Abstract

Transactions are a common solution for enterprise applications with high performance and concurrency demands. Over the last years, the use of Software Transactional Memories (STM) introduced a more expressive programming model allowing simpler and cleaner programs.

While dealing with concurrency problems, the use of transactions introduced computational overheads in the form of extra structures needed to implement the STM system and in the form of rollbacks.

If, at commit time, a transaction is marked as not committable, it must be restarted and all the work done meanwhile discarded. When this happens, a lot of possibly valid computations must be redone. This fact is even more wasteful in presence of long-running transactions.

Memoization is one way to solve the problem by trading time for memory usage. By caching the result yielded by a function if, in the future, the exact same computation should be redone, the system can use the previous result and prevent repeated work.

This paper describes the common uses of memoization and how it can be applied to solve the problem at hand, while taking advantage of a particular Software Transactional Memory (JVSTM).

Keywords: Memoization, transactions, partial re-execution
ATOM: Automatic Transaction-Oriented Memoization *

Hugo Rito and João Cachopo
Instituto Superior Técnico

Abstract. Transactions are a common solution for enterprise applications with high performance and concurrency demands. Over the last years, the use of Software Transactional Memories (STM) introduced a more expressive programming model allowing simpler and cleaner programs.

While dealing with concurrency problems, the use of transactions introduced computational overheads in the form of extra structures needed to implement the STM system and in the form of rollbacks.

If, at commit time, a transaction is marked as not committable, it must be restarted and all the work done meanwhile discarded. When this happens, a lot of possibly valid computations must be redone. This fact is even more wasteful in presence of long-running transactions.

Memoization is one way to solve the problem by trading time for memory usage. By caching the result yielded by a function if, in the future, the exact same computation should be redone, the system can use the previous result and prevent repeated work.

This paper describes the common uses of memoization and how it can be applied to solve the problem at hand, while taking advantage of a particular Software Transactional Memory (JVSTM).

Key words: Memoization, transactions, partial re-execution

1 Introduction

Enterprise applications, although difficult to define, are commonly described as involving complex and rich domain models, a great variety of user interfaces and large amounts of data to process, manage, and persist in some form of permanent storage system.

Even with modern multi-processor computers, performance continues to represent a big concern when designing and implementing a new system, especially for those meant to provide interaction with users who submit requests and wait for the respective response. The faster a request is processed and replied, the sooner it frees resources that become available for concurrent and subsequent computations.

* This work is being performed in the scope of the Pastramy project (PTDC/EIA/72405/2006).
So, a common requirement is the ability to process multiple requests simultaneously. But, with so many users concurrently using the system, it is sometimes difficult for programmers to ensure that, just because there are at least two users accessing some data or service at the same time, the system does not behave in an incorrect or unexpected manner.

One way to solve this problem, which is now the *de facto* standard in database systems, is to encapsulate the needed computation in an atomic transaction, relieving the programmer of all the concurrency and failure aspects. A transaction can be defined as a meaningful unit of work for the application which can incorporate a variable number of operations that only make sense if all or none are executed, either way, leaving the application in a consistent state.

Recent work [1] introduced Software Transactional Memory (STM), which brought the expressiveness of transactions to mainstream programming, leaving behind the cumbersome work of explicit lock-based constructions. The key idea behind STMs is the ability for programmers to specify which operations must be executed atomically, leaving to the STM the responsibility of providing the intended semantics, while maintaining as much parallelism and concurrency as possible.

To provide the aforementioned semantics, current implementations of STMs introduce an additional overhead to the system in the form of extra structures needed to enforce the atomicity of transactions (log [2], versioned boxes [3]) and in possible wasted work.

The latter, which is new to this approach, arises from the possibility of a transaction abort. Since each transaction works as if no other concurrent operations are having place in the system, somewhere during execution they must try to make their results permanent. The transactional system must then decide whether to apply the new results — commit the transaction — or issue a conflict signal — abort the transaction — due to an invalid action or inconsistent state production.

At commit time, the decision to abort a transaction is usually accompanied with its rollback and subsequent restart. This second execution constitutes a very real overhead since the cause for the restart usually only affects a portion of the computation and only a small part of the operations should be undone and then redone. That is, possibly the majority of the operations will produce the exact same results the second time they are executed, as they did in the original run. This problem is even worse, and more critical, as the number of concurrent transactions grow.

To exemplify I will use three scenarios on the same Web-based service which uses an STM and encapsulates each request from their users in a transaction. When the load of the system is low, because only a small number of users are accessing it, the number of aborted transactions is expected to be low, acceptable (since the cost of re-executing a transaction is not critical due to the fact that a lot of the system resources are not being used), and unnoticed by the requesting user.
Now let’s assume a particular point in time, where a large number of requests are concurrently being processed by the system, maxing its resources. Since for each request there is an associated transaction that handles it, there are a great number of transactions concurrently executing in the system, doing their computations oblivious of parallel executions. In this scenario, the probability of having conflicting transactions increases and the inevitable restarts are not so acceptable because transactions will keep consuming the scarce system resources until they finally commit, something that might happen just after a few restarts.

At last, instead of varying the number of transactions, just increasing the time each transaction takes to complete can be problematic. It is normal that there are services that take more time to complete than others, and so do their respective transactions. Long-running transactions are those which incorporate time-consuming operations and, because of that, take longer to finish. This extended duration will only increase the probability of conflicting with other transactions, which leads to restarts. In this case, something that by nature takes a lot of time to execute, will take longer and, if nothing is done otherwise, might never commit, continuing to waste system resources and not producing the intended result.

The problem of re-executing transactions is more noticeable in the two last scenarios, but even in the first one the system could be improved by avoiding computations it has already done in the past, just by caching the results yielded by previous executions.

Knowing the result produced by the execution of a service, even before that service is executed, leads to the obvious decision of spending time re-executing it, only to obtain an already known result, or to automatically use the recorded information, saving time and resources.

2 Goals

Memoization [4] is a widely known technique to improve the performance of a system since it can prevent unnecessary repeated computations from being executed. Although popular in pure functional programs, it is usually absent in object-oriented (OO) environments where objects are destructively changed. So, due to the popularity and wide-spread of the OO paradigm, it is now important to explore the benefits yielded by this optimization technique, in the most automatic way possible.

Regardless of the programming paradigm, current implementations [5, 6] of automatic memoization continue to depend on the programmer to explicitly specify which functions should be memoized. The memoization tool simply collects such information and modifies the program so that the selected functions are cached. With such behavior, in my opinion, this approach should only be addressed as “loosely-automated memoization” because the programmer is too involved in the memoization process.

Besides representing extra work for the programmer, current solutions can even produce erroneous behavior. By blindly following what is specified by the
programmer, if, by mistake, a function that should not be memoized is marked as
to be memoized, the automatic memoization system will memoize that function
and the program will not execute correctly. But even assuming that programmers
know what they are doing, since programs are not static entities, if a change in
a function prevents it from being memoized, it is difficult to ensure that all the
other marked functions that call the one that changed, are also unmarked.

Despite all that has been said, this solution is also deemed to fail because
it can only produce sub-optimal results. It is difficult to foresee which methods
will most benefit from memoization, and even if we could, once again, programs
change and so does the set of functions that can benefit from memoization. So
it is impracticable for programmers to find that set each time the program is
altered.

It is clear that the current approach is really the opposite of what it should
be. Instead of depending on the programmer to specify exactly what to memo-
ize, a truly, automatic memoization system should, dynamically, decide which
functions can and will be memoized. The programmer should be involved in
the process through new keywords that augment the expressiveness of the pro-
gramming language, in order to guide the decisions made by the memoization
system.

Just to increase the performance of a particular method, it is not uncommon
to discover classes that manage a simple internal cache to store some value that
is expensive to calculate or is likely not to change too often. This approach is
not desirable since it scatters responsibilities throughout all the system and is
error-prone because it may lead to a value not being invalidated after the data
it depends on changes. Therefore, replacing those caches with a manageable
and reliable system, like memoization, can solve the problem, while resulting in
better and cleaner code.

From all that has been said, it is clear that performance is very important, but
STMs’ main focus is on the correctness of operations and ease of programming.
It is true that by using transactions with optimistic locking policies, it is possible
to observe a boost in the performance of a system, but this fact does not negate
the need for further optimizations and improvements, nor hide the fact that
STMs introduce a set of structures and extra computation which penalize the
execution speed, in the normal case.

After a careful observation, one can conclude that all the information pro-
duced to provide the desired transactional semantic, can also be used to imple-
ment new functionalities. By sharing computation, these new functionalities can
help decrease the overhead of producing such information, while benefiting from
STM’s transactional semantic.

Since transactions can abort, in order to minimize the cost of a complete
re-execution, current solutions rely on programmers to specify certain points
of the program where to rollback. This approach can only be effective when
programmers are aware of the operations that might conflict and where is safe
to return to. With little effort, memoization can transparently avoid complete
transaction re-execution.
Thus, my work will focus on:
1. The construction of the appropriate mechanisms to memoize methods in an
   OO language (Java) with STM;
2. The development of a decision system that, without the intervention of pro-
   grammers, automatically chooses which methods can and are profitable to
   memoize, according to their run-time behavior;
3. Improving system performance, allowing programmers to drive the auto-
   matic process, through the introduction of new elements in the programming
   language that allow a smarter, better, and semantically richer memoization.

3 Related Work
In order to accomplish my goals, it is important to know how current solutions
tackle the problem at hand. So, I will start by introducing the concept of mem-
ioization through Section 3.1, focusing on how it works and the benefits it can
yield. Since I want to build a memoization system, it is also essential to under-
stand its main disadvantages, which will be addressed in Section 3.1.1, and how
to correctly implement a function cache, Section 3.1.3.

Incremental Computation will be briefly discussed in Section 3.1.2. Although
similar to memoization, its focus is on re-using computation and not on prevent-
ing a function from re-executing.

Section 3.2 will present two common issues that can be directly extracted
from the disadvantages discussed in the Section 3.1.1, while presenting different
approaches to tackle the problem.

The remainder of my research focused on transactions and STMs, which will
be the addressed in Section 3.3. This section will be important to understand
the common characteristics of STMs and the current solutions to the problem of
complete rollbacks. Section 3.3.2 introduces the concrete STM implementation
that will constitute the core of my work.

3.1 Memoization
Memoization, also known as function caching, is not a new idea, as a matter of
fact it was first introduced in 1968 [4] 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. Its applicability ranges
from artificial intelligence [6] to mathematical systems [7], and even software
configuration management systems [8].

The key idea behind this technique is that a great majority of functions are
deterministic and their result depends exclusively on the supplied arguments.
So, if a function is augmented with a cache which maps a set of arguments to
the result they produce, it is possible to prevent repeated work.

Unmemoized functions usually take some values as input and return some
other value computed from the supplied information. On the other-hand, a mem-
"...
int Fib(int n) {
    if (n <= 2) return 1;
    return Fib(n-1) + Fib(n-2);
}

(a) Recursive implementation in Java
(b) Computation-tree of Fib(5)

Fig. 1. The Fibonacci function is a mutually recursive mathematical function that generates a tree of function calls, as depicted in (b). Each node represents a step in the algorithm and the enclosed number the respective value of n.

A match will return the associated result to the caller. If, on the other hand, it is the first time such computation is needed, the requested result is calculated and, just before its return to the caller, a new entry is created in the cache representing this new arguments-to-result association.

With such behavior, memoization is particular useful when there is some form of repetition, may that be within a function call or over time.

The Fibonacci function is a classic, and merely academical, example of repetition within a function call. The small example present in Figure 1 demonstrates the extra and unnecessary work performed when a single routine calls itself recursively more than needed. Just by caching results as the function traverses down the recursion-tree, whole branches can be cut, thus saving time and resources.

An alternative to memoizing the function is to rewrite the code using a dynamic programming style. This approach, can lead to a better performance because it lacks the cache lookup, store, and retrieve overhead. The problem arises from the difficulty of producing an appropriate algorithm, since dynamic programming builds the solution in a bottom-up approach and programmers are trained to think in a top-down manner. So in a sense, memoization can be seen as an easy and automatic dynamic programming tool.

Although this kind of repetition is common in parsers or in computer graphics, its applicability is very low outside those specific areas of computation. Recursion is usually replaced by iteration and even when present, does not form a tree of pendent computations or have repeated pendent calculations.

Repetition over time is a lot more useful and broad since systems tend to have a sub-set of their services that is used considerably more often than the rest, so memoizing this sub-set of services can yield a performance boost.

To better exemplify, imagine a management system that controls all the administrative services and business logic of a university. The system interacts with its users (students, teachers, administrative staff) through a Web-based interface. It is normal that certain pages like the individual pages of each course,
since they contain relevant information about summaries, announcements and grades, are more visited than others. So, in this scenario, functions used to generate those pages will be repeatedly invoked, at least once for every access. Therefore, it is expected that memoizing these functions will lead to a better system performance, because many cache-hits will, not only compensate the time spent in cache lookups and stores, but also lead to less time to compute the requested page.

3.1.1 Disadvantages

Since the idea of memoization grew in a functional context, it only behaves well with pure functions, that is, functions in which: (1) the result is deterministically defined by the supplied arguments, and (2) cause no observable side-effects or I/O.

If a function’s result can only depend on the value of its arguments, that means it cannot access external values. In Java, this constraint prohibits a method to write to or read from the heap or static data, create objects, invoke native methods, invoke non-pure methods, throw explicit exceptions, for example. So, it is clear, that this constraint can limit too much the applicability of memoization solely because, in the normal case, it is difficult to find the set of all the values a function depends on, when this set contains not only the arguments but also external entities.

On the other hand, disallowing side-effects and I/O is truly a limitation, because most programs nowadays, specially in an object-oriented context, use an imperative programming style, or are designed to provide an interactive system. Therefore, an automatic memoization tool, will have to be extra careful not to memoize methods that produce side-effects or, even worse, solely exist to produce a change in the system (e.g., reset a counter), because a decision to skip those methods can lead to an inconsistent state.

Despite this fact, some [9] advocate that a change in program behavior can be acceptable, unimportant or beneficial, depending only on the programmer’s and user’s intent. This can happen in very particular cases:

– Elimination of output: A message is printed only once, rather than several times;
– Elimination of input: After receiving input, caching such information can save the burden of re-entering it;
– Elimination of break: After an error, and respective recovery, by caching the result saves the task of fixing it again.

Since this intent cannot be inferred just by code analysis, in the article mentioned above, the memoization system makes use of Interlisp [10] Masterscope to identify all the potential side-effects of executing a function, while leaving to the user, in an interactive way, the task of deciding what to do next.

Accepting side-effects will also have impact on the cache itself since in the presence of a cache-hit a copy of the cached result should be returned, because
it may be altered in subsequent computations. Also, when there is a miss and the function is normally executed, only copies of its result should be cached, for the same reason.

Also related to caches, there is the problem of too strict matches, that is, sometimes between two executions of the same function, the values of the arguments differ, but not enough to represent a change in the result. Such indifference is particularly common when dealing with floating point numbers where a change in the least significant digits of the number, may not have consequences on the result. To solve this problem, one can standardize the arguments or provide some clues to the matching algorithm.

A bigger problem takes place when there are not sufficient hits to compensate the overhead of maintaining and searching the cache, and so the function should not have been memoized. Two possible solutions are to delegate the decision on what to memoize to the programmer or dynamically, at run-time, decide the best to do.

I think the first solution is in fact a no-solution, because it requires an explicit intervention of the programmer, which I am trying to avoid. And even if it was acceptable, it is hard at code-time to foresee the number of times each candidate function will be, not only used but, used with repeated values. So the best solution will have to be one that, at run-time, decides what to memoize (assuming that at system startup none of the functions are to be memoized) or what not to memoize (assuming that at system start all of the functions are to be memoized). Therefore, the dynamic solution is completely transparent, flexible, less time-consuming to the programmer, and more powerful.

Memoization’s memory consumption was a great disadvantage in the past. In my opinion this is no longer a significant drawback since memory is very cheap these days and the run-time performance of a system is crucial for its perceptive value, so this will not be a major concern in my work.

3.1.2 Incremental Computation

Another way to take advantage of a function cache is through incremental computation [11], which concerns on how to reuse or adapt previous calculations. The idea is to calculate the result of a function from a previous calculation if the input varies only slightly.

Returning to the example of the university management system, suppose that students can consult their average grade. Since this value usually only changes at the end of the semester, the function responsible for the calculation (Figure 2) has been memoized. The idea behind incremental computation is to prevent the re-calculation of that value each time a new course grade is added.

One obvious way to implement this behavior is to, by hand, craft some sort of “update method” that given an already calculated result and a value representing the difference between the previous input and the new one, calculate the new value. However doable, specifying how to update the output in relation to a concrete variance in the input is not always easy, and represents extra work that must be debugged.
public Integer calcFinal(List<Integer> grades) {
    return sum(grades)/grades.size();
}

private Integer sum(List<Integer> grades) {
    if (grades.size() == 1) {
        return grades.get(0);
    } else {
        return sum(leftPart(grades)) + sum(rightPart(grades));
    }
}

Fig. 2. Java method that calculates the final grade from a list of course grades. The methods leftPart and rightPart, divide the list.

So the key idea behind this approach is decomposition. In order to exploit incremental computation, it is fundamental that two similar problems are broken down in a way that they share common sub-problems, because that way solving one problem will involve parts previously solved for the other.

Two decomposition schemes are proposed. I will focus on the chunky decomposition scheme for lists, where each element of the sequence is classified with a number directly extracted from its hash. Using this classification, the list is divided at the point positioned to the left of the rightmost element of the sequence with the highest classification.

To better understand, let’s go back to the example above. The original implementation of leftPartition and rightPartition simply decomposed the list in halves, returning the first half and the second half, respectively. When the list only contains 3 elements (Figure 3 (a)) the partition made by both approaches is the same, but when a new element is added it is possible to observe that the original decomposition scheme does not share common sub-problems with the its previous computation (Figure 3 (a) and (b)). On the other hand, the proposed chunky decomposition scheme will produce a hit in the cache.

This decomposition scheme shows that, in order to make problems share common sub-problems, it is important to construct algorithms based on the value of the arguments and not on their length.

3.1.3 Cache

The cache of a particular function can be seen as a map that associates a set of conditions that must be met, in the simpler case the value of each argument supplied to the function, to a value that represents the value of the function under those conditions. Since a function can depend on a variable number of values, not only large in number but also in size, instead of using those values as the cache key, a fixed-size hash, made through some canonicalization of such values, is used instead.
With this decision, some non-trivial equality tests can be avoided, but arises the possibility of two different sets of dependencies producing the same hash. To minimize the probability of such collision, long hashes must be used and generated carefully.

A classic problem when dealing with caches is deciding which elements to purge, when the cache is full. The most efficient algorithm would be to always discard the information that will never be needed again or will only be needed in a distant future. Since, generally, this is not possible to know in advance, there are some approaches that try to offer some more realistic solutions based on how frequent some information is used or when each entry was used for the last time. Least Frequently Used (LFU) discards first those entries that are used the least, by counting how often an item is needed, and Least Recently Used (LRU) discards the least recently used entries first, by keeping track of the last time an item was used.

Although similar to other forms of caching such as buffering or page replacement, a function cache differentiates itself from such cache eviction problems because the frequency of use of an entry in the cache varies based on what else is cached, and the cost to re-calculate an entry depends on the complexity of the function itself and on what else is cached.

William Pugh proposed [12] an improved replacement strategy that accounts for those differentiating properties. His practical algorithm stores the number of hits and the amount of time required to recompute each entry of the cache. The entries are stored within a fixed-size bucket sorted by Estimated Potential, which reflects the extra work that must be done if that entry is evicted.

Thus, the Estimated Potential of each entry incorporates the Estimated Recomputation Frequency, which estimates how many times a function is used, and the Estimated Recomputation Cost, that reflects the time it takes to compute the respective entry. In order to correctly capture this last value, an extra care must be taken.

---

**Fig. 3.** Visual representation of the calls made by \texttt{sum}. Each level represents a step in the algorithm. Grades are represented as numbers within a cell of the list. The division in chunks assumed that the classification (hash) of a grade is the same as its value.
int f1(int x, int y, int z) {
    if (x > 0) return y;
    else return z;
}

Fig. 4. Simple function.

To clarify we can look once more to Figure 2. In order to calculate its result, calcFinal makes use of another function sum. With this call relation, the execution-time of calcFinal will always be greater than that of sum. So, if these two values are stored in the same bucket and one must be evicted, a naive cache will choose to discard the value computed by sum. A more intelligent system will conclude that the expensive computation is done by sum, calcFinal simply divides two values. Therefore, in the situation described before, it should evict the value associated with calcFinal.

When a new entry must replace an old one in the cache, the purge algorithm finds the appropriate bucket and randomly selects an entry to discard, using a probability distribution that is highly skewed towards replacing a lower ranked entry, but can also evict a highly ranked one. This way if by mistake an entry was highly rated it does not stay in the cache forever.

A problem with this algorithm, is that upon a cache hit or a new store, the bucket must be re-sorted. This might be the reason why, in his implementation, Pugh used an eight entry bucket.

3.1.4 Precise Dependencies

As mentioned in Section 3.1.1, false negatives can be problematic, that is, due to a too strict match between the supplied arguments and those stored in the cache a recomputation is not prevented. This only happens because the dependency that exists between the result of a function and its respective arguments is too limiting, as if cache entries are constructed in an overly restrictive way, limiting their usability. This is the observation done here [8, 13] in the context of a software configuration management system for building large-scale software, similar to Make.

To illustrate, consider the function in Figure 4 and two consecutive calls:

1. f1(1, 2, 3);
2. f1(1, 2, 4);

As discussed, each cache entry, for a particular function, maps a variable number of arguments to the respective output they produce. In this case a naive approach will combine the three arguments of function f1 to construct the cache key. A careful observation can conclude that, after the first invocation to f1, the second invocation will result in a cache miss, due to different hash values, but there should be a hit on the cache entry created by the first call, since the value of z is not relevant to the outcome of the function.
This behavior can only be accomplished by noticing that the result of a function depends on two aspects: those which influence the result at call-time and those which influence it dynamically at execution-time. These two aspects, respectively, will form the primary and secondary parts of the cache key. In the above example, the primary key is simply formed by the function body (so that if the function is modified there will be no hit in the cache with that function’s previous body), while the secondary key includes the dynamic dependencies on the arguments x and y (x=1, y=2). Employing this scheme, the second call will now produce a hit and the return of value 2, without executing f1.

Returning to the example in Figure 4, the authors also highlight the fact that even with this scheme, the secondary key is still too strong, in the sense that the result only depends on x being positive, not necessarily equal to 1. Thus, a smarter caching policy could use predicates on the values of secondary keys instead of concrete values to improve performance. Unfortunately no further explanation is given on how to automatically extract these predicates or how to represent them.

3.2 Purity Analysis

Current implementations [9, 6, 7] of automatic memoization only address the problem of “how” to memoize functions, leaving the decision on “what” to cache to the programmer. Since in imperative programs side-effects are common, in order to develop a memoization system able to decide, without external help, what to cache, it is extremely important to find out which functions do not cause side-effects, that is, which functions are safe to memoize.

One way to do it is through source code analysis, which is the solution proposed here [14], where the introduced QPAL system transforms a Perl program into a semantically richer textual representation.

This system for each function f builds the list of all functions that can be called from f. If any of the callable functions by f produces side-effects, depends on the external state of the program, or was previously marked as not memoizable, f is also marked as unsafe. In the end, the system outputs all the memoizable and non-memoizable functions, accompanied with a list of variables that are the reason why a function is not cacheable. Such description will then serve as input to a memoization tool, which will perform the appropriate transformation on the cacheable functions.

A similar work [15] focused on finding the side-effect free methods present in a given set of Java classes. This work assumes that a method is free of side-effects if the observable state before and after its execution is exactly the same. All the static fields and all the heap objects transitively reachable from static fields or from the run-time call stack, constitute the observable state. This way a method can produce side-effects if they are not observable by the caller.

Unlike the previous work, that only looked for side-effects within methods, this solution must first decide what belongs to the observable state, and only then search for methods that might cause side-effects to that state.
In order to find the observable state, the first step is to analyse all the callable methods, those that are public in public classes, and construct the set of classes they can instantiate. From there, it must analyse methods with immediate side-effects, that is, methods that: (1) assign values to static variables or (2) create/modify instances of classes that belong to the observable state. The final step is to propagate that analysis to all the methods that, transitively, call methods which produce side-effects.

Both works, highlight the difficulty that arises when there are calls to external functions, which implementation is not available. The conservative approach is to mark all those functions as not cacheable, with the respective penalty of classifying all the functions that use them, also not cacheable. To solve this problem, the QPAL system introduces keywords which help guide the analyses with information about the nature of such functions.

Until now, we have looked at static methods to analyse the purity of programs. Both solutions are conservative because they classify a function as not cacheable when there is, at least, one path of execution where it may cause side-effects. So a function that rarely causes side-effects, will be classified as not cacheable even if, when executed, the path which leads to side-effects is not taken. To solve this problem, a run-time analysis must be done.

One such example [16], applied once more to Java, uses bytecode analysis and support several definitions of purity. So, looking only to the instructions that are executed, a method can be classified as: (1) Strongly Pure, if it behaves like a pure function, (2) Moderately Pure, if it is strongly pure and creates/alters objects that do not escape the method execution, (3) Weakly Pure, if it is moderately pure and reads information from the heap that is accessible from the supplied arguments, and (4) Once-Impure, if it is an impure method that behaves like a weakly pure one after the first invocation.

In their article, Xu et al. state that even once-impure methods are good candidates for memoization because they are executed at least once before being memoized, therefore, will not be deemed as impure just because mandatory class loading and initialization, during the first invocation, causes writes to memory.

So in their implementation, memoization is achieved by associating hash tables with once-impure methods. Once again, methods can only depend on the supplied arguments or information accessible through them. Primitive values are stored directly in the cache, whereas reference ones must be “flattened”, through recursively gathering the object type and primitive field values for all reachable types.

3.3 STM

The work on Software Transactional Memory [1] focused on providing programmers with a more expressive way to tackle the problem of multithreaded programs, now that multi-core processors are broadly available. The key observation made is that lock-based approaches are too distant from the way programmers usually think and, when dealing with complex systems, can lead to incorrect behavior, starvation or even dead-lock.
Therefore, instead of restricting access to data through locks, programmers should specify which operations must be atomically executed inside a transaction. A transaction is nothing more than a collection of operations that, from the point of view of the application, only makes sense in a “all or nothing” approach, that is, or all the operations are executed, or none is. It is STM’s responsibility to ensure this transactional semantic, while maintaining as much parallelism as possible.

In order to achieve this goal, an STM must ensure that each transaction executes in a context containing the state of the system at transaction start. Since this context can only reflect the changes made by the current atomic execution, STMs usually use, for each transaction, two private sets that represent the locations read (read-set) or written (write-set) within the transaction. These sets will also be important when the transaction tries to commit.

The commit of a transaction is its final operation and can yield one of two possible results:

- success - all the values modified by the transaction can and will be applied to the system state.
- fail - none of the values written should be applied to the overall system, the transaction aborts and will have to be re-executed. Usually, a transaction conflicts with another when it has read some data that the other has written and successfully committed.

Transactions can be decomposed into subtransactions, which, in turn, can be decomposed into more subtransactions, forming an arbitrarily deep hierarchy of nested transactions.

Nesting can be particularly useful [17] and in the context of this work I would highlight these two advantages over flat transactions:

- Intra-transaction parallelism: As transactions grow in number of operations so thus the opportunity for internal parallelism. Nested transactions offer an appropriate control structure to support supervised and safe intra-transaction concurrency.
- Intra-transaction recovery control: Without compromising the wellbeing of surrounding transactions, uncommitted subtransactions can be aborted and restarted.

3.3.1 Recovery

Flat transactions, once marked to be aborted, will restart and be completely re-executed. So large monolithic transactions increase the amount of work that must be redone in the presence of a restart. Nesting allows a fragment of work to be executed tentatively and, if failed, aborted without affecting work already accomplished. The system may then control what to do next: re-execute the failed fragment or pursue in a different direction. Therefore, being able to partially abort a transaction is not only useful for semantic reasons, but also essential to control the performance of the system.
Aborts are often issued when there is a conflict, but even more often the conflict can be resolved solely by undoing a portion of the transaction. Because of that, partial aborts can be particularly useful when non-conflicting operations are not undone, and then redone. To allow such behavior, there must be a way for a transaction to return to a specific location within its execution. Thus the notion of a checkpoint.

Checkpoints can be explicitly defined by the programmer or emulated through nested transactions. So, the amount of work that is lost in a transaction is defined by the number of checkpoints available in the code. The programmer must be aware of the appropriate balance because numerous checkpoints can be hard to maintain, but too few can lead to a lot of work that must be redone.

In order to alleviate programmers of this chore, Koskinen and Herlihy [18] proposed that checkpoints should be semantically rich locations within the transaction control flow, such that, from one checkpoint to the next there is a well-defined state transition, given by the semantics of the interleaved operations. They capture these locations by looking at operations that modify the objects managed by the STM, and automatically placing checkpoints before their execution.

3.3.2 The JVSTM

The Java Versioned Software Transactional Memory [19, 20] (JVSTM) is a particular object-oriented STM that was conceived with domain-intensive applications in mind and, unlike previous STMs, uses versioned boxes [3] to keep multiple versions of each transactional location.

At creation time each transaction is assigned an empty read-set, an empty write-set, and a version number obtained from a global counter, which is incremented every time a transaction successfully commits.

During their execution, transactions can access versioned boxes, which hold a tagged sequence of values. This history of values, corresponds to changes done to the box by a committed top-level transaction. A value tagged with the number $N$ was written by the transaction with the version number $N$.

Each time a transaction $Tx$ tries to write a value $val$ in a box $B$, the history of $B$ is not immediately changed, instead, a new mapping from $B$ to $val$ is added to the write-set of $Tx$. If the box has already been written in $Tx$, the previous mapping of $B$ in $Tx$’s write-set is replaced with $val$.

The read of box $B$, executed in the context of a transaction $Tx$, returns the last value wrote by $Tx$ to $B$, if such mapping exists in $Tx$’s write-set, or returns the entry in box $B$ with the highest tagged number that is no greater than $Tx$’s version number.

Thus, we can say that the read-set of a transaction represents the relevant state of the system as it was at the time the transaction started, and that the write-set constitutes a new view of the system only observable by the respective transaction.

Conflicts are detected only at commit-time by examining what is present in the read-set and write-set. The key idea behind this decision is that a transaction
can only be valid if it is trying to apply changes to a state that is indistinguishable from that observable when the transaction started.

With this condition, transactions that changed no boxes (read-only) or those that exclusively changed boxes (write-only) will never conflict. The first are void of impact, therefore there is nothing to be done to the system, and the latter assumed no state during their execution, so despite all the changes done meanwhile their behavior was not influenced, thus their commit will always be successful.

When dealing with read-write transactions the situation is not as simple, because at least one box was modified assuming the state of the system observable at transaction-start. So, in order to accept the commit, the transactional system must ensure that none of the boxes in the read-set of the committing transaction, changed after the transaction started. Only then, a top-level transaction can be considered valid, and its write-set applied to the global state.

The commit of a nested transaction will never fail because it just merges its read-set and write-set with its parent’s read-set and write-set.

The system speculatively assumes that a top-level transaction is read-only when it starts. If such assumption proves to be wrong, by detecting a write operation, the transaction is aborted and restarted as a generic read-write transaction. The complete restart its due to the fact that read-only transactions do not populate their read-set, which is important to the commit of read-write transactions.

There is one more source of aborts in the system and it is related to domain logic. Since the JVSTM was built to tackle the problem of constructing domain-intensive applications it provides a mechanism to enforce the consistency of the system and its data, through consistency predicates.

Consistency predicates are predicates that check whether a domain object satisfies some particular constraint. They are verified at the end of an atomic action in order to ensure that the system always transits from a valid state, to another valid state. A violation of such predicates, will prevent a transaction from committing and a restart is issued.

In summary, a JVSTM transaction, due to its optimistic and speculative construction, can be aborted if it conflicts, violates a consistency predicate, or was speculatively assumed as read-only when in fact needs to write some information.

4 Proposed Solution

Before I start describing my solution, it is important to mention that this work will be implemented as an extension to the Fénix Framework, a Java implementation of the JVSTM model previously described. Therefore, I will extensively use the transactional semantic offered by this STM, as well as the information it produces to keep track of the access patterns of each transactional execution.

The Fénix Framework is currently being used by the Fénix System, a university management system that incorporates the great majority of all on-line campus activities and related management services.
4.1 Building the Cache

Memoization’s key aspect is the function cache, because all the benefits it can yield are tightly coupled with its correct construction. An overly restricting cache can originate the re-execution of methods previously computed, while a wrongly constructed one can lead to incorrect behavior. So, I will start by describing how my solution constructs cache entries, focusing on how they store information and why that information is relevant to the applicability of memoization.

Capturing the set of values a method’s result depends on is fulcral, but current solutions do not address this problem because they impose that memoizable methods cannot depend on values that are not received as arguments or, at best, not reachable through them. Partially this is due to the difficulty of registering such information in a way that allows, in the future, the correct value to be retrieved and compared, but mainly because it is hard to find all the values a method accesses during its execution. As we will see, this is not a problem in the context of the JVSTM.

To understand why, it is important to carefully observe what happens when a method is executed within a transaction. As previously stated, the transactional system populates a read-set for each transaction in order to validate its commit. Each time a value is read inside a transaction, information regarding which versioned box was accessed is added to that transaction’s read-set. This way, the system fully captures the state of the application which is relevant to the outcome of a transaction.

Therefore, if every access a method does is registered by the transactional system, it is possible to memoize methods that access any kind of information, may that be internal or external to the method, knowing that all the relevant information will be registered in the read-set for future comparison.

The current implementation of the JVSTM uses a single read-set for each transaction, where it is not possible to distinguish which boxes where accessed by each method. So, in order to obtain the described behavior, it is important to augment the read-set to hold more detailed information regarding which methods accessed each of the registered boxes and what value they retrieved.

So, in order to implement memoization in Java, each method should be augmented with a cache mapping a set of versioned boxes read by the method, to the result obtained when the method was executed under those conditions. Just before a method is executed, it must check its cache to see if it holds information regarding the outcome of the requested computation. If so, the result is automatically returned and the method execution skipped.

To correctly commit the transaction, it is still important to add the cached read-set of a skipped method to the transaction’s global read-set because, even though the method did not execute, if it did, it would access those boxes.

Since several concurrent executions can try to obtain or add new information to the function cache, it is important to ensure that they do not create inconsistent entries, or cause lost-updates, for example. The simplest way to obtain a correct behavior is to make use of the transactional system to encapsulate cache
interactions within transactions. Therefore, I will use JVSTM’s transactions to guarantee a correct behavior when accessing this shared resource.

Using the read-set to track all the values a method depends on, will have another desired consequence on memoization’s applicability. The discussion in Section 3.1.4 made clear that recording precise dependencies can improve the number of cache hits, because fine-grained dependencies are more easily met then coarse-grained ones. Since versioned boxes are containers for values, each field of a class, for example, will be represented by a versioned box. So, if the result of a method \( m \) exclusively depends on the field \( f_1 \) of a class \( C_1 \), only \( f_1 \) will be added to \( m \)’s read-set. This way, if \( m \) is re-executed with the same value of \( f_1 \), even if all the other fields of \( C_1 \) have changed, there will be a cache hit and \( m \) will not re-executed. Therefore, we can conclude that the read-set holds fine-grained information about the values accessed during the execution of a particular method.

4.2 Automatic Memoization

Until now we have assumed that all the executed methods are to be memoized. Although there are two main reasons why that might not be the case:

– Some methods may produce side-effects;
– Some methods can be cheap to execute, but expensive to memoize.

Normally these decisions are left to programmers, but since one of the goals of my work is to produce an automatic memoization framework, the previously described approach is not acceptable. Therefore, the function cache must be complemented with a decision system capable of, at run-time, decide what is the best course of action and act accordingly.

In order to decide which methods are safe to memoize, the decision system can simply look to the write-set. Similar to the read-set, which stores all the values a method reads, the write-set holds all the boxes that the method wrote. So, when a called method returns a non-empty write-set indicates it has caused side-effects, therefore, for that particular read-set the method should not be memoized. As it is possible to observe, the decision not to cache the result of a method does not depend on if the method can produce side-effects, but rather depends on if that particular interaction (read-set) caused side-effects. Because of that, the execution of the same method, but with a different read-set that leads to a side-effect free execution, can culminate in the decision to memoize that particular interaction.

The decision whether to memoize, or not, an interaction that caused no side-effects, is not as simple because it depends on the time the method took to execute and, mainly, the time it will take to search the cache. Assuming that a method is not deemed as not to memoize due to time consumption, there is still the problem of methods that are rarely used with the same set of dependencies (read-set) and, because of that, there will not be sufficient cache hits to make the cache profitable. So, the decision system must collect information about the
time each method takes to complete and the respective cache hit-rate, in order
to heuristically decide what to do.

Therefore, the decision whether to create a new cache entry depends on the
time that particular interaction took to complete and on the number of successful
cache hits of the respective method, obtained from previous interactions. Since
methods, in Java, cannot be modified at run-time, all transactional methods
must be augmented with a function cache that might be deactivated, when the
method is not worthwhile to memoize.

So, in order to decide what to cache, during the execution of a transaction, the
memoization framework will, for each method, record its read-set, its write-set,
the value it returned, and time it took to complete. Only after the transaction
successfully commits, the memoization system will decide what to cache.

Methods with a non-empty write-set will certainly not have their respective
interactions added to the cache, while the rest will be cached in case the informa-
tion produced is expected to be relevant. Thus, an interaction with a non-empty
write-set will not be useful outside of the transaction, but it can be important
inside of the transaction if the transaction aborts.

The most usual reason why a transaction aborts is because it has read some
information that another transaction has modified. In that case, the transaction
is aborted at commit time, restarts, and gets a new version number. That version
number will be used to retrieve information from boxes and to tag values written
to them. So, before re-executing a method, or even looking at its cache, we can
first check if any of the boxes present in the method’s read-set were the reason
why the transaction had to be aborted. If not, we can simply add the previously
saved write-set to the global write-set, correctly re-tagged with the new version
number assigned to the transaction, add the saved read-set to the transaction’s
global read-set, and skip the re-execution of that method.

With this approach I can accomplish partial transaction re-execution in a
transparent way, without relying on explicit or implicit checkpoints. The memo-
ization system can decide precisely which methods must be re-executed, knowing
that all the results obtained from the aborted execution will be available when
the transaction restarts. This holds for transactions that aborted due to conflicts
or violations of consistency predicates.

It is also important to highlight that, since I use the read-set to capture
the set of values used by a method, I can memoize methods without side-effects
even though the result they produce might be altered in subsequent executions.
Also, as assumed by the JVSTM, since transactions that update are expected
to be low in number and to only write a sub-set of the values they read, even
read-write transactions can provide valuable cache information and benefit from
the existence of a function cache.

Long running transactions are expected to benefit the most from memoization
because they take a lot of time to complete, therefore, they are good candidates
to be cached. In the event one of these transactions aborts, it is also expected
that the reason of the conflict might be limited to a sub-set of the values read
and, therefore, the time it takes to re-execute can be cut down considerably.
Even though I have been focusing my discussion on the benefits of my solution when something goes wrong and an abort is issued, when no problems occur memoization can also help to improve the performance of the system, if it prevents time consuming methods from executing.

4.3 Read-Only Transactions

As described before, see Section 3.3.2, the JVSTM speculatively assumes that each newly created transaction is read-only. If this assumption proves to be wrong, once a write is attempted, the JVSTM aborts the transaction and restarts it as a generic read-write transaction. Unlike previously discussed restarts, this one does not happen because the system must be conservative and assume that a change, made by concurrent executions, will influence the outcome of the committing transaction. This abort solely happens because the system assumed something that latter proved to be wrong, therefore it is expected the second time the aborted transaction executes all the read-only operations executed until the one that triggered the restart, will yield the same result.

So it would be expected that memoization could prevent those methods from re-executing, thus saving time while compensating for the overhead produced by a bad decision on the nature of the transaction. But that is not the case. Memoization will only have the aforementioned desired behavior for those operations that were previously executed in the context of another read-write transaction.

I will explain, since read-only transactions do not save their read-set, memoization can only prevent methods from re-executing if they were previously cached in the context of another transaction. We must not forget that methods executed in a wrongly assumed read-only transaction did not record which boxes they read. So, even if the system cached the results yielded by each method, it could not establish the set of values they depend on in order to validate the commit.

The absence of the read-set will also affect the applicability of memoization when the transaction is indeed read-only. This lack of information will prevent the creation of new cache entries and if the exact same transaction is done in the future it will be re-executed.

So, it is possible to say that in the presence of read-only transactions memoization will have to behave slightly different, when compared to what happens with generic read-write transactions. After a cache hit it should not add any information to the read-set of the current transaction, because it would be unnecessary work, and it should not try to generate new information, for the reasons mentioned above.

Since the problem is the absence of the read-set, read-only transactions could be modified to populate their read-set, but by doing that we would be transforming read-only transactions in read-write transactions that, by chance, do not write boxes.

So, there are two possible solutions for this problem: (1) assume limited applicability of memoization in these cases, or (2) eliminate the speculative read-only transactions. Unfortunately this is a decision I cannot make just now, further
testing is needed in order to assess which of the proposed solutions is best. Therefore, I can only speculate why eliminating read-only transactions can yield a performance upgrade.

To eliminate speculative read-only transactions, we must substitute them for generic read-write transactions. This can be a good choice because we would be trading simplicity of execution for increased information to the memoization system. Increased information that could accelerate other transactions and even prevent former read-only transactions from executing at all.

The JVSTM was built assuming that in enterprise applications the great majority of operations are read-only, which are safe to memoize. In the Fénix System, these read-only transactions are mainly due to the fact that users interact with the system through a web-based interface that, dynamically, constructs web pages. Since memoization can be applied wherever there are methods responsible for the construction of time-consuming information, it is even possible to memoize a full request made to the system, because the data it depends on rarely changes, it is read-only, and will be extensively used. This is mainly the reason why I foresee that eliminating read-only transactions might increase the performance of the system.

4.4 Involving Programmers

All that has been said so far, assumed a fully automatic memoization system completely transparent to the programmer. But it is also important to involve programmers in this optimization process because they can provide valuable information not directly extractable from the program behavior or specification. Therefore, it is important to give programmers the chance to aid the memoization system, specially when methods execute I/O operations.

Until now, I have not addressed the problem of methods that perform I/O, but it is clear that, in the normal case, those methods should not be memoized. But as mentioned before, I allow methods, which produce side-effects, not to be re-executed inside a transaction because it is possible to achieve the same behavior whether they are re-executed or not. With I/O this is not as simple, since there are no version boxes associated to the outcome of the interaction. So, without further information, the method will be re-executed every time it is used inside a transaction\(^1\).

But now imagine a method that writes to a file, if the transaction restarts, that method will make another write. This second I/O interaction can be desired, unimportant, or even unacceptable, depending solely on the programmers intent. Therefore, programmers should be able to annotate methods that, if re-executed inside a transaction, should be skipped, if that is the desired behavior.

There are other examples where programmers can help the memoization system. For example, they can provide better ways to compare values in order to define when two floating point numbers must be considered equal. So, the

---

\(^1\) Assuming that there is a way to determine if a method executes I/O operations and, for that reason, should not be memoized.
programming language should be augmented with relevant constructs that help the memoization process and allow semantically richer decisions.

5 Validation

In order to validate my work, I will primarily use the STMBench7 [21] benchmark because it aims to provide a workload that is more realistic, when compared to other benchmarks. Since the target of my work is the Fénix System, it seems the natural choice to do. Besides, this benchmark was also used to validate the JVSTM, which is the basis of my work, so it will allow a comparative analysis on the performance before and after the implementation of my memoization system.

Prior to this work, we at the Software Engineering Group (ESW) of the Inesc-ID, adapted a set of representative benchmarks (DayTrader, OO7, RUBis, TPC-W) to the Fénix Framework. So those benchmarks will also be used to assess the difference in performance obtained by memoization.

In the tests I will conduct, it will be important to vary the number and the type of concurrent transactions, in order to conclude under which load memoization yields the best and worst results. And compare the behavior when the cache is already filled with valuable information (warm) and when it is completely empty (cold).

Since programmers will have the opportunity to aid the memoization system, it will also be important to assess the impact of these “tips” in the performance of the constructed solution.

I expect to demonstrate that large imperative systems can have a significant portion of functional computations, where memoization can be freely applied.

6 Acknowledgments

This work is being performed in the scope of the Pastramy project (PTD-C/EIA/72405/2006), which is funded by the Portuguese FCT (Fundação para a Ciência e Tecnologia).

7 Conclusion

In this paper I described the common uses of memoization, and how it can be applied to improve the performance of a system, while discussing how current implementations deal with side-effects and capture precise dependencies.

I have studied a particular STM, presented its main features, how it deals with conflicts and how it can be improved with the help of an automatic memoization.

My work differentiates itself from other implementations of memoization, not only because it will be implemented in an object-oriented environment, but essentially due to the fact it captures the full set of dependencies of a particular
method invocation, registers precise dependencies, and decides if a method can be memoized dynamically.

Finally, this paper described the main ideas of the work that will be implemented in the months to come, which ultimate goal is to be used in the Fénix System.

References