Atomic Instructions in Java

David Hovemeyer, William Pugh, and Jaime Spacco

Dept. of Computer Science, University of Maryland, College Park, MD 20742 USA
{daveho,pugh,jspacco}@cs.umd.edu

Abstract. Atomic instructions atomically access and update one or more memory locations. Because they do not incur the overhead of lock acquisition or suspend the executing thread during contention, they may allow higher levels of concurrency on multiprocessors than lock-based synchronization. Wait-free data structures are an important application of atomic instructions, and extend these performance benefits to higher level abstractions such as queues. In type-unsafe languages such as C, atomic instructions can be expressed in terms of operations on memory addresses. However, type-safe languages such as Java do not allow manipulation of arbitrary memory locations. Adding support for atomic instructions to Java is an interesting but important challenge.
In this paper we consider several ways to support atomic instructions in Java. Each technique has advantages and disadvantages. We propose *idiom recognition* as the technique we feel has the best combination of expressiveness and simplicity. We describe techniques for recognizing instances of atomic operation idioms in the compiler of a Java Virtual Machine, and converting such instances into code utilizing atomic machine instructions. In addition, we describe a runtime technique which ensures that the semantics of multithreaded Java[11] are preserved when atomic instructions and blocking synchronization are used in the same program. Finally, we present benchmark results showing that for concurrent queues, a wait-free algorithm implemented using atomic compare-and-swap instructions yields better scalability on a large multiprocessor than a queue implemented with lock-based synchronization.

1 Introduction

Wait-free data structures and algorithms have been an active area of research in recent years[3, 6, 12, 18], and have spawned a variety of applications[4, 8, 9]. They have the desirable property that when multiple threads access a wait-free data structure, stalled threads cannot prevent other threads from making progress. This avoids a variety of problems encountered with lock-based (blocking) synchronization, such as priority inversion and formation of convoys. Wait-free synchronization is especially useful when a data structure must be accessed in a context from which blocking is impossible or undesirable.

Wait-free data structures generally work by making small changes atomically, such that before and after the atomic operation the data structure is in a consistent state. These atomic operations are usually implemented using atomic machine instructions, in which one or more memory locations are accessed and updated atomically. For example, many architectures support an atomic *compare-and-swap*, or 'CAS', instruction. CAS atomically compares the contents of a single memory location against an input value, and if they are equal stores a second value in the memory location.

Atomic memory operations are easy to express in languages such as C and C++, in which it is possible to determine the address of any variable, array element, or field. (It is necessary to resort to assembly language to implement these operations, but most compilers support some form of inline assembly.) However, type-safe languages such as Java do not permit manipulation of arbitrary memory locations, nor do they permit the direct execution of arbitrary machine instructions. The problem of how to express atomic memory operations in such languages in a way that respects the language's safety guarantees is thus more challenging.

In this paper we consider several techniques for supporting atomic memory operations in the Java programming language[7]. We propose *idiom recognition* as a lightweight technique for expressing atomic instructions in a way that is simple to implement and is fully compatible with the semantics of the language.

While much of our discussion is specific to Java, the same basic techniques could also be used in other type-safe virtual machines, such as Microsoft's CLR (Common Language Runtime).

The structure of the paper is as follows. In Sect. 2, we give a general overview of atomic instructions and how they are used in wait-free algorithms. In Sect. 3, we describe several techniques for supporting atomic instructions in the Java virtual machine. In Sect. 4 we discuss several questions of strategy that arise in supporting atomic instructions. In Sect. 5, we describe an algorithm for recognizing instances of idioms which can be translated into atomic machine instructions within the Java virtual machine. In Sect. 6 we show performance results that show that support for atomic instructions allow the implementation of a concurrent queue that is more scalable than a comparable queue implemented with blocking synchronization. In Sect. 7 we describe related work. Finally, in Sect. 8 we summarize our findings and describe possibilities for future work.

2 Atomic Instructions

Atomic machine instructions atomically access and modify one or more memory locations. They have two primary advantages over using a lock to guarantee atomicity:

1. an atomic instruction is generally faster than equivalent lock-based code, and
2. atomic instructions execute in a finite amount of time, whereas acquiring a lock may block the executing thread for an unbounded period of time

For these reasons, atomic instructions are valuable in situations where a small update is made to a shared data structure, especially when the use of a lock might significantly reduce potential concurrency. Table 1 lists examples of atomic instructions and common commercial architectures in which they are supported in hardware[1].

Instruction	**Semantics**	**Supported by**
Compare-and-swap	Compare two values, update if equal	Sparc, IA32
LL/SC	Two instruction version of CAS	Alpha, PPC, MIPS
Double CAS	CAS on two memory locations	none
Atomic increment	Atomically increment an integer value	IA32
Atomic exchange	Exchange register and memory location	Sparc, IA32

Table 1. Examples of atomic instructions.

One promising application of atomic instructions is in the implementation of wait-free data structures and algorithms. A wait-free algorithm is one in which any thread attempting an operation on a shared data structure is guaranteed to succeed in a finite number of steps, regardless of the actions of other threads. Examples of algorithms and data structures for which efficient wait-free implementations exist include queues[12, 18], double ended queues[6], and union find[3].

A variety of applications have been implemented using wait-free algorithms and data structures. For example, because wait-free data structures do not block, they are ideal for communication between execution contexts that run at different priorities. This makes them useful for operating systems [8, 9], where high priority system threads (such as interrupt handlers) may need to access data structures that are also accessed by low priority threads. A wait-free double ended queue has also been used successfully in a work-stealing parallel fork/join framework[4].

Ad-hoc implementations of atomic instructions have also found their way into commercial implementations of the Java programming language. Several companies have determined that use of compare-and-swap in the implementation of `java.util.Random` could substantially improve performance on some benchmarks, and perhaps some real applications as well.

Considering the growing interest in wait-free algorithms and the potential of atomic instructions to increase the performance of Java libraries, we believe that there is a demonstrated need for a general-purpose mechanism to support atomic instructions in Java.

[1] 'LL/SC' stands for load linked/store conditional. Note that although double compare-and-swap, or DCAS, is not currently implemented as a hardware instruction on any mainstream architecture, it is a significantly more powerful primitive than single word CAS. Recent research[6, 8] suggests that DCAS would enable more effective wait-free algorithms, and therefore it is possible that in the future it will be implemented on commercial architectures in hardware.

A note on terminology: in this paper, we will use the term 'atomic operation' to refer to an operation on one or more memory locations that takes place atomically. We will use the term 'atomic instruction' to refer to the implementation of an atomic operation by a machine instruction or instructions. (Note that even when a particular atomic operation is not available as a single hardware instruction, it may be possible to synthesize from simpler instructions.)

3 Supporting Atomic Instructions in Java

In this section we discuss several ways of supporting atomic instructions in Java. After weighing the alternatives (see Sect. 3.2), we chose *idiom recognition* as the preferred approach. Idiom recognition is a semantically transparent compiler-based approach, in which instances of atomic idioms are translated into atomic instructions.

Before describing the alternatives we considered, it is worth discussing the criteria we used to evaluate them. We did not want to change the syntax or semantics of the language. More generally, we realized that support for atomic instructions was likely to be relevant only to specialized applications and libraries. Thus, we wanted a technique that was simple to implement, did not require expensive program analysis, and had no impact on programs that do not use atomic instructions. Finally, we wanted a technique that was not limited in the kinds of atomic operations which it could express.

3.1 Idiom Recognition

Our approach is to look for synchronized blocks that can be recognized as an idiom that can be implemented via an atomic operation. An example is shown in Fig. 1. The code shown is part of the implementation of a wait-free queue implementation based on an algorithm developed by Valois in [18].

To allow the programmer to indicate which blocks should be implemented using to atomic instructions, we define a new class, `javax.atomic.AtomicLock`. If the compile-time type of an object being synchronized on is a subtype of `AtomicLock`, we consider this a signal to the JVM to try to implement the synchronized block using atomic operations. If the JVM is unable to implement the block using an atomic instruction, it can issue a warning to the user. We refer to synchronized blocks implemented as atomic instructions as 'atomic synchronized blocks', and all others as 'ordinary synchronized blocks'.

Note that it is illegal to allow both atomic and ordinary synchronized blocks to use the same `AtomicLock`. If we did allow such sharing, atomic synchronized blocks might execute concurrently with ordinary synchronized blocks, a violation of Java's guarantee of mutual exclusion for blocks synchronized on the same lock. In Sect. 4.1 we discuss a runtime technique to ensure that atomic instructions are only executed if they respect Java semantics.

While this approach is semantically transparent, it isn't transparent from a performance point of view; it will not provide equal performance across all

```
import javax.atomic.*;
public class WFQ {
    // This lock marks blocks we want to implement using CAS.
    AtomicLock casLock = new AtomicLock();

    private static class Node {
        Node next;
        Object value; }

    // Head field of the queue.  The head node's next field
    // points to the node that contains the next value to
    // be dequeued.
    private volatile Node m_head = new Node();

    // Tail field of the queue.
    private volatile Node m_tail = m_head;

    public Object dequeue() {
        Node next, head;

        // try to advance the head pointer, until
        // either we observe an empty list or we succeed
        while ( true ) {
            head = this.m_head;
            next = head.next;

            if (next == null)
                return null; // We observed an empty queue

            // Attempt to use CAS to update head pointer to point
            // to the next node in the list.
            synchronized (casLock) {
                if (this.m_head == head) {
                    this.m_head = next;
                    break; // CAS succeeded!
                }
            }
        }

        // Head successfully updated; get value from new head node
        return next.value;
    }

    public void enqueue( Object value ) { ... }
}
```

Fig. 1. Part of wait-free-queue implementation.

platforms. If a user writes code implemented using a DCAS (double compare-and-swap) idiom, that code can only use DCAS instructions on platforms that support them.

3.2 Alternative Approaches

Before we delve further into the use of idiom recognition, we briefly describe alternative approaches to support for atomic operations.

Special Classes A simple way to support atomic instructions in Java would be as methods of special object types. For example, the Java runtime could provide a class called `AtomicMutableInteger`, shown in Fig. 2. This class supports get and set, atomic increment and decrement, atomic exchange, compare-and-swap, and double compare-and-swap. The default implementation of this class in the class library would use a private lock object to ensure the atomicity of its methods[2], in order to allow programs using the `AtomicMutableInteger` class to work as expected on any JVM. At runtime, an optimized JVM would convert the method calls into atomic machine instructions.

```
package javax.atomic; // provided by the runtime
public final class AtomicMutableInteger {
  private int value;
  // method implementations omitted
  public AtomicMutableInteger(int initialValue);
  public int getValue();
  public void setValue(int newValue);
  public void increment(void);
  public void decrement(void);
  public int exchange(int newValue);
  public boolean CAS(int expect, int update);
  public boolean DCAS(int expect1, int update1,
         AtomicMutableInteger other,
         int expect2, int update2);
}
```

Fig. 2. A special class to support atomic instructions on integer values.

The main advantage of this approach is simplicity of implementation. It is easy for a JVM to recognize special method calls and convert them to atomic instructions in the generated machine code. It is also easy for programmers to understand.

[2] Care must be taken to avoid potential deadlocks in the `DCAS()` method, since two locks would be involved.

The main disadvantage of this approach is its inflexibility; only those atomic operations explicitly named in the API would be supported. We would like to support as many operations as possible, but we do not want to clutter the API with operations that are only supported on a few platforms. Another problem is the duplication that would result from extending this approach to the full range of basic types supported by Java. For example, we would need a `AtomicMutableReference` class for operations on references, etc. This problem becomes worse when operations that operate on multiple independent memory locations are considered. For example, supporting double compare-and-swap for all combinations of operand types would require methods for the entire Cartesian product of basic types.

Furthermore, using special classes would introduce an additional level of indirection and memory consumption. If an application needed to perform CAS operations on an array of integers, the array would instead need to contain references to AtomicMutableIntegers. In addition to the performance penalty of this indirection, the additional allocation becomes more difficult in situations such as real-time Java where memory allocation is much more complicated (e.g., having to decide which `ScopedMemory` to allocate the `AtomicMutableInteger`s in).

Reflection Another possible technique to support atomic instructions is via Java's *reflection* mechanism. To support atomic operations, we could add additional methods to the `Field` class in `java.lang.reflect`. An example of how the `Field` class could support compare-and-swap is shown in Fig. 3. Note that the types of the operations' parameters are checked dynamically. Similar methods could be added to the `Array` reflection class to support atomic operations on array objects.

```
package java.lang.reflect;
public class Field {
  // ...

  // exception specification omitted for brevity
  public boolean CAS(Object obj, Object expect, Object update);

  // ...
}
```

Fig. 3. Adding support for compare-and-swap to `java.lang.reflect.Field`.

The main advantage of this approach is that it avoids the explosion of method variations needed in a statically typed approach, such as special classes. However, it does so at the price of requiring primitive types to be wrapped when used as arguments to atomic operations, and requiring dynamic type checks of operands.

There are many disadvantages to this approach. The primary problem is that under Java's synchronization semantics, it doesn't make sense to talk about an atomic memory operation in isolation. In particular, say you have an algorithm that needs, in different places, to read, to write or to CAS a field. How do you link those operations so that the write will be visible to the CAS? If you use synchronized blocks and idiom recognition, you just synchronize on the same lock object. But with the reflection based approach, you have normal reads/writes that need to interact with the CAS implemented through reflection.

Another disadvantage is that reflection can make it very hard for either programmers or static analysis to understand a program (most forms of static analysis are unsound in the presence of reflection).

A third disadvantage is that it is difficult to implement reflection in a way that doesn't incur a substantial performance penalty over standard field access. Finally, reflection also suffers from the problem that the API can support only a fixed set of atomic operations.

Magic As shown in the preceding two sections, both static and dynamic typing of method-based interfaces for atomic operations lead to difficulties. We could avoid these difficulties by bypassing the type system altogether, and allowing direct unsafe access to heap memory. For example, IBM's Jikes Research Virtual Machine[2, 10] (formerly known as Jalapeño) has a `VM_Magic` class which allows direct access to an object's memory, and Sun's HotSpot VM has similar `sun.misc.Unsafe` class (as of the 1.4 implementation). This class could be extended to support atomic operations. While it is reasonable for a JVM or runtime library to allow such unsafe operations to be called from core libraries that implement VM functionality, it is not reasonable to allow access to unsafe operations from application code.

4 Implementation Strategies

This section describes our high-level strategies for supporting atomic instructions in Java.

4.1 Runtime Issues

Recall from Sect. 3.1 that atomic synchronized blocks are atomic with respect to each other, but not with respect to ordinary (lock-based) synchronized blocks. Therefore, the JVM must prevent an `AtomicLock` from being used by both atomic and ordinary synchronized blocks.

We considered trying to use escape analysis[19] to prove that all synchronizations on a particular `AtomicLock` could be implemented using atomic operations. But we believe it would be impossible to devise an effective static analysis for this purpose that would be sound in the presence of dynamic class loading, native methods, reflection, and data races.

Instead, we depend upon run-time techniques to detect and handle standard monitor operations on subclasses of `AtomicLock`. Instances of `AtomicLock` are created by the JVM as though they are already locked; thus, any thread attempting to acquire a newly created atomic lock will be immediately suspended. Periodically (perhaps at each garbage collection), the system will check to see if any threads are blocked trying to acquire an atomic lock. If so, a bit is set in the lock to indicate that it is *spoiled*. Once the spoiled bit is set (and all threads are guaranteed not to be executing blocks synchronized on the lock, atomic or otherwise), the atomic lock reverts to ordinary blocking synchronization wherever it is used.

In JVMs using *thin locks*[5], spoiling a lock may be implemented by simply inflating it, avoiding the need for a dedicated spoiled bit in the lock data.

Because the JVM relies on a runtime technique to ensure the correctness of generated atomic instructions, it must preserve the original lock-based code for each atomic synchronized block, as shown in Fig. 4. It is important that when the system checks spoils an `AtomicLock`, all other threads are suspended and no thread is suspended at a point between checking the spoiled bit and performing the atomic operation.

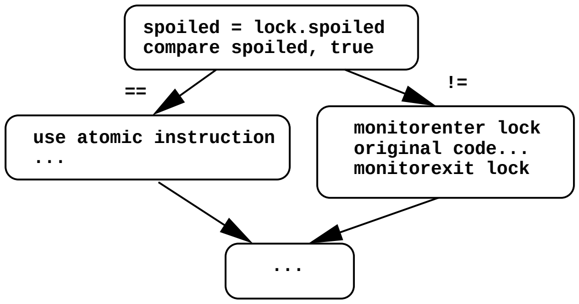

Fig. 4. Code generation pattern to check lock spoiled bit.

A consequence of this approach is that if an `AtomicLock` object is used in a way that cannot be supported using atomic operations, there will be a delay the first time the object is used in this way.

We could attempt to translate all synchronized blocks into atomic instructions; however, this strategy suffers from a number of complications. First, all potential atomic blocks must have the object they lock created as though it were locked until spoiled. Thus, all synchronized blocks suffer a performance hit the first time they are executed while the lock is spoiled, in addition to a small overhead for each execution in the form of an extra check and corresponding branch on the spoiled bit. Since many if not all synchronized blocks in a typical program will not be eligible for atomic substitution, we feel this approach is too heavy handed for our lightweight approach.

A necessary but useful feature of our design is graceful degradation. Assume you have a class that needs CAS operations for many of its functions, but also needs DCAS for one function. If the platform is unable to efficiently support DCAS using atomic instructions, it must implement it through lock-based mechanisms. However, while a lock-based DCAS operations is being performed, the system must not perform a CAS operation on one of the fields involved in the DCAS. With our implementation, rather than saying that the presence of a synchronization block that requires DCAS prevents us from translating any of the synchronized blocks into atomic operations, we can translate those that we support. If the code requiring the DCAS is involved, the system will revert to using lock-based synchronization for all blocks protected by the now spoiled `AtomicLock`.

4.2 Translation Issues

In some cases, it is possible to synthesize an atomic operation out of lower-level instructions. For example, if a block is recognized as an atomic increment, and the execution platform doesn't directly support atomic increment but does support CAS, we can implement atomic increment via CAS (load the value, perform increment in register, try to CAS the new value in on the assumption that the old value is still in the memory location, repeat until success). Now, this code can loop and under very high contention scenarios might be undesirable. However, the lock based approach to implementing atomic increment would likely perform even worse under a similar load.

It is also possible to use `AtomicLock`s to guard simple reads and writes. This would give semantics slightly stronger that declaring the field volatile, which might be useful in some cases[17]. It has the additional benefit that you can also guard reads/writes of array elements (which can't be declared as volatile), and interacts correctly with other operations on the field guarded by the same lock. This is a side benefit of using idiom recognition to support atomic instructions, and is possible because atomic locks simply mark places where it is possible for a synchronized block to be implemented without blocking.

4.3 Memory Model Issues

The Java Memory Model [11] defines the semantics of synchronization to include not only mutual exclusion, but also visibility of reads and writes. We provide exactly the same semantics for synchronized blocks implemented through atomic instructions. This is handled straightforwardly in our technique; generated atomic operations are treated as both compiler and processor memory barrier instructions; neither the compiler nor the processor must allow heap reads/writes to be reordered across an atomic operation. In some cases it may be necessary for the code generator to emit explicit memory barrier instructions to guarantee correct behavior.

5 Implementing Idiom Recognition

This section describes techniques for recognizing instances of atomic idioms and transforming them to use atomic instructions.

5.1 Recognizing Instances of Atomic Idioms

In order to transform instances of atomic operations into atomic instructions, the JVM must recognize them. Because it is not possible to express atomic instructions directly in Java bytecode, a technique for recognizing instances of atomic idioms in the JVM compiler's intermediate representation (IR) is needed.

Basics Figure 5 shows a template for how a compare-and-swap operation would look in a JVM's IR. We assume that the JVM's intermediate representation is a graph of basic blocks using three-address style instructions. We also assume that the IR is in single assignment form, since this greatly simplifies the task of determining the lifetime of a value in the IR. The template begins with a `monitorenter` on an atomic lock, which is simply an object whose static type is `javax.atomic.AtomicLock` or a subtype. The essential features of the CAS idiom are a load of a value from memory, a comparison against an expected value, a store of an updated value to the same location if the original and expected values were equal, and finally a `monitorexit` to release the atomic lock on both branches of the comparison. Additional instructions, indicated by `wildcard`, may be present provided they do not access or modify the heap or have effects observable by other threads; we refer to such instructions as 'incidental'. Note that the variable `atomicLock` appears three times in the template. We require that any template variable occurring multiple times matches the same code variable or constant value in each occurrence. In the context of the template, this ensures that the same lock is used for both the `monitorenter` and `monitorexit` instructions. Any part of an IR graph that matches this template may be transformed to use a hardware compare-and-swap or load linked/store conditional[3].

Our recognition algorithm works by treating each path through a potential idiom instance as a string to be matched by a corresponding path through the template pattern. We place some restrictions on the structure of potential idioms: we do not allow loops, and the only branching constructs we allow are `if` (comparing for equality or inequality only) and `goto`. This language is powerful enough to express all common atomic operations.

A complication arises for atomic operations which access multiple memory locations. For example, in a double compare-and-swap operation there are two ways the operation could fail, one for each comparison. However, the implementation of DCAS (either as a hardware instruction or an emulation in software) is likely to provide only a true or false result for the operation as a whole. This

[3] Note that if LL/SC is used, it is not possible to know the contents of the memory location if the SC fails. For this reason, the variable matched by `orig` in the template must be dead when the failure code for the operation is executed.

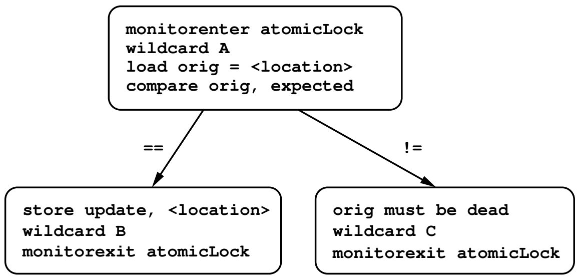

Fig. 5. A template for compare-and-swap in a JVM's IR.

means that a valid instance of the DCAS idiom must use a single code sequence for handling the failure of the operation. In other words, a valid instance of DCAS must be structured as a DAG, rather than a tree. An example of a DCAS template is shown in Fig. 6. There are two paths through the template which reach the block containing the failure code. When matching a potential instance of the DCAS idiom, the recognition algorithm must ensure that the IR code being matched is also structured with a single code sequence handling failure. Later on we will explain how we handle DAG-shaped templates.

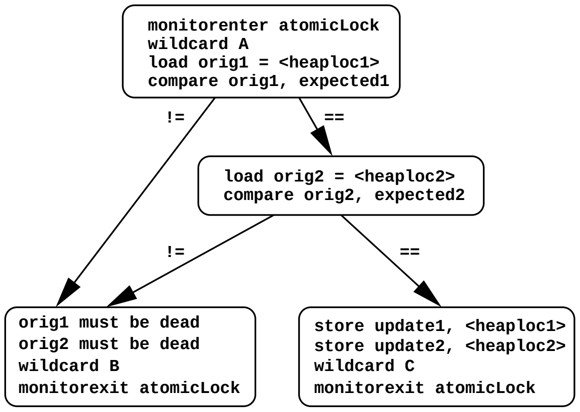

Fig. 6. An IR template for double compare-and-swap.

Language and Algorithm In order to test our pattern matching technique on code generated by a real JVM, we implemented a prototype within the Jikes Research Virtual Machine which analyzes methods for possible instances of atomic idioms and transforms them into atomic machine instructions[4]. The template patterns used in the prototype's idiom recognizer resemble instructions in Jikes RVM's optimizing compiler intermediate representation. The goal of the recognizer is to match the template against a method's IR. If a successful match is found, the matched part of the IR is transformed to use an atomic instruction or instructions[5].

Figure 7 shows the code generated by Jikes RVM for the `dequeue()` method from Fig. 1; comments indicate the start and end of the CAS idiom. Figure 8 shows how the template pattern for CAS is expressed using an augmented version of the same IR language. (This pattern is represented by a graph data structure within the prototype's idiom recognizer.) The `WILDCARD` template instructions match 0 or more 'incidental' instructions in the code. Incidental instructions do not directly affect the instructions comprising the atomic operation, but may help convey the result of the operation. Wildcards are labeled so that the matched instructions can be regenerated when the code is transformed. The `LOAD` and `STORE` template instructions access heap locations, and are an abstractions of `getfield`/`putfield` and `aload`/`astore` instructions, allowing atomic operations to be expressed on both fields and array locations. The `IFCMP` template instruction matches any if comparison (of which there are multiple variants in the Jikes RVM IR). The `ASSERTDEAD` template instruction ensures that the original value read from the heap location is not accessed in the failure branch, since when LL/SC is used to implement compare-and-swap, this value will not be available. (Note that `ASSERTDEAD` does not correspond to any actual code instruction.) Finally, the `FINISH` instruction indicates the termination of a path in the template graph.

The recognition algorithm is essentially a regular-expression match of each path through the template with a corresponding path in the IR code. Each template instruction is matched with a corresponding code instruction, except for `WILDCARD`, which may match a variable number of code instructions, and `ASSERTDEAD`, which does not match any code instructions. Each time a template instruction is matched, it is mapped to its corresponding code instruction(s), and any variables used in the template instruction are mapped to the corresponding variables in the code instruction. These maps are used by the algorithm to enforce two important properties.

First, the algorithm must ensure that the structure of the code graph is compatible with the structure of the template graph. Recall that recognizer templates may be DAGs, meaning that the same template instruction(s) will be applied multiple times. When this is the case, we need to ensure that each template

[4] The examples of IR code shown in this paper were produced by Jikes RVM; we have modified the format of its output to make it more concise and to eliminate redundant `goto` instructions, but have not otherwise removed or modified any generated instructions.

[5] We have not yet implemented the runtime check described in Sect. 4.1 to ensure the legality of the transformation.

```
label0:
  ir_prologue     local0 =
  yldpt_prologue
  bbend           bb0 (ENTRY)
label1:
  yldpt_backedge
  getfield        temp17 = local0, <WFQ.m_head>
  null_check      temp4(GUARD) = temp17
  getfield        temp18 = temp17, <WFQ$Node.next>, temp4(GUARD)
  ref_ifcmp       temp7(GUARD) = temp18, <null>, !=, label11
  goto            label9
  bbend           bb1
label11:
  ref_move        temp21 = temp18
  goto            label3
  bbend           bb11
label3:
  getfield        temp8 = local0, <WFQ.casLock>
  null_check      temp10(GUARD) = temp8
  monitorenter    temp8, temp10(GUARD)  ; *** start of CAS idiom ***
  bbend           bb3
label4:
  getfield        temp22 = local0, <WFQ.m_head>
  ref_ifcmp       temp13(GUARD) = temp22, temp17, !=, label6
  goto            label5
  bbend           bb4
label5:
  putfield        temp21, local0, <WFQ.m_head>
  null_check      temp25(GUARD) = temp8
  bbend           bb5
label15:
  monitorexit     temp8, temp25(GUARD)  ; *** end of CAS idiom ***
  goto            label8
  bbend           bb15
label6:
  null_check      temp29(GUARD) = temp8
  bbend           bb6
label16:
  monitorexit     temp8, temp29(GUARD)  ; *** end of CAS idiom ***
  goto            label1
  bbend           bb16
label7:
  get_exception   temp11 =
  null_check      temp30(GUARD) = temp8
  monitorexit     temp8, temp30(GUARD)
  null_check      temp15(GUARD) = temp11
  athrow          temp11
  bbend           bb7 (catches for bb15 bb5 bb16 bb6)
label8:
  getfield        temp16 = temp21, <WFQ$Node.value>, temp7(GUARD)
  bbend           bb8
label9:
  phi             temp31 = <null>, bb2, temp16, bb8
  return          temp31
  bbend           bb9
```

Fig. 7. IR produced by Jikes RVM for the `dequeue()` method from Fig. 1.

```
;;; Template for the compare-and-swap idiom.
;;; Variable ‘lock’ must be an AtomicLock.
label0:
  monitorenter lock, gv1(GUARD)
  WILDCARD A
  LOAD orig = <heaploc>
  IFCMP gv2(GUARD) = orig, expected, !=, label2
  goto label1
  bbend bb0 (ENTRY)
label1:
  STORE update, <heaploc>
  WILDCARD B
  monitorexit lock, gv3(GUARD)
  FINISH
  bbend bb1
label2:
  ASSERTDEAD orig
  WILDCARD C
  monitorexit lock, gv4(GUARD)
  FINISH
  bbend bb2
```

Fig. 8. Template pattern for the CAS idiom, expressed in the IR of Jikes RVM.

instruction maps to a single point[6] in the code graph, even if it is reached on multiple paths through the template graph. For example, in a template for the DCAS idiom (Fig. 6), we need to ensure that there is only a single code sequence to handle failure.

Second, it must ensure that a match results in a consistent mapping of template variables to code variables. For this purpose, our algorithm builds maps of variables in the template graph to variables, values, and heap locations in the code graph. These maps are used to detect inconsistent mapping of template variables to code values; for example, to ensure that the same lock is used for `monitorenter` and `monitorexit`, or to ensure that the same heap location is accessed by a `getfield` and a `putfield`.

Handling Heap References The template language uses ‘heap variables’ to name heap locations (such as fields and array elements). IR instructions may access the heap in three ways: through instance fields, static fields, and array elements. Because we require the IR to be in single-assignment form, heap references in the analyzed code can be modeled simply as tuples which store the type of reference, object reference variable, index value, and field name; each heap variable in the template is mapped to one such tuple.

[6] For this purpose, chains of `goto` instructions and their eventual target instruction are considered equivalent, since the `goto` instructions’ only observable effect is to change the program counter of the executing thread.

Integration with the JVM Our prototype is integrated into Jikes RVM as follows. The idiom recognizer looks for places where a `monitorenter` instruction accesses a value whose type is `AtomicLock` or a subtype. All known template patterns for atomic operations are applied at these locations[7]. If a template matches, the code is transformed to use an atomic instruction instead of ordinary blocking synchronization (as explained in Sect. 5.2).

Note that in general, a given template pattern will not match every code idiom that could possibly be implemented as an atomic instruction. Therefore, the JVM should document the exact form required to trigger the generation of an atomic instruction at runtime. Ideally, it should provide a stand alone program to check programs offline. Each attempted idiom (marked by synchronization on an atomic lock) would be matched against known templates; any failures would be reported.

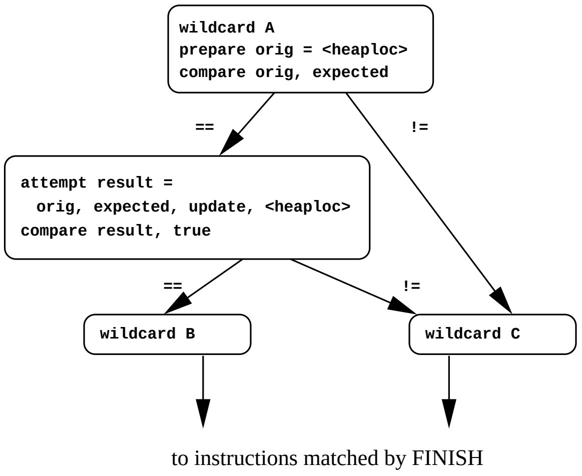

Fig. 9. Code generation template for compare-and-swap.

5.2 Code Generation

Currently, our prototype implements the code transformation of a matched instance of an atomic idiom using ad-hoc code, based on information collected by the idiom recognizer. A more general approach to code generation would be to use a third graph called the generation template, the structure of which is similar to the recognition template graph. An example of a generation template is shown in Fig. 9. Note that Jikes RVM uses `prepare` and `attempt` instructions

[7] Currently, only compare-and-swap is implemented.

to support atomic compare-and-swap. These instructions contain redundant information that allow code to be generated both for systems supporting a real compare-and-swap instruction as well as those supporting only LL/SC. To create IR code from the generation template, the JVM would substitute uses of template variables with their matching code variables, and expands code matched by `WILDCARD` instructions. The original `monitorenter` instruction would be replaced by the first generated instruction in the generation template's entry block, and each generated exit block would be connected to the instruction matched by the code instruction matched by the corresponding `FINISH` instruction in the recognition template.

6 Performance Results

To collect empirical evidence of the potential benefits of supporting atomic instructions in Java, we performed two experiments.

6.1 A Random Number Generator

In the first experiment, we measured the performance of a simple pseudo-random number generator class with an integer field for the seed[8]. We implemented two methods to generate the next random number in the sequence. The `nextIntCAS()` method uses the CAS idiom to update the seed, and the `nextIntSynch()` method is marked using the Java `synchronized` keyword. The code for these methods is shown in Fig. 10. We found that using Jikes RVM on a uniprocessor Power Macintosh G3, we could generate 1,000,000 random numbers in 312 milliseconds using the `nextIntCAS()` method, whereas it took 345 milliseconds to generate 1,000,000 random numbers using the `nextIntSynch()` method. This shows that the overhead of a CAS instruction is less than the overhead of acquiring and releasing a lock in the uncontended case. This result suggests that atomic instructions may be useful for expressing fine-grained optimistic concurrency in real code.

6.2 A Concurrent Queue

In the second experiment, we tested the performance of a wait free queue algorithm implemented with CAS instructions (developed by Valois[18]) against a lock-based queue (developed by Michael and Scott[12]). Because the lock-based queue uses two locks, it permits enqueue and dequeue operations to proceed in parallel, and was found by Michael and Scott to be the most scalable lock-based queue in their experiments.

[8] We based the algorithm on the `rand()` function from FreeBSD's C library. Originally, we intended to use the `java.util.Random` class from a commercial Java implementation; however, it uses a 64 bit seed, requiring a 64-bit CAS operation, which was not possible to support using Jikes RVM.

```
public class Random32 {
    private volatile int seed;
    private final AtomicLock atomicLock = new AtomicLock();
    private static int nextSeed( int origSeed ) { ... }

    public int nextIntCAS() {
        while ( true ) {
            int origSeed = this.seed;
            int newSeed = nextSeed( origSeed );
            synchronized ( atomicLock ) {
                if ( this.seed == origSeed ) {
                    this.seed = newSeed;
                    return newSeed;
                } } } }

    public synchronized int nextIntSynch() {
        int origSeed = this.seed;
        int newSeed = nextSeed( origSeed );
        this.seed = newSeed;
        return newSeed;
    }
}
```

Fig. 10. CAS and lock based implementations of a random number generator.

To add support for CAS, we modified OpenJIT[13, 14] to replace calls to special CAS methods with the Sparc `casa` instruction; essentially, this is the 'special classes' approach. However, the bodies of the CAS methods were in precisely the same form as the CAS idiom shown in Fig. 1, so our code would also work using idiom recognition to guide the transformation.

The benchmark consists of a number of threads performing pairs of enqueue and dequeue operations on a single queue. To simulate work associated with the queue items, after each enqueue and dequeue there is a call to a `think()` method, which spins in a loop adding to the value of one of the thread's fields. A 'think time' of n corresponds to $10n$ iterations of this loop. The enqueued objects were preallocated and the tests run with a large heap to ensure that object allocation and garbage collection did not affect the results. We also performed a dry run with one thread before collecting timing results, to remove dynamic compilation overhead.

We performed this experiment on a Sun Microsystems SunFire 6800 with 24 750 MHz UltraSparc-III processors and 24 GB of main memory. We tested three JVM versions:

- Sun JDK 1.2.2, Classic VM, using our modified version of OpenJIT
- Sun JDK 1.4, Hotspot Client VM
- Sun JDK 1.4, Hotspot Server VM

Figure 11 shows throughput for the wait-free and two-lock queues for the Classic and Hotspot VMs. ‘Throughput’ is a measure of the average rate at which the threads in the experiment perform enqueue/dequeue pairs.

The first graph shows throughput results for 0 think time, using increasing numbers of threads (up to the number of physical processors). Because there was no simulated work between queue operations, this scenario shows the behavior of the queue implementation under maximum contention. The wait-free queue using CAS shows demonstrates much better scalability than the lock-based implementation.

The second graph of Fig. 11 illustrates how the different queue and JVM implementations scale under more realistic conditions, where work is performed between queue operations. The HotSpot server VM, which contains an aggressively optimizing JIT compiler, significantly outperforms the other JVMs until we use 13 processors, at which point the HotSpot server performance tails off to closely match the performance of HostSpot client JIT. This occurs when contention for locks, rather than code quality, becomes the bottleneck. The throughput of the wait-free queue implementation shows nearly linear scalability.

This experiment demonstrates the potential of wait-free data structures implemented using atomic instructions to achieve significantly better scalability than lock-based synchronization when there is high contention. The results also suggest that the combination of a high performance compiler and support for atomic instructions would yield superior performance for any amount of contention.

7 Related Work

There has been a large amount of work on wait-free algorithms and data structures in recent years[3, 6, 12, 18], which have found a variety of applications[4, 8, 9]. This work has generally been based on type-unsafe languages such as C and C++. Because type-safe languages such as Java use garbage collection, they avoid a variety of implementation difficulties associated with wait-free algorithms, such as requiring type-stable memory. Our work has focused on trying to combine the strengths of type-safety and non-blocking synchronization.

In [1], Agesen *et. al.* describe a protocol for implementing low-level synchronization in Java. Their approach uses atomic instructions to implement a ‘metalock’ to protect the synchronization data for a lock object. While useful for building locks whose acquire and release operations are fast in the absence of contention, their work does not enable truly nonblocking synchronization.

In [16], Rinard describes a technique for automatically converting instances of lock-based synchronization into optimistic synchronization based on the CAS operation. His approach is based on *commutivity analysis*, which attempts to detect latent fine-grained parallelism in single-threaded object-based programs. In contrast, our approach uses a simple technique (idiom recognition) to allow the programmer to explicitly mark small blocks of code as constituting an atomic operation, allowing wait-free algorithms and data structures to be expressed.

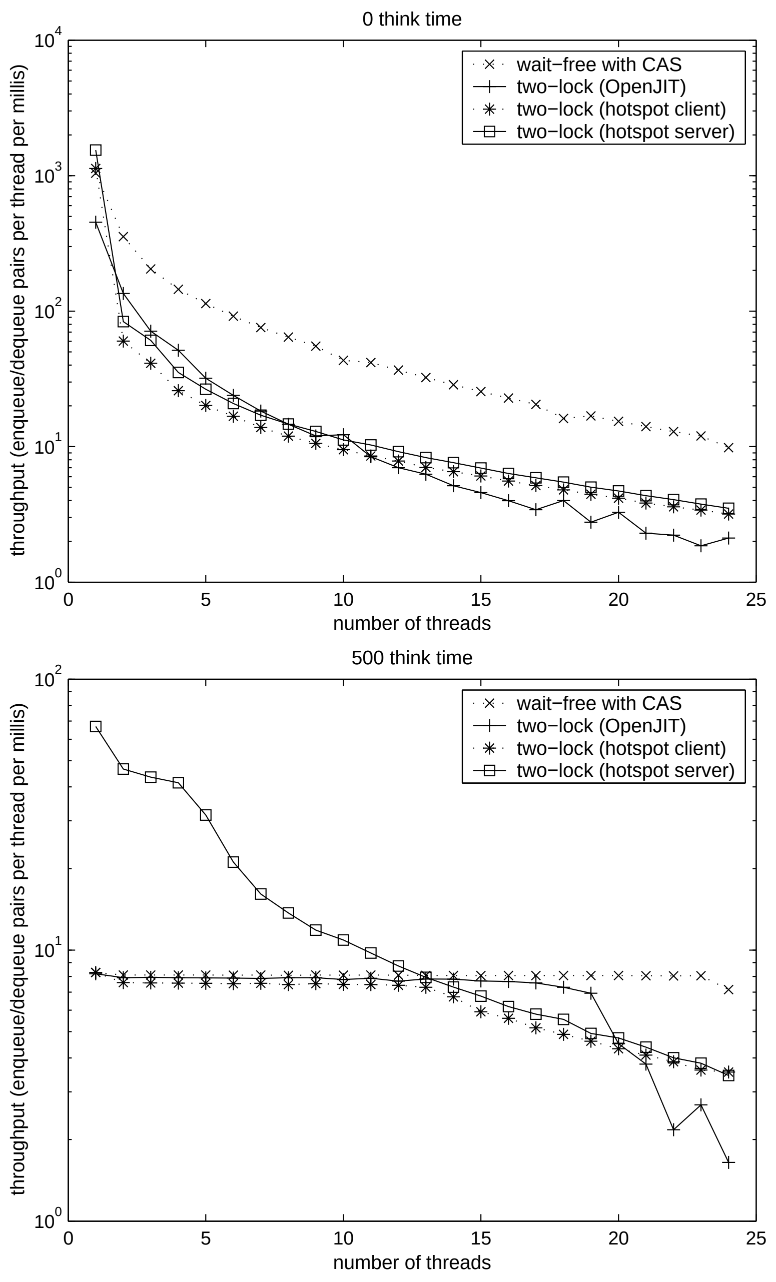

Fig. 11. Throughput results for 0 and 500 think time.

In [15], Pottenger and Eigenmann describe an application of idiom recognition to enable code transformations in an automatic parallelizing Fortran compiler. Their work serves as a good example of the benefit of recognizing idioms in order to perform targeted optimization. Our work differs in that the kinds of idioms we recognize are simpler, and are marked explicitly by the programmer (by atomic locks).

8 Conclusion

Atomic instructions have important applications, and can yield higher levels of concurrency than lock-based synchronization. This paper has presented a lightweight technique for supporting atomic instructions in Java without changing the syntax or semantics of the language. We also presented performance results from a microbenchmark which demonstrate that a wait-free queue implementation written in Java using atomic instructions is more scalable on a large multiprocessor than a comparable lock-based queue.

For future work, we would like to implement our run time check (spoiled bit) in Jikes RVM, and replace the current ad-hoc code generation code with a more general template-based approach. We would also like to explore to possibility of using atomic instructions in the implementation of standard Java classes (such as the collection classes), to determine if we can improve the performance of real concurrent applications.

9 Acknowledgments

We would like to thank Doug Lea for discussions on atomic instruction support in Java. We also would like to thank Matthew Katsouros for loaning us the Power Macintosh G3 we used to run Jikes RVM.

References

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization. In *ACM Conference on Object-Oriented Systems, Languages, and Applications*, Denver, CO, 1999.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, , and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), February 2000.

[3] Richard J. Anderson and Heather Woll. Wait-free Parallel Algorithms for the Union-Find Problem. In *ACM Symposium on Theory of Computing*, 1991.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[5] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin Locks: Featherweight Synchronization for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[6] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele, Jr. Even Better DCAS-Based Concurrent Deques. In *International Symposium on Distributed Computing*, pages 59–73, 2000.

[7] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification.* Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000.

[8] Michael Greenwald and David R. Cheriton. The Synergy Between Non-Blocking Synchronization and Operating System Structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[9] Michael Hohmuth and Hermann Härtig. Pragmatic Nonblocking Synchronization for Real-Time Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 217–230, 2001.

[10] Jikes RVM, http://oss.software.ibm.com/developerworks/projects/jikesrvm, 2001.

[11] Jeremy Manson and William Pugh. Semantics of Multithreaded Java. Technical Report CS-TR-4215, Dept. of Computer Science, University of Maryland, College Park, March 2001.

[12] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[13] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. In *European Conference on Object-Oriented Programming*, Cannes, France, 2000.

[14] OpenJIT: A Reflective JIT Compiler for Java, http://www.openjit.org, 2000.

[15] W. Pottenger and R. Eigenmann. Parallelization in the Presence of Generalized Induction and Reduction Variables. Technical Report 1396, Univ. of Illinois at Urbana-Champaign Center for Supercomputing Research & Development, January 1995.

[16] Martin C. Rinard. Effective Fine-grain Synchronization for Automatically Parallelized Programs Using Optimistic Synchronization Primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, 1999.

[17] Robert Strom and Joshua Auerbach. The Optimistic Readers Transformation. In *European Conference on Object Oriented Programming*, June 2001.

[18] J. D. Valois. Implementing Lock-Free Queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, 1994.

[19] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *ACM Conference on Object-Oriented Systems, Languages, and Applications*, Denver, CO, 1999.