

ATOM: Automatic Transaction-Oriented Memoization^{*}

Hugo Rito and João Cachopo

INESC-ID Lisboa/Technical University of Lisbon
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
`hugo.rito@ist.utl.pt`, `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, because it applies only to pure functions.

In this paper, we propose an extended memoization approach that takes advantage of the support provided by a Software Transactional Memory (STM) to remove many of the limitations of traditional memoization. We argue that our extended memoization system is the first to suit the needs of object-oriented programming, and show that it may be implemented almost for free in systems that use an STM.

We describe the major design and implementation decisions of our system and present results for a benchmark that show a 3-fold increase in the throughput of the system when using our memoization system.

Key words: Memoization, function caching, object-oriented 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 programming. The fundamental reason for this is that traditional memoization is limited to functions that produce no side-effects and do not access any shared mutable state. Thus, this limitation rules out most of the code that is used in traditional object-oriented programming.

In this work we describe the Automatic Transaction-Oriented Memoization (ATOM) system, which takes advantage of an implementation of a Software

^{*} This work was partially supported by the Pastramy project (PTD-C/EIA/72405/2006).

Transactional Memory (STM) for Java to extend the applicability of memoization to object-oriented programs.

To the best of our knowledge, ATOM is the first automatic memoization system that has any of the following benefits: (1) it allows memoizing methods that access shared state (rather than pure functions that depend only on their arguments), (2) it prevents errors in memoizing unintended methods, and (3) it allows memoizing methods that have side-effects. We argue that our extended memoization system is the first memoization system to suit the needs of object-oriented programming, and that it may come almost for free for systems that already use an STM.

In the next section we review what memoization is and what its major limitations are. Then, in Section 3, we introduce the concept of Software Transactional Memory, with special emphasis on the Java Versioned STM. In Section 4, we present the core ideas of our work, and then we describe its implementation in Section 5. In Section 6, we discuss how memoization can be particularly useful when combined with an STM and 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 well-known technique for improving the performance of programs in exchange of extra memory space: To avoid future redundant 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 with the currently supplied arguments. 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 representing this new arguments-to-result association.

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.

2.1 Limitations of memoization

Normally, memoization may be applied only to pure functions. So, it is not applicable to functions that are not deterministic, functions with behaviors that depend on the program's mutable state, or functions that have side-effects such as doing I/O or changing the program's state. Therefore, memoization is seldom applicable to methods of classes in an object-oriented program: Typically, these classes have internal state that affects the result of their methods.

```
public class Class1 {
    boolean slot;

    public int method1(int arg1, boolean arg2) {
        int res = /* Expensive computation */
        return res;
    }
}
```

Listing 1: Typical class in Java.

To better explain the limitations of applying memoization to object-oriented programs, we will use the simple example of a Java class shown in Listing 1.

If we want to memoize `method1`, we may 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. This is safe as long as the “expensive computation” expression is purely functional. But what happens if that expression accesses `slot`? For instance, consider the refinement of `method1` depicted in Listing 2.

```
public int method1(int arg1, boolean arg2) {
    int res = (slot
               ? /* Something expensive */
               : /* Something else expensive */);
    return res;
}
```

Listing 2: Example of a method that returns a result that is influenced by an object’s state—the value of `slot` of class `Class1`.

Assuming that the value of `slot` may change over time, it is clear that the simple solution 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.

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.

Consider now the method depicted in Listing 3. 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.

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

Listing 3: Example of a method that changes an object’s state and that, thus, cannot be memoized.

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 transactions 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, we 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 library¹ and it is the STM implementation that we use as a basis for the implementation of the ATOM system that we 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. These read-sets and write-sets are the key elements that will allow us to extend the applicability of memoization.

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

¹ The source code of the JVSTM is available at <http://web.ist.utl.pt/~joao.cachopo/jvstm/>

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

To overcome the limitations of memoization regarding shared state accesses, it is first necessary to understand what is the relevant state for a particular method's execution. We define relevant state as all the values that are significant for obtaining a particular result so that if at least one of those values change the result may be different.

Recalling the example of `method1` shown in Listing 2, the relevant state for the result returned by `method1` includes the value `slot`, besides, presumably, the values of `arg1` and `arg2`. The problem, as we discussed then, is finding all of the relevant state in more complex cases.

To solve this problem, we 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

If a method that we intend to memoize has side-effects, we may adopt one of the two following approaches:

- Offer the conventional semantics associated with memoization and prohibit the memoization.
- Extend the concept of memoization to imperative methods, so that the execution of a memoized method reproduces the side-effects that it would produce if it was not memoized.

Obviously, the first approach is simpler to implement. Still, it is traditionally left to the responsibility of the programmer to decide when to memoize or not a given method, which is error-prone and suboptimal (because the programmer may decide not to memoize a given method because it may produce side-effects, even though most of the times it does not).

Leveraging again on the support given by an STM, we argue that it is possible to automate this approach with negligible costs: If we execute a method inside an STM transaction, 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. Also, we are relying on the STM to be able to identify all of the possible side-effects, which may exclude side-effects such as doing I/O that are an open issue still in the area of STM research.

We turn our attention now to the second approach identified above of being able to memoize method calls with side-effects. Going back to the example shown in Listing 3, 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, correctly 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 our STM-based approach to memoization of object-oriented programs, we 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 memo cache

For each memoized method, a private instance of the class `MemoCache` must be added to the respective class. The rationale behind this decision is that we consider 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.

The `MemoCache` instance is responsible for providing all the memoization semantics. It assumes that there is an active transaction associated with the current method's execution and it is organized in two levels as shown in Figure 1.

We 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, we experimented with two caching policies. The first policy, implemented by `VersionMemoCache`, considers that a box still has the same value if and only if it is still in the

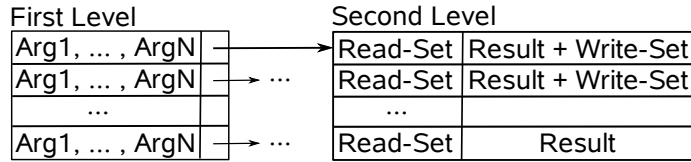


Fig. 1: Organization of the `MemoCache`. The first level is composed of all the information available at call time and maps to a second level which holds information only observable at runtime—that is, the captured read-set, the returned result, and, possibly, a write-set.

same version (meaning that no write occurred to this box). The second policy, implemented by `ValueMemoCache`, checks whether the current value of the box, regardless of its version, is equals to the value seen before; for this approach, we assume that the method `equals` is correctly defined for the values stored in the boxes.

5.2 The ATOM API

The `MemoCache` class, shown in Listing 4, constitutes the only visible interface for programmers.

```

public class MemoCache {
    public Object search(List<Object> args) { ... }
    public Object collectInformation(List<Object> args ,
                                    Object res) { ... }
}

```

Listing 4: Skeleton of the class `MemoCache`. The two methods shown are the basic operations needed to memoize methods with the ATOM system.

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, a `NotFoundException` is thrown. 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 a list with 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.

Because there is a very clear pattern to memoize methods, the ATOM system provides a method annotation to simplify the process: the `Memo` annotation.

Classes with methods that use this annotation are post-processed and rewritten, simplifying the task of memoizing methods: Programmers just need to express their intention, as shown in Listing 5, and the system automatically does the appropriate changes (shown in Listing 6).

As there are two possible caching policies, the `Memo` annotation may be parameterized with the appropriate `type: version` (which is the default) or `value`.

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

Listing 5: 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 6.

6 Memoizing transactions

Just like any other program, STMs could benefit from memoization, but we will see that applying memoization in a transactional system can be particularly useful.

In the JVSTM, as in most nonblocking STMs, changes made during a transaction are made permanent only at 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.

So, 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.

We argue that memoization can be very useful in this context 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.

There is a single module connected to a deep trinomial tree of assemblies. Base assemblies, situated at the bottom-level, are comprised of several composite parts, which contain a document and a graph of atomic parts. This graph is connected through connection objects allowing bottom-up as well as top-down traversals.

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, we made a first run of the benchmark with all of the 14 operations of the benchmark

```

private MemoCache method1_I_Z$cache;

public int method1(int arg1, boolean arg2) {
    List<Object> args = new ArrayList<Object>(2);
    args.add(arg1);
    args.add(arg2);

    Transaction.begin();
    try {
        int res = method1_I_Z$cache.search(args);
        Transaction.commit();
        return res;
    } catch (NotFoundException e) {
        /* First Execution, so fall-through to execute method */
    }

    int res = method1_I_Z$memoized(arg1, arg2);
    method1_I_Z$cache.collectInformation(args, res);
    Transaction.commit();
    return res;
}

private int method1_I_Z$memoized(int arg1, boolean arg2) {
    /* original method1's body */
}

```

Listing 6: Memoized version of `method1`, after being post-processed.

annotated with `@Memo(type="version")`. Then, we selected the operations with the higher cache-hit rate and performance boost, which resulted in the operations `Query2`, `Query7`, `ShortTraversal1`, `Traversal1` and `Traversal8`. Then, we 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.

We 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 3. 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 around 1 second, whereas for the remaining operations is below 10 milliseconds. 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 second mix almost doubles the throughput of the system for 4 threads.

The only case where the memoized version performs worse than the plain version is when we have 16 threads in the third mix, in which case both versions reduce their throughput. This result is typical of the STMBench7 when we have more threads than available processors, because the number of conflicts rise and so do the number of restarted transactions.

Overall, these results show that memoization is able to improve significantly the performance of read-dominated workloads. Our 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, our 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 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.

Ziarek and Jagannathan [9] first proposed the use of memoization to prevent the reexecution of operations that do not conflict, but their work assumed a

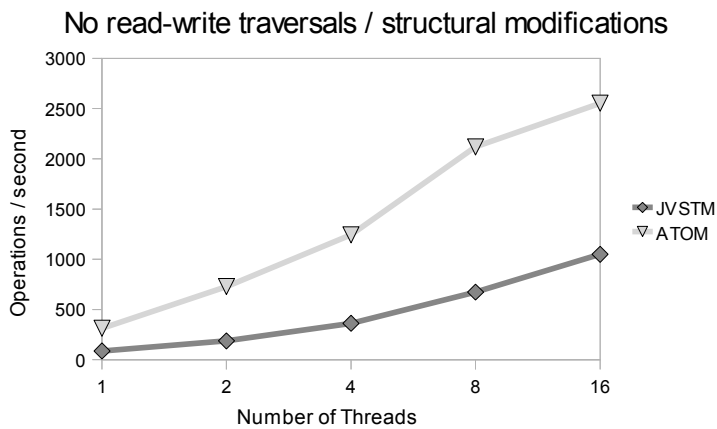


Fig. 2: Operations per second processed by the JVSTM with and without mem-
oization, for a read-dominated workload with all long read-write traversals and
structural modifications disabled.

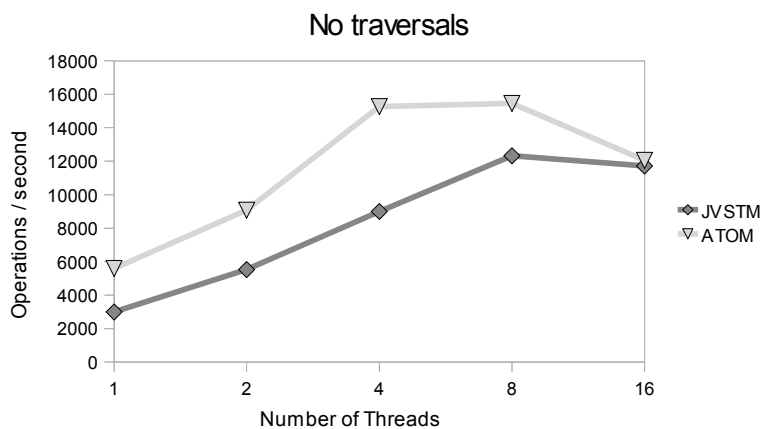


Fig. 3: Operations per second processed by the JVSTM with and without mem-
oization, for a read-dominated workload with all long traversals disabled.

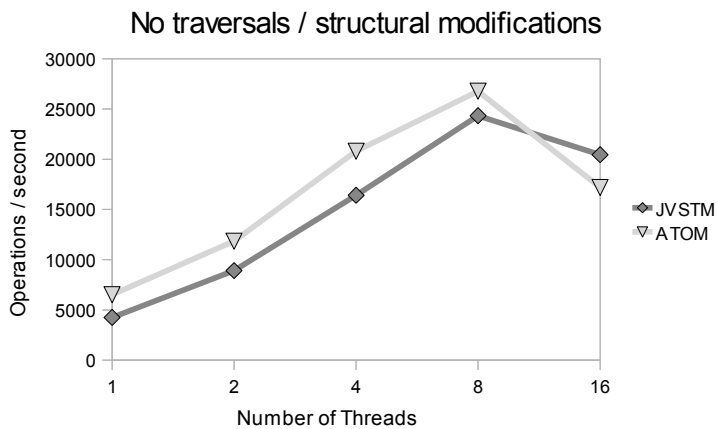


Fig. 4: Operations per second processed by the JVSTM with and without mem-
oization, for a read-dominated workload with all long traversals and structural
modifications disabled.

functional environment and only applies memoization to aborted transactions. By contrast, our work imposes no constraints on the characteristics of the system and allows the application of memoization to any kind of transaction.

8 Conclusions and future work

In this paper we 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.

Our 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.

We 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 we performed show an improvement of over three times the original throughput of the system.

A promising application of our 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 we intend to pursue in the future, as well as experimenting with other benchmarks.

References

1. Michie, D.: Memo functions and machine learning. *Nature* **218**(1) (1968) 19–22
2. Shavit, N., Touitou, D.: Software transactional memory. In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, ACM Press New York, NY, USA (1995) 204–213
3. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. *Science of Computer Programming* **63**(2) (2006) 172–185
4. Cachopo, J.: *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Technical University of Lisbon (September 2007)
5. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: A Benchmark for Software Transactional Memory. *ACM SIGOPS Operating Systems Review* **41**(3) (2007) 315–324
6. Hall, M., Mayeld, J.: Improving the performance of AI software: Payoffs and pitfalls in using automatic memoization. In: *Proceedings of the Sixth International Symposium on Artificial Intelligence*. (1993) 178–184
7. McNamee, P., Hall, M.: Developing a tool for memoizing functions in C++. *ACM SIGPLAN Notices* **33**(8) (1998) 17–22
8. Xu, H., Pickett, C., Verbrugge, C.: Dynamic purity analysis for java programs. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ACM Press New York, NY, USA (2007) 75–82
9. Ziarek, L., Jagannathan, S.: Memoizing multi-threaded transactions. In: *Workshop on Declarative Aspects of Multicore Programming*. (2008)