ATON: Automatic Transaction-Oriented Memoization

Hugo Rito

INESC-ID Lisboa/Technical University of Lisbon
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
hugo.rito@ist.utl.pt

Abstract. Memoization is a technique for improving the performance of a program, but it has been confined mostly to functional programming. I propose an extended memoization approach that takes advantage of the support provided by a Software Transactional Memory (STM) to remove many of the difficulties of applying memoization in non-functional programs. I argue that my memoization system is the first to suit the needs of imperative object-oriented programming, and show that it may be implemented almost for free in systems that use an STM. I describe the major design and implementation decisions of my system and present results for a benchmark that show a 14-fold increase in the throughput of the system when using my memoization system.

Key words: Memoization, function caching, object-oriented imperative programming, software transactional memory

1 Introduction

Memoization, also known as function caching, was first introduced in 1968 [1] in the context of Artificial Intelligence as a way for machines to learn from past experiences, as if programs could “recall” previous computations and thus avoid repeated work. The key idea behind it 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.

Memoization is often used in functional programming to increase the performance of a program, but is seldom used in other programming paradigms, such as object-oriented imperative programming. The fundamental reasons for this is that it is difficult to capture the list of relevant values for the output of a method if it accesses shared state and it is not possible to memoize a method if it produces side-effects.

In this work I describe the Automatic Transaction-Oriented Memoization (ATOM) system, which takes advantage of an implementation of a Software Transactional Memory (STM) for Java to extend the applicability of memoization to object-oriented imperative programs.

To the best of my knowledge, the ATOM is the first automatic memoization system that has any of the following benefits: (1) it makes easy memoizing

⋆ This work was partially funded by an FCT research grant in the context of the Pastramy project (PTDC/EIA/72405/2006).
methods that access shared state, (2) it prevents errors in memoizing unintended methods, and (3) it allows memoizing methods that have side-effects. I argue that my extended memoization system is the first memoization system to suit the needs of object-oriented imperative programming, and that it may come almost for free for systems that already use an STM.

In the next section I review what memoization is and what its major limitations are. Then, in Section 3, I introduce the concept of Software Transactional Memory, with special emphasis on the Java Versioned STM. In Section 4, I present the core ideas of my work, and then I describe its implementation in Section 5. In Section 6, I discuss how memoization can be particularly useful when combined with an STM and present some results obtained from the STM-Bench7 benchmark. In Section 7, I discuss related work, and, finally, in Section 8, I draw some conclusions and discuss future work.

2 Memoization

Memoization is a well-known technique for improving 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.

Unmemoized functions usually take some values as arguments and return another value that is computed from those arguments, regardless of whether that same computation was already done before. A memoized function, on the other hand, first checks whether it has already performed the asked computation in the past by searching its cache. If a mapping is found, it returns the stored result to the caller. If it is the first time that such computation is done, the requested result is computed and, just before its return to the caller, a new mapping is added to the cache.

A performance boost results when it is faster to search the cache for a match than to reexecute the function, and, of course, there is a cache hit (meaning that the result stored in the cache is returned). This gain in performance must be sufficient to compensate for possible cache misses and the cost of constructing and managing cache entries.

In pure functional programs, because the output of a function is fully defined by the supplied arguments, the memo cache simply holds an arguments-to-result association. As we will see in the next section, the same approach cannot be followed for non-functional environments.

2.1 Limitations of memoization

Consider the method method1 in Listing 1. If we want to memoize it, we insert a map in Class1 to store the memo information and change the body of method1 to search the map before computing the intended result. Assuming that the value of slot may change over time, it is clear that the solution used in functional programming, of just using the arguments of method1 to search in the memo cache, is not enough: The value of slot is relevant to determine which value to return from method1, also.
public int method1(int arg1, boolean arg2) {
    int res = (slot
        ? /* Something expensive */
        : /* Something else expensive */);
    return res;
}

Listing 1: Example of a method that returns a result that is influenced by an object’s state—the value of slot of class Class1.

A naive approach would be to extend the memo cache to include the value of slot in the keys of the cache map, too. 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, 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. This is one of the difficulties of applying memoization to object-oriented programs: Methods often depend on the state of other objects, many of which cannot be easily determined by code inspection alone. Still, in this example, method1 just reads values from the objects’ state.

public int method2() {
    int res = /* Expensive computation */
    slot = !slot;
    return res;
}

Listing 2: Example of a method that changes an object’s state and that, thus, cannot be memoized.

Consider now the method depicted in Listing 2. This method behaves almost like method1, but in addition to method1’s behavior it toggles the value of slot. A simple memoization solution cannot be applied here because method2 is not referentially transparent: Calling method2 does not have the exact same effect as replacing the method call with its return value. Again, methods such as this prototypical example are common in object-oriented programs.

3 Software Transactional Memory

Software Transactional Memories (STM) [2] introduced the concept of transaction in mainstream programming, mostly as a way to simplify concurrent programming. The key idea behind STMs is that programmers specify which operations must execute atomically, leaving to the transactional framework the responsibility of providing the intended atomicity semantics, while maintaining as much parallelism as possible.

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 locations, so that they may detect when two concurrent trans-
actions are interfering with one another, in which case the STM stalls, aborts, or restarts at least one of the transactions.

There are numerous STM implementations, varying considerably in how they ensure the atomicity of operations. Therefore, I will concentrate our discussion in a particular STM implementation, the Java Versioned STM (JVSTM) [3,4]. JVSTM is a multi-versioned STM implemented as a pure Java library and it is the STM implementation that I use as a basis for the implementation of the ATOM system that I describe in this paper.

The JVSTM introduces the concept of versioned boxes as a replacement for shared memory locations. Conventional memory locations keep only a single value but a versioned box is a container that holds a tagged sequence of values representing all of the changes made to that location by some transaction.

To detect conflicts among concurrent transactions, JVSTM logs into a per-transaction read-set which boxes, and respective versions, were read inside that transaction. Likewise, it logs into a write-set which boxes were written and with what value; only at commit-time will these values be effectively written to the boxes, and only if the transaction is valid.

4 Extending the applicability of memoization

To extend the applicability of memoization to object-oriented programs, as we saw in Section 2.1, we need to address two problems. First, we need to be able to capture all of the relevant state used to compute a result, other than the values received as arguments of a method; typically, this state includes values belonging to the program’s shared state. 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 correctly replicate their behavior. We shall show in this section how STMs can help to achieve both goals.

4.1 Finding all of the relevant state for memoization

Recalling the example of method1 shown in Listing 1, the relevant state for the result returned by method1 includes the value of slot, besides, presumably, the values of arg1 and arg2. The problem, as I discussed then, is finding all of the relevant state in more complex cases.

To solve this problem, I propose to use the support already provided by an STM. If method1 executes inside an STM transaction, then all of the memory read operations made within that transaction will be registered in the transaction’s read-set. Thus, at the end of the transaction we will know which values were read to compute the method’s result, thereby capturing all of the relevant state for the computation of this particular result.

4.2 Identifying side-effects

Leveraging again on the support given by an STM, I argue that it is possible to know that we can safely memoize a method, based on the fact that it produces

---

1 The source code of the JVSTM is available at [http://web.ist.utl.pt/~joao.cachopo/jvstm/](http://web.ist.utl.pt/~joao.cachopo/jvstm/)
no side-effects, if we execute the method inside an STM transaction. Thus, once it finishes, we may look at the transaction’s write-set to see whether the method wrote to any shared state; if it did, then we do not memoize this call; otherwise, we may memoize it as before. There is a subtle detail here which is worth mentioning. With this approach we are not identifying whether methods as a whole produce side-effects or not. Rather, we are identifying, on a per-call basis, whether that call produces side-effects or not.

To extend the concept of memoization to imperative methods, we must be able to reproduce the behavior of non-side-effect-free methods. Going back to the example shown in Listing 2, what does it mean to reproduce the behavior of method2, which has side-effects?

The externally observable behavior of method2 is the return of its result and the change made to slot. So, to reproduce the behavior of method2 it is necessary to replicate its return value and, also, change the value of slot.

The answer comes again from the STM. Looking at the write-set, we may see which boxes were written and with what value. So, it is possible to memoize method2 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

To implement my STM-based approach to memoization of object-oriented programs, I developed the Automatic Transaction-Oriented Memoization (ATOM) system. ATOM encapsulates memoized methods inside transactions to capture all of the relevant state for a particular result and to register possible write operations, so that it may apply them in future reexecutions of the same method.

5.1 The ATOM API

The Memo annotation constitutes the ATOM’s visible interface. Classes with methods that use this annotation are post-processed and rewritten. Thus, to use the ATOM, programmers just need to express their intention, as shown in Listing 3, and the system automatically does the appropriate changes (shown in Listing 4).

```java
@Memo
int method1(int arg1, boolean arg2) {
    /* method1’s Body */
}
```

Listing 3: Use of the annotation Memo to memoize the method method1. The necessary code is introduced by post-processing the bytecode, originating the code shown at Listing 4.

For each memoized method, a private instance of the class MemoCache is added to the respective class. The rational behind this decision is that I consider that, most often, the receiver of the message (the instance on which the method
private MemoCache $cache_method1_I_Z;

@Atomic
public int method1(int arg1, boolean arg2) {
    Object[] args = new Object[2];
    args[0] = arg1;
    args[1] = arg2;

    Object out = $cache_method1_I_Z.search(args);
    if (out != MemoCache.notFound) {
        return (Integer)out;
    }

    int res = // original method1's body */
    $cache_method1_I_Z.collectInformation(args, res);
    return res;
}

Listing 4: Memoized version of method1, after being post-processed. The Atomic annotation is a JVSTM annotation that marks a transactional 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.

The MemoCache instance is responsible for providing all the memoization semantics. The method search receives a list of arguments and is responsible for searching the cache for a hit. If the table lookup fails or none of the read-sets are valid, it returns a statically constructed instance of the class NotFound. Otherwise, the associated result is returned and, if necessary, the stored write-set is applied.

Responsible for collecting all the relevant information about a method’s execution, the method collectInformation receives the arguments passed to the original method, the result the original method will return, and it is responsible for extracting the read-set (and write-set) from the associated transaction.

First Level
Arg1, ... , ArgN
Arg1, ... , ArgN
... ...
Arg1, ... , ArgN

Second Level
Read-Set Result + Write-Set
Read-Set Result + Write-Set
... ...
Read-Set Result

Fig. 1: Organization of the MemoCache. The first level is composed of all the information available at call time 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.
The MemoCache is organized in two levels as show in Figure 1. I use a table lookup to search the first level of the MemoCache, using the supplied arguments as the key. Once we obtain a second level entry, the search algorithm iterates over all its entries looking for a valid read-set. 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, I experimented with three caching policies. The first policy, implemented by VersionRelevantState, 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). The second policy, implemented by ValueRelevantState, checks whether the current value of the box, regardless of its version, is equals to the value seen before; for this approach, I assume that the method equals is correctly defined for the values stored in the boxes. The third policy, implemented by IdentityRelevantState, checks if both the box and the cached entry reference the same (“==”) object instance.

6 Memoizing transactions

STMs introduce overheads in the form of intercepted memory accesses, extra memory to store transactional information, time spent validating transactions, and in the reexecution of conflicting transactions.

Just like any other program, STMs could benefit from memoization. But applying memoization in a transactional system can be particularly useful for at least two reasons. First, as it uses information that is already generated by the STMs, it amortizes that cost by speeding up memoized methods with no significant added cost. Second, because it can accelerate the reexecution of conflicting transactions by avoiding repeated computations.

6.1 Case study: the STMBench7 benchmark

The STMBench7 benchmark [5] is a highly customizable benchmark, developed for evaluating STM implementations. Its data structure is similar to CAD programs, consisting of atomic parts, assemblies, composite parts, connections, documents, manuals, and modules.

The STMBench7 benchmark allows for additional synchronization strategies to be implemented. So, a new one was introduced that wraps each operation with a JVSTM transaction. To test the effectiveness of the memoization approach, I made a first run of the benchmark to extract the operations that execute faster when memoized, which resulted in the operations Query2, Query5, Query6, Query7, ShortTraversal9, Traversal11, Traversal18, and Traversal19. Then, I ran a series of tests with these selected operations memoized to obtain the results presented below.

Each test ran for 120 seconds 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_13.

I present results for the JVSTM and for the JVSTM with ATOM, using 1, 2, 4, 8, and 16 threads. All tests were for a read-dominated workload and with one
of 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.

6.2 Experimental results
The results obtained for the three mixes of operations are shown in Figure 2 through 4.

These results show a clear increase in performance when using memoization in these test cases: The memoized version performs better than the plain JVSTM in almost all scenarios, achieving the best results in the first mix of operations (shown in 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. So, it makes sense that memoization gives the best results for long-traversals.

These results are even more expressive if we take into account the fact that the run of JVSTM without memoization uses optimized read-only transactions that do not log reads in a read-set, whereas the memoized version needs to compute a read-set to be able to memoize the operations. Thus, this means that the benefits of memoization pay off the cost of computing very large read-sets in this case.

Still, even if we disable all of the expensive traversals, leaving only short operations in the mix, the results depicted in Figure 3 and 4 show that memoization behaves better than the JVSTM without memoization, demonstrating clear advantages of using memoization even for operations that are not computationally demanding. The use of memoization in the third mix almost doubles the throughput of the system for 8 threads.

The only case where the memoized version performs worse than the plain version is when we have 8 and 16 threads in the second mix. This result shows that structural modifications negatively influence memoization because the state of the system constantly changes, reducing the number of cache hits.

Overall, these results show that memoization is able to improve significantly the performance of read-dominated workloads. My solution scales almost perfectly and given that the STMBench7 benchmark traverses the object graph but performs no operations on the leaves, it is reasonable to expect better results under a more realistic test.

7 Related work
Memoization has been the subject of investigation since it was first introduced in 1968. Many automatic memoization systems were introduced through the years for programming languages such as LISP [6] or C++ [7]. Compared to previous implementations of memoization, my solution is the first automatic memoization system that allows shared state dependencies, that validates choices made by the programmers, and that incorporates side-effects within the memoization process.

To identify functional methods in Java, Xu et al [8] proposed a dynamic solution based on escape and Java bytecode analysis. They applied memoization
Fig. 2: Operations per second processed by the JVSTM with and without memoization, for a read-dominated workload with all long read-write traversals and structural modifications disabled.

Fig. 3: Operations per second processed by the JVSTM with and without memoization, for a read-dominated workload with all long traversals disabled.

Fig. 4: Operations per second processed by the JVSTM with and without memoization, for a read-dominated workload with all long traversals and structural modifications disabled.
to weakly pure methods—those that only read shared state if it can be reached from supplied arguments and perform write operations as long as it is not to shared state. Even though it is an improvement over systems that memoize only pure functions, this approach is still too limited to be applied to object-oriented programs.

8 Conclusions and future work

In this paper I proposed to use Software Transactional Memories to extend the applicability of memoization. This extended memoization system allows for shared state dependencies, programmers' intent validation, and even the incorporation of side-effects in the memoization process.

My solution is to wrap methods with STM transactions to obtain all the information that is needed for the memoization system. This information comes from the STM, that already collects all the relevant information, which means that 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.

I implemented this approach in Java, as an extension to the Java Versioned STM, and discussed how memoization can help improve the performance of nonblocking STMs, specially in read-dominated workloads where the tests that I performed show an improvement of over fourteen times the original throughput of the system.

A promising application of my memoization approach is in the restarting of conflicting transactions, if we memoize most of the methods ran within a transaction. This is an area that I intend to pursue in the future, as well as a relevant state invalidation strategy.

References