic variables) and futures (read-only views of logic variables).

The added abstractions conservatively extend Java with respect to types, parameter passing, and concurrency. Futures support secure concurrent abstractions and are essential for seamless integration of single assignment variables into Java. These abstractions allow for simple and concise implementation of high-level concurrent programming abstractions.

Flow Java is implemented as a moderate extension to the GNU `GCJ/libjava` Java compiler and runtime environment. The extension is not specific to a particular implementation, it could easily be incorporated into other Java implementations.

The thesis presents three implementation strategies for single assignment variables. One strategy uses forwarding and dereferencing while the two others are variants of Taylor's scheme. Taylor's scheme represents logic variables as a circular list. The thesis presents a new adaptation of Taylor's scheme to a concurrent language using operating system threads.

The Flow Java system is evaluated using standard Java benchmarks. Evaluation shows that in most cases the overhead incurred by the extensions is between 10% and 50%. For some pathological cases the runtime increases by up to 150%. Concurrent programs making use of Flow Java's automatic synchronization, generally perform as good as corresponding Java programs. In some cases Flow Java programs outperform Java programs by as much as 33%.

Acknowledgments

I would like to thank my thesis advisors Seif Haridi and Christian Schulte for their help and support. I'm especially grateful to Christian Schulte, without his expert guidance navigating the jungles of academia, this thesis would not have been possible. He is also the author of the LaTeX macros used for typesetting the performance graphs in Chapter 5. I would also like to thank Seif Haridi and Per Brand for hiring me and giving me the opportunity to do the research underlying this thesis.

I am grateful to Andreas Rossberg who analyzed and pointed out a flaw in an early version of the Flow Java type system. Likewise I wish to thank Joe Armstrong for many inspiring conversations and for volunteering to proofread a draft of this thesis.

This work has been partially funded by the Swedish Vinnova PPC (Peer to Peer Computing, project 2001-06045) project.

Contents

Chapter 1

Introduction

This thesis presents the design, implementation, and evaluation of Flow Java, a programming language for the implementation of concurrent programs. Flow Java adds powerful programming abstractions for automatic synchronization of concurrent programs to Java. The abstractions added are single assignment variables (logic variables) and futures (read-only views of logic variables).

1.1 Motivation

Concurrent, distributed, and parallel programs fundamentally rely on mechanisms for synchronizing concurrent computations on shared data. To synchronize accesses to shared data most systems and languages for concurrent programming provide monitors and/or locks and condition variables. Implementing non-trivial concurrent programs using these synchronization mechanisms is known to be error prone. Flow Java provides a better way to construct concurrent programs which avoids explicit synchronization.

The goals of Flow Java are:

- Provide automatic synchronization for concurrent computations. The programmer should only have to consider *what* needs to be synchronized, not *how* and *when*.
- Provide efficient means for organizing concurrent computations as a collection of communicating tasks.
- Integrate seamlessly with standard Java. Recompilation should be the only requirement for using Java programs in Flow Java. This allows the Flow

Java programmer to take advantage of the wealth of standard Java libraries available.

- Have an efficient implementation. The implementation should have a reasonable overhead for using Flow Java features without unduly penalizing standard Java code. The implementation should make few assumptions on the underlying Java system, thus making it easily portable to different Java implementations.

1.2 Approach

1.2.1 Automatic Synchronization

Flow Java adds logic variables to Java. Logic variables in Flow Java are referred to as single assignment variables. Single assignment variables are initially unbound, that is they have yet to acquire a value. A thread accessing the content of a single assignment variable suspends until the variable's value becomes available. A single assignment variable acquires a value (becomes determined) by an operation called *bind*. This operation binds a single assignment variable to a Java object. This makes the single assignment variable indistinguishable from the object. Single assignment variables in Flow Java are typed and can only be bound to objects of compatible types.

1.2.2 Inter-task Communication

The automatic suspension of operations accessing the content of an unbound single assignment variable until its value becomes determined lends itself well to implementing abstractions such as streams [36]. Streams are a prerequisite for a port-based message passing system [22]. Once an initial message has been sent using the port, additional communication can occur through unbound single assignment variables embedded in the initial message (for example, for replies or further messages).

As the only operations on single assignment variables are the implicit synchronization and bind, a programmer wishing to connect two tasks via a shared single assignment variable has to create the variables before the tasks are created. To provide greater flexibility when constructing systems, a second operation on single assignment variables is introduced, *aliasing*. The alias operation allows two unbound single assignment variables to be made equal, equal in the sense that they will be indistinguishable from each other.

In order to allow safe abstractions where the ability to bind a single assignment variable is restricted, Flow Java provides a new kind of object, a *future*. The future

is a read only view of a single assignment variable. A future is always associated with a single assignment variable, when the variable becomes bound the future also becomes bound. A future has the same synchronization properties as a single assignment variable except that it cannot be bound. Futures are created by an overloaded type conversion operation, converting a single assignment variable to its ordinary Java type converts it to a future. A thread can, instead of a single assignment variable, share a future with other threads. The read-only property of the future ensures that only the thread having access to the single assignment variable can bind the future.

1.2.3 Integration with Java

For Flow Java to be useful for real applications, Flow Java must integrate seamlessly with standard Java libraries. The main issue is how single assignment variables should be visible to Java programs. Requiring that all objects passed to standard Java should be determined is unnecessarily restrictive. It would lead to unnecessary synchronization and also require a traversal of the involved Flow Java objects.

The approach taken in Flow Java is to add an implicit type conversion from single assignment variables to futures. This makes seamless integration with standard Java possible as methods in Java libraries will operate on futures. The synchronization properties of futures guarantees correct execution of standard Java code.

1.2.4 Efficient Implementation

The Flow Java implementation is based on the GNU `GCJ` Java compiler and the `libjava` runtime environment which implements ahead of time compilation of Java. The distinction between futures and single assignment variables is maintained purely by the compiler. The runtime system is only concerned with ordinary objects and synchronization objects (which represent single assignment variables and futures). The runtime system of `GCJ/libjava` uses the same object representation as C++. Flow Java objects are extended with a forwarding pointer field to support binding and aliasing. By using a specially constructed virtual method table for synchronization objects, automatic synchronization when invoking a method is possible without incurring a runtime overhead (apart from the extra memory required for the forwarding pointer). Synchronization on field accesses is on the other hand associated with a runtime overhead as the compiler generates code to check the binding status of the variable. The overhead of these checks can be reduced by compiler optimizations which remove redundant checks.

The infrastructure for Java monitors and locks is exploited for implementing thread suspension.

The binding and aliasing information manipulated and maintained by the runtime system uses the previously described forwarding pointer field in each object. Operations manipulating the information use a two layer architecture which separates concerns for correctness and atomicity from the representation of the equivalence classes formed by aliasing and binding. The forwarding based scheme for representing equivalence classes is selected after evaluating three alternative representations.

1.3 Source Material

The material in this thesis has previously been published in the following two internationally peer-reviewed papers:

- Frej Drejhammar, Christian Schulte, Seif Haridi, and Per Brand. Flow Java: Declarative concurrency for Java. In *Proceedings of the Nineteenth International Conference on Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 346–360, Mumbai, India, December 2003. Springer-Verlag. [11]. Won the Association of Logic Programming best application paper award 2003.

- Frej Drejhammar and Christian Schulte. Implementation strategies for single assignment variables. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2004)*, Saint Malo, France, September 2004. [10].

The thesis author is the main contributor to the design of Flow Java as well as the only implementor.

1.4 Thesis Contribution

The contribution of this thesis is the design, implementation, and evaluation of an extension to Java that makes logic programming technology for concurrent programming available in a widely used programming language. More specifically, it contributes the insight that futures as read-only variants of logic variables are essential for seamless integration of logic variables.

The thesis contributes implementation techniques for integrating logic variables and futures into object-oriented programming systems using a traditional thread-based concurrency model. Three different implementation strategies for

single assignment variables, previously known in logic programming, are adapted to a thread-based environment. The adaption takes locking, token equality, and updates into account. The thesis contributes a new two-layer architecture which separates the representation of single assignment variables from the operations required to ensure correctness and atomicity in an environment with operating system threads.

Additionally, the thesis clarifies how Taylor-based schemes need to be adapted to be compatible with thread-based concurrency, token equality, and update. Evaluation shows that the most crucial aspect for efficiency is to minimize the amount of memory accessed.

1.5 Thesis Organization

Chapter 2 presents Flow Java by presenting its features and giving illustrative examples. Chapter 3 shows how Flow Java can be applied to a non-trivial problem. Flow Java is used to implement a lift controller organized as a set of communicating tasks. The Flow Java implementation is discussed in Chapter 4 followed by its evaluation in Chapter 5. The thesis concludes with a summary of the thesis's contributions and presents plans for future work in Chapter 6.

Chapter 2

Flow Java

This chapter introduces Flow Java and presents some basic concurrent programming abstractions. Section 2.1 introduces single assignment variables, followed by futures in Section 2.3. Henceforth the term *synchronization variable* will used as a term for both single assignment variables and futures. Section 2.3 discusses aliasing of single assignment variables as an mechanism for constructing synchronization abstractions such as barriers. Section 2.5 describes types for synchronization variables. Finally, in Section 2.6, other languages with similarities to Flow Java are discussed.

The description of Flow Java in this chapter assumes basic knowledge of Java, as for example available in [3, 14].

2.1 Single Assignment Variables

Single assignment variables in Flow Java are typed and serve as place holders for objects. They are introduced with the type modifier `single`. For example,

```
single Object s;
```

introduces `s` as a single assignment variable of type `Object`.

Initially, a single assignment variable is *unbound* which means that it contains no object. A single assignment variable of type t can be bound to any object of type t. Types for single assignment variables are detailed in Section 2.5. Binding a single assignment variable to an object o makes it indistinguishable from o. After binding, the variable is said to be *bound* or *determined*.

Restricting single assignment variables to objects is essential for a simple implementation, as will become clear in Chapter 4. This decision, however, follows

```
1  ListCell currentTail;
2  ...
3  while(...) {
4    single ListCell newTail;
5    currentTail @= new ListCell(value, newTail);
6    currentTail = newTail;
7  }
```

Figure 2.1: Building a stream

closely the philosophy of Java which separates objects and primitive types such as integers or floats. For example, explicit synchronization in Java is only available for objects. Additionally, Java offers predefined classes encapsulating these restricted primitive types (for example, the class `Integer` storing an `int`). If single assignment properties are needed for a primitive type, one of the predefined classes should be used.

Flow Java uses `@=` to bind a single assignment variable to an object. For example,

```
Object o = new Object();
s @= o;
```

binds `s` to the newly created object `o`. This makes `s` equivalent to `o` in any subsequent computation.

The attempt to bind an already determined single assignment variable x to an object o raises an exception if x is bound to an object different from o. Otherwise, the binding operation does nothing. Binding two single assignment variables is discussed in Section 2.4. Note that the notion of equality used is concerned with the identity of objects only (token equality).

Note that in Flow Java it is the actual object which has the single assignment property, not the variable storing the reference. A variable containing a reference to a single assignment variable can be overwritten by the normal assignment operator.

The reason for the special `@=`-operator instead of overloading the `=`-operator with bind semantics if the left hand side is a single assignment variable is to allow the value of a field or variable declared `single` to be exchanged for a another single assignment variable. The need occurs for example when a stream [36] is built incrementally. A stream is an infinite list with an undetermined single assignment variable at its end. The stream is extended by binding the variable at its end to a new list cell whose tail is a new undetermined variable. For this to be efficient the creator of the stream has to keep track of the current end of the stream.

Consider the example in Figure 2.1, the tail of the stream is kept in the variable `currentTail` (line one). The current tail is bound to a new list cell in line five. The new cell is instantiated with a value and an unbound single assignment variable (`newTail`, on line four), that is the new tail. When the list cell has been added `currentTail` is updated with the new tail (line six). If the `=`-operator had been overloaded to represent bind the update would not be possible, as update would only be available for normal variables. In that case the single assignment tail would have to be encapsulated in a wrapper object. A new wrapper would have to be instantiated for each element added to the stream. The wrapper would be discarded as soon as the next element was added to the stream, producing unnecessary garbage.

2.2 Synchronization

Statements accessing the content of an undetermined single assignment variable automatically suspend the executing thread. These access statements are: field access and update, method invocation, and type conversion (to be discussed in Section 2.5).

For example, assume a class `C` with method `m()` and that `c` refers to a single assignment variable of type `C`. The method invocation `c.m()` suspends its executing thread if `c` is undetermined. As soon as some other thread binds `c`, execution continues and the method `m` is executed for `c`.

As all operations which access and/or need the contents of a single assignment variable suspends until the variable becomes bound, synchronization on its binding is truly automatic.

Automatic synchronization has two immediate benefits: It avoids needless explicit synchronization which artificially limits the available concurrency in a program; The programmer does not have to write error prone explicit code for synchronization or checks for determination.

Binding a synchronization variable forces all variables assigned to by the thread to be written back to main memory. Likewise a thread which resumes computation after having been suspended is guaranteed to flush all variables from its working memory. These semantics are analogous to the standard Java semantics for acquiring and releasing locks. They are needed for single assignment variables to be usable for coordinating access to shared variables.

2.2.1 Example: Network Latency Hiding

The mechanisms in Flow Java described so far allows us to easily implement constructs which concurrently computes a result which is needed only much later. A

```
1   single Integer answer1, ..., answerN;
2
3   issue(answer1);
4   issue(answer2);
5   ...
6   issue(answerN);
7
8   System.out.println(answer1);
9   System.out.println(answer2);
10  ...
11  System.out.println(answerN);
12
```

Figure 2.2: Masking network latency by pipelining

```
1  static public void issue(final single Integer result) {
2    new Thread() {
3      public void run() {
4        result @= syncRequest(); /* Bind */
5      }
6    }.start();
7  }
```

Figure 2.3: Spawning a thread to issue a request asynchronously

typical application of such a construction is in a distributed system. If network latency is high, unrelated requests to a remote node can be pipelined to hide network latency. In such a scenario all requests are issued in sequence and when all requests have been sent the results are processed. Consider the code fragment in Figure 2.2 which illustrates such a scenario, N requests are issued (lines three to six) and the answers are printed (lines eight to eleven). Printing the results will automatically synchronize on the reception of the answers.

Hidden in the `issue()` method is the functionality to spawn a new thread which sends the request, waits for the answer, and then binds the `answer` variable. Figure 2.3 contains the code for `issue()`. The single assignment variable to which the result will be bound to is passed as an argument to the method (line one). In lines two to six a new thread is created. The thread performs the remote request (the request is synchronous) in line four and then binds the answer to `result`.

A similar abstraction (a latch as described by Lea [26]) in plain Java requires roughly twice the number of lines of code, as it uses explicit synchronization for both storing and retrieving the result. Additionally, usage requires awareness that the result is encapsulated in an object. This is in contrast to Flow Java, where

the result is transparently available.

The Flow Java mechanisms described so far can be used to synchronize multiple threads. They allow for easy construction of patterns where one or more threads synchronize on the availability of a value, such as in the pipelining example shown above.

2.3 Futures

Single assignment variables can be shared among threads, sharing makes it possible to construct concurrent programs in which a thread controls other threads via shared variables. Consider a scenario in which two worker threads are given their input via a single assignment variable shared with a third controlling process. The synchronization properties of single assignment variables allows the workers to be programmed as if the input is available as they will automatically suspend until the input becomes determined. In this scenario the binding of the variable functions as a broadcast which sends the input to all waiting threads.

A problem with such a construction is that erroneous or malicious code in one of the workers can cause exceptions in the controller or trick the other worker to start processing unintended input by binding the shared variable. If the variable shared among the threads were a special kind of single assignment variable which only could be bound by the controller, but would retain the same synchronization properties as a normal variable, the integrity of the system could be guaranteed. To this end, Flow Java offers *futures* as secure and read-only variants of single assignment variables.

A single assignment variable x has an associated future f. The future is bound, if and only if the associated variable is bound. If x becomes bound to object o, f also becomes bound to o. Operations suspending on single assignment variables also suspend on futures.

The future associated with a single assignment variable, `v`, is obtained by converting `v`'s type from `single` t to t. This can be done by an explicit type conversion, but in most cases this is performed implicitly (see Section 2.5.2). A typical example for implicit conversion is invoking a method not expecting a single assignment variable as its argument.

In a secure implementation of the scenario described above the controller would be the only thread having access to the single assignment variable. The workers would only share the read-only future, thus protecting the integrity of the system.

Futures allow safe concurrency and synchronization constructs where the ability to bind a single assignment variable can be restricted to a subset of threads sharing a variable and its associated future.

```
1  single Object s;
2  Object sf = s;
3  f @= sf;
4  s @= new Object(); // f is bound
```

Figure 2.4: Why aliasing of futures violate the read only property of futures

2.4 Aliasing

With the operations on synchronization variables described so far, the ways in which to construct threads sharing synchronization variables is restricted. The shared variables have to be created first and then handed to the participating threads, either via their constructors or with a special initialization method. This is sufficient to create any sharing pattern but not very flexible. Using a special initialization method may also require extra explicit synchronization as Flow Java's automatic synchronization cannot be used to synchronize on a variable which has not yet been created. The problem can be eliminated by introducing an operation which aliases two unbound single assignment variables (makes them equal). Aliasing two single assignment variables x and y is done by x `@=` y. Binding either x or y to an object o, binds both x and y to o. Aliasing single assignment variables also aliases their associated futures.

Aliasing allows sharing patterns where the participating threads create single assignment variables as part of their initialization (for example in a static factory method) which returns the shared variable. A main program can then connect communicating threads by aliasing their single assignment variables.

Aliasing is only possible for single assignment variables, not for futures or a combination of a future and a single assignment variable. This restriction is essential to preserve the read only property of futures. If aliasing was available for futures a reference to a future would be sufficient to bind it. Consider the example in Figure 2.4. Assume `f` is a future, by creating a single assignment variable `s` (line one), retrieving its associated future `sf` (line two), we could alias `f` and `sf` (line three). But as we have access to `s` we could then bind `sf` and `f` by binding `s` (line four). This would effectively have removed the read-only attribute of `f`.

Aliasing is only possible for variables declared as having the same type. The reason for this restriction is to avoid inconsistencies in the language semantics. Consider the program fragment in Figure 2.5. If the alias in line three were allowed, would the alias in line four succeed? Looking at the declared types of `a` and `c` the operation is legal. But in that case, the alias in line three has no effect. Another possible interpretation of the fragment is that the alias in line three dynamically constrains the type of `a` to type `B`, this would lead to a type mismatch and a runtime error in line four. Neither of these alternative behaviors would be useful and as

```
1  class A; class B extends A; class C extends A;
2  single A a; single B b; single C c;
3  a @= b;
4  a @= c;
```

Figure 2.5: Inconsistencies if aliasing were allowed for variables of non-identical types

the current semantics have a simple implementation, aliasing of single assignment variables of different types is forbidden.

Aliasing provides a way to express equality among unbound variables, therefore Flow Java extends the equality test `==` such that x `==` y immediately returns true if x and y are two aliased single assignment variables. Otherwise, the equality test suspends until both x and y become determined or aliased.

Aliasing combined with the extended equality test allows Flow Java to borrow synchronization constructs from the field of Concurrent Constraint Programming. The example in Figure 2.6 of a barrier uses a technique which is often referred to as *short circuit* [32]. A barrier can in Flow Java be implemented by giving each thread two single assignment variables *prev* and *succ*. Before a thread terminates, it aliases the two variables. The main thread, assuming it spawns n threads, $t_0 \ldots t_{n-1}$, creates $n+1$ single assignment variables $v_0, \ldots, v_n$. It then initializes *prev* and *succ* as follows: $prev_i = v_i$ and $succ_i = v_{i+1}$ where i $(0 \leq i < n)$ is the index of the thread, thus sharing the variables pairwise among the threads. The main thread then waits for v_0 to be aliased to v_n as this indicates that all threads have terminated.

In Flow Java the algorithm can be implemented as shown in Figure 2.6. The `prev` and `succ` variables are stored in the instance when it is created with the constructor in line four. The actual computation is done in `run()` in line eight and finishes by aliasing `prev` to `succ` in the next line.

The main function `spawn()` creates the threads and waits until they have completed. Each loop iteration creates a new single assignment variable `t` and a thread running the computation. The final check suspends until all threads have terminated and hence all variables have been aliased.

2.5 Types

Variables of type t in Java can refer to any object of a type which is a subtype of t. To be fully compatible with Java's type system, single assignment variables follow this design. A single assignment variable of type t can be bound to any object of type t' provided that t' is a subtype of t.

```
1  class Barrier implements Runnable {
2      private single Object prev;
3      private single Object succ;
4      private Barrier(single Object p, single Object s) {
5          prev = p; succ = s;
6      }
7      public void run() {
8          computation();
9          prev @= succ;
10     }
11     public static void spawn(int n) {
12         single Object first; single Object prev = first;
13         for(int i = 0; i < n; i++) {
14             single Object v;
15             new Thread(new Barrier(prev, v)).start();
16             prev = v;
17         }
18         first == prev;
19     }
20 }
```

Figure 2.6: A short circuit barrier

```
1  class A;
2  class B extends A;
3
4  void doit(single A a) { }
5  single A a;
6  doit(a);
7  single B b;
8  doit(b);
```

Figure 2.7: `single` does not denote a subtype, therefore the calls are legal

Note that the `single` keyword is a type modifier just like `final` and `volatile` and does not denote a subtype. Assume we have a base class `A` and a subclass `B` of `A` as in the program fragment in Figure 2.7. If we define a method as in line four, we can invoke it with arguments of both type `single A` and `single B` as in lines six and eight. If `single` would denote a subtype, `single B` would not be a subclass of `single A` which would make the invocations illegal. That `single` is a type modifier is also the reason why synchronization variables, with the semantics of Flow Java, cannot be expressed in standard Java (see also Section 4.7).

2.5.1 Aliasing

Aliasing two single assignment variables x and x' with types t and t' respectively, is only correct if it statically can be determined that $t = t'$.

2.5.2 Type Conversions

Type conversion can also convert the type of a synchronization variable by converting to a type including the `single` type modifier. Widening type conversions immediately proceed. The operation is always safe in that it can statically be proved to be correct. A narrowing type conversion on an undetermined single assignment variable suspends until the variable is determined. Allowing a narrowing conversion on an undetermined synchronization variable would be equivalent to allowing aliasing of single assignment variables of different types as discussed in Section 2.4.

The type conversion syntax present in standard Java is overloaded to provide access to the future associated with a single assignment variable. The future is obtained by a conversion from `single` t to t.

A conversion from a single assignment variable to a future is implicitly done each time a single assignment variable is passed to a method or assigned to a variable of non-single assignment type. The implicit conversion to a future is essential for seamless integration of single assignment variables. Conversion guarantees that any method can be called, in particular methods in predefined Java libraries. The methods will execute with futures and execution will automatically suspend and resume depending on whether the future is determined or not. This approach is different from other language extensions such as CC++ [9] which enforce that anything visible to components written in the base language must be determined before execution can proceed. An advantage of this approach is that it allows reuse (linking) of code written in the base language without recompilation. The requirement that everything visible to the base language must be determined may lead to unnecessary or premature synchronization which reduces the amount of parallelism available.

The Flow Java approach maximizes the available parallelism at the cost of recompilation and the overhead for automatic synchronization. On the other hand it allows data structures such as collection classes written in the base language to be used for synchronization objects.

A drawback with this approach is that standard Java collection classes cannot be directly used to store single assignment variables. When the variable is added it will automatically be converted to a future as part of the method invocation. This drawback can be circumvented by encapsulating the single assignment variable in a wrapper object.

2.6 Related Approaches

The design of Flow Java has been inspired by concurrent logic programming [36] and concurrent constraint programming [33, 32, 37], and distributed programming [17]. The main difference is that Flow Java does not support terms or constraints in order to be a conservative extension of Java. On the other hand, Flow Java extends the above models by futures and types. The closest relative to Flow Java is Oz [37, 28, 41], offering single assignment variables as well as futures. The main difference is that Oz is based on a constraint store as opposed to objects with mutable fields and has a language specific concurrency model. As Oz lacks a type system, conversion from single assignment variables to futures is explicit.

Another closely related approach is Alice [1] which extends Standard ML by single assignment variables (called promises) and futures. Access to futures is by an explicit operation on promises but without automatic type conversion. Alice and Flow Java share the property that futures are not manifest in the type system.

The approach to extend an existing programming language with either single assignment variables or futures is not new. Multilisp is an extension of Lisp which supports futures and threads for parallel programming [16]. Here, futures and thread creation are combined into a single primitive similar to the thread spawning construct in Section 2.2.1. Multilisp is dynamically typed and does not offer single assignment variables and in particular no aliasing. Another related approach is Id with its I-structures [4]. I-structures are arrays of dataflow variables similar to single assignment variables without aliasing. A field in an I-structure can be assigned only once and access to a not yet assigned field will block.

Thornley extends Ada as a typed language with single assignment variables [39]. The extension supports a special type for single assignment variables but no futures and hence also no automatic conversion. The work does not address to which extent it is a conservative extension to Ada, even though it reuses the Ada concurrency model. It supports neither aliasing nor binding of an already bound single assignment variable to the same object. A more radical approach by the same

author is [40]. It allows only single assignment variables and hence is no longer a conservative extension to Ada.

Chandy and Kesselman describe CC++ in [9] as an extension of C++ by typed single assignment variables without aliasing together with a primitive for thread creation. CC++ does not provide futures. Calling a method not designed to deal with single assignment variables suspends the call. This yields a much more restricted concurrency model.

The approach to extend Java (and also C#) with new models for concurrency has received some attention. Decaf [34] is a confluent concurrent variant of Java which also uses logic variables as the concurrency mechanism. Decaf does not support futures and changes Java considerably, hence requiring a complete reimplementation. Hilderink, Bakkers, et al. describe a Java-based package for CSP-style channel communication in [19].

An extension to C# called *Polyphonic C#* is described in [6], where the system is implemented by translation to standard C#. Polyphonic C# adds a mechanism for asynchronous message sending by adding asynchronous functions which spawn new threads. Polyphonic C# also adds a construct for receiving messages to the language. Message reception is through a construct called a *chord* which is a method definition associated with a list of messages. An invocation of such a method will suspend until the messages in the list have been received.

While the two latter approaches use models for concurrent programming different from synchronization variables, they share the motivation to ease concurrent programming in Java or C# with Flow Java.

Another Java dialect is `jcc` [31] which redefines Java's concurrency model. In `jcc` threads are isolated from each other and communicate through message sending. The message contents are copied during sending, only immutable objects are shared among threads. The language compiles to standard JVM bytecode and can transparently use Java libraries as long as they do not make use of threads.

Chapter 3

Programming in Flow Java

This chapter illustrates the current best practice in Flow Java programming by describing the implementation of a non-trivial application, a lift controller.

The chapter starts with an overview of concurrency abstractions in Section 3.1. Section 3.2 describes some of these abstractions as adapted to Flow Java. The design of the lift controller is then described in terms of these abstractions in Section 3.3. Finally, Section 3.4 relates the implementation to hypothetical implementations in other languages, including standard Java, and discusses the key insights gained in implementing the controller.

3.1 Concurrency Abstractions

One of the motivations for Flow Java is to avoid explicit synchronization for shared data. A powerful approach to writing concurrent programs which avoids explicit synchronization is to construct programs as a set of communicating tasks. A task is a separate thread of control which sends and accepts messages. Messages sent to a task are stored in a FIFO queue. Messages can selectively be read from this queue by the task.

The task abstraction can be found in many languages which have been designed with concurrency support built in such as Ada [21], Concurrent C [13], and Erlang [2] (Where, for the last two, tasks are called *processes*). Here the task abstraction is visible at the syntactic level with constructs for defining task types, accepting and sending messages. Ada and Concurrent C have task types, this is in contrast to Erlang where a message can be any Erlang term. Task types characterize tasks by the types of messages it accepts. Tasks accepting the same set of messages are said to be of the same task type.

3.2 Tasks in Flow Java

Flow Java tasks are typed and internally structured as state machines. Messages sent to the task are buffered in a message queue from which the task selectively can read messages. Compared to other languages which has task support built in, Flow Java requires the programmer to implement his own task abstractions. This section starts by describing how states and messages are represented in Section 3.2.1. Then the implementation of message delivery and message queues is described in Section 3.2.2. Finally task creation is described in Section 3.2.3.

3.2.1 Messages and States

Messages are instances of message classes. A message instance encapsulates the information contained in it.

The states of the state machine are represented as instances of a base state class, `State`. The `State` class contains a protected field `self` which is the task's message queue.

Messages are delivered to a state by calling a handler method with an argument of the message type, that is, `handle(A msg)` handles a message `msg` of type `A`. The method returns either `null`, to indicate that the state does not accept the message, or a state object to indicate that the message was accepted. This setup allows for simple construction of state machines by defining an abstract base class with a method for each possible message. The methods in the base class all return `null`. States are built by subclassing the base class overriding the methods for the messages which are accepted in that particular state. The base class which defines all messages accepted by the task is called the *task type*, as it serves the same purpose as Ada task types [21].

It is desirable for the framework delivering messages to states to be fully generic. For this to be possible the framework cannot directly call the `handle` method of the state as it would require an explicit type conversion to the task type of the recipient. Therefore all message classes implement the interface `Message` which specifies a single `deliver(Object state)` method. This method is responsible for converting its argument to the correct task type and invoking the `handle` method for the message.

3.2.2 Message Handling

The message queue implements buffering of messages which have not been processed yet, as well as storing messages which are not accepted by the current state. The message queue preserves the order of sent messages. It guarantees that a task will receive the oldest message which is accepted in the current state.

The message queue implementation can be decomposed into four functional units:

- Adding messages to the message queue.
- Buffering messages not yet delivered to the task.
- Delivering messages to the task.
- Storing messages which were not accepted by the task.

Buffering of messages not yet delivered to the task is handled by a linked list with an undetermined single assignment variable at its end. Such a list is called a *stream* [36] (see also Section 2.1). The synchronization properties of Flow Java will automatically suspend a thread iterating over the list when it reaches the unbound synchronization variable at its end.

New elements can be appended to the list by binding the single assignment variable at the end to a new list cell, the tail of which is unbound. The binding of the tail element will automatically awaken a thread suspended on the tail.

Streams are easily implemented in Flow Java, they are simply a list cell whose elements are declared as `single` and two access methods to access the head and tail, see Figure 3.1. The stream can grow arbitrarily long and is therefore suitable for buffering messages delivered to the task.

A problem with streams is that it is hard to allow multiple threads to simultaneously append elements to the end of the stream without introducing race conditions. This problem can be avoided by introducing an object which holds a reference to the end of the stream and a synchronized method which creates and appends a new list element. The synchronized method will then arbitrate among multiple threads appending to the stream. The arbitrating object is usually called a *port* [22] and is the mechanism by which messages are added to the message queue.

The implementation of a port is shown in Figure 3.2. The `send()` method (line eight) appends a new element to the stream by: creating a new unbound tail (line nine); binding the existing tail to a new stream element containing the message which is sent `o` (line ten); and finally update the private field containing the tail to the new tail (line eleven).

Delivery of newly arrived messages and storage of messages not yet accepted by the task are intertwined. Undeliverable messages are stored in a doubly linked list which is organized as a queue. The main loop which drives the task is shown in Figure 3.3. The loop first tries to deliver the messages in the queue (line seven), if it is accepted the message is unlinked from the list (line eleven) and attempted delivery restarts from the head of the queue (line five), otherwise the next message

```
1  public class Stream {
2      private single Object head;
3      private single Stream tail;
4
5      public Stream(single Object o, single Stream tail) {
6          this.head = o;
7          this.tail = tail;
8      }
9
10     public single Stream get_tail() {
11         return tail;
12     }
13
14     public single Object get_head() {
15         return head;
16     }
17 }
```

Figure 3.1: A stream

```
1  public class Port {
2      private single Stream tail;
3
4      public Port(single Stream stream) {
5          tail = stream;
6      }
7
8      public synchronized void send(Object o) {
9          single Stream new_tail;
10         tail @= new Stream(o, new_tail);
11         tail = new_tail;
12     }
13 }
```

Figure 3.2: A port

is tried (line nine). When the list is exhausted of deliverable messages the stream is accessed (line 19). The driver loop does not access the stream directly but uses an iterator with the operations `read()`, to return the current message (if no message is available it suspends until one is available), and `next()` (advances the internal position of the iterator to the next message). When a message has been read it is delivered to the current state (line 20), if it is accepted, message delivery starts over from the beginning of the queue (line 27).

3.2.3 Task Creation

Tasks are encapsulated by instances of the `Task` class. The class has a single public `create` method, shown in Figure 3.4, which takes an initial state. The method creates a port and a stream and then creates a new thread which starts executing the infinite message delivery loop described in Section 3.2.2. The method then returns the port.

3.3 A Lift Controller

The use of a lift controller to demonstrate concurrent programming is inspired by the examples in Erlang and Oz in [2] and [41] respectively. This example is modeled after a lab given in the course 2G1512 at KTH [35].

The lift controller controls a system of three lifts in a six floor building. Each floor except the bottom and top floors have two buttons, one for calling a lift to go downwards and one to go upwards (the top and bottom floors only have a single down respectively up button). Inside the lift-cabins there are buttons for each floor.

There is a task for each lift cabin handling the low level control of the lift, that is opening/closing of the doors and going to a certain floor. The low level controller is in the implementation called a `Cabin`. For each lift there is also a high-level controller managing a record of scheduled stops, called a *Controller*. There is one task, called *Floor* for each floor which receives a message when one of the call buttons on that floor are pressed. It is responsible for asking all high level lift controllers for an estimate of the time required to serve the request and give the request to the lift with the smallest waiting time. The low level controller communicates with the high-level controller receiving orders to go to a floor and telling the high level controller when it has arrived.

This section describes the implementation of one of the cabins in Section 3.3.1, it then discusses system initialization in Section 3.3.2. Section 3.3.3 shows how Flow Java's automatic synchronization can be used to simplify the implementation by reducing the number of states in a task.

```
1  public final void run() {
2      while(true) {
3          /* Try the queue */
4          ListIterator q = queue.listIterator(0);
5          while(q.hasNext()) {
6              Msg msg = (Msg)q.next();
7              State new_state = msg.deliver(state);
8              if(new_state == null) {
9                  continue; // This message is ignored
10             } else {
11                 q.remove(); // This message was accepted
12                 state = new_state;
13                 q = queue.listIterator(0);
14             }
15         }
16
17         /* Nothing found in the queue, try the stream */
18         while(true) {
19             Msg msg = (Msg)stream.read();
20             State new_state = msg.deliver(state);
21             stream.next(); // Advance the queue
22             if(new_state == null) {
23                 queue.addLast(msg); // Message was not accepted,
24                 // queue it
25             } else {
26                 state = new_state; // Message was accepted
27                 break;
28             }
29         }
30     }
31 }
```

Figure 3.3: The message delivery loop in a task

```
1   public final static Port create_task(State initial) {
2       single Stream s;
3       Port p = new Port(s);
4       initial.install_port(p);
5       Thread handler = new Thread(new Task(new StreamIterator(s),
6                                            initial));
7       handler.start();
8       return p;
9   }
```

Figure 3.4: Method for creating a new task

3.3.1 A Sample Task: The `Cabin`

To illustrate the implementation of a task we will now describe the `Cabin` task which controls a single lift cabin. The task implements the state machine in Figure 3.5. The machine has five states:

stopped The elevator is standing still with the doors closed. In this state it accepts the `goto(n)` message which is an order to go to floor `n`. The message is represented by the class definition in Figure 3.6. Note how the message calls a method of the correct task type.

running In this state the cabin is moving to a floor. To animate the simulation it uses a timer provided by the system to periodically receive a `timeout` message. On reception of the timeout it updates the cabin position and if the cabin has arrived at the desired floor it transitions to the opening state.

opening In this state the lift cabin is in position and is opening its doors. It uses the same timeout mechanism as the running state.

open In this state the cabin's doors are open, here a single longer timeout suffices. When it expires the state changes to closing.

closing This state is similar to opening but here the doors are closing. When the doors are closed it transitions to stopped. When the transition occurs it sends an `arrived` message to its controller.

The `Cabin`'s task type is defined in Figure 3.7. Apart from being a specification of the accepted messages it also defines information shared between all states, in this case: a reference to the hardware controlling the physical motors, buttons, etc (line four); the lift identity (line two) which is needed for interacting with the external lift hardware; the controller responsible for this lift (line three); the

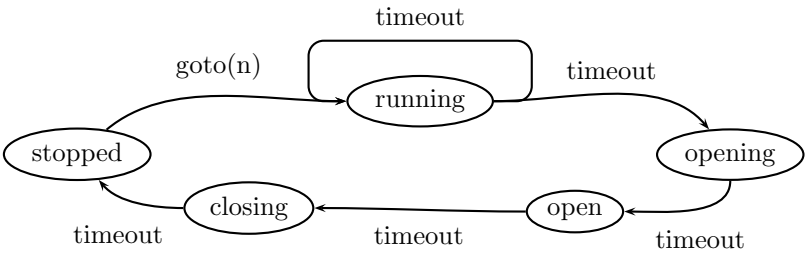

Figure 3.5: The *Cabin* state machine

```
1   public class Goto implements Message {
2       public int floor;
3       Goto(int floor) {
4           this.floor = floor;
5       }
6
7       public State deliver(State in) {
8           return ((Cabin.CabinState)in).handle(this);
9       }
10  }
```

Figure 3.6: Representation of a `goto(n)` message

current cabin position (line five). The constructor in line 15 is used when the initial cabin state is created during initialization. The constructor in line seven preserves the shared information and is used when the state machine makes a transition. The two non-accepting handlers for `goto(n)` and `timeout` are defined on line 22 and 26.

In Figure 3.8 the code for the stopped state is shown. As the stopped state is the initial state, it defines a constructor initializing the shared information (line two) as well as a constructor used when changing from the closing state (line five). The stopped state only accepts the `goto(n)` message, the handler is defined in line nine. It creates and returns a new running state using a constructor which takes the current state (to preserve the current shared information) and a destination floor.

The running state is defined as in Figure 3.9. As this state is only entered when the stopped state does a transition on a `goto(n)` it has only a single constructor (line three). The constructor uses a timer provided by the system to send itself a `timeout` message after 50 ms (line six). The handler for the timeout is defined on line nine. If the cabin has arrived at the desired floor it changes to the open state (line eleven). Otherwise it schedules a new timeout (line twelve) and updates the internal position (line 14) as well as the hardware (line 15).

```
1  abstract public class CabinState extends State {
2      protected int no;
3      protected Port controller;
4      protected Hardware hw;
5      int pos;
6
7      CabinState(CabinState old) {
8          super(old);
9          no = old.no;
10         controller = old.controller;
11         hw = old.hw;
12         pos = old.pos;
13     }
14
15     CabinState(int no, Hardware hw, Port controller, Timer t) {
16         this.no = no;
17         this.controller = controller;
18         this.hw = hw;
19         this.pos = 0;
20     }
21
22     public State handle(Goto msg) {
23         return null;
24     }
25
26     public State handle(Timeout msg) {
27         return null;
28     }
29 }
```

Figure 3.7: The cabin task type

```
1  public class Stopped extends CabinState {
2      Stopped(int n, SockReader sr, Port controller, Timer t) {
3          super(n, sr, controller, t);
4      }
5      Stopped(CabinState old) {
6          super(old);
7      }
8
9      public State handle(Goto msg) {
10         return new Running(this, msg.floor);
11     }
12 }
```

Figure 3.8: The cabin stopped state

```
1  public class Running extends CabinState {
2      int dest;
3      Running(CabinState old, int dest) {
4          super(old);
5          this.dest = dest;
6          Timer.delay(50, self, new Timeout(null));
7      }
8
9      public State handle(Timeout msg) {
10         if(pos == dest) // We have arrived, open doors
11             return new Opening(this);
12         Timer.delay(50, self, new Timeout(null));
13
14         pos += dest < pos ? -1 : 1;
15         hw.setpos(no, pos); // Animate
16         return this;
17     }
18 }
```

Figure 3.9: The cabin running state

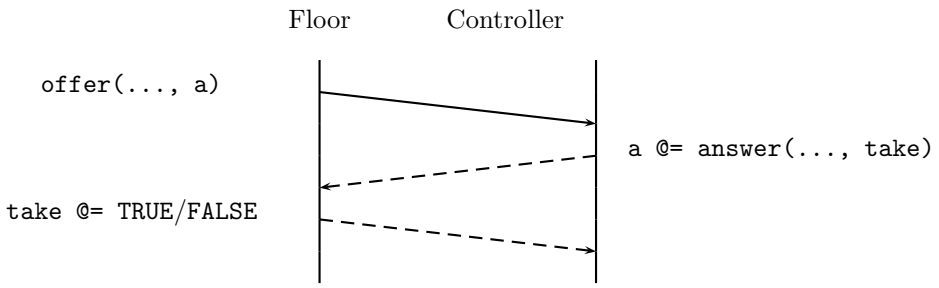

Figure 3.10: An `offer` dialog

3.3.2 System Initialization

In the lift controller the low-level controller and the high-level controller communicate asynchronously. This means that they both must have access to each other's ports thus creating a cyclic dependency. This dependency is easily handled by exploiting single assignment variables. The initial states are simply given port futures which are then bound to the real ports by the main thread when both tasks have been created.

3.3.3 Inter-task Synchronization

When tasks communicate asynchronously explicit message sending is needed. This is the case with the controller and the cabin, the controller sends a `goto(n)` to the cabin. The cabin will, when it has reached the floor, send an `arrived` message back to the controller.

If RPC (Remote Procedure Call) semantics is desired this can be implemented by introducing an extra state which only accepts the reply message (this corresponds to using a separate receive statement in Erlang or Ada). Single assignment variables provide a simple way to implement RPCs. Consider the interaction between a controller and a floor in the lift controller, see Figure 3.10. The floor sends an `offer(Direction d, int floor, single Answer a)` to the controller, where `a` is an unbound variable. The controller replies by binding `a` to an `answer(int time, single Boolean take)` message (dashed lines represents communication through single assignment variables), where `time` is the controller's estimate of the time it will require to serve a request for floor `floor` in direction `d`. Using this technique makes the handler for the `offer` message very simple, as shown in Figure 3.11. No extra states are needed as the handler synchronizes on the binding of `take` in line five.

```
1  public State handle(Offer msg) {
2      // Tell estimated time
3      msg.answer @= new Answer(time(msg.floor, msg.dir));
4
5      if(msg.answer.take.booleanValue())
6          stop_at(msg.floor, msg.dir); // Schedule the stop
7      return this;
8  }
```

Figure 3.11: The handler for the `offer` message in a cabin

```
1  // Send a offer to each lift, then collect the answers
2  for(int i = 0; i < noof_lifts; i++) {
3      single Answer a;
4      ans[i] = a;
5      lifts[i].send(new Offer(floor_no, msg.dir, a));
6  }
7
8  // Pick the fastest
9  int fastest = 0;
10 for(int i = 1; i < noof_lifts; i++) {
11     if(ans[i].time < ans[fastest].time) {
12         ans[fastest].take @= Boolean.FALSE;
13         fastest = i;
14     } else
15         ans[i].take @= Boolean.FALSE;
16 }
17 ans[fastest].take @= Boolean.TRUE;
```

Figure 3.12: Selecting and notifying the fastest lift

As mentioned before, the floor task sends an `offer(Direction d, int floor, single Answer a)` to each of the controllers and then assigns the request to the fastest lift. The automatic synchronization in Flow Java and the previously described technique for RPCs makes the selection surprisingly easy. The implementation also allows for the controllers to handle offers in parallel. The program fragment in Figure 3.12 shows how this is implemented. An offer is sent to each lift (line five) which contains the unbound variable created on line three. The future associated with the variable is stored in the array `ans` (line four). When an offer has been sent to each lift the array is traversed and lifts that are slower than the currently fastest lift are told to ignore the request (line 15). Finally when all lifts have been considered the one receiving the request is informed (line 17).

3.4 Implementation in Other Languages

Implementing a system as a set of communicating tasks boils down to three main issues:

Message delivery. Flow Java uses a state machine, the states of which are represented by instances of a state class which is derived from the task type. The state class overrides the message handling methods for the messages it accepts in that state.

Message Passing. Each task has a port to which messages are sent. The message delivery framework reads one message at a time from the stream associated with the port and tries to deliver it to the task. Messages which are not accepted in the current state are queued.

Initialization. The task may have cyclic dependencies if two communicating tasks perform asynchronous communications as both processes need to know the other task's port. This is handled by using futures for the counterpart's port during initialization. Cyclic dependencies can also be broken by passing asynchronous replies via single assignment variables instead of an explicit message send.

Implementing the lift controller in a language which provides built-in support for sending and receiving messages among tasks simplifies the implementation. Languages which have this support are for example: Ada, Concurrent C, and Erlang. Using one of these languages removes the need for the infrastructure developed in Section 3.1, that is the message representation and the classes representing the task's states as well as the code for message handling and queuing. What requires more work to emulate in one of these languages are the initialization and the mechanism for asynchronous replies using synchronization variables. It would require either explicit message sending or the implementation of an abstraction of synchronization variables in the base language.

An implementation in Flow Java's closest relative, Oz, can use the same mechanisms as Flow Java for initialization and replies but would, just as Flow Java, have to explicitly emulate the message handling.

An implementation in standard Java could emulate the Flow Java implementation by implementing synchronization variables by either creating type-specific wrapper classes emulating the future or sacrifice the static typing with a generic class as in [25]. See Section 5.3 for an estimate of the overhead for such an approach. Even if the programmer chooses to use a type specific wrapper it does not allow direct access to object fields (see Section 4.7 for a discussion on the feasibility of emulating futures in standard Java).

Chapter 4

Implementation

Flow Java can be implemented by extending an existing Java implementation. Compared to Java a Flow Java implementation adds support for synchronization variables. This includes primitives for aliasing, binding and synchronization in the runtime system. The compiler is also extended to parse the syntax specific to Flow Java and to generate code supporting automatic synchronization.

Additionally the Flow Java extensions must be compatible with the garbage collection and multi-threading facilities of the underlying Java implementation. The Flow Java implementation described in this thesis uses a novel two-level architecture for the representation of single assignment variables. The architecture separates concurrency issues from the underlying representation of aliased variables.

The extensions described in this chapter can easily be implemented in any Java runtime environment using a memory layout similar to C++. The extensions are not limited to Java, they can equally well be applied to other object-oriented languages such as C# or C++.

The Flow Java implementation is based on the GNU `GCJ` Java compiler and the `libjava` runtime environment. They provide a virtual machine and the ability to compile Java source code and byte code to native code. The runtime system uses the same object representation as C++. Garbage collection is provided by a conservative collector.

The runtime system does not distinguish between futures and single assignment variables as this distinction is maintained by the compiler. Both kinds of synchronization variables are in the runtime system represented by *synchronization objects*.

This chapter starts by giving an overview of the `GCJ`/`libjava` runtime en-

vironment in Section 4.1 and the implementation of synchronization objects in Section 4.2. The implementation of synchronization objects is factored into two parts: one part dealing with concurrency and synchronization aspects, described in Section 4.3, and one part describing three different strategies for representing synchronization objects (Section 4.4). The prior description is parametric with respect to the underlying variable representation strategy. Section 4.5 describes the compiler support for Flow Java. Flow Java-specific optimizations are discussed in Section 4.6. Section 4.7 discusses alternative ways of implementing Flow Java. The description of the implementation concludes with a summary in Section 4.8.

4.1 The `GCJ/libjava` Runtime Environment

The Flow Java implementation is based on the GNU `GCJ` Java compiler and the `libjava` runtime environment. They provide a virtual machine and the ability to compile Java source code and byte code to native code. This section describes the base system.

4.1.1 Object Representation

The `GCJ/libjava` implementation uses a memory layout similar to C++. An object reference points to a memory area containing the object fields and a pointer, called *vptr*, to a virtual method table, called *vtab*. The *vtab* contains pointers to object methods and a pointer to the object class. The *vtab* also contains a garbage collector descriptor. The memory layout is the same for classes loaded from byte code and native code. Instances of interpreted classes store pointers in their *vtab* to wrapper methods which are byte code interpreters. The byte code interpreters are instantiated with byte code for the methods during class loading.

4.1.2 Memory Management

`libjava` uses a conservative garbage collector developed by Hans Boehm [7]. The collector is originally intended to be used for C and C++ but works equality well with Java as the object representation in C++ and `libjava` is the same.

4.1.3 Suspension

The `GCJ/libjava` runtime uses operating system threads. For example, on x86-linux `pthreads` [20] are used. Explicit suspension and resumption in Java is implemented by the `wait()`, `notifyAll()`, and `notify()` methods. The methods are present in all Java objects. A thread suspends if it calls `wait()` on an object.

The thread resumes execution when another thread calls either `notifyAll()` or `notify()` on the same object. The difference between `notifyAll()` or `notify()` is that `notifyAll()` will awaken all threads suspended on an object, `notify()` will only wake up one thread, which thread waken up is not specified.

The wait/notify functionality is made available to the Flow Java runtime as two functions, `prim_wait`/`prim_notifyAll`, each taking the waiting/notified object as an argument. The functions interface with the underlying system-level thread implementation.

4.1.4 Monitors

Orthogonal to the wait/notify mechanism is the monitor which is present in each Java object to support synchronized methods. The lock associated with the monitor is made available to the Flow Java runtime by the two functions `lock` and `unlock`.

4.2 Implementing Synchronization Objects

Synchronization objects are allocated on the heap and contain minimal information to support aliasing.

All objects which are aliased to each other are in some sense equivalent, we call the set of objects which are aliased to each other the *equivalence class*. The implementation strategies discussed in this chapter select one element from the equivalence class as *leader*.

Equivalence classes are maintained in two layers. An upper layer (described in Section 4.3) handles the language level operations and makes them safe and atomic. The lower layer (described in Section 4.4) handles the representation and maintenance of equivalence classes.

4.2.1 Binding

When a synchronization object is bound to an object o, its internal information is updated to point to o. Binding is implemented by the primitive `bind(a,b)`. It is infeasible to allocate synchronization objects which are large enough to contain the largest possible object in the system. Therefore, a synchronization object contains a pointer to its value. This in contrast to systems using tagged pointers where logic variables are simply overwritten during binding.

4.2.2 Aliasing

Aliasing creates or extends an equivalence class by merging two, possibly singleton, equivalence classes with the primitive `alias(a,b)`. The aliasing operation modifies the internal information of the synchronization objects to maintain the equivalence relation (equality).

4.2.3 Synchronization

The runtime system sometimes suspends execution until a synchronization object becomes determined. The primitive `waitdet(r)` suspends until its argument becomes determined and then returns the determined object.

Synchronization objects do not use the same virtual method table as ordinary objects. Entries in the *vtab* of a synchronization object point to stub functions which are created by the runtime system during class loading. The stub suspends the executing thread until the object becomes determined, using `waitdet(r)`, and then restarts the method invocation. This provides automatic synchronization of method invocations without a runtime penalty for method invocations on ordinary objects.

4.3 Concurrency and Aliasing

Atomic aliasing and binding are required by Flow Java. In contrast to other systems supporting logic variables (for example, PARMA [38], WAM [42, 5], or even Mozart [29, 27]), the runtime system of Flow Java provides concurrency by using operating system threads. The primitives implementing synchronization and atomic bind/alias are more complex as the operations must be made safe and atomic without resorting to a "stop the world" approach.

This section describes how binding, aliasing, and synchronization operations can be implemented using `lock` and `unlock` (see Section 4.1.4). First the low-level operations manipulating equivalence classes are described in Section 4.3.1. The invariants applying to the use of these operations are described in Section 4.3.2. The operations on synchronization variables are then described in terms of these low-level operations in Sections 4.3.3 to 4.3.7.

4.3.1 Operations

The operations manipulating equivalence classes do so through a set of low-level primitives:

`ll_is_so(r)` A predicate which tests whether `r` is a synchronization object.

`ll_bind(a, b)` updates the internal representation of the equivalence class `a` to bind it to `b`.

`ll_alias(a, b)` updates the representation of `a` and `b` by merging their equivalence classes.

`ll_leader(r)` returns the leader of the equivalence class `r`.

`ll_compress(orig, new)` Shortens the reference chain of `orig` to point directly to `new` if the representation needs or supports it.

4.3.2 Invariants

The following invariants apply to the use of the low level primitives:

1. The leader of a bound object is the object itself.
2. An equivalence class is only modified if the lock for its leader is held by the modifying thread.
3. Leader locks are acquired in order of increasing address of the leader.
4. Binding an equivalence class notifies all threads suspending on its leader by `prim_notifyAll`. The lock of the leader is still held by the binding thread.
5. If two equivalence classes are merged, the leader at the highest address is notified by a call to `prim_notifyAll` while its lock is still being held by the modifying thread.
6. All low level primitives except `ll_leader(r)` and `ll_is_so(r)` take leaders as arguments.

4.3.3 Bind

The `bind(a,b)` primitive (defined in Figure 4.2) binds the synchronization object `a` to `b`. It first acquires the determined value of `b` by using `waitdet()` (which will suspend if `b` is not already determined). It then uses `ll_leader(a)` to find the leader of `a` and acquire its lock. If another thread is modifying the equivalence class this may require multiple iterations.

When the lock has been acquired the binding is checked for validity. The equivalence class is updated by `ll_bind()`. `prim_notifyAll` is then called on the leader to wake up all threads suspended on the leader. Finally the lock for the leader is released.

```
1   jobject alias(jobject a, jobject b)
2   {
3     bool as, bs;
4     jobject low, high;
5     while(true) {
6       a = ll_leader(a);
7       b = ll_leader(b);
8       as = ll_is_so(a); bs = ll_is_so(b);
9       if(!as && !bs) {
10        if(a == b)
11          return a;
12        throw TellFailureException;
13      } else if(as && bs) {
14        if(a < b) {
15          low = a; high = b;
16        } else {
17          low = b; high = a;
18        }
19        lock(low); lock(high);
20        if(low == ll_leader(low) && high == ll_leader(high))
21          break;
22        unlock(high); unlock(low);
23        continue;
24      } else {
25        if(as)
26          return bind(b, a);
27        return bind(a, b);
28      }
29    }
30    if(!valid_alias(low, high)) {
31      unlock(high); unlock(low);
32      throw TellFailureException;
33    }
34    ll_alias(low,high);
35    prim_notifyAll(high);
36    unlock(high); unlock(low);
37    return low;
38  }
```

Figure 4.1: The primitive `alias`

```
1  jobject bind(jobject a, jobject b)
2  {
3      b = waitdet(b);
4      while(true) {
5          a = ll_leader(a);
6          lock(a);
7          if(ll_leader(a) == a)
8              break;
9          unlock(a);
10     }
11     if (!bind_is_valid(a, b)) {
12         unlock(a);
13         throw error;
14     } else if(a == b) {
15         // Nothing to do
16     } else {
17         ll_bind(a, b);
18         prim_notifyAll(a);
19     }
20     unlock(a);
21     return b;
22 }
```

Figure 4.2: The primitive `bind`

```
1  jobject waitdet(jobject o)
2  {
3      if(!ll_is_so(o))
4          return o;
5      jobject t = o;
6      while(ll_is_so(o)) {
7          o = ll_leader(o);
8          lock(o);
9          if(ll_is_so(o) && ll_leader(o))
10             prim_wait(o);
11         unlock(o);
12     }
13     ll_compress(t, o);
14     return o;
15 }
```

Figure 4.3: The primitive `waitdet`

4.3.4 Aliasing

Aliasing of synchronization objects is implemented by `alias`. The definition of `alias` is shown in Figure 4.1. In order to be thread safe, `alias` iteratively acquires the locks of the two leaders (the `while`-loop in line five). The lock of the leader with the lowest address is acquired first to prevent deadlock (the ordering is handled by the `if` on line 14, the locking in line 19). When both locks have been acquired, a check is made to verify that no other thread has modified the equivalence classes. If so the locks are released and the aliasing starts anew (line 20). If one of the leaders is determined the operation is turned into a bind (line 25-27). If both objects are undetermined leaders the aliasing operation is tested for validity (line 30) and the equivalence classes are merged (line 34). The leader with the highest address is then notified (line 35) and the locks are released (line 36).

4.3.5 Synchronization

The `waitdet` primitive suspends the currently executing thread until its argument becomes determined.

Only the `bind(a,b)` primitive changes the status of a synchronization object from unbound to bound. The invariants maintained by `alias(a,b)` and `bind(a,b)` (invariants four and five) guarantee the following property: if the leader for an equivalence class changes or all members become bound, `prim_notifyAll` is called on the leader when its lock is held by the thread doing the modification. Therefore `waitdet(r)` can be implemented as shown in Figure 4.3. It is based on a loop which uses `ll_leader(r)` and terminates when a determined object is found (line six). If an undetermined leader is found, the lock associated with the leader is acquired (line eight). If the object is still undetermined (line nine) `prim_wait` is called to wait for the leader to be updated (line ten). When `prim_wait` returns, the lock is released (line eleven) and the loop continues. Requiring the thread to acquire the lock before calling `prim_wait` guarantees that no binding or aliasing notifications are lost. For representations which can make use of path-compression `ll_compress` is executed as a final step (line 13).

4.3.6 Method Invocation

As described in Section 4.2.3 synchronization objects do not use the same virtual method table as ordinary objects. Instead they have a synchronization virtual method table (*sync. vtab.*) which contains pointers to stub methods with code as in Figure 4.4. The synchronization virtual method table is constructed by the runtime system during class loading. The class loader allocates and initializes a memory area for each class containing the table and the stubs.

```
1   void meth_stub(jobject *this, ...)
2   {
3     this = waitdet(this);
4     goto METHOD_ADDRESS;
5   }
6
```

Figure 4.4: A synchronization virtual method table (*sync. vtab.*) stub

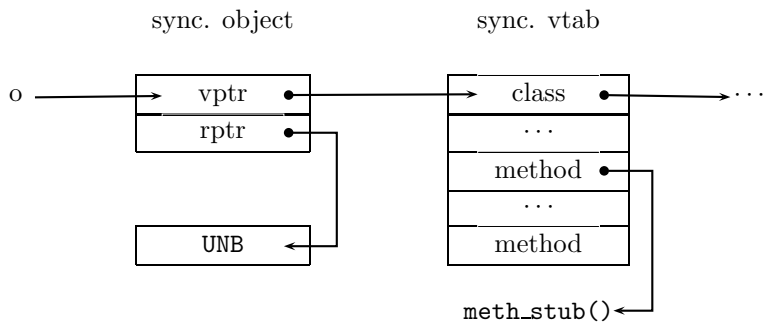

Figure 4.5: Unbound synchronization object in Flow Java.

The stub uses the runtime primitive `waitdet` (see Section 4.3.5) which suspends until its argument becomes determined. When the object is determined the stub replaces the object reference pushed on the stack as part of the method invocation with the determined object. The invocation is then continued by dispatching to the corresponding method in the standard *vtab*. This achieves transparent redirection of method invocations without a runtime penalty when synchronization objects are not used. Figure 4.5 shows an unbound synchronization object.

4.3.7 Reference Equality

The reference equality operators `==` and `!=` are implemented by the runtime primitive `refeq` and its negation. `refeq` suspends until either both its arguments have been determined or aliased to each other. The primitive `refeq` is defined as shown in Figure 4.6. The code makes use of the helper function `lock_leader` (Figure 4.7) which iteratively acquires the lock for the leader of its argument.

The `refeq` primitive executes in an infinite loop, first it acquires the leaders of its arguments (line five and six). If both objects have the same leader, it returns `true` (line nine). If both arguments are referring to determined objects it returns `false` (line twelve). If only one of the arguments is determined it suspends on the

undetermined argument (line 17 and 22).

Recall that suspension is implemented using the primitives `prim_wait` and `prim_notifyAll` (Section 4.1.3) which only operate on one object at a time. If the reference equality function is given two undetermined objects it makes use of the invariant that the aliasing operation updates the *rptr* of the synchronization object at the higher address to point to the object at the lower address. The explicit ordering allows the operators to only suspend on the object at the higher address as this is the object that will receive the notification if both objects are aliased to each other (line 28). This is sufficient as the only case when the operators return before both objects are determined is when they are aliased to each other.

4.4 Maintaining Equivalence Classes

The description of the operations in Section 4.3 defined the semantics of the low level operations (named `ll_<`*name*`>`). This section describes three different schemes for implementing the underlying representation. By construction of the high level operations the operations modifying equivalence classes (`ll_bind` and `ll_compress`) can assume exclusive access. The only exception is `ll_compress` which is allowed to shorten a reference chain without holding the lock as it does not change the interpretation of a determined equivalence class.

This section describes three representations for equivalence classes. First a scheme based on a forwarding pointer is described in Section 4.4.1. This is the representation used by the standard Flow Java implementation. Then an variant of Taylor's scheme [38] adapted to a language with update and token equality (non structural equality) is described in Section 4.4.2. Then finally Section 4.4.3 shows an optimization to Taylor's scheme in a concurrent setting.

4.4.1 Forwarding

This scheme is similar to the forwarding pointer scheme used in the WAM [5]. An equivalence class is represented as a tree of synchronization objects rooted in the leader. A bound equivalence class has a determined object at its root.

Synchronization objects are in this scheme allocated as two-field objects containing a redirection-pointer field *rptr* and the *vptr*. Normal objects also have a *rptr*, the *rptr* is used to indicate binding status and is also used as a forwarding pointer. In standard Java objects the *rptr* points to the object itself.

The *rptr* of a synchronization object can be: a sentinel `UNB` (for unbound), a pointer to a determined object, or a pointer to a synchronization object. A sentinel is used instead of `null` as otherwise an undetermined synchronization object would be indistinguishable from an object bound to `null`.

```
1   jboolean refeq(jobject a, jobject b)
2   {
3     while(true)
4       {
5         a = ll_leader(a);
6         b = ll_leader(b);
7
8         if(a == b)
9           return true;
10
11        if(!ll_is_so(a) && !ll_is_so(b))
12            return false;
13
14        if(is_determined(a)) {
15            b = lock_leader(b);
16            if(ll_is_so(b))
17                prim_wait(b);
18            unlock(b);
19        } else if(is_determined(b)) {
20            a = lock_leader(a);
21            if(ll_is_so(a))
22                prim_wait(a);
23            unlock(a);
24        } else {
25            jobject high = a < b ? b : a;
26            high = lock_leader(high);
27            if(ll_is_so(high))
28                prim_wait(high);
29            unlock(high);
30        }
31      }
32  }
```

Figure 4.6: The primitive `refeq`, used to implement `==` and `!=`

```
1   jobject lock_leader(jobject o)
2   {
3     o = ll_leader(o);
4
5     while(true) {
6         lock(o);
7         if(ll_leader(o) == o)
8             return o;
9         unlock(o);
10    }
11  }
```

Figure 4.7: The helper function `lock_leader`

The *rptr* for all objects increases the memory requirements, but requires only one pointer dereference and a comparison to determine whether an object is a synchronization object (that is `o->rptr != o`). To save memory the *rptr* could be present only in synchronization objects. But as `libjava` does not have tagged pointers, the test whether an object is a synchronization object would have additional runtime overhead. There are at least two ways to implement such a test. The first emulates tagged pointers by allocating vtables in a special area. The vtable pointer is then tested to see if it is inside this area. This approach is troublesome as the area cannot be of fixed size, and testing would have to be aware of the current area size and location. The second approach makes use of the reference to an object's class object which is present in each vtable (that is both the synchronization and normal vtable). The vtable is dereferenced to reach the class object which is in turn dereferenced to acquire the reference to the synchronization vtable, that is `o->vtab->class->svtab == o->vtab`. The test requires at least three pointer dereferences and a comparison.

The primitives in the forwarding scheme are as follows:

`ll_is_so(r)` An object is a synchronization object if it is not `null` and its *rptr* is not pointing to the object itself. This operation takes constant time.

`ll_bind(a, b)` Binding is implemented by changing the leader's *rptr* to point to the object `b`. Again, this operation takes constant time.

`ll_alias(a, b)` Aliasing is implemented by allowing a synchronization object's *rptr*-field to point to another synchronization object. The operation updates the *rptr* of the synchronization object at the higher address to point to the object at the lower address. This makes the "high" object *aliased* and the "low" object the *leader* of the joined equivalence class, Section 4.3 (Synchronization) motivates the ordering. The operation takes constant time.

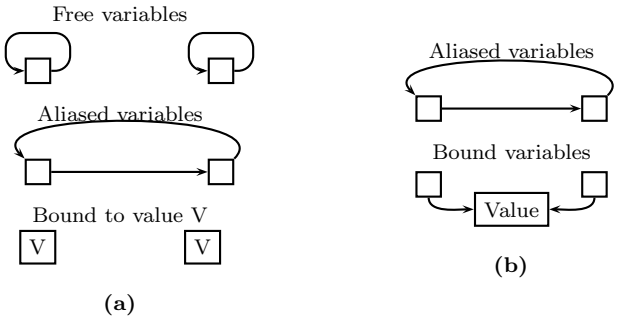

Figure 4.8: Variable representation in Taylor's scheme: a, plain; b for Flow Java.

`ll_leader(r)` follows the *rptr* of its argument until it finds an object which is either determined or which has its *rptr* set to `UNB`. The found object is returned. The worst-case runtime of this operation is linear in the number of objects forming the equivalence class.

`ll_compress(orig, new)` The conservative garbage collector used in the runtime system does not shorten or remove chains of aliased objects. Therefore path compression [30] is implemented by `waitdet` (see Section 4.3.5) which dereferences synchronization objects. The `ll_compress(orig, new)` primitive simply updates the *rptr* of `orig` to contain `new`.

4.4.2 Taylor

In this adaption of Taylor's scheme [38] an equivalence class is represented as a cycle containing all elements of the class. The element at the lowest address is defined as the leader.

Taylor's scheme is a conceptually simple scheme to represent free variables in Prolog. It avoids arbitrarily long reference chains as in the WAM by representing a free variable by a special reference type with a single pointer field. A single free variable contains a reference to itself, thus making it a member of a one-element cycle. When two free variables are aliased their cycles are merged by exchanging the pointer values of the objects being aliased. Binding is implemented by traversing the cycle and overwriting the variables with the value to which they are bound. Figure 4.8(a) graphically shows how variables are represented in Taylor's scheme.

Taylor's scheme cannot be used for Flow Java without some modifications.

```
1   single Object a, b;
2   Object v = new Object();
3   a @= b;
4   a @= v;
5   bool result = a == b; // result is false
```

Figure 4.9: The effects of token equality

Overwriting single assignment variables as part of the binding operation is troublesome. Single assignment variables would have to be allocated as large as the size of the largest object which could be stored in the variable. The largest size of a compatible object is not necessarily available to the runtime system when the variable is created as classes can be loaded at runtime.

Another problem is that token equality is implemented by pointer comparison. Consider the example in Figure 4.9. As `a` and `b` are at different addresses the equality test on line five will return false although `a` and `b` should be equivalent after the aliasing on line three. Even if equality in Java was defined on the contents of the objects, Taylor's scheme would still be incompatible with Flow Java. An update of `a` would not modify `b` even though `a` and `b` are aliased.

In Flow Java Taylor's scheme can be used to reduce the number of dereferencing steps needed to get the value of a determined single assignment variable to one. Instead of overwriting the single assignment variable during the binding, the forwarding pointer is overwritten to point to the determined object, as in Figure 4.8(b).

Limiting the length of the reference chains is attractive but has drawbacks. When synchronization objects are bound, Taylor's scheme will modify all objects in the cycle even if only one thread is interested in the value. The forwarding scheme will only update objects which are accessed (see `waitdet`, Section 4.3.5).

As the `libjava` garbage collector is conservative the system is unable to collect a cycle of synchronization objects unless all references to the cycle are unreachable.

Taylor's scheme leads to the following implementation of the low level primitives:

`ll_is_so(r)` A sentinel in place of the forwarding pointer is used to indicate a bound object. A special case is `null` which cannot be dereferenced but is not a synchronization object. This operation takes constant time.

`ll_bind(a, b)` Traverses the cycle overwriting the forwarding fields of the variables with `b`. This operation is linear in the number of elements in the cycle.

`ll_alias(a, b)` Aliasing merges the cycles by exchanging the forwarding pointer values. This operation takes constant time.

`ll_leader(r)` traverses the cycle. If a determined object is found (this only occurs if another thread is modifying the cycle concurrently) it is returned. Otherwise the object at the lowest address is returned. This operation is linear in the number of elements in the cycle.

`ll_compress(orig, new)` This operation does nothing since no compression is performed.

4.4.3 Hybrid

The hybrid scheme removes the linear time complexity of the `ll_leader(r)` primitive by maintaining a field in all synchronization objects pointing to the leader of the equivalence class.

Compared to the Taylor scheme, only the following operations change:

`ll_alias(a, b)` Merges the cycles as in Taylor followed by choosing a new leader for the now merged cycle. The leader at the lowest address is selected as the new leader. The half-cycle which is assigned a new leader is traversed and the leader pointer is updated. This operation takes linear time in the size of the cycle.

`ll_leader(r)` The value of the leader is simply returned in constant time.

4.5 Compiler Support for Flow Java

To support Flow Java, the compiler's code generation strategy has been modified: Object references are checked for determination before being dereferenced; Single assignment variables are initialized when entering scope; Reference equality operators (`==` and `!=`) are translated into calls to runtime primitives; The binding/aliasing operator, `@=`, is translated into a call to a runtime primitive; The argument to a narrowing conversion is checked for determination before being performed. The type system of Flow Java allows the compiler to statically determine if an application of the `@=` primitive is an alias or bind operation.

4.5.1 Dereferencing

When an object reference is dereferenced to access a field, the compiler wraps the reference in a call to the runtime primitive `waitdet`. The primitive suspends the executing thread, if the reference is undetermined, and returns a reference to the object to which it is bound. See Section 4.3.5 for a detailed description of `waitdet`. This wrapping of references is correct but unnecessarily strict as not all accesses

need to be wrapped. The optimizations that are implemented in Flow Java are further elaborated in Section 4.6.

4.5.2 Initialization of Single Assignment Variables

When a single assignment variable enters scope the compiler generates code which allocates a new unbound synchronization object and initializes the variable with a reference to the synchronization object.

The compiler's code generation when compiling constructors is also augmented to initialize all single assignment fields to newly allocated synchronization objects before the body of the constructor is run on the object.

4.5.3 Operators Implemented as Runtime Primitives

The bind and alias operator, `@=`, is translated into a call to the runtime primitive `alias` if both arguments are single assignment variables. If only the left hand argument is a single assignment variable it is translated into a call to `bind`.

The reference equality operators `==` and `!=` are also implemented as calls to the runtime primitive `refeq` and its negation.

4.5.4 Narrowing Conversions

The compiler wraps the argument of a narrowing conversion of a single assignment variable in a call to `waitdet`, thus forcing the variable to be determined before conversion occurs. Code generation then proceeds as for ordinary narrowing conversions.

4.6 Compiler Optimizations

Dereferencing all references by a call to the runtime primitive `waitdet`, as described in Section 4.5.1, is correct but not needed in many cases. For example, when accessing the fields of `this` (the self object), `this` is always determined as the executing thread synchronized on the binding of the object which now is `this` when invoking the current method.

A second optimization critical for the performance of the Flow Java implementation is to optimize `waitdet` for the non-suspending case. This is done by annotating the conditionals in the primitive with GCC-specific macros for telling the optimizer the most probable execution path.

A third optimization avoids repeated calls to `waitdet` within a basic block, if a reference is known to be constant inside the block. For example, the program

```
1  class Vec {
2    int x, y;
3    public void add(Vec v) {
4      x += waitdet(v).x; y += waitdet(v).y;
5    }
6  }
```

Figure 4.10: Unnecessary calls to `waitdet`

```
1  public void add(Vec v) {
2    v = waitdet(v);
3    x += v.x; y += v.y;
4  }
```

Figure 4.11: Optimized version of the code in Figure 4.10

fragment in Figure 4.10 can be rewritten as in Figure 4.11. Inside `add(Vec v)` `v` is constant and only a single call to `waitdet` is necessary.

This optimization has previously been described in the context of PARLOG in [15]. In Flow Java it is implemented by exploiting the common subexpression elimination in GCC by marking `waitdet` as a pure function (its output only depends on its input). Evaluation shows that this optimization yields a performance improvement of 10% to 40% for real programs.

4.7 Alternative Implementation Strategies

During the development of Flow Java an implementation based on a precompiler was considered. A naïve precompiler implementation would, to implement single assignment variables, for each class create a wrapper class boxing object references. The wrapper class would have a boolean flag and a reference to the determined object. The precompiler would then generate explicit unboxing code for each field and method access. This is not a feasible implementation strategy as it incurs an overhead for all method invocations.

A natural improvement to the naïve strategy would be to use Java's type system to remove the explicit boxing and unboxing. This is not feasible as the single assignment aspect of variables cannot be expressed in Java. The reason for this is that Java's type system only allows subtyping. Consider the example in Figure 4.12 where the single assignment variables are represented as a subclass. The invocation of `doit` on line nine is illegal as `SingleB` is not a subtype of `SingleA`. If ordinary objects were subclasses of single assignment variables a corresponding example could be constructed, but in this case a method with the signature `void doit(A a)` would be the problem.

```
1  class A;
2  class B extends A;
3  class SingleA extends A;
4  class SingleB extends B;
5  void doit(SingleA a) {
6  ...
7  }
8  SingleB b;
9  doit(b); // illegal, B is not a subtype of A
```

Figure 4.12: The difference between a subtype and a type modifier

Worth noting is that this problem cannot be be worked around by having the precompiler create a method for each `single` subclass of A as classes extending `B` could be loaded at runtime.

4.8 Summary

Flow Java is implemented by extensions to the `GCJ/libjava` runtime system and compiler. The extensions are straightforward and only require an object representation similar to C++. They are fully compatible with systems using operating system threads. They could therefore be applied to other languages as for example C#. The Flow Java compiler maintains the distinction between futures and single assignment variables. The runtime system represent both kinds of synchronization variables as synchronization objects. Aliasing and binding is implemented with a simple forwarding scheme. The compiler is responsible for generating explicit determination checks where needed. The conservative garbage collector used by `libjava` makes it necessary to implement path compression in the dereferencing primitives as the collector cannot safely modify objects in memory.

Chapter 5

Evaluation

This chapter contains a performance evaluation of Flow Java. In Section 5.1 the three equivalence class implementations described in Section 4.4 are compared to each other. The implementation with least overhead is then used to assess the overhead incurred by Flow Java to code not making use of Flow Java functionality (Section 5.2). In Section 5.3 the performance of a Flow Java program is compared to a similar program implemented in plain Java. The chapter concludes with a summary of the findings in Section 5.4.

5.1 Equivalence Class Implementations

To measure the performance of the three different implementation schemes, four benchmark sets were used: constructing an equivalence class of size n (the benchmarks are named `cr.f`, `cr.t`, and `cr.h` where `.f` is for forwarding, `.t` for Taylor, and `.h` for hybrid); aliasing two equivalence classes of size n each (`al.f`, `al.t`, and `al.h`); binding an equivalence class of size $2n$ (`bi.f`, `bi.t`, and `bi.h`); and accessing a bound value of an equivalence class through all its members repeatedly (`ac1.f`, `ac1.t`, and `ac1.h` for first time access, `ac2.f`, `ac2.t`, and `ac2.h` for second time access).

5.1.1 Methodology

All benchmarks have been run on a 3GHz Intel Pentium 4 with 1GB RAM and hyperthreading enabled. Each benchmark has been run a hundred times and the mean time for each set has been calculated. The standard deviation of the

individual runtimes is for all cases less than 6.5 percent which is small enough to not change the relative performance of the three implementation schemes.

5.1.2 Random Allocation

The benchmarks have been performed with synchronization objects allocated at random addresses. This captures the situation where synchronization objects are allocated by different program parts. It is also a typical memory layout after garbage collection.

Figure 5.1 shows the results of the `cr.*`-benchmarks involving n objects. The equivalence class is constructed by adding one element at a time. The Taylor scheme (`cr.t`) is slowest due to scanning the whole cycle to find the leader (quadratic complexity).

The forwarding (`cr.f`) and hybrid (`cr.h`) schemes also have quadratic complexity. On average they follow an indirection path of length $n/2$. As the entire chain fits into the cache the actual scanning time is dwarfed by the time taken to handle cache misses (linear in the number of unique memory locations accessed). Therefore, in practice, building an equivalence class is done in $O(n)$. As `cr.h` accesses more memory than `cr.f`, `cr.h` is marginally slower due to more cache misses.

For aliasing and binding the caching effects dominate here as well, see Figure 5.2. For aliasing, two chains of length n are aliased to each other. `al.f` and `al.h` execute in linear time, but they execute in more or less constant time for the cycle lengths considered. The hybrid scheme has a much larger constant overhead for aliasing as it updates the leader pointer in half of its resulting cycle (n elements). Pure Taylor is slowest as it accesses all objects in both cycles ($2n$). The difference in performance for bind is less pronounced as both `bi.f` and `bi.h` access all elements.

Also for accessing the value of a bound equivalence class through its members caching effects dominate. Figure 5.3 shows the time required for accessing all elements the first (`ac1.*`) and second (`ac2.*`) time. As to be expected the forwarding scheme is slowest as it accesses the largest amount of memory. The hybrid scheme is slower than the pure Taylor scheme as it accesses more memory. Looking at the time required for the second access it is clear that path compression has little impact compared to the effect of a hot cache in the Taylor based schemes.

5.1.3 Ordered Allocation

The same set of benchmarks has also been performed with synchronization objects allocated in order. The objects have been ordered in memory such that the forwarding based scheme constructs the longest possible forwarding chains.

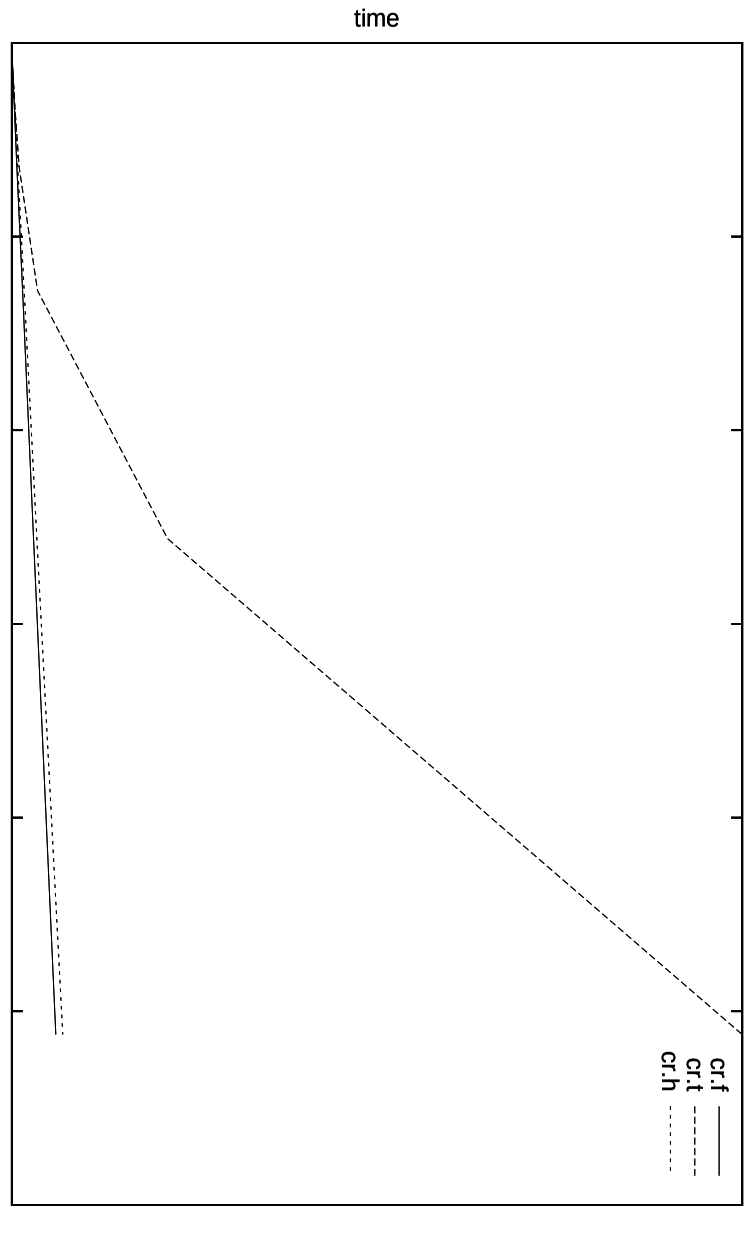

Figure 5.1: Time vs. size for constructing equivalence classes

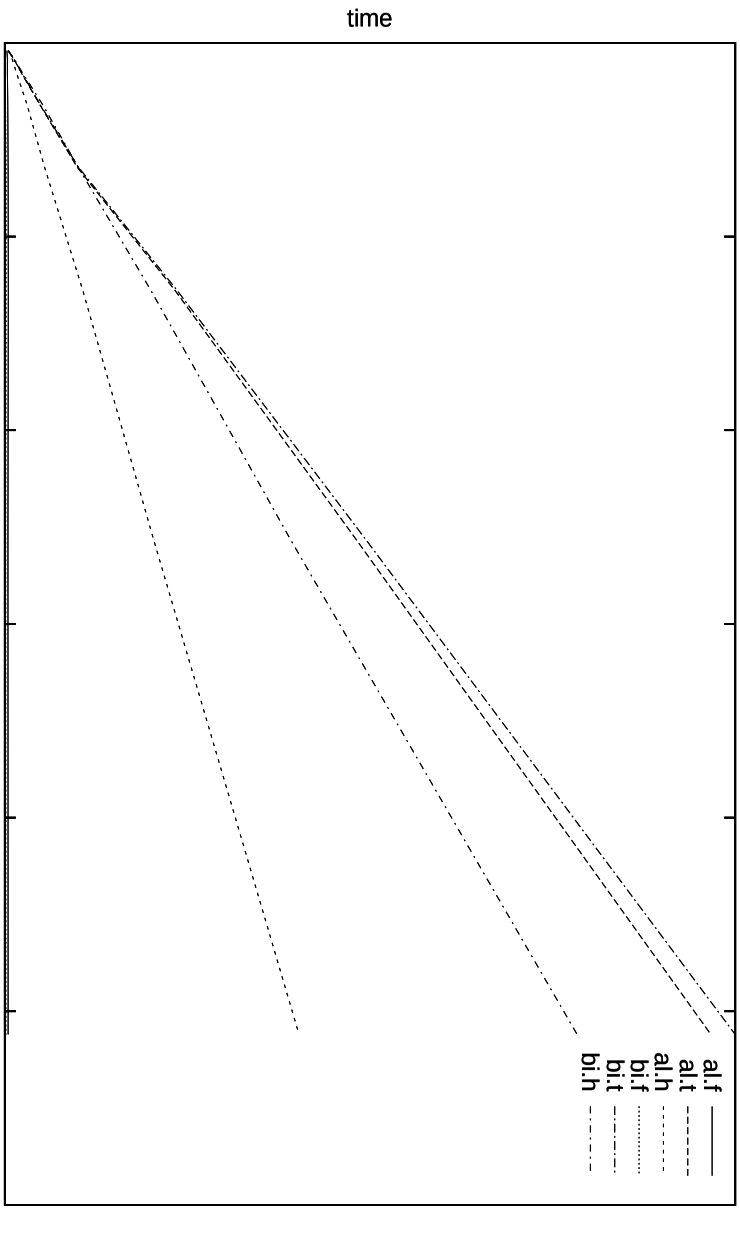

Figure 5.2: Time vs. size for aliasing and binding equivalence classes

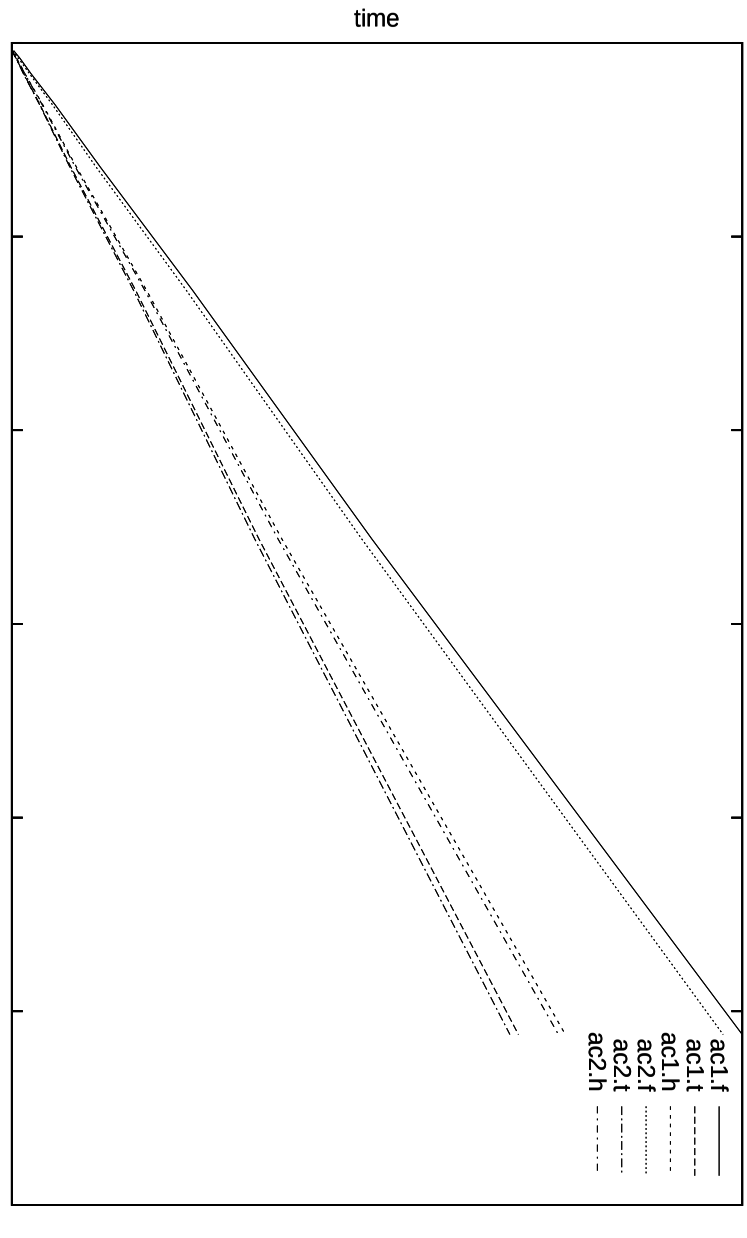

Figure 5.3: Time vs. size for accessing equivalence classes

For creating equivalence classes `cr.f` shows the same relative performance as for random allocation. This is due to the low overhead for traversing the elements already loaded in the cache by the previous aliasing operation. For aliasing, `al.t` and `bi.t` outperform `al.h` and `bi.h`. This is because synchronization objects are smaller for the Taylor scheme. Hence more objects fit into the cache and also accessing one object might already prefetch part of another object into the cache.

Even if the experiment is set up to maximize the length of the forwarding chains, and neutralize the effect of path compression, the measured time for accessing a bound class is linear in the number of elements. This has been verified with an instrumented `waitdet` primitive which counts the number of forwarding hops taken.

5.1.4 Summary

The benchmarks show that the time required to handle cache misses dominates to such a large extent as to make the quadratic components insignificant. To maximize performance one should minimize the amount of memory accessed, as multiple accesses to memory already in the cache is very fast. The Taylor based scheme is the slowest as it accesses all objects in the equivalence class for all operations. The forwarding based scheme accesses, on average, half of the objects in the equivalence class and is therefore the fastest. The hybrid scheme behaves as the forwarding scheme but is penalized as it has a larger memory footprint, and updates half the cycle when performing an alias and the complete cycle when doing a bind. The best of these schemes is therefore the scheme based on forwarding.

It would be interesting to study the performance of the forwarding scheme combined with a more aggressive path compressor which updates all elements in a path as soon as the leader is found.

5.2 Determining the Overhead of Flow Java

To determine the performance impact of the Flow Java extensions compared to the unmodified `GCJ/libjava` the Java Grande benchmark suite [12] has been used. The Flow Java implementation used for benchmarking uses the standard forwarding scheme to represent equivalence classes.

5.2.1 Methodology

The benchmark suite has been run using both the Flow Java compiler and the standard Java compiler. The suite has been run 20 times which is sufficient to limit the standard deviation of the run-times among runs of the same benchmark

to less than 3%. Most benchmarks show a standard deviation of less than 1%, a single benchmark has a deviation of 12% due to jump prediction failures (discussed later). All benchmarks have been run on a standard PC with a 1.7GHz Pentium 4 and 512MB main memory running Linux 2.4.21.

When the current Flow Java implementation was started the equivalence class implementation chosen was the forwarding scheme. That decision was later, when a functioning Flow Java system existed, substantiated by a comprehensive evaluation. Therefore the hardware used to determine the overhead of Flow Java is not the same as when the relative performance was determined.

5.2.2 Results

In this section the benchmark results are presented as graphs, as for example in Figure 5.4. A lightly shaded bar extending above the horizontal axis indicates a speedup. A dark bar extending downwards indicates a slowdown. The names above the bars are the Java Grande benchmark name. The number shown above or below each bar is the speedup/slowdown in percent. A speedup value of 100% means that Flow Java is twice as fast as Java. A slowdown of 100% means that Java is twice as fast as Flow Java.

The Flow Java extensions have very little impact on operations on primitive types as shown in Figure 5.4 and Figure 5.5. This is to be expected as Flow Java does not modify the handling of primitive types. The same is true for operations in `java.lang.math` such as `abs()`, `sin()`, `cos()`, etc. Anomalies show up in the benchmarks for `abs(int)` and `sin()` where the slowdowns are 12% and 69% respectively. As `abs(long)` just has an overhead of 0.02% and the standard deviation is as large as 12% it indicates that the slowdown is due to branch prediction failure coupled to scheduling in the operating system. For `sin()` the situation is the same but here branch prediction is consistent (standard deviation 0.03%) which gives an overhead of 69% just as for `abs()`, `cos()` does not show any slowdown.

Overhead for control constructs and exceptions are all less than 6% as shown in Figure 5.6 which is to be expected as Flow Java does not change the implementation of these constructs.

The increased size of Flow Java objects due to the forwarding pointer is visible in benchmarks studying object creation. The results shown in Figure 5.7 are for a benchmark which creates arrays of increasing size. It is noticeable how the increased size of allocated objects changes the performance of the memory allocator. Similar results are shown by the benchmark in Figure 5.8 which creates objects with: none, one, two, and four fields of type `int` (Simple:nField); Objects with four `float` and `long` fields (Simple:4fField and Simple:4LField). The Subclass

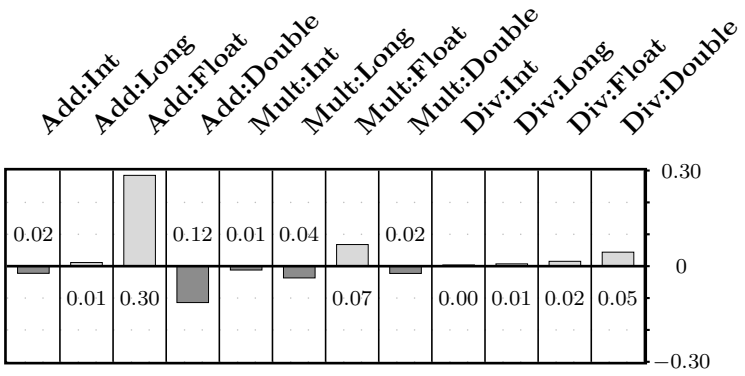

Figure 5.4: Operations on primitive types

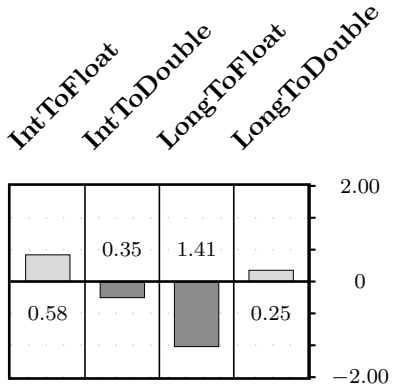

Figure 5.5: Casts of primitive types

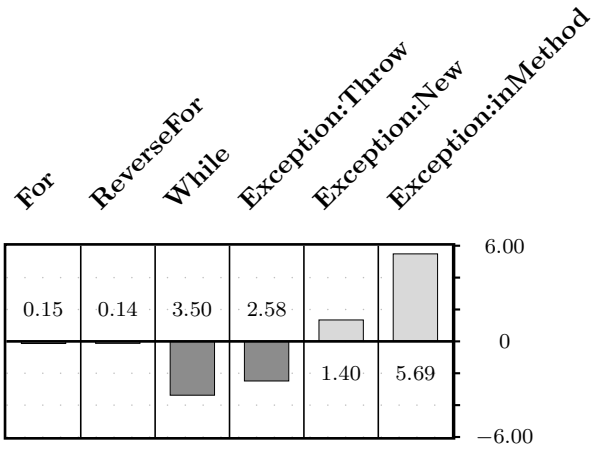

Figure 5.6: Control constructs and exceptions

benchmark creates objects of a subclass extending the class in Simple:0Field and shows comparable performance. Cplx and Cplx:Constr creates an object with one field but uses complex constructors. Therefore the allocator performance has less impact than in the Simple:∗ cases. Generally the slowdown due to memory allocation is less than 20% but can sometimes be as large as 40%.

The other main contributor to slowdown in Flow Java is the checks for determination before dereferencing. For method calls the special vtable does not incur any overhead except for calls to methods declared final which are resolved statically and require a determination check. This is clearly visible in the benchmark in Figure 5.9 where Same:FinalInstance have a slowdown of 13%.

The results shown in Figure 5.10 are for benchmarks measuring overhead for (left to right): assigning a scalar local variable to a scalar local variable; assigning a scalar member variable to a scalar member variable; assigning a scalar class variable to a scalar class variable. For these three the overhead is negligible as the compiler does not generate a determination check when dereferencing `this` and accessing local variables (which are on the stack). The next three benchmarks perform the same operations with fields of an array, here a determination check is generated before dereferencing the array when accessing instance and class variables. The next four benchmarks are the same as Same:∗:Instance and Same:∗:Class but here the variables assigned are not accessed through `this`, thus

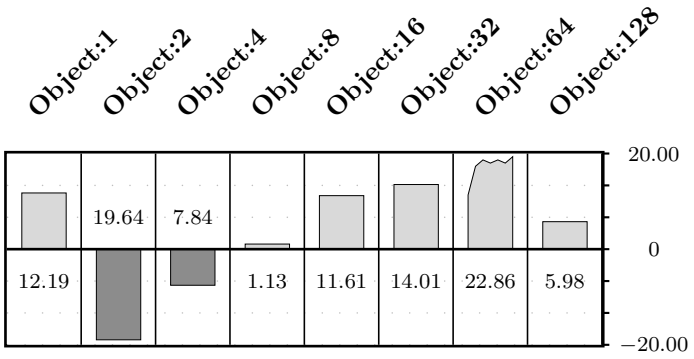

Figure 5.7: Array creation

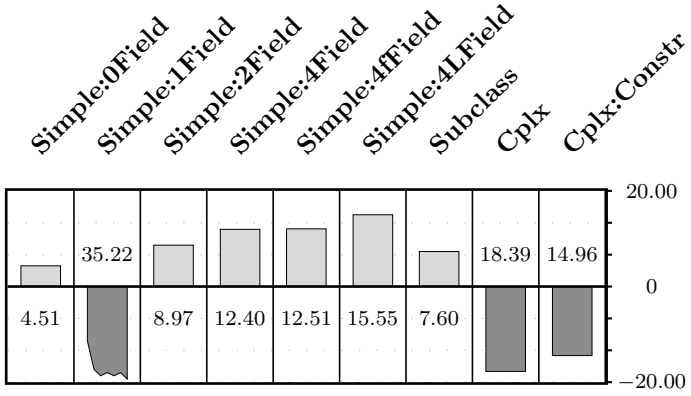

Figure 5.8: Create object

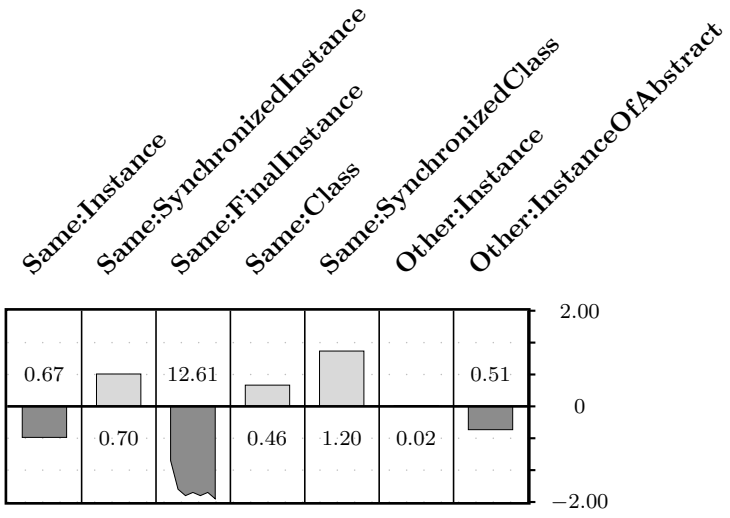

Figure 5.9: Performance of method invocations

requiring extra determination checks. Here it is noticeable how the common subexpression elimination in the compiler manages to keep the class reference live and therefore avoids a determination check while accessing class variables. The maximum overhead can be as large as 226% but is typically below 20%.

The Java Grande benchmarking suite also contains larger benchmarks, the results for these are shown in Figure 5.11. What is noticeable is the large slowdown for benchmarks in which member methods make frequent accesses to instance variables of other objects and/or allocate many small objects. For these the overhead can be as large as 150% but is generally below 50%. For a majority of the benchmarks the overhead is less than 10%.

5.3 Flow Java Versus Plain Java

This section evaluates the performance of concurrent programs using synchronization variables, either natively or emulated in standard Java.

Three benchmarks have been used:

- The *Issue* abstraction as described in Section 2.2.1.
- Merge sort, *MSort*, for linked lists using recursive concurrency, with a limit on the number of concurrent threads. Synchronization between recursion

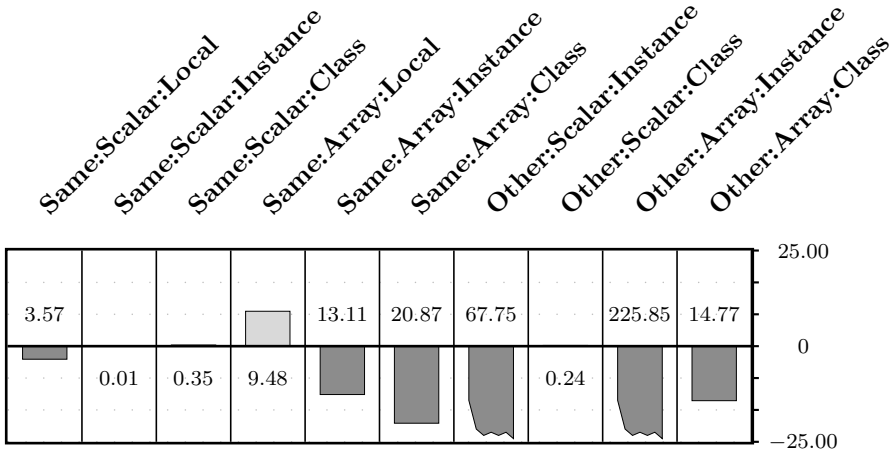

Figure 5.10: Assignment performance

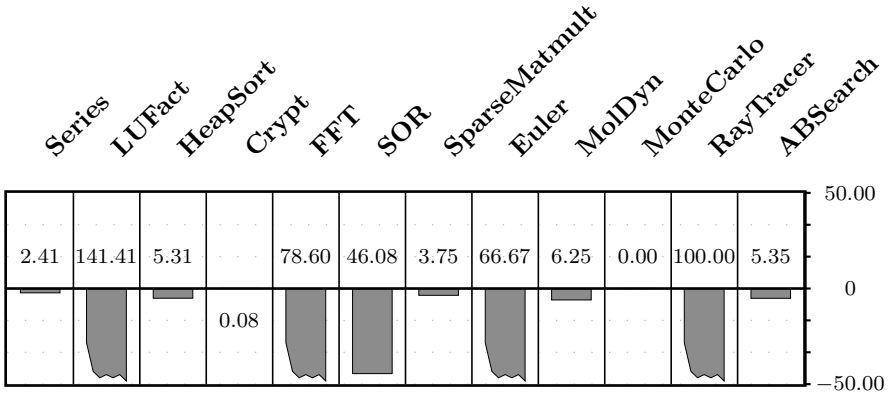

Figure 5.11: Larger benchmarks

levels is implicit through futures.

- A producer-consumer program, *PC*, in which a consumer communicates with a producer over a stream. The consumer requests data from the producer by appending a new stream element containing an unbound synchronization variable to the stream. It then synchronizes on the binding of that variable. The producer iterates over the stream binding the synchronization variable in each element.

The standard Java versions of the benchmarks emulate synchronization variables by using type-specific wrapper classes which are instantiated with an ordinary object when bound. The wrapper defines an access method for each field of its parent type, as well as a bind method. The methods check a status flag indicating the determination status and suspend if the object is unbound, otherwise they forward the invocation to the ordinary object.

5.3.1 Methodology

The benchmarks have been run on the same machine as described in Section 5.1.1. The benchmarks were run repeatedly and the mean time for each benchmark has been calculated. As the benchmarks use concurrency the scheduling of threads in the operating system has a large impact, after 8000 runs the standard deviation of the runtimes is still as large as ten percent for *PC* and less than five percent for *Issue* and *MSort*.

5.3.2 Results

The results of the benchmarks are presented in Figure 5.12. The figure uses the same type of graph as the performance graphs in Section 5.2.2.

As can be expected, the primitive operations (`wait` and `notify`) performed by the programs are the same, the performance of *Issue* and *PC* are comparable to their Java counterparts. For *MSort* on the other hand, the Flow Java implementation is 33% percent faster. This is mostly due to the increased memory footprint of the emulated synchronization variables. An emulated synchronization variable requires three words of storage (vptr, determination flag, and forwarding pointer), a Flow Java synchronization variable only needs two (vptr and forwarding pointer). The increased memory footprint is only noticeable in this benchmark as the other two benchmarks have a much smaller working set. The *MSort* operates on a large linked list, the other two only operate on one element at a time after which it can be garbage collected.

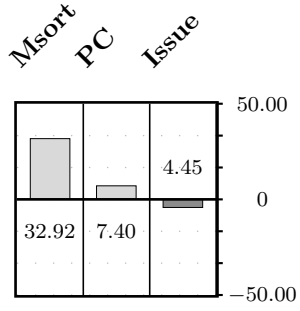

Figure 5.12: Flow Java versus Java implementing the same abstractions

5.4 Summary

Compared to standard Java, Flow Java incurs both a runtime and memory overhead. Memory overhead is always present as the forwarding pointer used increases the size of all objects by a word. This overhead is most noticeable during allocation of small objects where it is up to 40%.

Runtime overhead comes both from the increased memory footprint for the larger Flow Java objects and the extra checks for suspension which are performed when an object is dereferenced. These checks incur an overhead for access to non-member fields. Optimizations for multiple accesses to the same field and accesses relative to `this` reduce the impact to between 10% to 50%. Some pathological cases exhibit as much overhead as 150%. For multithreaded programs exploiting the implicit synchronization of Flow Java the runtime overhead is negligible compared to a standard Java implementation using explicit synchronization. For some benchmarks the Flow Java implementation even outperforms the standard Java version by as much as 33%.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis presents the design, implementation and evaluation of Flow Java. Flow Java is a superset of Java which adds logic variables as a mechanism for automatic synchronization. The Flow Java variant of logic variables are referred to as single assignment variables.

A thread accessing an undetermined single assignment variable will automatically suspend until the single assignment variable becomes determined. A single assignment variable becomes determined when a bind operation is performed on it. Binding assigns the single assignment variable a value, the operation is monotonic and atomic.

Single assignment variables are typed and can only be bound to values of a compatible type. The single assignment type does not form a subtype of the base type. This is the reason for the flexibility of single assignment variables and also the reason for why single assignment variable types cannot be expressed by the Java type system without sacrificing type precision.

Single assignment variables can be aliased, made equal, while still undetermined, this is useful for creating abstractions in which threads share variables. Without aliasing such abstractions must be created top-down, where the shared variables are created first, then the threads. Aliasing allows the threads to be created individually and then connected through aliasing.

To aid security and correctness when sharing single assignment variables among threads, Flow Java adds futures which are read-only views of single assignment variables. A thread having access to a future can not bind it. The future can only be bound through the single assignment variable associated with the future. A

future is obtained by converting the single assignment type to its standard Java type.

Futures are also the basic mechanism for integration with Java. By adding an implicit type conversion from single assignment types to their standard type when calling a method or assigning a variable, interoperability with Java is achieved. The drawback with this approach is that single assignment variables cannot be stored in standard Java collection classes as the variables are automatically converted to futures. If needed this problem can be circumvented by encapsulating the single assignment variable in a wrapper object.

Single assignment variables and futures are ideal for creating abstractions such as ports which are used to construct concurrent programs as sets of message passing tasks. Message encoding, reception, and sending can be implemented in Flow Java itself but requires some extra work from the programmer.

The Flow Java implementation includes both a compiler and a runtime system. The distinction between single assignment variables and futures is maintained solely by the compiler. The runtime system operates on *synchronization objects* which represent both single assignment variables and futures.

For high performance it is essential to limit the memory requirements and accesses required for maintaining the aliasing and binding information. Flow Java uses a forwarding based scheme, which incurs an overhead of one pointer field per object.

Flow Java incurs a moderate runtime overhead for sequential programs due to the need to check for determination before dereferencing a variable. The overhead is reduced by optimizations which only do the check before the first access and remove it completely when accessing fields of `this`. The runtime overhead is generally below 50%. For concurrent code which makes use of the automatic synchronization provided by Flow Java, the overhead is generally negligible. For some examples the Flow Java implementation even outperforms standard Java.

6.2 Future Work

The Flow Java language and system presented in this thesis still has areas open to refinement and optimizations. The following sections present ideas for future work.

6.2.1 Abstractions

The programming abstractions adapted to Flow Java in Chapter 3 for sending and receiving messages are not optimal. The need for the programmer to create explicit message and state classes is cumbersome. Ideally, only the task type

should have to be defined and message sending and reception should have special syntactic constructs such as in Ada or Concurrent C. Such an refinement could be implemented as syntactic sugar for the implementation presented here. Generics as added to Java in version 1.5 could also be used to ease the implementation [8]. Message reception by a special construct could be implemented by manipulating the message queue directly thus freeing the programmer from creating explicit states.

Generic collection classes are also a natural way in which to integrate synchronization variables with the standard Java collection classes.

6.2.2 Distributed Flow Java

Flow Java is but a first step in the development of a platform for high-level and efficient distributed and parallel programming. Work to add distribution support similar to the support available in Oz [18] to Flow Java, based on a language-independent distribution middleware [24] is planned. The goal is to make these powerful abstractions available in a widely used language such as Java while being based on a high-level and declarative model of concurrent programming.

6.2.3 Improved Compilation

The current Flow Java compiler is conservative when generating code for dereferencing variables. Currently the optimizations described in Section 4.6 only operate on the level of basic blocks. A further optimization would be to analyze and optimize across basic block boundaries. The compiler could also be made to emit two versions of the generated code for each method. One with full suspension checking during dereferencing and one without, the fully checking code would then divert to the version without checking as soon as arguments and variables are known to be determined.

6.2.4 Flow Java in Other Implementations

This thesis argues for and claims that the extensions needed for Flow Java are minor and make few assumptions on the underlying Java implementation. It would be interesting to substantiate this claim by implementing Flow Java by extending other Java implementations as for example Kaffe [23].

6.2.5 Flow Java Functionality in Other Languages

It would be interesting to port the Flow Java functionality to a language which have built in support for inter-task message passing such as Ada. Another ap-

proach would be to combine C++ or C# with synchronization variables and message passing.

Bibliography

[1] Alice Team. The Alice system. Programming Systems Lab, Universität des Saarlandes. Available from `www.ps.uni-sb.de/alice/`.

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall International, Hemel Hempstead, Hertfordshire, UK, 1996.

[3] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[5] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming Series. The MIT Press, Cambridge, MA, USA, 1991.

[6] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[7] Hans Boehm. A garbage collector for C and C++. Available from `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[8] Gilad Bracha. Generics in the java programming language. Available at `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.

[9] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Aki Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, USA, 1993.

[10] Frej Drejhammar and Christian Schulte. Implementation strategies for single assignment variables. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2004)*, Saint Malo, France, September 2004.

[11] Frej Drejhammar, Christian Schulte, Seif Haridi, and Per Brand. Flow Java: Declarative concurrency for Java. In *Proceedings of the Nineteenth International Conference on Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 346–360, Mumbai, India, December 2003. Springer-Verlag.

[12] Edinburgh Parallel Computing Centre (EPCC). The Java Grande forum benchmark suite. Available from `www.epcc.ed.ac.uk/javagrande`.

[13] Narain Gehani and William D. Roome. *The concurrent C programming language*. Silicon Press, 1989.

[14] James Gosling, Bill Joy, and Guy Steele. *The Java Programming Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[15] Steve Gregory. *Parallel Logic Programming in PARLOG*. International Series in Logic Programming. Addison-Wesley, Reading, MA, USA, 1987.

[16] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[17] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.

[18] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[19] Gerald Hilderink, André Bakkers, and Jan Broenink. A distributed real-time Java system based on CSP. In *3rd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, Newport Beach, CA, USA, March 2000. IEEE Computer Society.

[20] IEEE Computer Society. *Portable Operating System Interface (POSIX)—Amendment 2: Threads Extension (C Language)*. 345 E. 47th St, New York, NY 10017, USA, 1995.

[21] International Organization For Standardization, International Electrotechnical Comission, Intermetrics Inc. *Annotated Ada Reference Manual*, v6.0 edition, December 1994. ISO/IEC 8652:1995.

[22] Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects. In Gul Agha, Peter Wegner, and Aki Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, USA, 1993.

[23] Kaffe.org. Kaffe.org. Available from `http://www.kaffe.org/`.

[24] Erik Klintskog, Zacharias El Banna, Per Brand, and Seif Haridi. The design and evaluation of a middleware library for distribution of language entities. In Vijay A. Saraswat, editor, *Advances in Computing Science - ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference Proceedings*, volume 2896 of *Lecture Notes in Computer Science*, pages 243–259, Mumbai, India, December 2003. Springer.

[25] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Addison-Wesley, Reading, MA, USA, 2000.

[26] Doug Lea. *Concurrent Programming in Java, Second Edition*. The Java Series. Addison-Wesley, Reading, MA, USA, 2000.

[27] Michael Mehl. *The Oz Virtual Machine: Records, Transients, and Deep Guards*. Doctoral dissertation, Universität des Saarlandes, Im Stadtwald, 66041 Saarbrücken, Germany, 1999.

[28] Michael Mehl, Christian Schulte, and Gert Smolka. Futures and by-need synchronization for Oz. Draft, Programming Systems Lab, Universität des Saarlandes, May 1998.

[29] Mozart Consortium. The Mozart programming system, 1999. Available from `www.mozart-oz.org`.

[30] Dan Sahlin and Mats Carlsson. Variable shunting for the WAM. Research Report R91-07, Swedish Institute of Computer Science, Kista, Sweden, 1991.

[31] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. jcc: Integrating timed default concurrent constraint programming into java. In *Progress in Artificial Intelligence*, volume 2902 / 2003 of *Lecture Notes in Computer Science*, pages 156–170. Springer-Verlag, Heidelberg, 2003.

[32] Vijay A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA, USA, 1993.

[33] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, USA, January 1990. ACM Press.

[34] Tobias Sargeant. Decaf: Confluent concurrent programming in Java. Master's thesis, School of Computer Science and Software Engineering, Faculty of Information Technology, Monash University, Melbourne, Australia, May 2000.

[35] Christian Schulte. Datalogi 2, 2003. Course Homepage `http://www.imit.kth.se/courses/2G1512/`.

[36] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[37] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[38] Andrew Taylor. *High Performance Prolog Implementation*. PhD thesis, University of Sydney, Australia, 1991. `ftp://ftp.cse.unsw.edu.au/pub/users/andrewt/phd_thesis.ps.gz`.

[39] John Thornley. Integrating parallel dataflow programming with the Ada tasking model. In *Proceedings of the conference on TRI-Ada '94*, pages 417–428, Baltimore, ML, USA, 1994. ACM Press.

[40] John Thornley. Declarative Ada: parallel dataflow programming in a familiar context. In *Proceedings of the 1995 ACM 23rd annual conference on Computer science*, pages 73–80, Nashville, TE, USA, 1995. ACM Press.

[41] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, MA, USA, 2004.

[42] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Menlo Park, CA, USA, October 1983.