Implementation Strategies for Single Assignment Variables

Frej Drejhammar[1,2] Christian Schulte[1]
[1] IMIT, KTH - Royal Institute of Technology, Sweden
[2] SICS - Swedish Institute of Computer Science, Sweden
{frej,schulte}@imit.kth.se

Abstract

Flow Java integrates single assignment variables (logic variables) into Java. This paper presents and compares three implementation strategies for single assignment variables in Flow Java. One strategy uses forwarding and dereferencing while the two others are variants of Taylor's scheme. The paper introduces how to adapt Taylor's scheme for a concurrent language based on operating system threads, token equality, and update of data structures. Evaluation of the strategies clarifies that the key issue for efficiency is reducing memory usage.

1 Introduction

Flow Java attempts to simplify concurrent programming in Java by conservatively extending Java with single assignment variables (logic variables). Clearly, the most important aspect of implementing Flow Java concerns maintaining single assignment variables. Motivation, related work, and an implementation based on forwarding and dereferencing are presented in [2].

This paper discusses and compares three different implementation strategies for single assignment variables in Flow Java. In addition to the forwarding scheme, we discuss two schemes based on maintaining aliased (equal but not yet bound) single assignment variables in a circular data structure originally due to Taylor [10].

Other approaches based on Taylor's scheme are [4, 9, 8]. They have in common that they carefully investigate the interaction with search by optimizing trailing. This paper naturally takes a different perspective. Firstly, Flow Java implements concurrency with operating system threads. This means that all operations on variables must take this concurrency model into account and use locking to guarantee atomicity. Locking is made deadlock free by exploiting ordering of objects in memory. This is different from Prolog where order is used for trailing. Secondly, in contrast to Prolog and HAL, equality in Flow Java is based on object identity (token equality).

The paper makes the following specific contributions: it develops and evaluates two Taylor-based schemes for maintaining single assignment variables in a truly concurrent setting. The schemes differ in the asymptotic complexity of locking data structures. It presents how to use a Taylor-based scheme in a language with token equality and update. It evaluates the performance of the different schemes and identifies memory usage as the key criterion for good performance.

While the implementation is based on the GNU `GCJ` Java compiler and the `libjava` runtime environment, the techniques presented in this paper apply to any Java runtime environment using a memory layout similar to C++. The techniques are not limited to Java, they can equally well be applied to other object-oriented languages such as C#.

Plan of the Paper. The next section gives a brief overview of Flow Java. Section 3 describes an architecture for implementing Flow Java which is parametric with respect to different implementations of single assignment variables. This is followed by the different implementation strategies for variables. Section 5 evaluates the different strategies and the next section concludes.

2 Flow Java

Flow Java is a conservative extension to Java which adds single assignment variables (a variant of logic variables) to Java. This section provides a brief overview of how single assignment variables are supported in Flow Java, more details including the discussion of related work, types, and futures for security and seamless integration can be found in [2].

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* containing no object. A single assignment variable of type t can be bound to any object of type t. Binding a single assignment variable to an object o makes it indistinguishable from o. After binding, the variable is *bound* or *determined*.

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 called *aliasing* and is discussed below. Note that equality 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. 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`.

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.

3 Implementation Architecture

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. Garbage collection is provided by a conservative collector. Extensions to the runtime system and to the compiler implement binding, aliasing, and synchronization on *synchronization objects* as implementations of single assignment variables.

3.1 The `GCJ/libjava` Runtime Environment

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 memory layout is the same for byte code and native code.

Suspension. The `GCJ/libjava` runtime uses operating system threads. For example, on x86-linux `pthreads` [3] are used. Explicit suspension and resumption in Java is implemented by `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 wait/notify functionality is made available in the `libjava` 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.

Monitors. Orthogonal to the wait/notify mechanism is the monitor which is present in each Java object required for synchronized methods. Internally to `libjava` the lock associated with the monitor can be acquired and released with the two functions `lock` and `unlock`.

3.2 Implementing Synchronization Objects

Synchronization objects are allocated on the heap containing the minimal information to support aliasing. We refer by *equivalence class* to all synchronization objects aliased to each other. The implementation strategies discussed below select one element from the equivalence class as *leader*.

Equivalence classes are maintained on two layers. An upper layer handles the language level operations and makes them safe and atomic. The lower layer (described in Section 4) handles the representation and maintenance of equivalence classes.

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 has at least one forwarding step to its value. This in contrast to tagging where logic variables are simply overwritten during binding.

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).

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

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 method invocation. This provides automatic synchronization of method invocation without a runtime penalty for method invocation on ordinary objects.

3.3 Concurrency and Aliasing

Atomic aliasing and binding are required by Flow Java. In contrast to other systems supporting logic variables (for example, PARMA [10], WAM [11, 1], or even Mozart [6, 5]), 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.

Operations. This section describes how binding, aliasing, and synchronization operations can be implemented using `lock` and `unlock` (see Section 3.1). The operations manipulate equivalence classes through a set of primitives (low-level primitives, starting with `ll_`):

`ll_is_so(r)` 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`. Binding an equivalence class binds all synchronization objects in the equivalence class.

`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.

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

1. The leader of a determined 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 being 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.

Bind. The `bind(a,b)` primitive (defined in Figure 1) 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). Then it 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.

Aliasing. Aliasing of synchronization objects is implemented by `ll_alias`. In order to be thread safe, `alias` iteratively acquires the locks of the two leaders. The lock of the leader at the lowest address is acquired first to prevent deadlock. The definition of `alias` can be found in Figure 1.

```
jobject alias(jobject a, jobject b)
{
  bool as, bs;
  jobject low, high;
  while(true) {
    a = ll_leader(a);
    b = ll_leader(b);
    as = is_so(a); bs = is_so(b);
    if(!as && !bs) {
      if(a == b)
        return a;
      throw TellFailureException;
    } else if(as && bs) {
      if(a < b) {
        low = a; high = b;
      } else {
        low = b; high = a;
      }
      lock(low); lock(high);
      if(low == ll_leader(low) &&
         high == ll_leader(high))
        break;
      unlock(high); unlock(low);
      continue;
    } else {
      if(as)
        return bind(b, a);
      else
        return bind(a, b);
    }
  }
  if(!valid_alias(low, high)) {
    unlock(high); unlock(low);
    throw TellFailureException;
  }
  ll_alias(low,high);
  prim_notifyAll(high);
  unlock(high); unlock(low);
  return low;
}
```

```
jobject bind(jobject a, jobject b)
{
  b = waitdet(b);
  while(true) {
    a = ll_leader(a);
    lock(a);
    if(ll_leader(a) == a)
      break;
    unlock(a);
  }
  if (!bind_is_valid(a, b)) {
    unlock(a);
    throw error;
  } else if(a == b) {
    // Nothing to do
  } else {
    ll_bind(a, b);
    prim_notifyAll(a);
  }
  unlock(a);
  return b;
}

jobject waitdet(jobject o)
{
  if(!is_so(o))
    return o;
  jobject t = o;
  while(is_so(o)) {
    o = ll_leader(o);
    lock(o);
    if(is_so(o))
      prim_wait(o);
    unlock(o);
  }
  ll_compress(t, o);
  return o;
}
```

Figure 1: Primitive operations, alias, bind, and waitdet

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 4 + 5) guarantee the following property: if the leader for an equivalence class changes or all members become bound, then `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 1. It is based on a loop which uses `ll_leader(r)` and terminates when a determined object is found. If an undetermined leader is found, the lock associated with the leader is acquired. If the object is still undetermined `prim_wait` is called to wait for the leader to be updated. When `prim_wait` returns, the lock is released 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 final step.

4 Maintaining Equivalence Classes

The description of the operations in Section 3.3 defined 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 hypothetical 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 4.1. Then an variant of Taylor's scheme [10] adapted to a language with update and token equality (non structural equality) is described. Then finally Section 4.3 shows an optimization of Taylor's scheme in the concurrent setting.

4.1 Forwarding

This scheme is similar to the forwarding pointer scheme used in the WAM [1]. An equivalence class is represented as 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. Standard Java objects have their *rptr* pointing 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 as otherwise an undetermined synchronization object would be indistinguishable from an object bound to `null`.

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 increase runtime. 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 are implemented 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 is constant time.

`ll_bind(a, b)` Binding is implemented by changing the leader's *rptr* to point to the object `b`. Again, this operation is 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" the *leader* of the joined equivalence class, Section 3.3 (Synchronization) motivates the order. The operation is constant time.

`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. This operation takes linear time in the number of objects forming the equivalence class (worst-case).

`ll_compress(orig, new)` The conservative garbage collector used in the `libjava` runtime does not shorten or remove chains of aliased objects. Therefore path compression [7] is implemented by `waitdet` (see Section 3.3) which dereferences synchronization objects. The `ll_compress(orig, new)` primitive simply updates the *rptr* of `orig` to contain `new`.

4.2 Taylor

In this adaption of Taylor's scheme [10] 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 2(a) graphically shows how variables are represented in Taylor's scheme.

Taylor's scheme can not be used for Flow Java without some modifications. 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:

```
single Object a, b;
Object v = new Object();
a @= b;
a @= v;
bool result = a == b; // result = false
```

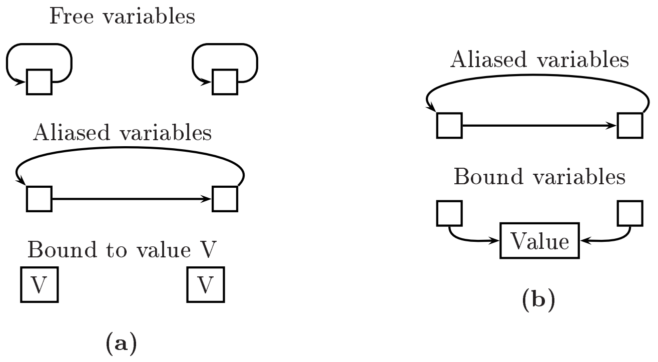

Figure 2: Variable representation in Taylor's scheme: a, plain; b for Flow Java.

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.

Taylor's scheme can in Flow Java 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 2(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 3.3).

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 is 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 is constant time.

`ll_leader(r)` traverses the cycle. If a determined object is found (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 is a noop.

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 plain 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 is linear time in the size of the cycle.

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

5 Evaluation

To measure the performance of the three different implementation schemes, we use four benchmark sets: 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).

Methodology. All benchmarks have been run on a 3GHz Intel Pentium 4 with 1GB RAM. 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.

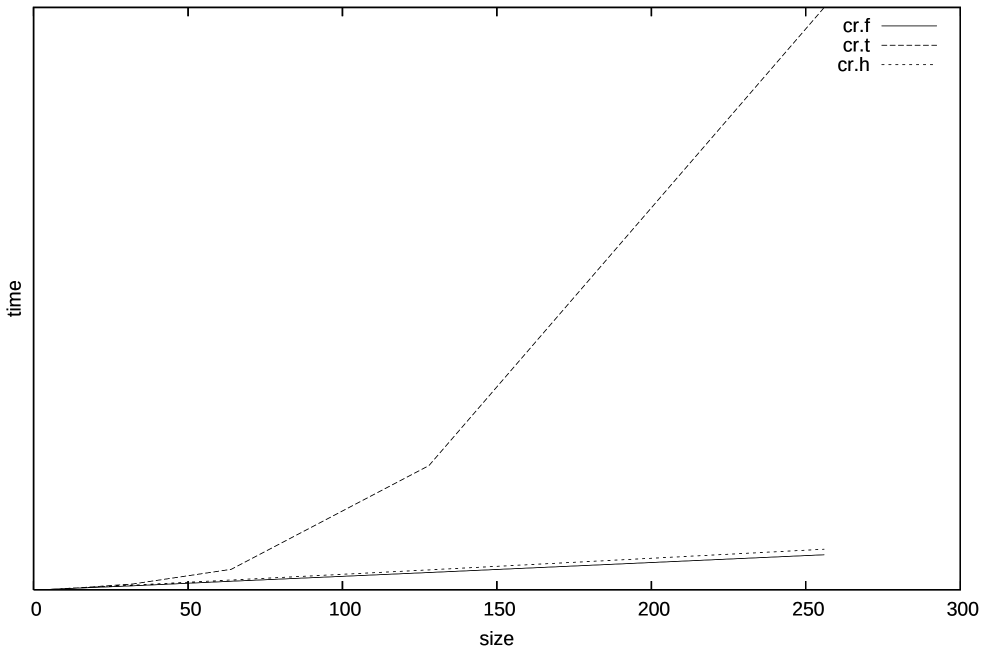

Figure 3: Time vs. size for constructing equivalence classes

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.

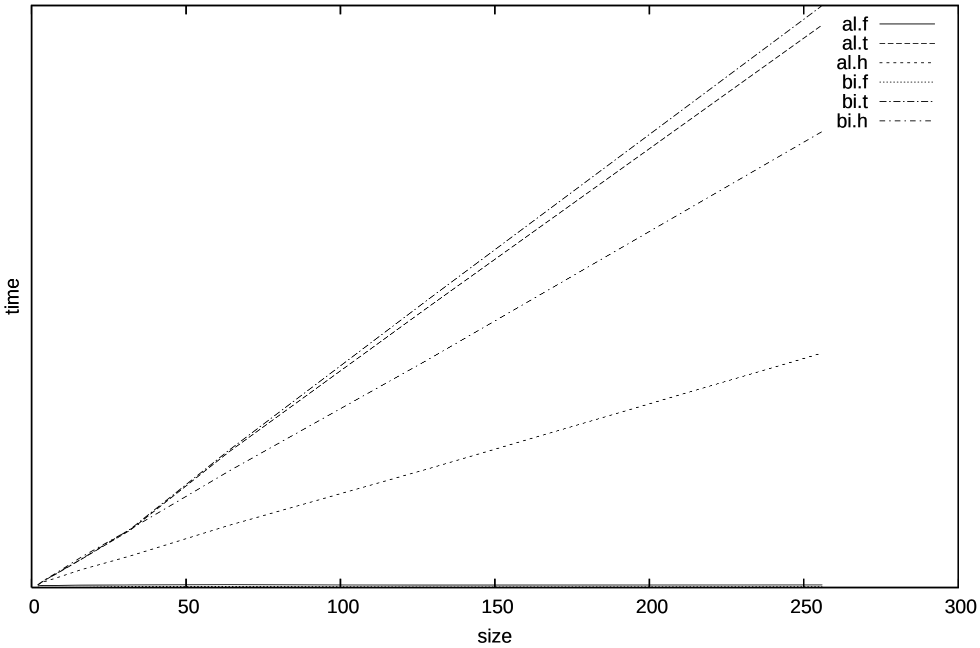

Figure 4: Time vs. size for aliasing equivalence classes

Figure 3 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 4. For aliasing, two chains of length n are aliased to each other. `al.f` and `al.h` are linear time, but they are 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 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.

Ordered Allocation. The same set of benchmarks has also been conducted 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.

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

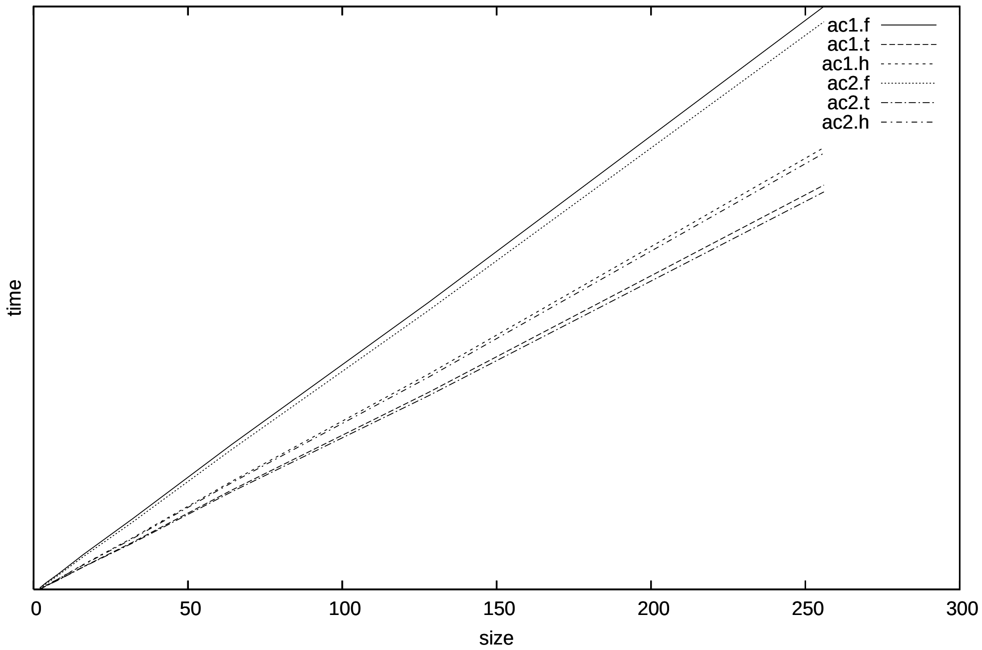

Figure 5: Time vs. size for accessing equivalence classes

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 ($O(n^2)$).

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 essentially for free. With these selection criteria, the forwarding scheme is best.

6 Conclusion

The paper presents three different implementation strategies for single assignment variables which take locking, token equality, and updates into account. The implementation factorizes out the operations concerned with manipulating the different implementations of single assignment variables.

The paper clarifies how Taylor-based schemes need to be adapted to be compatible with thread-based concurrency, token equality, and update. Evaluation establishes that the most crucial aspect for efficiency is to access as little memory as possible.

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

References

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

[2] F. Drejhammar, C. Schulte, S. Haridi, and P. Brand. Flow Java: Declarative concurrency for Java. In *Proceedings of the Nineteenth International Conference on Logic Programming*, volume 2916 of *LNCS*, pages 346–360, Mumbai, India, Dec. 2003. Springer-Verlag.

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

[4] T. Lindgren, P. Mildner, and J. Bevemyr. On Taylor's scheme for unbound variables. Technical Report 116, Computer Science Department, Uppsala University, Oct. 1995.

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

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

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

[8] T. Schrijvers, M. G. de la Banda, and B. Demoen. Trailing analysis for HAL. In *International Conference on Logic Programming*, volume 2401 of *LNCS*, pages 38–53. Springer-Verlag, 2002.

[9] T. Schrijvers and B. Demoen. Combining an improvement to PARMA trailing with trailing analysis. In *Proceedings of the 4th international ACM SIGPLAN conference on Principles and practice of declarative programming.* ACM Press, 2002.

[10] A. Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, Sydney, Australia, 1991.

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