Improving Software Performance with Automatic Memoization

Marty Hall and J. Paul McNamee

Automatic memoization transforms existing procedures in a computer program into versions that “remember” previous computations and thus avoid recalculating the same results or those similar to ones computed previously. The technique works without requiring the programmer to make changes to the existing source code, thus providing a simple and transparent method of dramatically accelerating the execution of certain types of functions. This article discusses the development of a practical automatic memoization library, how it can be applied, technical obstacles encountered during development, and experiences using it to significantly improve the performance of two large systems in the Submarine Technology Department of the Applied Physics Laboratory.
(Keywords: Computer science, Dynamic programming, Memoization, Optimization.)

INTRODUCTION

Software performance is a critical issue in most computer systems, particularly in the large complex systems necessitated by Department of Defense requirements. This issue has remained a crucial one, even as computer hardware speeds have increased, as a result of increased demand for fidelity in simulations, stricter accuracy requirements, application to more complex problems, and an ever-increasing expectation for new features. However, many optimizations that increase execution speed do so by making the software larger, more complicated, and harder to maintain. Automatic memoization is a technique that can greatly improve the efficiency of several types of software routines while keeping the code simple and easy to maintain.

What Is Automatic Memoization?

Many functions perform redundant calculations. Within a single function invocation, several subfunctions may be invoked with exactly the same arguments, or, over time in a system run, a function may be invoked by different users or routines with the same or similar inputs. This observation leads to the conclusion that in some cases it is beneficial to store the previously computed values and only perform a calculation in situations that have not been seen previously. This technique is called “memoization.” The manual version of the technique generally involves building lookup tables and is a standard technique in dynamic programming. This process, however, is often tedious and time-consuming, and it requires significant modification and debugging of existing code. An “automatic” memoization facility is one in which existing functions can be programmatically changed to cache previously seen results in a table. These results will then be returned when the functions are invoked with arguments seen previously. This procedure can be done without manually changing the source code, thus providing a simple, modular, and transparent way to dramatically accelerate certain types of functions.

Although the idea of automatic memoization is not new, it has not been taken seriously as a practical software engineering tool intended for widespread use by programmers in industry. However, experience with a prototype facility has shown that automatic memoization is even more widely applicable than previously thought. It must be stressed that with software development in industry, ease of use is a paramount consideration. In virtually all software development programs, there is a trade-off between productivity (development speed and software maintenance costs) and program efficiency (execution speed). Providing a method to get a large percentage of the efficiency gains with a small fraction of the effort can be a tremendous advantage. This is the role that automatic memoization fills.

An Example

Consider the following naive recursive version of `Fib`, a function to calculate the *N*th Fibonacci number given *N*, using the definition that the first two Fibonacci numbers are 1 and any other is the sum of the previous two:

```
int Fib(int N){
  if(N<3)
     return(1);
 else
     return(Fib(N–1)+Fib(N–2));
}
```

This version is inefficient, since the second recursive call to `Fib` repeats a calculation performed by the first recursive call. In fact, the time required to compute `Fib(N)` by this method grows exponentially with *N*. However, by adding a single keyword, the programmer can tell the automatic memoization system to convert the code into a memoizing version that operates in time proportional to *N* for the first calculation and in constant time if the same or a lower value is computed subsequently. This constant-time performance is achieved by implementing the lookup table using “hashing,” a technique whereby the time required to retrieve an entry from a table is independent of the size of the table. This capability is illustrated in Fig. 1, which shows the number of operations required for the memoized and unmemoized versions. Now, memoization is not required for such a trivial application as calculating Fibonacci numbers, as an iterative version or a smarter recursive version can get the same results as the first invocation of the memoized version. However, the point is that repetition can cause a dramatic increase in execution time, and many problems do not have such readily accessible alternatives.

APPLICATIONS

Automatic memoization has been applied in three main ways: to prevent recursive repetition within a function call, to prevent repetition over time, and to make persistent lookup tables.

Repetition within a Function Call

The Fibonacci example illustrated the use of memoization to eliminate repetition within a function call. The `Fib` implementation is typical of a large class of algorithms that have elegant recursive definitions that are simply unusable for most real problems because of repetition. The conventional response to this situation is to manually rewrite the algorithm in a dynamic programming style.[1] Because such manual rewriting is typically very difficult and time-consuming, and involves the risk of introducing new bugs, it is frequently inappropriate in the dynamic, rapid-prototyping context of complex software development. An attractive alternative is to use an automatic memoization package

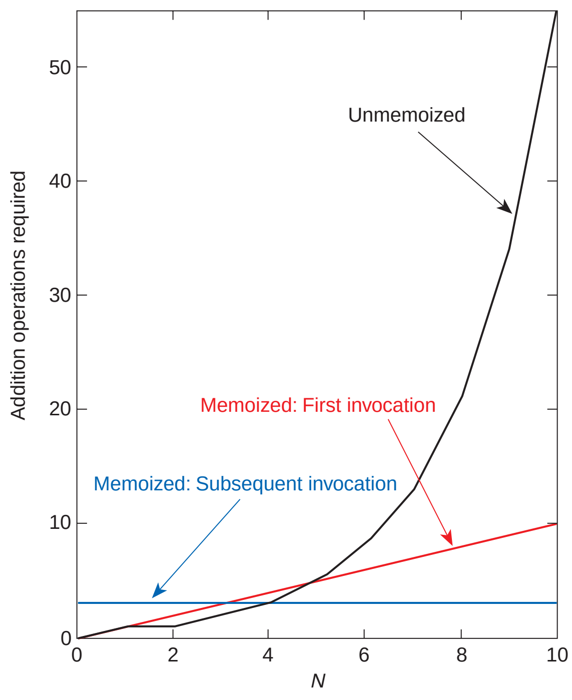

Figure 1. Number of additions required to compute `Fib(N)` using the naive recursive implementation.

to convert the elegant but inefficient algorithm into a computationally tractable one. For example, a simple recursive parser for context-free grammars may parse the same part of speech many times, resulting in execution time that is an exponential factor of the length of the input. Norvig[2] has shown that through the application of memoization, such an algorithm can obtain the same polynomial performance as the best-known context-free grammar parsing methods (chart parsing[3] or Earley's algorithm[4]). Given the intricacy of those methods, it is unlikely that a typical software developer would reinvent them, whereas a simple recursive parser is relatively obvious to developers even without experience in formal languages or parsing.

Thus, memoization can be viewed as an automatic dynamic programming tool. Rather than the difficult and time-consuming process of determining the correct order in which to construct the subpieces in a bottom-up manner, the simple, top-down algorithm can be memoized to achieve the same performance.[1] This is the typical application of memoization in the literature. This alternative is attractive, of course, only if the memoization package can address the practical problems faced in real applications.

Repetition Over Time

In a team programming environment, different sections of the system, written by different programmers, may access the same function. Alternatively, in an interactive system, the user may invoke calculations at different times that make use of some of the same pieces. In these cases, there is no central routine that could manage the calling sequence, and it is common to have the routine in question manage its own global data structure to remember previous results. Memoization provides an efficient, convenient, and reliable alternative.

For instance, Fig. 2 shows a tactical decision aid for planning submarine operations in the Defense Advanced Research Projects Agency (DARPA) Signature Management System[5] developed by APL's Submarine Technology Department. It shows the predicted overall probability of detection of a submarine for various choices of heading and speed, drawn on a polar plot with the angle θ indicating heading (0 corresponding to due north), and the radius r corresponding to speed. Each (r,θ) pair (arc) in the display is coded with a color indicating the cumulative probability of detection for the submarine if it were to operate at that course and speed. This display is used as a high-level tool in mission planning and thus presents highly summarized information. Each point represents a single number for probability of detection, which is a composite of all the potential detection methods (signatures). The user frequently is interested in the contribution of individual signature components to this composite. Since the probability of detection of each component is memoized before it is combined into the composite, any component contributing to a composite point on the display can be retrieved almost instantly. Taking advantage of this caching, we developed the display of Fig. 3 with very little additional bookkeeping.

Whenever the user moves the computer mouse over the composite detectability display (Fig. 2), the corresponding speed and course for the point under the mouse is calculated. The individual components are then calculated by simply using the original functions, with their relative values shown in the bar charts. As a result of the effects of memoization, the component values can be calculated and graphed as quickly as the user can move the mouse.

This example illustrates the ease of use that automatic memoization provides. A real-life requirement that was expected to require a lengthy development process turned out to be trivial to implement and test.

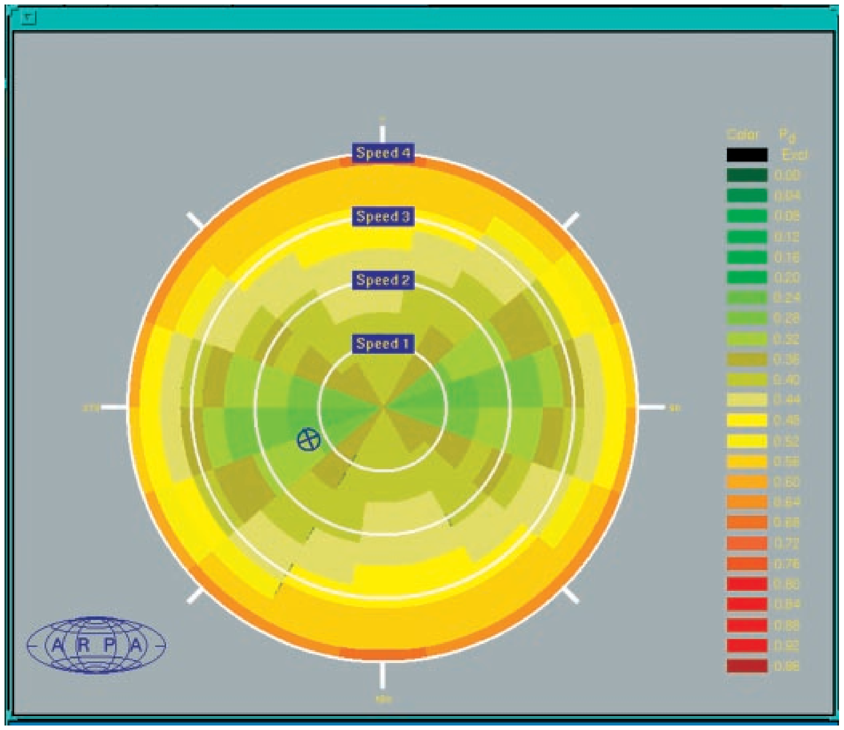

Figure 2. Signature Management System detectability planning screen. The probability of detection (P_d) for the next 24 h is shown for the composite signature at a depth of 300 ft.

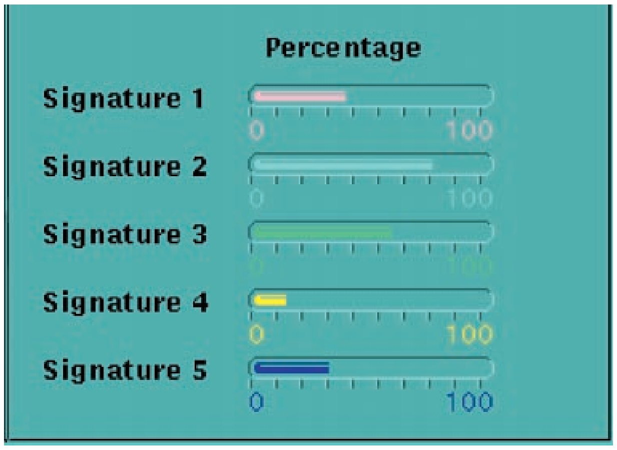

Figure 3. Individual signature components. Relative signature strengths are shown. Depth = 300 ft, heading = 100°.

Precomputation: Persistent Memo Tables

The preceding subsections show that memoization can eliminate the repeated invocation of expensive calculations. Memoization is also useful even when the first invocation of a function is too expensive to perform at run time. Memoization provides an alternative to handcrafting a lookup table of results and determining on paper the range of values that must be stored. Instead, the function is memoized interactively and run off-line on the cases of interest. The contents of the automatically generated (hash) table are then automatically saved to disk and later used to seed the hash table for the function when it is reloaded.

In this situation, the ease of use of memoization particularly pays off. If the situation were permanent, it might be feasible to build conventional lookup tables. However, for temporary conditions (e.g., running multiple simulations in the same environment), it would likely not be worth the effort to build the tables. Furthermore, unlike a handcrafted lookup table, unexpected inputs will still result in correct results since values not found in the table will still be calculated by the original function. This approach was used extensively on the DARPA Signature Management System and was found to be highly effective.

TECHNICAL OBSTACLES

When an automatic memoization library intended for general-purpose application is being implemented, several technical impediments need to be overcome. Most of these issues were identified from our experience with applying the Lisp-based library on Submarine Technology Department programs.

Overly Strict Matching on the Input Argument

In the Signature Management System, the arguments to a function differed in many cases but not by enough to result in significant changes in the output value, where "significant" was defined in the particular context in which the function was being used. Such differences were particularly common when floating point numbers were used as input. For instance, calculating the position of a submarine by two different methods might result in latitude and longitude values that differ in the tenth decimal place even though these differences are insignificant in the models being used.

The standard solution[6] has been to allow the user to define a "match" predicate that is used at lookup time to compare the current input argument to each of the stored parameters until a match is found. However, using the standard solution means that the entire cache might have to be searched each time, greatly limiting the effectiveness of memoization as an optimization technique. Rather than trying to do this type of match at lookup time, our approach has been to try to standardize the values at the time of the original call and then perform an exact match at lookup time. This approach has the benefit of allowing a wide range of user-defined matches but still supporting a fast and easy hash table lookup when checking the cache for matches.

Limiting the Size of the Cache

One of the problems with the current automatic memoization package is its all-or-nothing approach. Either all calculations are stored or none is stored. This is widely viewed as one of the major problems of memoization,[7] but the only solution offered in the literature (using an ordered list of cache values instead of a hash table[6]) requires one to dramatically increase the time required to access the cache on average since access time is now proportional to the number of entries in the table. Furthermore, the solution of Michie[8] and Popplestone[9] is to use a cache ordered by recency of access. Instead, we developed an approach whereby excessive entries were eliminated on the basis of frequency of use, and an algorithm was designed that supported this in time independent of the size of the table.

Cache Consistency

Most of the literature on memoization discusses its use in a purely functional environment where side effects are not possible and, thus, memoization is guaranteed safe. Even in such an environment, however, once caches become persistent, problems could arise as code evolves over time. The cache could be flushed if a memoized function's definition changed. But what if some function that was called (perhaps indirectly) changed? The stored table might now contain invalid data. Furthermore, few real-life programs are produced in purely functional environments, and one of the foremost goals of the proposed effort is to produce techniques and tools that are useful to real-world developers.

Determining if a given function is free of side effects is a provably undecidable problem, and recent heuristic attempts at automatically determining which functions are appropriate for memoization have proved unfruitful.[10] We instead propose an interactive determination assisted by the memoization system. Rather than always returning the cached value, if it has one, at the user's request, the system can recompute a certain percentage of cases and compare the newly computed value with the cached one, warning the user if differences exist. Similarly, when loading a saved memoization table from disk, the user can specify that a certain percentage of the entries will be recalculated at load time or that, at run time, a cache retrieval has a certain probability of being recalculated. This recalculation not only alerts the user to improperly memoized procedures but also helps to maintain the integrity of the cache as the system evolves over time.

Implementation in Conventional Languages

The Lisp automated memoization library relies upon several high-level features of the language, including built-in support for hash coding and functions that can be created and modified at run time. C++ lacks these features, and although hashing can be implemented, there is no way to create functions at run time.[11] As a result, our approach to automatic memoization in C++ has been to use a source code to source code translation system. Furthermore, storing arguments in a type-independent manner to be used in hashing is difficult for C++ classes with complex copy constructors, for structures whose fields get "padded" in an implementation-specific manner, and for variable-length arrays.

Despite these difficulties, C++ is widely used, and automated support for memoization in C++ would benefit many existing programs. Work is currently under way to address these problems, and an early prototype (see the following section) has been effective in an ongoing DARPA program. We hope to apply the C++ library to additional APL programs soon.

CASE STUDIES

Case Study 1: Determining Magnetic Detectability of Submarines

Table 1 gives timing statistics for the Signature Management System Magnetic Anomaly Detection module, timed after various uses of memoization were put into effect. Memoization provided a 24-fold increase in speed, even before persistent tables were used. Benchmarks can be misleading, however, and in many places the code would have been written differently if the automatic memoization package had not been available. Nevertheless, the results are illuminating, especially since they represent improvements over the original, baseline system.

Table 1. Timing of the Magnetic Anomaly Detection module of the Signature Management System.

Use of memoization	Time (s)	Relative speedup (cumulative)
Original	48	1.0
Conventional optimization added	36	1.3
Repetition over time prevented	24	2.0
Repetition within a function call prevented	2	24.0
Persistent tables added	0.001	48,000.0

Case Study 2: Detectability Planning Display

Given the diverse uses of our memoization library by various programmers on the Signature Management System program, an attempt was made to estimate its overall contribution to the system. To do this, the default display (shown in Fig. 2) was run both in the default mode and then with all memoization turned off. The timing results, which represent the final high-level computations of the system, are shown in Table 2.

Speed improved by a factor of 631, and the amount of temporary memory (garbage) allocated improved by a factor of 4822. Although other approaches would undoubtedly have been attempted if memoization had not been available, this result is illustrative for two reasons. First, it assesses the overall contribution of automatic memoization to the entire system. The system was built by multiple programmers from four companies over the course of several years, and the programmers were free to choose whatever techniques seemed most beneficial and convenient to them. Second, it shows that automatic memoization not only assisted in easily recomputing portions of the computations (by preventing repetition over time) but also dramatically improved the performance of the original computation of the display.

Case Study 3: The DARPA/Submarine Technology Department ISSIPS Magnetics Module

The Submarine Technology Department is currently developing the Integrated Submarine Stealth Information Processing System (ISSIPS) in C++, and one of the modules involves calculation of magnetic detectability. Given the success of the Lisp-based library on

Table 2. Overall contribution of memoization to the Signature Management System.

Version	Time (s)	Bytes allocated
Memoized	4.06	615,784
Unmemoized	2,562.74	2,969,392,724

the Signature Management System Magnetic Anomaly Detection module, when we completed a C++-based prototype we applied it first to this module. An 18-fold increase was obtained, even without the use of persistent tables.

FUTURE WORK

The Lisp-based automatic memoization library is relatively mature, and technology-transfer funding from the Army Research Office focuses on transferring it to a commercial Common Lisp vendor (Franz, Inc., the largest Lisp vendor worldwide). The C++-based library is in a prototype stage. This year's work focuses on its application to additional APL programs to gain experience with issues specific to conventional languages. In addition, new work planned for the upcoming year focuses on applying memoization to the Java programming language. Java is a ripe target for an automatic memoization system for the following reasons: It is often run interpreted, it is used on the World Wide Web, it has built-in support for hashing, it has been plagued by performance problems, and it is rapidly growing in popularity. However, the fact that its object-oriented model and type system are much stricter than those of C++ presents new technical difficulties.

CONCLUSIONS

Automatic memoization has been shown to be an effective optimization tool in large real-world systems. Its use in higher-level languages like Lisp is simple and has resulted in huge performance gains and more maintainable code on Navy- and DARPA-funded programs in the APL Submarine Technology Department. Application in lower-level languages like C++ is more difficult, but early results from ISSIPS are promising. However, more experience is required to identify which approaches work best. We solicit the opportunity to apply the C++ and Java libraries to additional APL programs.

REFERENCES

[1] Cormen, T., Leiserson, C., and Rivest, R., *Introduction to Algorithms*, McGraw Hill, New York (1991).

[2] Norvig, P., "Techniques for Automatic Memoization with Applications to Context-Free Parsing," *Computational Linguistics*, **17**, 91–98 (1991).

[3] Kay, M., "Algorithm Schemata and Data Structures in Syntactic Processing," in *Readings in Natural Language Processing*, Morgan Kaufmann, pp. 205–228 (1986).

[4] Earley, J., "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM* **6**(2), 431–455 (1970).

[5] Wenstrand, D., Dantzler, H. L., Hall, M., Scheerer, D., Sinex, C., and Zaret, D., "A Multiple Knowledge-Base Approach to Submarine Stealth Monitoring and Planning," *Proc. DARPA Associate Technol. Symp.*, pp. 112–122 (1991).

[6] Michie, D., "Memo Functions: A Language Feature with Rote Learning Properties," Research Memorandum MIP-R-29, Dept. of Machine Intelligence and Perception, University of Edinburgh (1967).

[7] Field, A. J., and Harrison, P. G., *Functional Programming*, Addison-Wesley (1988).

[8] Michie, D., "Memo Functions and Machine Learning," *Nature* **218**(1), 19–22 (1968).

[9] Popplestone, R. J., "Memo Functions and POP-2 Language," Research Memorandum MIP-R-30, Dept. of Machine Intelligence and Perception, Edinburgh University (1968).

[10] Mostow, J., and Cohen, D., "Automating Program Speedup by Deciding What to Cache," *Proc. Int. Joint Conf. on Artificial Intelligence* (IJCAI-85), pp. 165–172 (1985).

[11] Koenig, A., "Function Objects, Templates, and Inheritance," *J. Object-Oriented Programming*, **8**, 65–84 (Sep 1995).

THE AUTHORS

MARTY HALL is a senior computer scientist in the Advanced Signal and Information Processing Group of the Milton S. Eisenhower Research and Technology Development Center at APL. He received a B.S. in chemical engineering from The Johns Hopkins University in 1984 and an M.S. in computer science from The Johns Hopkins University in 1986, and is currently working on his Ph.D. thesis in computer science at the University of Maryland Baltimore County. His technical interests include artificial intelligence, the World Wide Web and Java software development, and simulation. Before joining the Research and Technology Development Center, Mr. Hall was a subcontractor in the Submarine Technology Department. For the past 8 years he has been an adjunct faculty member in The Johns Hopkins University part-time graduate program in computer science. His e-mail address is Marty.Hall@jhuapl.edu.

J. PAUL McNAMEE is a computer scientist in the Submarine Technology Department at APL. He received a B.S. degree in electrical and computer engineering in 1990 and an M.S. degree in computer science in 1996, both from The Johns Hopkins University. He joined APL in 1994 and has worked on several decision support systems for the analysis and real-time monitoring of submarine detectability. His technical interests include applied artificial intelligence, optimization of algorithms, and development of software engineering tools. Mr. McNamee is a member of the Association of Computing Machinery and the Institute for Electrical and Electronics Engineers. His e-mail address is Paul.McNamee@jhuapl.edu.