Automating Program Speedup by Deciding What to Cache

Jack Mostow* and Donald Cohen
USC Information Sciences Institute**
4676 Admiralty Way
Marina del Rey, California 90292

Abstract

A common program optimization strategy is to eliminate recomputation by caching and reusing results. We analyze the problems involved in automating this strategy: deciding which computations are safe to cache, transforming the rest of the program to make them safe, choosing the most cost-effective ones to cache, and maintaining the optimized code. The analysis extends previous work on caching by considering side effects, shared data structures, program edits, and the acceptability of behavior changes caused by caching. The paper explores various techniques for solving these problems and attempts to make explicit the assumptions on which they depend. An experimental prototype incorporates many of these techniques.

1. Introduction

Optimizing a program by hand is expensive, both directly and indirectly. The process itself is time-consuming and error-prone, while the optimized program is more complex and therefore difficult to maintain. Ideally we would like to write and maintain unoptimized programs and let the machine take care of optimizing them. This would free programmers to write simpler, more understandable, less efficient code, secure in the knowledge that unnecessarily expensive computation would be optimized away. While this is done to some degree by current optimizing compilers, this paper shows how more could be done by exploiting knowledge not available to standard compilers.

We focus on a single general strategy for program speedup, namely (software) caching: storing and reusing the results of computations. Caching is usually thought of as trading space for time. The cost is the space used for storing results. The benefit is the time not spent recomputing. However, caching can also reduce storage costs by eliminating the extra space that would have been allocated during recomputation. Caching can be thought of as a simple form of learning, insofar as the program tends to speed up over time.

While caching has received considerable theoretical attention [Marsh 70, Lenat 79, Bird 80, Anderson 81, Neches 81, Rosenbloom 83], especially for applicative languages, specialized knowledge representations, and production systems, we chose to investigate it in the context of general Interlisp programs [Teitelman 78]. Toward this end we constructed an experimental prototype called **Memoize** that installs caches in Interlisp programs, including itself.

The actual operation of installing a cache is trivial: a function, say f(x, y), is transformed to store its results in a table (the cache) indexed by the values of the arguments x and y. (In Interlisp, this can conveniently be done by ADVISEing f.) Before the transformed function actually computes a result, it looks in its cache, and if the entry is already there, it returns the result computed before. Thus it saves the expense of recomputing at the cost of storing and accessing the cache. Whether an entry is considered to be "already there" depends on the equivalence criterion used to compare arguments against stored keys. Some caches use an EQ test, while others use EQUAL; one can imagine using others, though **Memoize** does not.

We will refer to this program transformation as "memoizing the function f." We can also talk about memoizing program fragments other than functions. For example, the expression (f (g x)) can be memoized by rewriting it as (fg x), defining fg(x) as (f (g x)), and memoizing the function fg. Similarly, the computation of g and h in the expression (f (g (h (i x)))) can be cached by folding the expression into (f (gh (i x))), defining gh(y) as (g (h y)), and memoizing gh.

Deciding which parts of a program to memoize, and how, is far from trivial. For example, memoizing a function may require substantial modifications to the rest of the program. While the development of **Memoize** was greatly facilitated by such tightly integrated Interlisp facilities as the Masterscope program analyzer, the Pattern Match Compiler, the structure editor, and the ADVISE facility, the power of Interlisp forced us to confront many problems that do not arise in simpler languages. As we shall see, not all of these problems yield to complete, fully automatic solutions. The rest of the paper discusses the issues of safety, cost-effectiveness, and maintaining the optimized program.

2. Safety

Optimization should be "safe," i.e., *the optimized program should accomplish the same purpose as the unoptimized version.* In the case of memoizing, one might (and we used to) think that ensuring safety is straightforward: before memoizing a program fragment, analyze it to make sure that memoizing it will yield a program with equivalent behavior. As we shall see, this view is naive. First, equivalence lies in the mind of the beholder, and cannot be determined solely by looking at the code itself. Second, some changes in program behavior caused by memoizing are actually desirable; the real issue is not whether memoizing preserves "equivalence," but whether the behavior changes are acceptable. Third, safety is not a context-free property of the program fragment, but depends on the rest of the program and on how the fragment is memoized.

Therefore we define the safety of memoizing in terms of preserving the acceptability of the unmemoized program's behavior. (We ignore the issues that arise if this behavior is

---

*Current address: Rutgers University Department of Computer Science, Hill Center - Busch Campus, New Brunswick, New Jersey 08903.

**This research was performed at USC-ISI and supported by the Defense Advanced Research Projects Agency under contract No. MDA903 81 C 0335. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

nondeterministic.) By this definition, it is always safe to recompute a result rather than retrieve it. Notice that this definition may be inappropriate for programs whose correct behavior relies on their being memoized. The ability to write such programs is a potentially important benefit of automated memoizing, since it permits a simpler programming style.

Whether a change in program behavior is unacceptable, unimportant, or beneficial depends on the user's intent. For example, suppose a program is instrumented to measure the time spent in some function. If the program is being run to measure machine speed, then memoizing it would defeat the purpose. On the other hand, if the goal is to find and eliminate performance bottlenecks, then memoizing the very same program may be desirable even though it changes the result computed (i.e., the timing).

As this rather extreme example illustrates, information about which changes are acceptable cannot infallibly be inferred from the code. From a specification of what the program is supposed to do, and a derivation of the code from that specification, it might be possible to determine which changes were acceptable and which were not. However, even in the absence of such information, fairly reliable guesses can be made based on program analysis, default assumptions, and the conservative principle that it is always safe (if inefficient) to refrain from memoizing.

Memoizing a program can change its behavior in several ways. For example, it affects time and space costs, preferably reducing at least one. We assume here that changes in time and space costs are acceptable with respect to safety -- although in the case of an embedded real-time system, or limited storage, they clearly would not be. We now discuss other changes in program behavior, and the factors that influence their acceptability.

2.1. Side effects eliminated by caching

By eliminating recomputation, caching eliminates the side effects performed during the recomputation. **Memoize** considers such changes important enough to preclude caching or require user confirmation.

A computation specifically executed for side effect should not be memoized if it is important for the side effect to occur every time the computation is invoked. For example, a function that clears the screen should not be memoized, even if it always returns the same value. While a program fragment with indispensable side effects should not be memoized, it is sometimes possible to move the code that performs the side effects outside of the fragment. However, such code motion is outside the scope of this paper or **Memoize**'s capabilities.

Sometimes eliminating side effects by caching can actually improve the program.

Eliminated output: The user may not care (or may even rejoice) if memoizing the program causes it to print the same message once rather than 47 times.

Eliminated breaks: If a computation generates an error from which the user can recover, caching the result saves the trouble of fixing it again. And again...

Eliminated input: If the computation prints out a question and inputs an answer, caching it may usefully reduce the question-answering burden on the user, provided the answer is explicitly indexed under the variables on which it depends. Section 5.2 discusses how this technique was used to prevent **Memoize** from asking the same question repeatedly.

Identifying all possible side effects of a computation can be difficult for people. **Memoize** uses Masterscope to identify potential direct and indirect side effects of executing a given program fragment. To assist the user in deciding whether it is safe to memoize the fragment, **Memoize** displays its possible side effects and the calling paths by which indirect side effects might be invoked.

These side effects include setting non-local variables, evaluating expressions constructed at runtime (EVAL and APPLY, whose effects Masterscope does not attempt to predict), smashing data structures (RPLACA, PUTPROP, PUTD, etc.), and performing I/O. (For purposes of analysis, it is convenient to regard functions like TIME as a form of I/O, i.e., reading a clock.) If a program is viewed at a low enough level, allocation (e.g., CONS) is also a side effect; however, this is considered separately in section 2.4.

2.2. Eliminating infinite recursion

A useful behavior change introduced by memoizing concerns infinite recursion. Whenever the memoized code starts a cached computation for which no result has been stored, it first creates an empty cache entry. When it completes the computation, it fills in the result. If a cache lookup ever retrieves an empty entry, it means that in the absence of caching, the computation would have invoked itself with the very same parameters, i.e., would have entered an infinite recursion. In this case, the retrieval code halts and warns the user about the problem. (Section 3 describes mechanisms that handle the case where the function alters its own parameters or the global state on which it depends.)

2.3. Introducing extra computations

A memoized function should not invoke user code in situations where the unmemoized version would not, since doing so can have arbitrarily bad effects. **Memoize** must take care to avoid this problem.

If the function's arguments aren't always evaluated left-to-right before entering its body, memoizing it may cause arguments to be evaluated unnecessarily, out of order, or in the wrong environment. For example, memoizing the function AND so as to evaluate all its arguments would cause the call (AND (LISTP x) (CAR x)) to break when x was bound to a non-list, even though the unmemoized version would not. **Memoize** therefore refrains from memoizing (NLAMBDA) functions that evaluate their arguments conditionally.

The same problem can be introduced by folding a program fragment into a function, e.g., folding the expression (AND (LISTP x) (CAR x)) into (f (LISTP x) (CAR x)), where f(y, z) is defined as (AND y z). Similarly, folding the expression (f (g x) (g x)) into (h (g x)), where h(y) is defined as (f y y), will cause (g x) to be computed once instead of twice. This could introduce errors if g has side effects that need to be executed twice.

2.4. Allocation eliminated by caching

Reusing a result instead of recomputing it means that different calls on a memoized function may return the same (EQ) data structure where the unmemoized version would have constructed separate (but EQUAL) copies.

This change can be viewed as returning a substitute for the result that would have been computed by the unmemoized version. Whether the substitute is acceptable depends on assumptions made elsewhere in the program. If, as is typically the case, the only allocation information available to the program

is EQness, then changes in allocation affect programs only insofar as they affect data structure reuse. (The safety of caching becomes more problematic for programs that exploit additional allocation information, such as the memory location of an object (LOC), or the ability to iterate over all objects in memory.) Implicit assumptions may constrain how the function can be memoized, e.g., how the cache compares the arguments passed to the memoized function against the cache index, and whether it returns the cached result or a copy of it. Moreover, these assumptions may constrain reuse not only of complete results but of their substructures.

These assumptions cannot be directly determined from the code, since they depend on what behavior is considered acceptable. The obvious solution is to let the programmer identify them. Unfortunately, declaring such assumptions explicitly and keeping them consistent with changing code imposes the same kind of burden as maintaining documentation. To at least assist the user in bearing this burden, the machine can try to detect what happens to the results of computations in the unmemoized code.

Masterscope is inadequate for this purpose because it performs no data flow analysis. However, useful information can be derived by installing experimental caches. An experimentally memoized function recomputes results even when they are already stored. Comparing the stored and recomputed results detects reuse. For example, if the two are EQ, the function is reusing the same structure. Smashing can be detected by storing a copy of the result as well as the original, and comparing them later. If they are no longer EQUAL, the original has been smashed. Moreover, if the copy is not EQUAL to a later result computed for the same arguments, that result may depend on global state that has changed in the interim. Similar techniques can be used to detect argument-smashing, a problem discussed in Section 2.5.1.

We now discuss the effects of various allocation constraints.

2.4.1. When results must be reused

Sometimes program correctness requires a function to reuse certain of its results (or their substructures). If the unmemoized function satisfies this requirement, so will the memoized version, provided the cached result itself is returned rather than a copy.

Returning the same data structure is typically important when the result will be used as a name rather than as a value. For example, if the memoized function returns a node in some structure to a caller who wants to modify the structure, it may be important to return the original rather than a copy.

2.4.2. When results can be reused

If different calls to a function cause it to create different copies of the same result, but the program does not rely on this property, memoizing it can save space and time by reusing a single copy. Besides the time saved by not recomputing results, an additional savings occurs when a consumer of the results uses the EQUAL predicate to compare them. The EQUAL test succeeds much faster on large data structures when they are EQ.

2.4.3. When results can be canonicalized

The case above can be generalized. Sometimes there are large equivalence classes of inputs to a function, i.e., nothing in the program relies on the differences between the results that would have been returned by the unmemoized function. In such cases, the cached result can be reused even more by relaxing the equivalence test between inputs and cache indices. (A closely related idea for increasing reuse of cached results is discussed in Section 4.2.)

A particularly common case is where EQUAL but non-EQ inputs can safely retrieve the same result. The obvious advantage is that less space is needed to store results, and fewer results need to be computed. A possible disadvantage is that cache lookup with a more general equivalence criterion may be slower, e.g., EQUAL is slower than EQ. On the other hand, canonicalizing results in this way may permit EQUAL tests elsewhere in the program to be reduced to EQ. This is a common optimization tactic, but the analysis required to determine where it can be applied is beyond the capabilities of Masterscope. Notice also that for a program written to rely on the canonicalizing effect of caching, safety cannot be defined simply in terms of preserving the behavior of the unmemoized code.

2.4.4. When results must not be reused

Sometimes a program requires that a function *not* reuse (part of) some result, because the results returned at two different times are to be used in incompatible ways. Typically one caller smashes the result, and the next one wants the old value. Result-copying is typically used to satisfy such requirements. Since memoizing a function can cause it to reuse results where the original version did not, caching can interfere with such a requirement. The memoized function can be made to satisfy the requirement by copying the cached result before returning it, but this approach has some problems:

- How much of the structure needs to be copied? There's no point in doing a full COPY of a list if copying only the top level will suffice.
- How much of the structure is safe to copy? Other parts of the program may require corresponding substructures of the copies to be EQ.
- The copying operation might not terminate (the structure might be circular).
- The time and space costs of copy-on-retrieval may defeat the advantage of caching.

2.5. Assumptions introduced by memoizing

As we have seen above, safe caching must satisfy assumptions made by code that uses the memoized function. However, the memoizing transformation defined in Section 1 introduces code that makes assumptions of its own.

For example, this code assumes that no part of the code for cache lookup will itself be memoized, since doing so could cause infinite recursion. This problem is familiar to implementors of facilities like BREAK and ADVISE.

2.5.1. Cache-smashing

Memoize assumes that the cache will not be altered in a way that corrupts its ability to retrieve the correct result. It is reasonable to assume the integrity of cache structure built by the caching mechanism itself, since this structure did not exist in the unmemoized version. However, the cache also incorporates possible "Trojan Horses" -- the data structures passed to and returned from the memoized function. If user code retains pointers into these structures, we have to worry about it smashing them.

Different lines of reasoning can be used to show the absence of such danger. In order of increasing generality:

The cache is never smashed. This is certainly the case if the program doesn't smash the arguments it passes to the memoized function or the results returned. To prevent argument-smashing, we can copy the arguments before incorporating them into the cache and pass the originals to the function (in case the function

is tempted to smash them itself!). If the result incorporates argument structure that other parts of the program can smash, we can cache a copy of the result. To protect against a caller smashing the returned result, we can use copy-on-retrieval. Of course the allocation changes introduced by any copying must be acceptable (see Section 2.4). For example, argument-copying requires that canonicalizing be acceptable, since finding the copied arguments in the index will require EQUAL as the equivalence criterion.

The function does not access the parts of its input that might get smashed. Therefore, it would have done the same thing with the smashed input as it did in the first place. Notice that in cases where the result of the function shares structure with the input, smashing can change a valid cache entry into another valid cache entry that was never explicitly computed. For example, if we cache the identity function, (lambda (x) x), smashing an input won't invalidate the cache.

All changes to a cache entry leave it valid, i.e., the (modified) result is an acceptable substitute for the one that would have been computed for the (modified) arguments. For example, a function that counts the number of NILs in a list will not be affected by an operation that reorders the list.

Of course, the analysis required to support such arguments is far beyond the capabilities of Masterscope, and if it is done anyway (e.g., by hand), one must still watch out for smashing done by the user at the top level or in breaks.

3. Transforming programs to make memoizing safe

So far, we have only considered the safety of the memoizing transformation defined in Section 1 (and simple variants that make copies). Naturally, much more can be done if one is willing to change other parts of the program, such as those making assumptions about the code to be memoized. For example, suppose the function F is required to return a copy of its result because one caller will smash it. If that one caller were changed to make a copy of the result and use it instead of the original, then F could be memoized safely with no copying.

The most common obstacle to caching is the problem of side effects. The safety of memoizing a function depends on two classes of side effects. Section 2.1 discussed the safety of eliminating side effects invoked (directly or indirectly) by the function itself. We now discuss how to make memoizing safe in spite of side effects remaining in the program. In particular, we show how the program can be transformed to prevent retrieval of results rendered obsolete by changes in global state.

Side effects that affect the integrity of a cache can be viewed generally as smashing the input or output of the memoized function. The input is considered to include all data on which the function depends, both the explicit arguments passed as parameters and the implicit arguments accessed in the function, such as free variables, property lists of atoms, and even the function's definition. We view all changes to such data as forms of smashing (e.g., setting a global variable is smashing its value cell). Since implicit and explicit arguments are really the same in this view, we see that one reaction to dependence on global state is to treat it like an explicit argument and index on it. Conversely, the approach to global state described below can be applied to explicit arguments as well.

Input to a memoized function may contain substructure that is not actually accessed by the computation. Given some way to record which part actually is used, the cache could safely be indexed on only that part, for example by building a discrimination tree that only tests the relevant parts. This could be worthwhile for a computation that ignores most of its input data. A related idea is described in Section 4.2.

The general strategy for dealing with global state dependencies requires cooperation from the other parts of the program:

1. Detect state changes that might cause cache entries to become invalid.
2. Figure out which cache entries might be affected.
3. Do something about them.

Since detecting and reacting to state changes is an additional cost, and it is always safe to recompute, different schemes may be appropriate in different situations:

Don't cache.

Discard an entire cache at the end of some computation. This method is useful if the memoized function is executed many times by the computation and depends on state information changed between (but not during) invocations of the computation, e.g., arguments passed to some procedure containing the computation. In particular, if the computation corresponds to a program block, making the cache a local variable of the block will cause it to be discarded when the block is exited.

Empty the cache for a memoized function in response to any side effect that might smash the cache or change the function's implicit arguments.

Discard individual cache entries in response to any side effect that might smash them or change the implicit arguments used to compute them.

Update the affected entries without recomputing them, i.e., use finite differencing [Paige&Koenig 82]. This technique requires knowledge about the nature of the side effect as well as sophisticated program analysis. Section 3.3 describes a weaker technique that reduces the amount of recomputation without requiring such knowledge or analysis.

We now discuss methods for detecting state changes, identifying the cache entries they might invalidate, and restoring cache validity.

3.1. Detect state changes

To make a memoized program respond to side effects, we must identify operations that change global state and transform them to do something about the caches they might affect. This triggering can be done with various degrees of selectivity depending on the degree of sophistication with which the program is understood.

Modify primitives: At the lowest level, Lisp's structure-smashing primitives (RPLACD, SETA, SET, etc.) could be altered to trigger the desired response directly. This approach requires no understanding of the program and would catch all structure-smashing operations, many of which would not invalidate any cached results. It is not actually used by **Memoize**.

Modify functions: If one can identify the (higher level) functions in the program that modify state on which caches depend, those functions can be ADVISEd to take appropriate action.

Modify invocations: Sometimes one can determine not only which functions, but which invocations affect a cache. For example, if a cache is sensitive to the global variable V, then (SETQ U ...) will not affect it. In this case, the particular

invocations can be transformed to take appropriate action. **Memoize** uses Masterscope to find these invocations, and uses Interlisp's structure editor to insert appropriate trigger code around each one.

It is also necessary to respond to state-changing operators invoked interactively by the user. **Memoize** uses the hooks Interlisp provides for this purpose (e.g., MARKASCHANGED and SAVESETQ).

3.2. Identify suspect cache entries

After noticing a change, it is necessary to identify the affected cache entries. This can be done with various degrees of expense and precision, as long as we err on the side of thinking that a valid entry is invalid. Again, greater understanding of the program can increase precision at constant expense, or reduce the expense for constant precision.

Brute force: Great precision with no knowledge is possible at great expense. The very low-level solution would be to record every memory cell accessed by the cached computation. A cell could then be used to index a table of the cache entries that depend on it. This is not done by **Memoize**.

Static analysis: Sometimes one can determine which modifications can possibly affect which caches, e.g., the memoized function depends on the value of the variable V, and we can determine which operations affect the value of V. Masterscope recognizes dependence on variables, fields of records, function definitions, etc., and installs code to react to changes in them. However, Masterscope misses some dependencies due to the problem of aliassing: smashing U might affect V, if U and V share structure. **Memoize** relies on the programmer to identify code that could affect the implicit arguments of a function in non-obvious ways.

Dynamic dependency recording: Code that accesses global state can be altered to record (at runtime) what state an individual entry depends on. This is more expensive than static analysis, but more precise, especially when not all elements of a cache depend on the same state. In different cases, the improved precision may or may not pay off. Again, **Memoize** relies on the user to point out non-obvious code that might smash state accessed by the function.

A special case of dependency is when the computation of a new cache entry retrieves an existing one, which in turn depends on other data. If that data is ever changed, it will invalidate the old entry, which in turn will invalidate the new one. Thus we need not record the dependency of cache entries on data they use only via other entries.

3.2.1. Static versus dynamic dependency analysis

Memoize decides heuristically whether to predict dependencies statically or record them dynamically.

Dependencies on a global variable are determined by static Masterscope analysis. This determines with reasonable precision which updates might invalidate which caches. E.g., if f uses the global variable V, then changing the value of V is liable to affect f.

Dependencies on properties and fields are recorded dynamically, since static analysis would not be sufficiently precise. Suppose a memoized function contains the expression (GETPROP x 'COLOR). With the dynamic approach, we can tell which atom's COLOR a cached result depends on. If we relied on Masterscope's static analysis, we'd have to consider every entry in the cache suspect whenever *any* atom changed COLOR.

To determine dependencies on function definitions, **Memoize** uses a combination of approaches. It uses static analysis to find the functions invoked by a cached computation, since Masterscope determines these with high precision. To record function definitions accessed in other ways, e.g., with GETD, **Memoize** uses the dynamic method, since static analysis lacks precision in such cases.

3.3. Update invalidated result

When a cached result is identified as potentially invalid, something must be done to prevent it from being retrieved -- or any other cache entries that depend on it. The obvious thing to do is simply delete the entry, since if it is ever needed again, it will have to be recomputed anyway. (Notice that if in the course of computing the result, the memoized function first accesses some data and then smashes it, the entry to be deleted will still be empty at the time of smashing. If the function shoots itself in the foot in this way, the result eventually computed should be considered correct for this invocation but not for the next one. But watch out for recursion!)

Deleting a suspect result can be very wasteful if it is likely still to be valid and there are other entries that depend on it. This typically occurs when the cached computation is a many-to-one function (e.g., NULL, LENGTH) of the changed state information. We now present two solutions to this problem; their relative effectiveness will be described in Section 4.3.

3.3.1. Eager recomputation of suspect results

One technique for avoiding the deletion of valid cache entries that depend on a suspect result is to recompute it immediately. If the new result is equivalent to the old one, nothing else needs to be done; the entries that depend on it are still valid. If not, the entries that depend directly on the changed result are suspect, and the process repeats up the chain of dependencies. Since the dependency structure is finite and acyclic (see Section 2.2), this process is guaranteed to terminate.

Eager recomputation is not always safe, since the computation might never have occurred in the unmemoized version. (Section 2.3 describes similar hazards.) For example, suppose the memoized expression is 1/X, where X is a global variable. Eager recomputation after setting X to zero would generate an error, even if the original program never evaluated the expression when X was zero.

Memoize knows that functions which have no side effects and never generate errors, e.g., LISTP, are safe to eagerly recompute. When these restrictive conditions are not met, it asks the programmer whether eager recomputation is safe.

3.3.2. Propagation and exoneration of suspicion

An alternative technique that prevents retrieval of invalid results and avoids unnecessary recomputation involves marker propagation. Results that depend directly on changed data are marked "Must Recompute." Results that depend indirectly on changed data are marked "Might Need to Recompute" since they depend on entries that might or might not change when they are recomputed. The cost of marker propagation is limited, as in the case of eager recomputation. Also, when an entry already marked "Might Need to Recompute" is encountered, no entries that depend on it need to be marked.

When an entry marked "Must Recompute" is subsequently retrieved, it is duly recomputed. If the new result is equivalent to the old one, the entry is marked "OK." Otherwise, the new result is stored and the entries that depended directly on the old result are marked "Must Recompute."

When an entry marked "Might Need to Recompute" is retrieved, an attempt is made to exonerate it without recomputing it, by checking the entries on which it depends. If all these supporting entries prove to have been valid, the original entry is marked "OK" and returned. If any of these entries was suspect, the attempt to retrieve it will (recursively) cause it to be exonerated or recomputed. If its value has changed, the exoneration process is aborted and the original entry is recomputed.

The supporting entries must be retrieved in the order in which they were used during the original computation, so this order must be recorded. Retrieving them in another order may be unsafe, as in the case of eager recomputation.

The propagation technique avoids unnecessary recomputation: a result is recomputed only if it depends directly on changed data (including other cache entries) *and* would have been recomputed in the uncached computation.

4. Cost-effectiveness

Just because a computation is safe to cache doesn't mean that doing so is worthwhile. A cache should only be installed if it is likely to be cost-effective. Moreover, just as the safety of caching depends on how it is implemented, so does its cost-effectiveness. Thus the issue is not just to decide whether a given computation is cost-effective to cache, but rather which parts of it to cache, and how, so as to maximize cost-effectiveness.

The impact of a cache on memory usage depends on such factors as the number of entries, the size of each entry, the degree of structure sharing between entries, and how long each entry is stored (and hence unavailable for garbage collection).

The impact on time depends on such factors as the hit rate (how often a retrieval replaces recomputation), the cost of cache lookup, the frequency of side effects that render cache entries suspect, and the cost of responding to them.

A cache provides a net speedup over its lifetime if doing without the cache would take more time, i.e.,

$$\sum_{calls} \textit{Computation time}$$

$$>$$

$$\textit{Installation time} +$$

$$\sum_{misses} (\textit{Lookup time} + \textit{Computation time}) +$$

$$\sum_{hits} (\textit{Lookup time} + \textit{Retrieval time}) +$$

$$\sum_{updates} \textit{Update time} +$$

$$\textit{Bookkeeping overhead},$$

where $calls = hits \cup misses$.

Predicting cost-effectiveness entails predicting the values of these variables. This section discusses factors that influence those values and heuristics for estimating them. Of course, the ultimate test of a cache's cost-effectiveness is to try using it. We view the heuristics presented below as techniques for identifying promising candidates. While **Memoize** uses these heuristics to decide what to cache, it collects statistics on actual cache performance, e.g., hit rate, update frequency, and how much time was spent computing each cached result. Such data could be used, either by the user or by the machine, to decide which caches to keep.

4.1. Computation cost

The time to execute an expression depends on how long each operation takes and how many times it's invoked. The latter quantity depends on things like how many times loops are executed and the relative frequency of the branches in a conditional. These depend on runtime data distribution and are hard to predict analytically (though see [Kant 79]).

Memoize uses a simple one-bit heuristic theory of computational complexity: functions and expressions are either cheap or expensive.

- Built-in Lisp functions are classified individually.
- Memoized functions are cheap.
- User input is expensive.
- Recursive functions are expensive.
- Expressions containing loops are expensive.
- Expressions that call expensive functions are expensive.
- Other functions and expressions are cheap.

To classify a compiled function whose definition is unavailable for analysis, **Memoize** can either ask the user, or instrument the function (with Interlisp's BREAKDOWN) and postpone classifying it until data is available.

4.2. Hit rate

While the exact hit rate of a cache cannot be predicted *a priori*, certain useful characterizations can be made based on static program analysis. In deciding which parts of a computation to cache, **Memoize** uses the intuitive notion that the hit rate of a computation decreases as the variability of its input increases. In particular:

Hit rate is a decreasing function of the ***amount of changing global state information*** used in the computation.

Hit rate is a decreasing function of the ***frequency of global state changes*** that invalidate cached results.

Hit rate is a decreasing function of the ***number of possible values of a cache parameter.*** This number is bounded by the domain of the parameter type; for example, a boolean parameter takes on at most two values.

Hit rate is a decreasing function of the ***number of parameters.*** If it's worth caching an expression, and that expression contains subexpressions with fewer parameters, they may be worth caching as well, since their hit rates will be higher.

More generally, ***many-to-one mappings reduce variability*** and therefore can be used to increase hit rate. If (f (g x)) is the computation to be cached, and g is a many-to-one mapping, it's better (in terms of hit rate) to index a cache for f by the values of (g x) than to index a cache for the composite function fg on the values of x. However, if the computation has the form (f (g x) (h x)), where g is a many-to-one mapping and h is not, then using (g x) instead of x as an input parameter will not increase the hit rate. That is, increasing the hit rate requires masking the variability in x along every input path. Moreover, this argument must be applied to the total set of inputs -- for example, a cache for (f (CAR x) (CDR x)) might as well be indexed on x.

4.2.1. Choosing cache indices to maximize hit rate

A variable used in a cached computation can be treated either as an explicit cache parameter or as an implicit state variable. This decision should be made so as to maximize the hit rate. A variable whose values tend to recur should be used to index the cache. Conversely, if an explicit parameter will be stable over

many successive calls, and then change without returning to its former value, it might as well be treated as a global variable. Of course, if every call to a function is going to have different parameter values, there is no advantage to memoizing it.

Memoize uses a simple heuristic for classifying variables: treat function arguments and local variables bound outside the memoized expression as parameters, and everything else as global state variables. This heuristic works well for the examples encountered so far, where the global state variables have been longish lists, but may be worth refining, for example to treat global flags as parameters.

4.3. Update cost: eagerness versus suspicion

The relative costs of the two schemes described in Section 3.3 for responding to cache-invalidating side effects depend on a tradeoff between unnecessary recomputation and unnecessary propagation.

The marker propagation scheme performs a subset of the computation the unmemoized version would have performed. In good cases, only a small subset is performed; in the worst case, the same amount of computation occurs, plus the additional overhead of marker propagation. A cache entry can never be marked "Might Need to Recompute" or "Must Recompute" more often than it would have been computed in the absence of caching. Therefore this overhead is at worst proportional to the amount of original computation. This estimate must be adjusted slightly since entries that depend directly on a changing piece of global state will be marked "Must Recompute" every time that piece changes, even though propagation only occurs the first time. The adjustment for each such piece of state is proportional to its update frequency times the number of entries that depend on it directly.

The eager recomputation scheme (when safe) runs the risk of recomputing a result unnecessarily, i.e., either the entry will never be retrieved again, or it will need to be updated again before it is. However, when the result turns out to be unchanged, this scheme avoids the overhead of propagating markers to other entries.

The two strategies can be mixed together, even on the same cache. At update time, **Memoize** decides which one to use by predicting their relative costs based on the history of the suspect cache entry. Of course this computation may itself take longer than the time saved by choosing the better scheme. This decision mechanism illustrates a recurring issue in caching: the cost of information. An alternative scheme could use heuristic approximations. The degraded accuracy might well be offset by eliminating the costs of recording and using a detailed history of the entry.

5. Maintaining memoized code

Optimizing a program, whether automatically or manually, tends to make it more complex and harder to maintain. The optimizations introduce assumptions that may be violated by subsequent changes to the program.

The obvious solution to this problem is analogous to recompilation: let the user maintain the unoptimized code, and reoptimize after each change. This approach is grossly inefficient, especially for the sort of "exploratory programming" [Sheil 83] characteristic of AI research. Such programming, illustrated by **Memoize**'s own evolution, involves a repeated cycle of running the program, deciding to change its behavior, making the necessary edit, and resuming execution. Inserting a full-fledged reoptimization phase after every program edit would be wasteful:

- Massive reoptimization would itself be time-consuming, and wasteful to the extent that it was simply reinstalling identical caches.
- If reoptimization were not fully automatic, it would include the expense of making the user answer the same questions as before, which would be wasteful to the extent that the answers had not changed.
- Re-memoizing from scratch would mean discarding all existing caches, even though their entries might not be invalidated by the edit. These entries would have to be recomputed if they were needed again.

On the other hand, we don't want the inefficiency of unoptimized code. What's needed is something like efficient incremental recompilation. Our solution is to let the user edit the memoized code, and have the machine (with some help from the user) do whatever is needed to keep the program safe.

5.1. Which caches are affected by a program edit

Program edits can affect caching in various ways:

- Existing cache entries may become invalidated.
- A memoized function may become unsafe to memoize.
- An existing cache may cease to be optimally cost-effective.
- An unmemoized function may become worthwhile to memoize.

Memoize only worries about the first two cases. When a function becomes unsafe to memoize, its cache is removed. **Memoize** then considers re-memoizing the function. Failure to deal with the last two cases may affect the efficiency of the edited program, but not its safety.

It is easy to see that editing a function may affect its own safety and that of its callers. When a function is edited, **Memoize** finds any memoized functions related to it in this way, and warns that their caches may no longer be safe.

It is harder to identify other caches affected by editing a function. Some such effects could be noticed straightforwardly For example, if a function is modified to set a global variable, any memoized functions that access the variable would be affected. However, other cases would require sophisticated data flow analysis. Moreover, no matter how good the tools for such analysis were, they could not detect changes in the programmer's intent and implicit assumptions about the code.***

5.2. Speeding up re-optimization by reusing information

The cost of re-memoizing can be further reduced if all the information about the program used during the original memoizing process is recorded. Information known to depend only on properties of the program that haven't changed is still valid. This includes the results of program analysis as well as the user's answers to questions about the program.

Memoize caches both kinds of information, but records its dependencies on the program only incompletely. The problem with caching a result of program analysis is that it may be invalidated by a change almost anywhere in the program. For example, changing f to call g may change whether h indirectly calls i. The problem with remembering the user's answer to a question about the program is that which properties of the program the answer depends on is implicit in the mind of the user. (See Section 2.1.)

*** Barring a breakthrough in ESP research.

6. Conclusion

We have tried to systematically analyze the problems associated with software caching in a general programming environment. These problems include deciding which parts of a program are safe to memoize, transforming the rest of the program to make them safe, choosing the most cost-effective ones to memoize, and maintaining the optimized code. This analysis extends previous work on caching by considering side effects, shared data structures, beneficial behavior changes, and program edits.

The insights we have gained include:

- The right question to ask about a program transformation like **Memoize** is not whether it leaves the program's behavior unchanged, but whether the changes it makes are acceptable.
- There is not enough information in a program to determine which changes are acceptable.
- Safety and cost analysis should be viewed as deciding how to rearrange a computation into cached and uncached parts, rather than a simple binary choice about whether to memoize a given function.
- This analysis requires a deep understanding of the program to be optimized, involving questions that are in general undecidable, e.g., "Is the result of function F ever smashed?," or depend on implicit assumptions, e.g., "Does the program rely on the difference between the values returned for EQUAL inputs?"

In addition, the paper presents some non-obvious techniques for eliminating recomputation:

- A simple mechanism that detects certain forms of infinite recursion.
- Various schemes for detecting and preventing assaults on cache integrity.
- Various schemes for identifying invalidated results.
- A one-bit heuristic theory of computational complexity.
- Exploitation of many-to-one mappings to increase hit rate and reduce update cost.
- Methods for maintaining cached results and cache safety in the face of program edits.

Memoize itself constitutes a contribution of sorts, but one that should not be misinterpreted. While it serves as a demonstration of many of the techniques described here, it is not a practical tool and should not be evaluated as one. For example, any statistics on its performance would be distorted by the extensive bookkeeping it performs for experimental purposes. The amount of speedup obtained by caching can be arbitrarily high or low, depending on the program to be memoized; evaluating a real tool would require applying it to a "representative" sample of programs. Moreover, applying such a tool to existing programs would say nothing about how much simpler the programs might have been if they had been developed with the tool in mind in the first place.

The main value of **Memoize** has been as an exploratory vehicle: the bugs and opportunities it has exposed have greatly improved our insights into caching. One of these insights concerns the additional work required to extend **Memoize** into a useful tool. We had originally hoped that the process of installing caches in an Interlisp program could be made essentially automatic. We now understand much more clearly the obstacles to achieving that ideal.

Acknowledgements

We would like to thank Bill Swartout for asking some good questions and Bob Balzer for suggesting this problem in the first place.

References

[Anderson 81] J. A. Anderson (ed.), *Cognitive Skills and their Acquisition*, Erlbaum, 1981. Contains papers presented at 1980 Carnegie Symposium on Cognition, Pittsburgh, PA.

[Bird 80] R. S. Bird, "Tabulation techniques for recursive programs," *ACM Computing Surveys* 12, (4), 1980, 403-417.

[Kant 79] E. Kant, "A knowledge-based approach to using efficiency estimation in program synthesis," in *IJCAI-6*, pp. 457-462, Tokyo, Japan, 1979.

[Lenat 79] D. B. Lenat, F. Hayes-Roth, and P. Klahr, "Cognitive economy in artificial intelligence systems," in *IJCAI-6*, pp. 531-536, Tokyo, Japan, 1979.

[Marsh 70] D. Marsh, "Memo functions, the Graph Traverser, and a simple control situation," in B. Meltzer and D. Michie (eds.), *Machine Intelligence 5*, pp. 281-300, American Elsevier, New York, 1970.

[Neches 81] Neches, R., *Models of Heuristic Procedure Modification*, Ph.D. thesis, Department of Psychology, Carnegie-Mellon University, 1981.

[Paige&Koenig 82] R. Paige and S. Koenig, "Finite differencing of computable expressions," *ACM TOPLAS* 4, (3), July 1982, 402-454.

[Rosenbloom 83] Rosenbloom, P. S., *The Chunking of Goal Hierarchies: A Model of Practice and Stimulus-Response Compatibility*, Ph.D. thesis, Carnegie-Mellon University, 1983. Comp. Sci. Tech. Rep. #83-148.

[Sheil 83] Beau Sheil, "Power tools for programmers," *Datamation*, February 1983, 131-144.

[Teitelman 78] Teitelman, W. , *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.