Memoization of Methods Using Software Transactional Memory to Track Internal State Dependencies

Hugo Rito
INESC-ID Lisboa/Technical University of Lisbon
hugo.rito@ist.utl.pt

João Cachopo
INESC-ID Lisboa/Technical University of Lisbon
joao.cachopo@ist.utl.pt

Abstract
Memoization is a well-known technique for improving the performance of a program, but it has been confined mostly to functional programming, where no mutable state or side-effects exist.

Most object-oriented programs, however, are built around objects with an internal state that is mutable over the course of the program. Therefore, the execution of methods often depends on the internal state of some objects or produces side-effects, thus making the application of memoization impractical for object-oriented programs in general.

In this paper, we propose an extended memoization approach that builds on the support provided by a Software Transactional Memory (STM) to identify both internal state dependencies and side-effects, hence removing many of the limitations of traditional memoization.

We describe the Automatic Transaction-Oriented Memoization (ATOM) system, a thread-safe implementation of our memoization model that requires minimal learning effort from programmers, while offering a simple and customizable interface.

Additionally, we describe a memoization advisory system that collects per-method performance statistics with the ultimate goal of aiding programmers in their task of choosing which methods are profitable to memoize.

We argue that ATOM is the first memoization system adequate to the unique characteristics of object-oriented programs and we show how memoization can be implemented almost for free in systems that use an STM, presenting the reasons why this synergy can be particularly useful in transactional contexts.

We show the usefulness of memoizing object-oriented programs by applying memoization to the STMBench7 benchmark, a standard benchmark developed for evaluating STM implementations. The memoized version of the benchmark shows up to a 14-fold increase in the throughput for a read-dominated workload.

Categories and Subject Descriptors D.1.5 [PROGRAMMING TECHNIQUES]: Object-Oriented Programming

General Terms Performance

Keywords Memoization, Object-Oriented Programming, Software Transactional Memory

1. Introduction
Memoization [11] is a well-known technique for improving the performance of functional programs in a transparent way, without changing their semantics. The key idea behind memoization is that we may speedup the execution of a function if we maintain a cache of previous computations and return results from that cache instead of computing the results again. Because the result returned by a pure function depends only on the arguments supplied to the function, it is sufficient to use those arguments as the cache-key.

That simple approach, however, does not work in general for object-oriented programs. In the object-oriented programming paradigm, objects maintain internal state that influence the objects’ behaviors and, in general, this state changes over time. This differentiating characteristic of object-oriented programs hinders the application of memoization because the outcome of a method often depends not only on the arguments supplied to the method but also on the internal state of objects, which must be, therefore, taken into consideration when building the cache-key.

Objects with mutable internal state raise problems to concurrent programs, also. In a context where concurrent threads may read from and write to shared objects, programmers need to ensure that threads observe the state of shared objects and perform actions on that state in a consistent manner. To tackle this problem, much of the recent work on concurrent programming explores the idea of using a Software Transactional Memory (STM) [14], which introduces the notion of atomic actions, or transactions, into the programming model.

With an STM, programmers specify which operations must execute atomically, leaving to the STM the responsibility of providing the intended semantics, while maintaining as much parallelism and concurrency as possible. Typically, this is accomplished by intercepting and registering all accesses to (transactional) memory locations in a per-transaction log, which is used to validate transactions and to ensure atomicity and isolation between threads.

In this paper, we present a solution to the problem of applying memoization to object-oriented programs that is based on the information collected by a software transactional memory runtime. We assume that every mutable memory location is under the control of an STM, meaning that a program may need to be transactified before we apply our memoization solution to it. The details of how a program may be transactified, however, are out of the scope of this paper: We will assume that either the underlying runtime system provides already an STM-based support for the execution of the program, or the program was correctly transactified before (either manually, or automatically through a transactification tool such as JaSPEx [2]).
Our proposal was implemented as the Automatic Transaction-Oriented Memoization (ATOM) system, a Java implementation of our memoization model that extends the Java Versioned STM (JVSTM). To the best of our knowledge, ATOM is the first automatic memoization system that addresses the unique characteristics of object-oriented programs because: (1) it allows memoizing methods that depend on the internal state of objects (rather than only pure functions that depend only on their arguments), (2) it prevents errors in memoizing unintended methods, (3) it allows memoizing methods that have side-effects, and (4) it offers a memoization advisory tool that aids programmers in choosing which methods are profitable to memoize.

In the next section we review what memoization is and show the pitfalls of applying memoization in object-oriented programs. We briefly introduce the concept of Software Transactional Memory in Section 3, with special emphasis on the Java Versioned STM. In Section 4, we present the core ideas of our work, and then, in Section 5, we describe its implementation. In Section 6, we present some results obtained from the STMBench7 benchmark. In Section 7, we discuss related work. and, finally, in Section 8, we draw some conclusions and discuss future work.

2. Memoization

Memoization is a function-level optimization technique that improves the performance of programs in exchange of extra memory space: To avoid future repeated work, each memoized function is augmented with a cache that stores the results of previous computations. A performance increase is possible if it is faster to search the cache for a match than to reexecute the original function, and, of course, there is a cache hit (meaning that the result stored in the cache is returned).

To apply memoization it is essential that we define what are the relevant values for a computation, because only doing so will we be able to build a cache-key that allows a correct classification of computations as repeated or not.

We classify a particular value as relevant for a computation’s result if the result of the computation may be different when the aforementioned value changes. Likewise, we define the list containing all the relevant values that are significant for a computation as the relevant state of said computation. In functional contexts it is simple to build the relevant state because the result of a function is fully defined by the function’s arguments. Thus, the list of arguments received as input by the memoized function may be used as the cache-key. As we will see, the same approach may not apply to methods in object-oriented contexts.

2.1 The pitfalls of applying memoization in object-oriented programs

To introduce the problems of applying memoization to object-oriented programs and later illustrate our approach, we shall use a minimalist example of an application from the banking domain. In particular, suppose that the application deals only with two types of entities: accounts and clients.

Accounts have a current balance, which corresponds to some monetary amount in some particular monetary currency. Clients may have any number of accounts, can deposit money to any account, but can issue withdrawals only from accounts that they own. These operations change the respective account’s current balance in accordance with the amount deposited or withdrawn. Clients may also query the bank for their total balance—that is, the sum of the balance of all the accounts that they own.

This minimalist example is representative of typical object-oriented programs, where some objects have an internal state that may change over time (in this case, both the account’s and the client’s state may change). In the following, we give a brief overview of the problems that this mutable state poses to memoization.

2.1.1 The problem of depending on the internal state of objects

Consider the implementation of the Client and the Account classes shown in Listing 1. Furthermore, assume that, for performance reasons, we want to memoize the getTotalBalance method.

```java
class Account { long balance; }

class Client {
    Set<Account> accounts;

    long getTotalBalance() {
        long total = 0;
        for (Account acc : accounts) {
            total += acc.balance;
        }
        return total;
    }
}
```

Listing 1. Java implementation of the classes Client and Account.

To do it safely, we need to identify what may possibly change a client’s total balance—that is, what are the relevant values for the output of the method getTotalBalance.

A client’s total balance may change every time the client opens a new account, closes an account, or changes the balance of one of its accounts. Thus, we need to reexecute the method getTotalBalance every time that there is a change on the content of the slot accounts or on the value of the slot balance of one of the Account instances contained in the accounts slot.

From this example it is clear that the approach followed in functional programs, of using the function’s list of arguments as the cache-key, cannot be directly applied in object-oriented programs, because the list of relevant values encompasses not only the list of arguments supplied to the memoized method, but also values read from the internal state of objects.

Knowing that the list of arguments is not suited to classify computations as repeated or not, a naive approach would be to include in the cache-key the content of the set accounts and the balance of each Account instance contained within the accounts set. In fact, in this simple example that may solve the problem. Yet, in the general case, of more complex methods where the set of fields accessed is much larger and the relevant state depends on which execution path is taken or on what other methods are called, it is unfeasible to determine manually the correct set of fields to use in the cache-key.

Another problem with internal state dependencies is that it is not simple to store in the cache information regarding the locations from where relevant values were retrieved, so that, in future reexecutions, we can inspect those cached locations to check if they still hold the same value.

2.1.2 The problem of side-effects

Without risking a semantic change, we can skip the execution of a method only if the method is referentially transparent. A method is referentially transparent if calling the method has the exact same effect as replacing the method call with its return value.

This means that memoization is not applicable to methods with side-effects, such as, for instance, methods that do I/O or that change the internal state of some object. The problem is that methods with side-effects are common in object-oriented programs and
represent a tangible risk when memoized: Skipping their execution after a cache hit may lead to an inconsistent system state.

This impossibility directly influences how memoization is done in imperative systems because, unlike what happens in functional environments, it is no longer safe to memoize any method of the system. Thus, to guarantee correct system semantics, we need to classify methods as producing side-effects or not.

In polymorphic environments and in the presence of deep and complex call-chains, doing this classification manually is time-consuming, hard, and error-prone. Moreover, even assuming that we could identify methods without side-effects, nothing prevents future iterations of the system to introduce side-effects, either directly or indirectly, on methods’ executions that had none before. So, after each change to the system, we have to revise all memoized methods to look for potential side-effects. In practice, this means that, without support from the memoization tool, we will never be sure that a method’s execution that produces side-effects will not be skipped due to being inadvertently memoized or being called inside a memoized method.

Thus, an adequate memoization tool would remove this burden from the programmers shoulders, by automatically identifying methods with side-effects. Once we know which methods have side-effects, it is easy to prevent the memoization of those methods. Yet, if we allow only a reduced subset of the system to be memoized, this subset may not be enough to obtain more than marginal improvements in performance. On the other hand, if we relax this prohibition, allowing methods that produce side-effects to be memoized, we extend the applicability of memoization to any method of an imperative system. The difficulty here is how to replicate the behavior of a memoized method that is not referentially transparent—that is, how to preserve the semantics of the program in the presence of memoized methods that may produce side-effects.

3. Software Transactional Memory

The advent of multicore architectures highlighted the necessity for adequate parallel programming abstractions that help programmers build systems that take full advantage of the underlying hardware platform. That is the main motivation behind the work on Software Transactional Memory, which brought the expressiveness of transactions to mainstream programming, leaving behind the cumbersome work of explicit lock-based constructions.

From the perspective of an STM, operations executed within a transaction do not have a special meaning associated with them: They are just a series of reads from and writes to shared memory locations. STMs intercept these accesses to shared memory and log into a per-transaction read-set what locations were read and into a per-transaction write-set what locations were written.

Changes made during a transaction are made permanent only at the transaction’s end (commit time) and only when the transactional system can assure that there are no consistency violations. So, the commit of a transaction is responsible for detecting conflicts and can yield one of two possible results:

- success—all of the values written by the transaction were applied to the shared system state.
- fail—none of the values written were applied and the transaction should be restarted.

Transactions can be decomposed into subtransactions, which, in turn, can be decomposed into more subtransactions, forming an arbitrarily deep hierarchy of nested transactions. A nested transaction is created every time a new transaction is started in the context of a surrounding transaction.

The JVSTM [4] is a multi-versioned object-oriented STM implemented as a pure Java library that was conceived with read-dominated, domain-intensive applications in mind.

The JVSTM introduces the concept of versioned boxes (VBox) to keep multiple versions of each shared mutable location. A VBox instance holds a sequence of values that corresponds to a successful assignment made to the box by a successfully committed transaction. Each element of this history is tagged with a version number that corresponds to the number of the transaction that made the change.

To use the JVSTM, programmers need to transactify their programs. In Listing 2, we show the changes needed to make both the Client class and the Account class transactional. We replaced the Client’s accounts field with an instance of a transactional versioned set VSet, we encapsulated slot balance from the class Account inside a versioned box, and we made the getTotalBalance method atomic by using the @Atomic annotation.

```java
class Account { VBox<Long> balance; }
class Client { 
  VSet<Account> accounts;
  @Atomic
  long getTotalBalance() { 
    long total = 0; 
    for (Account acc : accounts) { 
      total += acc.balance.get(); 
    } 
    return total; 
  }
}
```

Listing 2. JVSTM-based implementation of the Client class and the Account class from the banking domain example. The bold-faced lines represent new code that was not present in the original implementation of both classes. The method get of a versioned box returns the current value stored in the box.

The JVSTM makes a clear distinction between read-write transactions and read-only transactions. A transaction is read-write if it writes to at least one box, and read-only otherwise. Given that, in the JVSTM, read-only transactions never conflict, registering all the boxes that are read during the transaction in the transaction’s read-set is useless work. Thus, only read-write transactions keep a read-set. Because it is not possible to know beforehand a transaction’s nature, the JVSTM speculatively assumes that a transaction is read-only when it starts, revising this assumption as soon as the transaction tries to write to a box, in which case the transaction is restarted as a read-write transaction.

4. Extending the applicability of memoization

As we saw in Section 2.1, to extend the applicability of memoization to object-oriented programs we need to address two main problems. First, we need to be able to capture all the relevant values for a method’s result, which typically include not only the values received as arguments, but also values belonging to the internal state of some objects. Second, we need to identify which methods have side-effects, so that we may choose either to not memoize them or to collect sufficient information to replicate correctly their behavior. We show in this section how we propose to use STMs to achieve both goals.

1 Source code available at http://web.ist.utl.pt/joao.cachopo/jvstm/
4.1 Finding the relevant state

To solve the problem of constructing the relevant state for a method’s result, we propose to use the support already provided by an STM: If a memoized method executes inside an STM transaction, then all of the memory read operations made by the execution of the method will be registered in the transaction’s read-set. Thus, at the end of the transaction we will know which values were read by the method, thereby capturing the relevant state for this particular method’s result.

This approach has a second advantage. If we recall what was discussed in Section 2.1.1, to correctly handle relevant states, besides registering which values were read, we must be able to store in the cache information regarding the locations from where those values were retrieved from.

Once again the underlying transactional system offers the solution for this problem. Because versioned boxes reify the concept of mutable locations, we can simply store in the cache all the instances of versioned boxes belonging to the read-set, knowing that a particular versioned box instance uniquely identifies a memory location and allows the memoization system to query its current content.

Thus, by storing the read-set in the cache, the next time the memoized method is called we can check if each versioned box that belongs to the stored read-set preserves the same value as when the method originally executed. If so, there is a cache hit and the method’s execution is skipped. Otherwise, we must reexecute the method.

4.2 Identifying side-effects

If we are trying to build an automatic memoization tool that is appropriate for imperative contexts, we must be aware that dealing with methods that are not referentially transparent is crucial not only for a successful memoization process, but more importantly, for determining the applicability of this optimization technique. Thus, we must adopt one of the two following approaches regarding side-effects: (1) offer the conventional semantics associated with memoization, not memoizing methods that produce side-effects, or (2) extend the concept of memoization to methods with side-effects.

Obviously, the first approach is simpler to implement because we need only to identify methods with side-effects and prohibit memoization in such cases. Given that all methods are executing inside an STM transaction to capture the internal state dependencies, once the transaction finishes we may look at the transaction’s write-set to see whether the method wrote to any memory location; if it did, then we do not memoize this call; otherwise, we may memoize it as described before.

A beneficial property of this approach is that we do not classify methods as a whole as producing side-effects or not. Rather, we identify whether a particular method execution produces side-effects or not, maximizing the possibility of using traditional memoization.

Turning our attention now to the second approach identified above, to memoize a method that produces side-effects we need to reproduce its external behavior upon a cache hit. This external behavior includes replicating its return value, as it is done with referentially transparent methods, but also correctly changing all the memory locations that would be written if the method executed.

Once more we may use the STM to obtain the intended behavior: Looking at the write-set, we may see which boxes were written and with what value. So, it is possible to memoize any method that produces memory side-effects if we store the write-set in the cache and in the future, after a cache hit, we iterate over the associated write-set and reapply all the changes as an additional step of the memoization process.

5. The ATOM system

In this section we introduce the Automatic Transaction-Oriented Memoization (ATOM) system. The ATOM system implements the STM-based approach, described in Section 4, of using a transaction’s read-set to capture the relevant state for a particular result and the write-set to identify side-effect-free executions or to register possible write operations, so that it may apply them in future reexecutions of the same memoized method.

5.1 The ATOM API

The ATOM system is implemented as a pure-Java library and, given that we tried to keep its interface as small and simple to use as possible, provides only a single interface for programmers—the @Memo annotation.

Classes with methods that use this annotation are post-processed and rewritten, thus programmers just need to express their intention, as shown in Listing 3. The automatic transformation is done as a step of the compilation phase and uses the ASM [3] library for bytecode manipulation.

```
@Memo
long getTotalBalance() {...}
```

Listing 3. Use of the annotation @Memo to memoize the method getTotalBalance.

For each annotated method, the transformation process inserts in the class a new slot, named after the method to memoize, to hold an instance of a MemoCache. Because Java allows for method overload (several methods may have the same name as long as they differ from each other in terms of the type of the input arguments), to ensure the uniqueness of this name as well as an unambiguous relationship between the memoized method and its respective method cache, each inserted slot’s name is prefixed with “$cache,” and suffixed with the type of the arguments of the memoized method. If the annotated method is static the transformation process inserts a static slot to hold the memo cache.

Then, each memoized method’s body is changed by adding a preamble that does a cache search complemented with the respective decision of whether to execute the method or not, and by preceding each return instruction with a call to the memo cache method that collects information regarding the execution. When this process ends, each new class definition is written over the original class file.

Listing 4 shows the body of a memoized method after being transformed. As we can see, all the memoization behavior is implemented by the MemoCache class, which will be discussed in the next section.

The decision to augment each memoized method with a memo cache relates to the fact that, most often, the receiver of the message (the instance on which the method is being invoked on) influences the outcome of the method. Thus, by spreading the cache by all of the objects of a class, rather than having a single cache for all of them, naturally partitions the cache and simplifies its maintenance, because when an object is garbage-collected, so is the portion of the cache that belongs to it.

5.2 The MemoCache class

The MemoCache class, shown in skeletal form in Listing 5, implements the memoization strategy described in Section 4. Each in-

---

2 Source code available at http://web.ist.utl.pt/hugo.rito/
3 The @Memo annotation provides also the same semantics of the @Atomic annotation, and, thus, replaces it.
class Client {
  VSet<Account> accounts;
  MemoCache $cache_getTotalBalance;

  @Atomic
  long getTotalBalance() {
    Object[] args = new Object[];
    Object res = $cache_getTotalBalance.search(args);
    if (res != MemoCache.notFound) return res;
    long total = 0;
    for (Account acc : accounts) {
      total += acc.balance.get();
    }
    $cache_getTotalBalance.collectInformation(args, total);
    return total;
  }
}

Listing 4. The memoized version of method getTotalBalance. The changes done by the ATOM are highlighted in boldface.

The memo cache (Figure 1) is organized in two levels: The first level is composed of all the information available at call time—the arguments supplied to the memoized method—and maps to a second level which holds information observable only after the method executes—that is, the captured read-set, the returned result, and, possibly, a write-set.

5.3 Implementation of the memoization cache

The memo cache (Figure 1) is organized in two levels: The first level is composed of all the information available at call time—the arguments supplied to the memoized method—and maps to a second level which holds information observable only after the method executes—that is, the captured read-set, the returned result, and, possibly, a write-set.

Because the JVSTM is a multi-versioned STM, we may have concurrent threads executing in different versions of the system. So, to maximize the probability of a cache hit, we allow threads that are executing in a more recent state of the system to update the memo cache without overriding entries that may be useful for lagging transactions. Thus, the second level of the memo cache contains not one but a fixed-sized array of entries. Given that, in general, it is not possible to determine the maximum number of concurrent threads, we limit the size of the array to the number of available processors.

A cache search begins with a table lookup on the first level of the MemoCache, using the list of arguments supplied to the memoized method. Once we obtain a second level array, the search algorithm iterates over all the second level entries looking for a valid read-set, starting with the most recently added one. Intuitively, a read-set is valid if and only if all the boxes in it still hold the same value as when the read-set was stored in the cache.

To see whether a box still has the same value as before, the ATOM system offers three caching policies: version, value, and identity.

The version caching policy considers that a box still has the same value if and only if it is still in the same version (meaning that no write occurred to this box). It is theoretically the fastest of the three caching policies because it just needs to do a simple integer equality test to check if one entry of the relevant state is still valid, but may lead to less hits: from the moment one box changes value, any cache entry that depends on that particular box will forever be invalid. It is also the policy that consumes less memory, because it saves only the versioned boxes read by the memoized method, not caching the values that were read.

The value caching policy checks whether the current value of the box, regardless of its version, is equal to the value seen before. This second policy type may lead to more hits than the previous because it allows a box to change value and still maintain cache entries valid, provided that when the cache entries are checked, the box has the same value as it did when the entries were created. It is also the only cache policy that gives programmers the opportunity to control the validation process by defining their own custom implementation of the method equals, which is used to compare cached values.

The drawbacks of this cache policy are that it assumes that the method equals is correctly defined for the values stored in the boxes and that its time complexity depends on the complexity of such method.

The identity caching policy checks whether each cached versioned box references the same object in memory as the value that was read—that is, if both objects are "=="-equal. If so, then the entry is still valid. This last policy offers not only conventional semantics used in reference models to compare values for equality, but also constitutes an interesting compromise between the other two caching policies: a simple and fast comparison test that allows a box to change value and still maintain cache entries valid.

The decision to implement three caching policies was made to offer programmers an opportunity to choose which equality policy adapts better to the situation—that is, which will translate to a better performance. The memoization system allows this choice to be made on a per-method basis, placing no restrictions on the caching policy used for a particular method.

5.4 Memoization strategy

The addition of new cache entries complies with the semantics defined in Section 4—that is, programmers can opt either to memoize executions that produce side-effects, or not. Hence the MemoCache implements two memoization strategies: read-only and write-allowed.
The read-only memoization strategy follows the traditional memoization behavior of disallowing executions with side-effects to be memoized, whereas the write-allowed memoization strategy caches any method execution, saving the write-set and reapplying it after a cache hit.

5.5 Parameterizing @Memo annotations

Given that the ATOM system allows for three types of comparison policies and also two memoization strategies, the @Memo annotation may be parameterized to inform the transformation process what is the combination of caching policy and memoization strategy that the programmer intends to use in the annotated method. We finally present, in Listing 6, the detailed implementation of the @Memo annotation.

```java
@Retention(RetentionPolicy.CLASS)
@Target(ElementType.METHOD)
public @interface Memo {
    CachePolicyType type() default CachePolicyType.VERSION;
    MemoStrategy strategy() default MemoStrategy.READONLY;
}
```

Listing 6. Implementation of the @Memo annotation.

The attribute type accepts three possible values—VERSION, VALUE, and IDENTITY—and defines how the cache will compare values for equality—using the version of the box, the value of the box, or the object referenced, respectively.

On the other hand, the attribute strategy controls how side-effects are handled by the memoization system. If parameterized with READONLY, any method execution that writes will not be memoized, whereas with WRITEALLOWED all calls to the memoized method will be cached, regardless of whether they produced memory side-effects or not, and the side-effects will be reapplied in future cache-hits, if necessary.

Both parameters are optional and, by default, the instrumentation tool inserts in @Memo annotated methods a read-only cache that uses the version of the box to decide if the value that it holds is still valid.

5.6 Improving cache searches

The complexity of the read-set validation algorithm grows linearly with the number of boxes that need to be validated and depends on the cache policy chosen. Thus, it is a potential bottleneck of the system that we would like to optimize. One way to do so is by validating a read-set only when it is strictly necessary.

In a program that uses the JVSTM, the system evolves through discrete states that are tagged with the number of the transaction that committed and created the state. Combining this fact with the observation that a read-set is known to be valid at creation time and after a successful search, we decided to augment each second level entry of the cache with a list of valid states where we store all the system states for which the read-set is valid.

This way, before validating a read-set, we first check whether the current transaction’s number is stored in the log. If so, we return the cached result, if not we validate the read-set as usual and, if the read-set is indeed valid, we add the current transaction’s number to the log.

5.7 Memory management in the MemoCache

To control memory consumption, the MemoCache adopts two distinct approaches, one for each level of the method cache.

The first level uses weak references to hold the values of the map—that is, second level entries are only weakly referenced. Using weak references helps controlling the amount of memory used by the method cache because, as specified in the Java memory model [9], weak references, unlike strong ones, do not prevent their referents from being made finalizable, finalized, and then reclaimed by the garbage collector (GC). Hence, if at a certain point in time the GC algorithm decides that it needs to free some memory, entries in the cache will be dropped, without affecting cache semantics.

On the other hand, the second level of the cache, as mentioned before in Section 5.3, relies on a fixed-size number of entries, equal to the number of available processors, with a simple round-robin policy.

If a memoized method m1 calls another memoized method m2, m2’s relevant state contains the nested transaction’s read-set, whereas m1’s relevant state contains both the boxes read by m1 and by m2. If we store both relevant states in the cache, m1’s cache entry will contain a subset of boxes that correspond to an entry in m2’s cache. This replication of information is undesirable considering that read-sets are usually very large.

Thus, we decided to use the runtime call hierarchy of memoized methods to compose second level cache entries. Each cache entry now stores only the boxes that are locally read by a method and has a list of other cache entries that correspond to all memoized methods that were called by the current method. With this new approach, m2’s relevant state remains the same, but m1’s relevant state now contains only the boxes read by m1 plus a reference to the relevant state of m2.

5.8 The memoization advisory system

As discussed in Section 2, memoization results in a performance boost only when it is faster to search the cache for a match than to reexecute the original method, and, of course, there is a cache hit. From this observation comes a crucial conclusion: Memoization may deteriorate the performance of the system if programmers memoize methods that never execute under the same state or that take less time to execute than the time it takes the memoization system to search the cache.

Although fundamental, the task of maximizing system performance is difficult, error-prone, and arduous if accomplished by code inspection alone or by reasoning about the intended system behavior. This problem becomes even more complex in the case of the ATOM system because programmers need to worry not only about which methods are beneficial to memoize, but also which memoization strategy and comparison policy is the best to use for each method.

To simplify the whole selection process, we introduced the TestMemoCache class. This new type of method cache subclasses the original MemoCache and is designed to collect per-memoized-method information.

Unlike the conventional MemoCache that enforces a single memoization strategy and a single cache policy, the TestMemoCache is in fact composed of six independent MemoCache instances, one for each possible combination of memoization strategy and cache policy.

Thereby, upon a search request, the TestMemoCache queries each of its private MemoCache instances for a hit, collecting information about the time each took to issue a response and if the response was a cache hit or a cache miss. Each of the six searches is done inside a transaction that is aborted as soon as the TestMemoCache collects enough information about the request. This extra work is necessary to correctly recreate the execution scenario where only one cache is used. In particular it prevents side-effects done by write-allowed caches to influence the outcome of subsequent searches and computations.

Independently of any combination of responses given by each of the six individual caches, a search request made to the TestMemoCache results always in a cache miss, to force the execution
of the original body of the memoized method. This way, the TestMemoCache can collect the time it would take the normal version of the code to execute and compare it with the various memoized solutions.

It is important to note that the time a memoized method takes to complete is equal to the time spent on the cache search, if there is a cache hit, or is equal to that time plus the time spent on executing the method’s original body and on creating the new cache entry, if the cache yielded a miss.

The runtime information collected by the TestMemoCache is then saved to the file system and used as input to the MemoAdvisor class, responsible for calculating the total number of cache hits and cache misses, average execution time, and expected speedup over the unmemoized version, for each combination of cache strategy and comparison policy.

This overall information is presented to the programmer divided in two sections: methods where it is expected that memoization can accelerate their execution and those that do not. The advisory system complements such information with which combination of memoization strategy and caching policy type is best suited for each memoized method. It is then up to the programmer the task of choosing the final configuration of the system.

Listing 7 shows a possible output for the MemoAdvisor with two memoized methods. Only expected average times are shown.

5.9 Extending JVSTM transactions

Our memoization implementation depends crucially on the support offered by the transactional system. More specifically, on the ability to obtain from the JVSTM the read-set and the write-set, as well as the guarantee that all memory accesses are correctly registered in the respective set. But the JVSTM was not designed with memoization in mind. For example, the JVSTM uses read-only transactions as a way to eliminate the overheads of logging memory accesses when they are not relevant, whereas in the ATOM system all memory accesses are relevant, even for read-only methods.

Thus, we decided to extend the already available transactions with a new type of transaction—the MemoTransaction. This choice allowed us to implement memoization-specific semantics and optimizations. Going back to the example given above, to keep all benefits of read-only transactions and still allow correct memoization we created read-only memo transactions that log to the read-set only when executing inside a memoized method.

In concurrent environments, multiple threads may execute concurrently and STMs use the write-set as a buffer for the changes that a transaction wishes to do. These changes will be applied if the transactional system can guarantee that the committing transaction does not conflict with a previously committed one. In this validation process, the read-set is used to ensure that the committing transaction did not read a transactional location that another already committed transaction wrote.

Hence, even in the simpler case where we memoize a read-only method, there are side-effects that may change the semantics of the program: All transactional methods write to the transaction’s read-set. Thus, to replicate the behavior of a memoized transactional method, besides replacing the method call with the cached result, and possibly applying the cached changes, the memoization system needs to populate the transaction’s read-set with the versioned boxes that would be read by the method if it executed normally.

One observation made during early tests with the ATOM system was that it spent too much time populating the read-set to ensure correct transactional semantics and that this action was negatively influencing the performance of the memoization system, especially for methods that read a large number of versioned boxes.

Given that the information registered in the read-set is necessary only if at commit time the transaction is read-write—the JVSTM’s validation task first looks at the transaction’s write-set and it uses the read-set only if the write-set is non-empty—we improved the performance of the memoization system by not populating the read-set when it is known to be not necessary. For example, we may skip the population step after a cache hit on a top-level, read-only, memoized method, because top-level read-only transactions never fail.

The problem is that, after a cache hit on a memoized method executing inside a nested transaction, populating the read-set is unnecessary only if the nested transaction is read-only, its parents are read-only, and other nested transactions that may execute in the context of this transaction are also read-only. Thus, when a nested transaction commits it is not possible to know in advance if the read-set will be necessary in the future, or not.

Because the conservative approach of populating the read-set when a memoized nested transaction commits is very inefficient, even more inefficient in read-dominated contexts where the frequency of read-write transactions is low, the ATOM system enforces a speculative read-only policy on memoized methods.

Upon a cache hit, a promise is added to the running memo transaction regarding the values that would be added to the read-set. This promise is realized immediately if the transaction is already read-write. Otherwise, it is delayed until a future nested transaction tries to write to a box.

6. Evaluation

To evaluate the usefulness of memoization in object-oriented programs we tested the ATOM system with the STMBench7 benchmark. The data structure of the STMBench7 benchmark is similar to those used by CAD programs and consists of a set of graphs and indexes that are shared and are concurrently accessed by a configurable number of threads.

We extended the default implementation of the benchmark, which is lock-based, with a new version that uses the JVSTM. The JVSTM version encapsulates each mutable field of design-library objects inside a versioned box, uses versioned box operations to access the fields, and executes each operation of the benchmark inside a transaction. In our test with the ATOM we replaced the JVSTM transactions with ATOM memo transactions.

Additionally, we created a version of the benchmark, from now on referred as “Plain”, that is the default implementation of the benchmark minus all the locks. The Plain version does not use the JVSTM and was used solely to assess the overheads of using an STM on sequential, single-threaded programs.

In total, the STMBench7 benchmark implements 45 distinct operations. These operations are classified according to their category and their type. In the original benchmark, we can disable all the long traversals, all the structure modification operations, or both. We extended it to allow us to disable only the read-write long traversals, leaving active the read-only long traversals.

The benchmark executes an operation by calling the method performOperation, that either reads or updates the design-object graph, depending on the operation instance on which this method is invoked on. We decided to memoize only the various implementations of the method performOperation because that is where we believe memoization will yield the best performance boost. Unfortunately, many implementations of the method performOperation use random numbers to, for example, index the design-object graph. Thus, we had extra care to memoize deterministic methods only, which led, for example, to not being able to memoize structure modification operations.

To select which methods to memoize, we made a first run of the benchmark with all of the 14 remaining operations annotated with the Memo annotation and using the TestMemoCache. This preliminary test run for 120 seconds with all long read-write traver-
The machine used for these tests can run up to 8 real threads or 16 logical threads with hyperthreading.

Normal READONLY_IDENTITY

The results for the read-write and write-dominated workload can be found in [12].

<table>
<thead>
<tr>
<th>Query7.performOperation()</th>
<th>Memo Time</th>
<th>Normal Time</th>
<th># of Hits</th>
<th># of Misses</th>
<th>SpeedUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITEALLOWED_VERSION</td>
<td>561221</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
<td>15.83</td>
</tr>
<tr>
<td>WRITEALLOWED_VALUE</td>
<td>558751</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
<td>15.83</td>
</tr>
<tr>
<td>WRITEALLOWED_IDENTITY</td>
<td>558458</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
<td>15.83</td>
</tr>
<tr>
<td>READONLY_VERSION</td>
<td>558149</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
<td>15.83</td>
</tr>
<tr>
<td>READONLY_VALUE</td>
<td>558015</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
<td>15.83</td>
</tr>
<tr>
<td>READONLY_IDENTITY</td>
<td>557800</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
<td>15.83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ShortTraversal6.traverse(lstbench7/core/AtomicPart);</th>
<th>Memo Time</th>
<th>Normal Time</th>
<th># of Hits</th>
<th># of Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITEALLOWED_VERSION</td>
<td>145072</td>
<td>125019</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>WRITEALLOWED_VALUE</td>
<td>137762</td>
<td>125019</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>WRITEALLOWED_IDENTITY</td>
<td>136493</td>
<td>125019</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>READONLY_VERSION</td>
<td>127957</td>
<td>125019</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>READONLY_VALUE</td>
<td>127917</td>
<td>125019</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>READONLY_IDENTITY</td>
<td>127873</td>
<td>125019</td>
<td>0</td>
<td>104</td>
</tr>
</tbody>
</table>

Listing 7. Output of the MemoAdvisor with two methods of the STMBenchmark7 benchmark memoized. The time is in nanoseconds.

sals and structural modifications disabled, under a read-dominated workload. We then decided to memoize all the operations that, on average, executed faster when memoized. Therefore, we memoized the operations Query2, Query5, Query6, Query7, ShortTraversal19, Traversal1, Traversal8, and Traversal9.

We present results for a read-dominated workload and with three possible mixes of operations: (1) all operations except long read-write traversals and structural modifications, (2) all operations except long traversals (both read-write and read-only), and (3) all operations except long traversals and structural modifications.

We ran each test five times, removed both the best and the worst throughput values, and averaged the three remaining throughput values. All tests ran for 120 seconds using 1, 2, 4, 8, and 16 threads in a Dual-Quadcore Intel Nehalem-based Xeon E5520, with 12Gb of RAM running Ubuntu Linux 9.04, and Java SE version 1.6.0.16. While the test ran, no other relevant processes were executing in the system.

6.1 Comparing the JVSTM and the ATOM

We show the STMBenchmark7 benchmark throughput results obtained for the various mixes of operations and a read-dominated workload with the JVSTM and with the ATOM in Figure 2.

These results show a clear increase in performance when using memoization. The memoized version performs better than the JVSTM in almost all scenarios, achieving the best results in the first mix of operations (shown in the leftmost graph of Figure 2), where the throughput of the system increases by a factor of 14.

The first mix of operations includes long read-only traversals, which are the most computationally intensive operations in the benchmark—the maximum time to completion of long traversals is over half a second, whereas for the remaining operations is below 9 milliseconds. So, it makes sense that memoization gives the best results for long-traversals.

For the second mix of operations, as we can see in the second graph of Figure 2, for 1, 2, or 4 threads memoization continues to behave better or at least as good as the JVSTM with speculative read-only transactions. The same cannot be said for 8 or 16 threads.

To understand these results it is important to remember that in this second mix of operations we turned on all structural modifications. So, the state of the system is constantly changing, reducing the number of cache hits. Further, more cache misses translate directly to more operations that are not skipped and, thus, concurrently add new cache entries, generating contention in the cache.

This problem with operations that change the structure of the design-object graph, is confirmed by the third mix of operations. We disabled once again all structural modifications and, as we can see, the ATOM outperforms the standard solution, demonstrating clear advantages of using memoization even for operations that are not computationally demanding.

With the second mix of operations, we can see a reduction in performance for both the JVSTM and the ATOM, with 16 threads. This result is typical of the STMBenchmark7 when we have more threads than available processors, because the number of conflicts rises and so do the number of restarted transactions.

Overall, our solution scales well and, given that the STMBenchmark7 benchmark traverses the object graph but performs no operations on the leaves, it is reasonable to expect better results under a more realistic test because the unmemoized version of the system would take longer to complete each operation.

6.2 The STMBenchmark7 Plain version

Because memoization is to be used also in single-threaded environments, where the introduction of an STM is semantically irrelevant but computationally heavy, it is important to assess if the overhead imposed by the usage of a software transactional memory to capture internal state accesses is high enough to negate the performance benefits extractable from memoization.

We show in Table 1 the throughput results for a read-dominated workload and the various operations mixes in a single-threaded environment with the Plain, the JVSTM, and the ATOM versions of the benchmark.

As we can observe, using the JVSTM in single-threaded programs results always in a degradation of performance. At best, the JVSTM version of the benchmark processes half the operations per second, when compared with the non-transactional version. Hence, even though in a single-threaded environment transactions never conflict or abort, STMs introduce a significant overhead that discourages their use.

The same cannot be said about the ATOM version of the benchmark: In the scenario with long read-only traversals, the ATOM is by far the best approach, improving the performance of the Plain

The machine used for these tests can run up to 8 real threads or 16 logical threads with hyperthreading.
version by a factor of 7. This result is even more expressive considering that even for single-threaded environments, the ATOM ensures that the memo cache is accessed in mutual exclusion and read-sets and write-sets are correctly populated and validated.

As expected, in the last two mixes of operations, where the amount of read-only expensive operations is low or the system state is constantly changing, the ATOM performs worse than the non-transactional version, but even on these scenarios, the ATOM outperforms the JVSTM.

Thus, we argue that memoization can be very useful in transactional systems as a way to reduce the overheads imposed by STMs, thereby promoting their adoption. Memoizing methods called inside a transaction may accelerate future executions of the transaction and even reexecutions of the same transaction if the transaction conflicts and then restarts.

Overall, we can conclude that when there are computationally expensive repeated read-only operations, the performance boost we obtain from using the ATOM is enough to compensate not only the time spent on cache operations, but also the overheads imposed by the use of an STM.

7. Related work

Memoization has been the subject of investigation since it was first introduced in 1968. Over the years, many automatic memoization systems were introduced for programming languages such as LISP [8] or C++ [10]. Compared to previous implementations of memoization, our solution is the first automatic memoization system that allows internal state dependencies, that validates choices made by the programmers, and that incorporates memory side-effects within the memoization process.

Incremental computation is a technique that captures the runtime behavior of a function and then adapts its future outputs by recomputing only the parts of the computation that changed. This is accomplished using dependence graphs [1, 5] to capture the data dependencies of computations and by finding the parts of the graph that change between function calls. Although promising, it is a work that still needs to mature. Its current formulation does not deal with concurrent systems and still lacks a mainstream language implementation.

Our approach detects at runtime if a method produces side-effects. Another solution would be to do this classification statically at source code level as proposed by Franke et al. [6] or Rountev [13]. Static solutions have the advantage of introducing no runtime overheads, which is essential to obtain the best speedup possible. The biggest problem with static analysis is that they are conservative: a method is classified as not cacheable when there is, at least, one path of execution where it may cause side-effects.

Xu et al. [15] also applied memoization to Java programs. They dynamically identify methods that are safe to memoize using a solution based on escape analysis and Java bytecode inspection. In their work, memoized methods may do heap reads as long as the retrieved values are reachable from the supplied arguments. They allow such behavior by “flattening” reference arguments—recursively gathering object type and primitive field values for all reachable types—and by incorporating such information in the cache-key. Such solution does not solve entirely the problem of internal state reads because it does not account for reads of static fields or instance fields that influence the outcome of the method, and were not received as input. For example, in the scenario described in Section 2.1.1, and implemented in Listing 1, their solution would still fail to register the appropriate relevant state.

Our solution combines memoization with STMs, but Ziarek and Jagannathan [16] were the first to apply memoization in transactional environments. The aim of their work was to use memoization to prevent the reexecution of operations that do not conflict in transactional environments, as a way to accelerate forced reexecutions and not as way to speedup repeated operations, as we propose. Another difference between both works is that Ziarek and Jagannathan’s memoization system assumes that a method’s result depends solely on the list of arguments, not capturing internal state dependencies.

<table>
<thead>
<tr>
<th>Version of the Benchmark</th>
<th>Plain</th>
<th>JVSTM</th>
<th>ATOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>No read-write traversals / No structural modifications</td>
<td>154</td>
<td>79 (0.51)</td>
<td>1131 (7.34)</td>
</tr>
<tr>
<td>No traversals</td>
<td>6291</td>
<td>2311 (0.37)</td>
<td>3186 (0.51)</td>
</tr>
<tr>
<td>No traversals / No structural modifications</td>
<td>7719</td>
<td>3460 (0.45)</td>
<td>6564 (0.85)</td>
</tr>
</tbody>
</table>

Table 1. Operations per second processed by the Plain, the JVSTM, and the ATOM versions of the benchmark, for a read-dominated workload under the three different mixes of operations and one thread. In parentheses is shown the speedup relative to the Plain version.
8. Conclusion

The main goal of our work was to make memoization more appealing and appropriate to the unique characteristics of object-oriented programs. In particular, we identified the difficulty of constructing the list of relevant values for the output of a method, when this list includes the arguments of the method as well as values read from the internal state of some objects, as the major obstacle to using memoization in object-oriented systems. The fact that methods may also change the state of objects, i.e., produce memory side-effects, is an additional factor that hinders and influences the applicability of memoization in object-oriented programs.

In this paper we proposed to use Software Transactional Memory to extend the applicability of memoization. Our solution is to wrap methods in transactions to obtain all the information that is needed for memoization.

We described an automatic memoization tool—the ATOM system—that implements our extended memoization model for object-oriented programs in a flexible and easy-to-use way, requiring minimum knowledge from programmers. The ATOM system captures automatically the relevant state for a memoized method’s result, identifies side-effect-free methods that are safe to memoize, and captures memory write operations to the internal state of objects so that they can be reapplied in future reexecutions of the method, thus extending memoization even to methods that are not referentially transparent.

We developed a memoization advisory tool that aids programmers to find the methods that can benefit from memoization. Because the ATOM system offers three caching policies and a couple of memoization strategies, the advisory tool not only lists methods that memoization can accelerate, but also what is the best combination of caching policy and memoization strategy to use, in a per-method basis.

For now, the selection of memo methods requires programmers to memoize all the methods of the system, or at least those that they expect will execute repeated work, and then run some representative portion of the use cases of the application. In the future we intend to automate the selection process and even allow the ATOM system, at runtime, to turn on/off memoization in methods, adapting the system to its current workload.

We can conclude from our tests that as long as there are read-only repeated operations, the ATOM system is able to improve the performance of the underlying system in a transparent, simple, and flexible way. We have shown as well that, regardless of whether, as a whole, the system is functionally pure or not, programmers can expect the same behavior and beneficial characteristics of memoization in object-oriented contexts if they adopt our STM-based solution.

Because STMs already do all of the expensive work of collecting the relevant information for the memoization tool, this extended memoization approach comes for free for a system that already uses an STM. Moreover, because it increases the performance of the system at almost no extra cost, it amortizes the upfront cost of using an STM, thereby promoting the adoption of STMs.

We strongly believe that STMs have much to gain from memoization and we intend to explore such synergy in the future. For once, memoization can be used to accelerate the reexecution of transactions that abort at the validation step. In fact, this comes for free if all transactions are memoized and may even allow transactions that will conflict to reach the validation step faster. Second, because memoization gives a different end to the information already being generated by STMs, it lowers the perceived cost of constructing it in the first place.

Because the effectiveness of the ATOM system is strongly correlated with the time spent on cache operations, we intend to experiment in future work with different cache strategies. In particular, we will explore an alternative to the read-set validation algorithm that, instead of asserting if each versioned box contained in the cache is still valid, relies on an invalidation protocol that automatically removes from the cache all entries that are made invalid by a successful write to a versioned box.

Acknowledgments

We wish to thank the reviewers for all the insightful comments, which have helped us to improve the final version of the paper.

References