Flow Java: Declarative Concurrency for Java

Frej Drejhammar[1,2], Christian Schulte[1], Per Brand[2], and Seif Haridi[1,2]
{frej,schulte}@imit.kth.se {perbrand,seif}@sics.se

[1] Institute for Microelectronics and Information Technology
KTH - Royal Institute of Technology
Electrum 229, SE-16440 Kista, Sweden
[2] SICS - Swedish Institute of Computer Science
Box 1263, SE-16429 Kista, Sweden

Abstract. Logic variables pioneered by (concurrent) logic and concurrent constraint programming are powerful mechanisms for automatically synchronizing concurrent computations. They support a declarative model of concurrency that avoids explicitly suspending and resuming computations. This paper presents *Flow Java* which conservatively extends Java with single assignment variables and futures as variants of logic variables. The extension is conservative with respect to object-orientation, types, parameter passing, and concurrency in Java. Futures support secure concurrent abstractions and are essential for seamless integration of single assignment variables into Java. We show how Flow Java supports the construction of simple and concise concurrent programming abstractions. We present how to moderately extend compilation and the runtime architecture of an existing Java implementation for Flow Java. Evaluation using standard Java benchmarks shows that in most cases the overhead is between 10% and 40%. For some pathological cases the runtime increases by up to 75%.

1 Introduction

Concurrent, distributed, and parallel programs fundamentally rely on simple, powerful, and hopefully even automatic mechanisms for synchronizing concurrent computations on shared data. Powerful programming abstractions for automatic synchronization are logic variables and futures (as read-only variants of logic variables). Logic variables for concurrent programming have been pioneered in the area of concurrent logic programming [19], concurrent constraint programming [17, 16, 20], and distributed programming [11].

This paper presents Flow Java which conservatively extends Java with variants of logic variables referred to as *single assignment variables*. By *synchronization variables* we refer to both single assignment variables and futures. Synchronization variables provide a declarative concurrent programming model allowing the programmer to focus on *what* needs to be synchronized among concurrent computations [24]. This is in contrast to synchronized methods in Java where explicit actions to suspend and resume computations are required. Moreover, a programmer is always faced with the complexities of shared mutable state in a concurrent setting (even if not necessary). Synchronization variables in particular support the organization of concurrent computations in a dataflow style.

Flow Java. Flow Java is a conservative extension of Java with respect to types, object-orientation, parameter passing, and concurrency. Flow Java programs without synchronization variables compute as their Java counterpart and Flow Java classes integrate seamlessly with Java classes. As a consequence, Flow Java programs can take advantage of the wealth of available Java libraries.

Single assignment variables in Flow Java are typed and can only be bound to objects of compatible types. Type-compatible single assignment variables can be aliased allowing easy construction of powerful concurrency abstractions such as barriers. Statements automatically synchronize on variables being bound.

To achieve security as well as seamless integration, futures serve as read-only variants of single assignment variables. Futures share the synchronization behavior of single assignment variables but can only be bound through their associated single assignment variables. Futures are used in a novel way to achieve conservative argument passing essential for seamless integration. Passing a single assignment variable to a method not expecting a single assignment variable automatically passes the associated future instead.

Implementation. Flow Java is implemented by a moderate extension (less than 1200 lines) to the GNU `GCJ` Java compiler and the `libjava` runtime environment. Source code for Flow Java is available from `www.sics.se/~frej/flow_java`. The implementation slightly extends the memory layout of objects to accommodate for single assignment variables and is independent of native code or bytecode compilation. Binding and aliasing are implemented using common implementation techniques from logic programming systems. These techniques reuse the functionality for synchronized methods in Java to implement automatic synchronization. We present important optimizations that help reducing the cost incurred by single assignment variables and automatic synchronization.

Evaluation of Flow Java on standard Java benchmarks shows that synchronization variables incur an overhead between 10% and 40%. For some pathological cases runtime increases by up to 75%. Some rare cases currently show an excessive slowdown of up to two orders of magnitude. This slowdown is due to problems in the underlying optimizer and not related to the extensions for Flow Java. Evaluation uses native code obtained by compiling Flow Java programs.

Contributions. The general contribution of this paper 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 new insight that futures as read-only variants of logic variables are essential for seamless integration of logic variables. Additionally, the paper contributes techniques for integrating logic variables and futures in implementations based on objects with a predefined concurrency model.

Structure of the Paper. The next section introduces Flow Java by presenting its features as well as showing concurrency abstractions programmed from these features. The implementation of Flow Java is discussed in Section 3 followed by its evaluation in Section 4. Flow Java is related to other approaches in Section 5. The paper concludes and presents concrete plans for future work in Section 6.

2 Flow Java

This section introduces Flow Java and presents some concurrent programming abstractions. The first section introduces single assignment variables, followed by futures in Section 2.2. The next section discusses aliasing of single assignment variables. Finally, Section 2.4 details types for synchronization variables. We assume some basic knowledge on Java as for example available from [2, 8].

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 2.4. Binding a single assignment variable to an object o makes it indistinguishable from o. After binding, the variable is *bound* or *determined*.

Restricting single assignment variables to objects is essential for a simple implementation, as will become clear in Section 3. This decision, however, follows closely the philosophy of Java to restrict the status of non-object data structures such as integers or floats. For example, explicit synchronization in Java is only available for objects. Additionally, Java offers predefined classes for these restricted data structures (for example, the class `Integer` storing an `int`) which can be used together with single assignment variables.

Binding. 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 2.3. Note that the notion of equality used is concerned with the identity of objects only (token equality).

Synchronization. Statements that access the content of a yet undetermined single assignment variable automatically suspend the executing thread. These statements are: field access and update, method invocation, and type conversion (to be discussed later). Suspension for synchronization variables has the same properties as explicit synchronization in Java through `wait()` and `notify()`.

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 not determined. As soon as some other thread binds `c`, execution continues and the method `m` is executed for `c`.

Example: Spawning concurrent computations with results. This examples shows an abstraction for concurrently computing a result that is needed only much later and where synchronization on its availability is automatic. A typical application is network access where concurrency hides network latency.

The following Flow Java fragment spawns a computation in a new thread returning the computation's result in a single-assignment variable:

```
class Spawn implements Runnable {
    private single Object result;
    private Spawn(single Object r) {
        result = r;
    }
    public void run() {
        result @= computation();
    }
}

public static void main (String[] args) {
    single Object r;
    new Thread(new Spawn(r)).start();
    System.out.println(r);
}
```

`Spawn` defines a constructor which takes a single assignment variable and stores it in the field `result`. The method `run` binds the single assignment variable to the result of some method `computation` (omitted here). The `run` method is invoked when creating a thread as in the method `main`. After thread creation, `r` refers to the result of the spawned computation. Execution of `println` automatically suspends until `r` becomes bound.

A similar abstraction in plain Java requires roughly twice the lines of code which uses explicit synchronization for both storing and retrieving the result. Additionally, usage requires awareness that the result is wrapped by an object. This is in contrast to Flow Java, where the result is transparently available.

2.2 Futures

The abstraction presented above is easily compromised. The main thread can, unintentionally or maliciously, bind the result, thus raising an unexpected exception in the producer of the result. 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 is obtained by converting the type from `single` t to t. This can be done by an explicit type conversion, but in most cases this is performed implicitly. A typical example for implicit conversion is calling a method not expecting a single assignment variable as its argument. 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.

Example: Secure spawning. The design flaw in the previous example is rectified by using a future to pass the result of the spawned computation. The following addition of a class-method (static method) to `Spawn` achieves this:

```
public static Object spawn() {
    single Object r;
    new Thread(new Spawn(r)).start();
    return r;
}
```

As `spawn` is not declared to return a single assignment variable, the result is automatically converted to a future.

2.3 Aliasing

Single assignment variables in Flow Java can be aliased (made equal) while still being unbound. 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.

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.

Example: Barriers. A frequent task in multi-threaded programs is to create several threads and wait on their termination. This is usually achieved by an abstraction called barrier. A barrier can be implemented in Flow Java by giving each thread two single assignment variables *left* and *right*. Before a thread terminates, it aliases the two variables. The main thread, assuming it spawns n threads, creates $n + 1$ single assignment variables $v_0, \ldots, v_n$. It then initializes *left* and *right* as follows: $\mathit{left}_i = v_i$ and $\mathit{right}_i = v_{i+1}$ 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 described algorithm has the following implementation:

```
 1   class Barrier implements Runnable {
         private single Object left;
         private single Object right;
         private Barrier(single Object l, single Object r) {
 5           left = l; right = r;
         }
         public void run() {
             computation();
             left @= right;
10       }
         public static void spawn(int n) {
             single Object first; single Object prev = first;
             for(int i = 0; i < n; i++) {
                 single Object t;
15               new Thread(new Barrier(prev, t)).start();
                 prev = t;
             }
             first == prev;
         }
20   }
```

The `left` and `right` variables are stored in the instance when it is created with the constructor on line 4. The actual computation is done in `run()` on line 8 and finishes by aliasing `left` to `right` on 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. This technique is often referred to as *short circuit* [16].

The previous example shows that variable aliasing can be useful for concurrent programs. Let us consider a design variant of Flow Java without variable aliasing. Then, binding would need to suspend until at least one of the two single assignment variables (or futures) is determined. Hence, an object would be passed as a token through binding resulting in more threads being suspended in contrast to early termination with aliasing. Early termination is beneficial and might even be essential as resources become available to other concurrent computations. This is in particular true as threads are costly in Java.

2.4 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 an object of type t' provided that t' is a subtype of t.

Aliasing. The same holds true for aliasing of variables. Aliasing two single assignment variables x and x' with types t and t' respectively, restricts the type of both variables as follows: the type is restricted to t, if t is a subtype of t'; it is restricted to t', if t' is a subtype of t. Note that there is no need for type intersection as Java only supports subtypes created by single inheritance.

Type conversions. Type conversion can also convert the type of a single assignment variable by converting to a type including the `single` type modifier. Widening type conversions immediately proceed. A narrowing type conversion proceeds and modifies the type of the single assignment variable.

A widening type conversion on a future also immediately proceeds. A narrowing type conversion on a future will instead suspend until the associated variable becomes determined. This is in accordance with the idea that futures are read only, including their type.

3 Implementation

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 `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 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.

To implement Flow Java, extensions to the runtime system as well as to the compiler are needed. These extensions implement the illusion of transparent binding and make method invocations and field accesses suspendable.

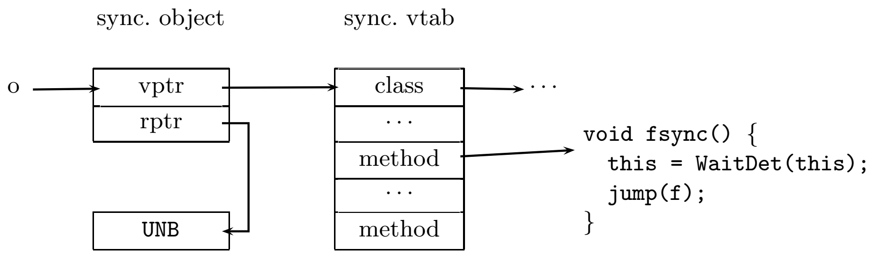
The extensions described in this section can easily be implemented in any Java runtime environment using a memory layout similar to the layout described below. The extensions are not limited to Java, they can equally well be applied to other object-oriented languages such as C#.

3.1 Runtime System Extensions

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

Object representation. To support synchronization objects, a redirection-pointer field (*rptr*) is added to all objects. Standard Java objects have their *rptr* pointing to the object itself. Synchronization objects are allocated as two-field objects containing the *rptr* and the *vptr*. Initially, the *rptr* is set to a sentinel `UNB` (for unbound). Figure 1 shows an unbound synchronization object.

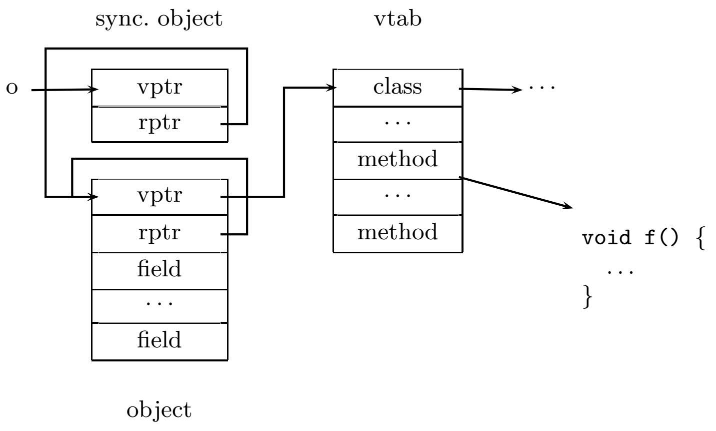
The reason for using the sentinel `UNB` instead of a null pointer to represent an undetermined synchronization object is that a null pointer would make an undetermined synchronization object indistinguishable from a synchronization object bound to `null` (being a valid object reference in Java).



```
void fsync() {
  this = WaitDet(this);
  jump(f);
}
```

Fig. 1. An unbound synchronization object in Flow Java.

Binding. When a synchronization object becomes determined, its *rptr* is set to point to the object to which it is bound, as shown in Fig. 2. This scheme is similar to the forwarding pointer scheme used in the WAM [4]. To guarantee atomicity the per object lock provided by the standard Java runtime system is acquired before the *rptr* is changed.



```
void f() {
  ...
}
```

Fig. 2. A bound synchronization object in Flow Java.

The *vptr* field of a synchronization object is initialized to a synchronization *vtab* containing a synchronization wrapper for each of its methods. A wrapper waits for the synchronization object to become determined using a runtime primitive. When the object has been determined, the object reference pushed on the stack as part of the method invocation is replaced with the determined object. The invocation is continued by dispatching to the corresponding method in the standard *vtab*. This supports transparent redirection of method invocations without a performance penalty when synchronization objects are not used.

Suspension. Suspension is handled by the `wait()` and `notifyAll()` methods implemented by all Java objects. Calling `wait()` suspends the currently executing thread until another thread calls `notifyAll()` on the same object. To guarantee that no notifications are lost, the thread must acquire the lock associated with the object before calling `wait()` and `notifyAll()`. For `wait()` and `notifyAll()` to work as intended they cannot be wrapped in the synchronization *vtab* as it would lead to an infinite loop.

When the *rptr* of an object is modified, its `notifyAll()` method is called to wake up all threads suspended on that object. Suspending operations always follow the *rptr* chain to its end before calling `wait()`.

Aliasing. Aliasing of synchronization objects is implemented by allowing a synchronization object's *rptr*-field to point to another synchronization object. The aliasing operation updates the *rptr* of the synchronization object at the higher address to point to the object at the lower address. The ordering allows the efficient implementation of the equality operators `==` and `!=`. They follow the *rptrs* in order to determine reference equality. They suspend until either both arguments have been determined or aliased to each other. As suspension is implemented by the `wait()` and `notifyAll()` methods, care must be taken to only suspend on one object at a time but still guarantee the reception of a notification if the objects are aliased to each other. The explicit ordering allows the operators to suspend on the object at the higher address.

To guarantee atomicity, aliasing iteratively follows the *rptr*-chains until the locks of the synchronization objects at the ends are acquired or an determined object is found (which is analogous to binding). To prevent deadlock between threads aliasing the same variables, the lock of the object at the lowest address is acquired first. The minimal amount of locking required to guarantee atomicity is the reason for choosing this scheme instead of Taylor's scheme [21], where all involved synchronization objects would require locking.

Type conversion. When the type of a synchronization variable is restricted by a narrowing type conversion, the *vptr* of the corresponding synchronization object is updated to point to the synchronization *vtab* of the new type. Bind and alias operations are checked for validity during runtime using the type information provided through the class reference in the *vtab*.

The primitives following *rptr* chains implement path compression by updating the *rptr* of the first synchronization object to the last object in the chain.

3.2 Compiler Extensions

The compiler is responsible for generating a call to a constructor which initializes single assignment variables to undetermined synchronization objects. The initialization is done each time a single assignment variable enters scope.

When a reference is dereferenced to access a field, the compiler wraps the reference in a call to a runtime primitive. The primitive suspends the currently executing thread, if the reference is undetermined or returns a reference to the object to which it is bound. This behavior is correct but unnecessarily strict as not all accesses need to be wrapped. This is further elaborated in Sect. 3.3.

The bind/alias operator, `@=`, is translated into a call to runtime primitives implementing bind or alias depending on the type of the right-hand argument. The reference equality operators `==` and `!=` are also implemented as calls to runtime primitives described above.

Widening conversions are handled exactly as in standard Java. For narrowing conversions, the reference undergoing conversion is wrapped in a call to a runtime primitive. The primitive suspends the currently executing thread until the reference becomes determined.

3.3 Optimizations

Dereferencing all references by a call to a runtime primitive (named `WaitDet`) is correct but not needed in many cases. For example, when accessing the fields of `this` (the self object), `this` is always determined when executing a member 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:

```
class Vec {
  int x, y;
  public void add(Vec v) {
    x += v.x; y += v.y;
  }
}
```

Inside `add(Vec v)` the `v` is constant and can therefore be transformed into:

```
  public void add(Vec v) {
    v = WaitDet(v);
    x += v.x; y += v.y;
  }
```

thereby avoiding one call to `WaitDet`. This optimization has previously been described in the context of PARLOG in [9]. It is implemented by exploiting

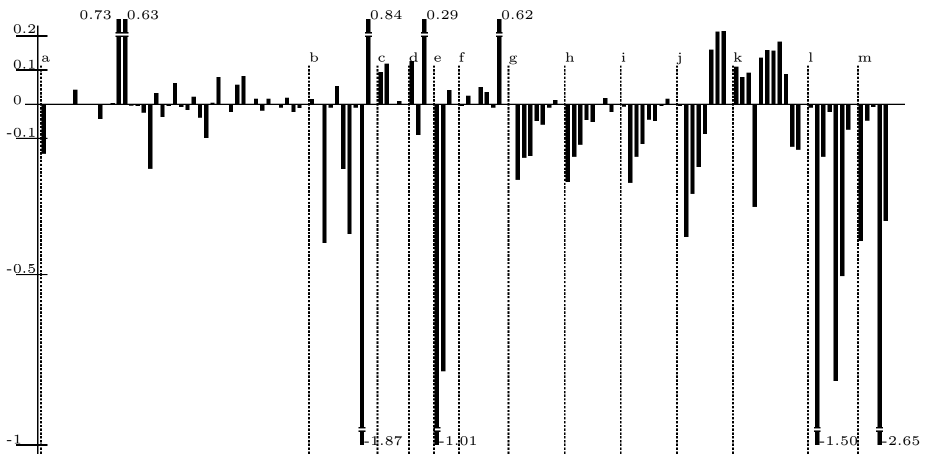


Fig. 3. The relative performance of Flow Java compared to Java.

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 Evaluation

This section answers two questions: How large is the overhead for programs not using the Flow Java extensions? How does a program using Flow Java features perform compared to a similar program implemented in plain Java?

Benchmarks used. To determine the performance impact of the Flow Java extensions we used the Java Grande benchmark suite [7]. The suite has been run using both the Flow Java compiler and the standard Java compiler. The suite has been run 20 times where the standard deviation is less than 2%.

We use two benchmarks to measure the performance of a Flow Java program compared to a program in standard Java implementing the same functionality. The first benchmark, *Spawn*, is based on the `Spawn` abstraction described in Sect. 2.2. The second benchmark (*Producer-Consumer*) is based on a on-demand producer-consumer scenario, where a producer and a consumer thread communicate over a stream. The two benchmarks have been run using the same methodology as above.

All benchmarks have been run on a standard PC with a 1.7GHz Pentium 4 and 512MB main memory running Linux 2.4.21.

Java Grande benchmark suite. Figure 3 shows the results of running the Java Grande benchmark suite. A value of 1.0 means that the Flow Java program is twice as fast (100% speedup), whereas a value of −1.0 means that the Flow Java program is twice as slow as the plain Java program. A solid line indicates the performance of one benchmark, a dotted lines separates groups of benchmarks. The groups are as follows: (a) arithmetic and math operations, (b) assignment operations, (c) casts, (d) exceptions, (e) loops, (f) method invocation, (g-j) creation of arrays with 1, 2, 4, 8, 16, 32, 64 and 128 elements of different types ((g) `int`, (h) `long`, (i) `float`, (j) `Object`), (k) instance (object) creation, (l) small computing kernels, and (m) real applications.

The large slow-downs apparent in groups (b) and (e) are due to a problem in the current implementation of Flow Java. In some cases, the optimizer makes a wrong decision for conditional jumps in loops. Running hand-edited assembler code removes the slowdown for (e) and reduces it to 40% for (b).

Disregarding the previously described problem, the Flow Java extensions show very little impact on arithmetic, conversion, loops and method invocation (groups (a), (c), (e), (f)). This is as expected, since the underlying mechanisms are the same in Flow Java.

The overhead due to the extra redirection pointer in each object instance is noticeable for object creation. It penalizes allocation of small objects with as much as 40%, but is typically around 15%. For larger objects the performance is comparable to plain Java (groups (g), (h), (i), (j), each group creating successively larger arrays, (k) creates single instances, and (d) creates exceptions). These effects are probably due to particularities of the memory allocator.

What is very apparent is the overhead incurred by the check that a reference is determined before actually dereferencing it (Sect. 3.2). The check involves a call as opposed to a pointer dereference with constant offset in plain Java. The overhead is noticeable for real applications in the Java Grande benchmarks ((l) and (m) in the figure). The slowdown ranges between 75% to 45% for programs in which member methods make frequent accesses to instance variables of other objects. The two very large slow-downs in (l) and (m) are due to the previously described optimizer problem as well as the penalty for allocating small objects.

Flow Java versus plain Java. Performance of a Flow Java program is comparable to a plain Java program implementing similar abstractions (*Spawn* and *Producer-Consumer* benchmarks). The Flow Java version of *Spawn* performs slightly worse than the traditional implementation (by 2%), whereas the Flow Java *Producer-Consumer* is slightly faster (by 6%).

5 Related Work

The design of Flow Java has been inspired by concurrent logic programming [19] and concurrent constraint programming [17, 16, 20]. 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 [20, 15, 24], 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 without aliasing (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 [10]. Here, futures and thread creation are combined into a single primitive similar to the spawn abstraction in Section 2.2. 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 [3]. 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 [22]. 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 does neither support aliasing nor binding of an already bound single assignment variable to the same object. A more radical approach by the same author is [23]. It allows only single assignment variables and hence is not any longer a conservative extension to Ada.

Chandy and Kesselman describe CC++ in [6] 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 being 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#) by new models for concurrency has received some attention recently. Decaf [18] is a confluent concurrent variant of Java which also uses logic variables as its concurrency mechanism. It 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 [13]. An extension to C# using Chords is described in [5], where the system is implemented by translation to C# without Chords. 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.

6 Conclusion and Future Work

Flow Java minimally extends Java by single assignment variables and futures which yield a declarative programming model of concurrency. Single assignment variables and futures in Flow Java are compatible with the object-oriented design of Java, its type system, and its model of concurrency. Futures are essential for both security as well as seamless integration.

The implementation of Flow Java is obtained by small extensions to the `GCJ` implementation of Java. The extensions for synchronized variables are straightforward and only make few assumptions on how objects are represented. Java's concurrency model provides the necessary support for automatic synchronization. The implementation is shown to be not only moderate in the amount of modifications required, also the efficiency penalty incurred is moderate.

Flow Java is but a first step in the development of a platform for high-level and efficient distributed and parallel programming. We will add distribution support similar to the support available in Oz [12] to Flow Java based on language-independent distribution middleware [14]. Our 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. Additionally, we plan to investigate how to implement Flow Java by extending other Java implementations.

Acknowledgments. We are grateful to Joe Armstrong, Guido Tack, and the anonymous reviewers for giving helpful comments on the paper. This work has been partially funded by the Swedish Vinnova PPC (Peer to Peer Computing, project 2001-06045) project.

References

1. Alice Team. The Alice system. Programming Systems Lab, Universität des Saarlandes. Available from `www.ps.uni-sb.de/alice/`.
2. Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.
3. 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.
4. Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* Logic Programming Series. The MIT Press, Cambridge, MA, USA, 1991.
5. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. In Boris Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, volume 2374 of *Lecture Notes in Computer Science*, pages 415–440, Málaga, Spain, June 2002. Springer-Verlag.
6. 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.
7. Edinburgh Parallel Computing Centre (EPCC). The Java Grande forum benchmark suite. Available from `www.epcc.ed.ac.uk/javagrande`.
8. James Gosling, Bill Joy, and Guy Steele. *The Java Programming Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.
9. Steve Gregory. *Parallel Logic Programming in PARLOG.* International Series in Logic Programming. Addison-Wesley, Reading, MA, USA, 1987.

10. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
11. 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.
12. Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.
13. 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.
14. Erik Klintskog, Zacharias El Banna, and Per Brand. A generic middleware for intra-language transparent distribution. SICS Technical Report T2003:01, Swedish Institute of Computer Science, January 2003. ISSN 1100-3154.
15. Michael Mehl, Christian Schulte, and Gert Smolka. Futures and by-need synchronization for Oz. Draft, Programming Systems Lab, Universität des Saarlandes, May 1998.
16. Vijay A. Saraswat. *Concurrent Constraint Programming.* ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA, USA, 1993.
17. 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.
18. 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.
19. Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.
20. 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.
21. Andrew Taylor. Parma - bridging the performance gap between imperative and logic programming. *The Journal of Logic Programming*, 1–3(29):5–16, 1996.
22. John Thornley. Integrating parallel dataflow programming with the Ada tasking model. In *Proceedings of the conference on TRI-Ada '94*, pages 417–428. ACM Press, 1994.
23. 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. ACM Press, 1995.
24. Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* The MIT Press, Cambridge, MA, USA, 2003. Forthcoming.