Compile-Time Function Memoization

Arjun Suresh, Erven Rohou, André Seznec

HAL Id: hal-01423811

https://hal.inria.fr/hal-01423811

Submitted on 31 Dec 2016

Compile-Time Function Memoization

Arjun Suresh *

The Ohio State University, USA
suresh.86@osu.edu

Erven Rohou   André Seznec

Inria/IRISA, France
first.last@inria.fr

Abstract

Memoization is the technique of saving the results of computations so that future executions can be omitted when the same inputs repeat. Recent work showed that memoization can be applied to dynamically linked pure functions using a load-time technique and results were encouraging for the demonstrated transcendental functions. A restriction of the proposed framework was that memoization was restricted only to dynamically linked functions and the functions must be determined beforehand. In this work, we propose function memoization using a compile-time technique thus extending the scope of memoization to user defined functions as well as making it transparently applicable to any dynamically linked functions. Our compile-time technique allows static linking of memoization code and this increases the benefit due to memoization by leveraging the inlining capability for the memoization wrapper. Our compile-time analysis can also handle functions with pointer parameters, and we handle constants more efficiently. Instruction set support can also be considered, and we propose associated hardware leading to additional performance gain.

Categories and Subject Descriptors D.4.3 [*Programming Languages*]: Processors—Compilers; D.4.3 [*Programming Languages*]: Processors—Optimization; B.3 [*Hardware*]: Memory Structures

Keywords compilation, optimization, memoization, performance, instruction set extension

1. Introduction

Memoization – saving the results of a sequence of code execution for future reuse – is a time-old technique to improve the run time of programs. Over the years, memoization has been proposed at various levels of implementation ranging from function level in software to instruction level in hardware. Function memoization – where the granularity of memoization is a function – aims at limiting repeated function executions, thereby saving CPU cycles. As with other memoization schemes, there is a trade-off between saved CPU cycles and increased data storage. Overheads come

from storing function results in a table, and looking-up the table at the next invocation. The challenge with memoization is to minimize these overheads and gain maximum performance. The smaller the amount of code being saved by memoization, even smaller must be the overhead. This implies a very fast lookup mechanism and a table usage which should not hamper the normal execution of the program. Further, identifying the functions for memoization and applying a suitable memoizaton technique is also important.

Though function memoization have been proposed long back, there was no pure software approach with performance benefit, until recent work by Suresh et al. [27] where dynamically linked functions were targeted for memoization and performance benefit was demonstrated for a variety of transcendental functions. Their main advantage is that they intercept functions at load-time and they provide a way for memoizing dynamically linked functions without the need of source code and recompilation. However, their technique is restricted to dynamically linked functions, and the target functions must be determined beforehand.

We found that most of the pure function codes are either transcendental functions or are inlined by the compiler due to very small size (like expressions used for indexing arrays). Still, pure and expensive functions are present in case of recursions and applications which are critical to numerical computation (cases where a closed form solution is not available). Also there are cases of function usage with pointers and the pointed value can be used for memoization. Further, by capturing constants at call site, we can optimize and handle more functions (by reducing arguments for memoization) for memoization. Thus we propose a compile-time memoization technique which can handle all these cases. We also propose a hardware model for our memoization scheme which can increase the potential benefit of memoization.

The remaining part of the paper is organized as follows: Section 2 first reviews related work. We then present the advantages of our compile-time approach in Section 3. Section 4 then describes our implementation, how we identify potential candidates for memoization and the involved challenges. Section 5 details the experimental setup used and the results of our memoization technique. Section 6 details our hardware proposal for memoization and estimates the additional performance gain. Section 7 concludes.

2. Related Work

Memoization has been implemented at various levels of execution: instruction level, block level, trace level and function level, using both hardware and software techniques. At functional level the term "memo function" was coined by Donald Michie [21] in 1968.

At instruction level Richardson [23] dealt with trivial and redundant computations. Citron et al. [9] propose a hardware-based technique that enables multi-cycle computation in a single-cycle.

In a follow up paper on memoization, Citron et al. [10] showed the scope of memoizing trigonometric functions $(\sin, \cos, \log, \exp \ldots)$ on general purpose or application specific floating-point processors. To enable memoization for user written function they propose two new instructions – to lookup and update a generic MEMO TABLE. By doing such a hardware implementation they achieved a computation speedup greater than $10\,\%$. They also compared the same using a software mechanism which could not yield a speed up but gave a slowdown of $7\,\%$.

González et al. [13] explore hardware techniques for trace-level reuse. In instruction-level memoization a single instruction is reused while in trace-level memoization a dynamic sequence of instructions are reused. At block level Huang et al. [16] investigate the input and output values of basic blocks and find that these values can be quite regular and predictable, suggesting that using compiler support to extend value prediction and reuse to a coarser granularity may have substantial performance benefits.

Memoization has been used in programming languages such as Haskell [1] and Perl [2]. The presence of closure [22] in functional programming languages like Lisp, Javascript etc. give a ready-to-use mechanism for the programmer to write the memoization code. McNamee and Hall [19] propose automatic memoization at source level in C++.

When memoization is used in procedural languages, special techniques are needed to handle procedures with side effects. Rito et al. [25] used Software Transactional Memory (STM) for function memoization, including impure functions. Their scheme provides a memoization framework for object oriented languages which are traditionally difficult for memoization as the outcome of a method often depends not only on the arguments supplied to the method but also on the internal state of objects. Tuck et al. [28] used software-exposed hardware signatures [3] to memoize functions with implicit arguments (such as a variable that the function reads from memory). An extension of simple memoization is used by Alvarez et al. [1] in their work on fuzzy memoization of floating point operations. Here, multimedia applications which can tolerate some changes in output are considered and memoization is applied for *similar* inputs instead of strictly identical (several inputs produce the same result). A similar technique is also used by Esmaeilzadeh et al. [11] in their work about neural accelerators to accelerate programs using the approximation technique. The basic idea is to offload code regions (marked as approximatable by the programmer) to a low power neural processor. Since approximation enhances the scope of memoization, the technique gave them very good results.

Memoization is also used in graphical domains and in a recent work, [2] Arnau et al. proposed a hardware scheme in GPUs for saving redundant computations across successive rendering of frames. Long et al. [18] propose a multi-threading micro-architecture, Minimal Multi-Threading (MMT), that leverages register renaming and the instruction window to combine the fetch and execution of identical instructions between threads in SIMD applications. The main idea in MMT is to fetch instructions from different threads together and only splitting them if the computation is unique.

In this work, we propose function memoization at software level which can be applied across different languages using a compile-time approach. Our framework uses large memoization tables to capture longer intervals of repetition that benefit computationally intensive pure functions at the software level without programmer intervention and we evaluate the benefit on current architecture. Our experimental results show that software memoization is capable of producing speed-up in current architectures over current compiler techniques. We also propose a hardware scheme for additional performance gain of our memoization approach.

The most closely linked work to ours is the work by Suresh et al. [27] where memoization is shown to be working effectively for dynamically linked *math* functions. We outline the major difference as follows:

1. our technique applies to any function: part of the executable or from a dynamically linked library;

2. memoized function are identified automatically, without the need to manually predetermine them;

3. memoization code is combined with the source code, and hence usually inlined by the compiler resulting in significant speedups;

4. we handle global variables, pointers and constants;

5. we propose hardware scheme for additional improvement of the memoization benefit.

3. Compile-Time Memoization

Doing memoization at compile-time enables both user written as well as library functions to be memoized. It also provides a way to handle some impure functions such as those which read/update global variables, have pointer arguments etc. Also, constants can be identified at call sites and possibly optimized out. Inlining the memoization wrapper is another advantage given by enabling memoization at compile-time. Though small pure functions in user code are usually inlined by the compiler, there are cases like recursion with large/unknown depth, dynamic linking, different object modules without inter-procedural optimization etc. where inlining fails. To leverage inlining for memoization, we provide a small inline wrapper for memoization that performs the table lookup and resorts to a function call only in case of a miss. This increases the effect of memoization and lowers the threshold needed for memoizing a function. As an example, a call to function `float pure_foo(float arg)` is replaced by a call to the following wrapper, potentially inlined.

```
float _memoized_pure_foo(float arg) {
    if (lookup(&result, arg))
        return result;
    else {
        result = pure_foo(arg); /* call original function */
        update_table(arg, result);
        return result;
    }
}
```

In addition to providing compiler optimizations, inlining also saves a function call whenever there is a hit in the table. In order to measure the performance impact of saving this call we put a dummy call to a dynamically linked function in a custom code having 100 million function calls. The run time of the inlined memoization code made a time difference of $400\,$ms which would mean $\frac{400.10^{-3}}{100.10^{6}} = 4\,ns$ on average for a single call. We ran our experiment on a CPU running at 2.3 GHz and hence the run time amounts to $4 \times 2.3 \approx 9$, which means that approximately 9 clock cycles are saved on average for a single call. So, even without considering possible compiler optimizations on the wrapper code, we expect to

---
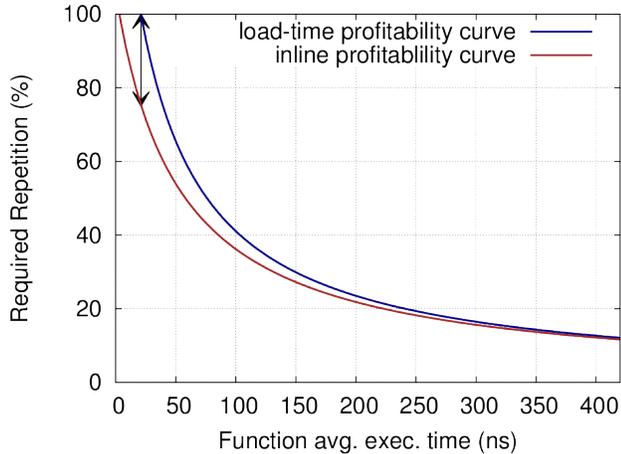
Figure 1. Profitability: load-time vs inlining

Algorithm 1 LLVM Module Pass
***

Require: Source Module
Ensure: Function calls replaced by their memoized wrapper and produces function meta for memoization
  for each function f **do**
    if IsMemoizable(f) **then**
      MakeMemoizedFunc(f);   ▷
Copies the function prototype and create a new function (name starting with memoized _) prototype with empty body
      ReplaceCallSite(f);   ▷ Replaces all calls
to original function with calls to new function (the function body will be available only during link time)
      OutputFunMeta(f);   ▷ For input to IFM
    end if
  end for
***

Figure 2. Flow Diagram of Memoization

Algorithm 2 Identifying memoizable functions
***

Require: Function
Ensure: Identifies if input function can be memoized
  function IsMemoizable(f)
    if IsDeclaration(f) OR IsIntrinsic(f) OR IsVarArg(f) OR MayBeOverridden(f) **then**
      return false;
    end if
    if IsMemoizableLib(f) **then**
      return true;
    end if
    count = 0;   ▷ Initializing count for keeping track of recusive calls
    if IsProperArguments(f) AND IsGlobalSafe(f) AND CheckFunctionCalls(f) **then**
      return true;
    end if
    return false
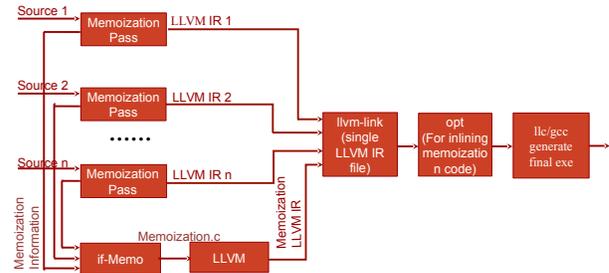 **end function**
***

save 9 clock cycles by inlining during a hit in the memoization table. Suresh et al. [27] performed a profitability analysis to quantify the required repetition rate H based on the function execution time (see Equation 1), where t_{mo} is the miss overhead, T_f the original execution time, and t_h the hit time. Figure 1 shows how inlining the wrapper increases the applicability of memoization, in particular for short function (say, under 100 ns). In particular, when the load-time approach reaches its limit (100 % repetition required to have a benefit), inlining lowers the requirement to 75 %, a more feasible threshold.

$$H > \frac{t_{mo}}{T_f + t_{mo} - t_h} \tag{1}$$

In our work, we aim to provide memoization capability to a code using a compile-time technique. During compilation stage we should identify which functions are potential candidates for memoization and then provide memoization capability for them which is done by adding a memoization wrapper for them. The memoization wrapper must have inlining capability in order to get the potential benefit of avoiding a dynamic function call during a hit in the memoization table. Impurities for function memoization in the form of global variable usage, pointer arguments are also to be handled.

4. Implementation

We implemented our technique in the LLVM compiler [17]. To support Fortran applications, we relied on the `dragonegg` GCC plugin. It uses the GCC front-end to parse the source code, but replaces optimizers and code generators by those of LLVM. We developed an LLVM optimizer pass at module level since we are analyzing all functions in a module and possibly modifying them. This pass iterates through each function and marks them if they are memoizable. If so, we create a template for memoization wrapper and replace each call site of the function, with a call to the corresponding memoization wrapper. The prototype of the wrapper memoization function is same as that of the original function except the name being prefixed with `__memoized_` and parameter order possibly being changed under the condition detailed in Section 4.1. The pseudocode for our approach is given in Algorithm 1.

We considered functions with up to three parameters as the chance of repetition goes down and the overhead increases when

the number of parameters increases. Constant arguments at call sites do not count towards this limit as per the algorithm detailed in Algorithm 6.

4.1 Function Templates and Parameter Ordering

To avoid a combinatorial explosion of the number of function types to be handled, we sort the parameters by type and our function prototype is generated as per this new order. We generate the call string for the original function during this stage itself and hence it preserves the actual call order. This is exemplified as follows.

Let the original function prototype be:

```
void foo(float arg1, int* arg2, int arg3)
```

Algorithm 3 Identifying if function is safe for memoization

---

  function CHECKFUNCTIONCALLS(f)
    for each instruction I in f **do**
      if ISCALLINSTRUCTION(I) **then**
        calledFunction, f ← GETCALLEDFUNCTION(I)
        if ISPURE(f) OR I = f **then**
          continue
        end if
        count++
        if count = 10 **then**   ▷
We only check up to 10 levels of function calls as more than 10 levels of distinct pure function calls is highly unlikely
          return false
        end if
        if ISMEMOIZABLE(f) **then**
          continue
        end if
        return false
      end if
    end for
  end function

Algorithm 4 Identifying if input function has safe arguments

---

  function ISPROPERARGS(f)
    for each argument A of f **do**
      if ISPOINTERTYPE(A) **then**
        if ISMEMOIZABLEPOINTER(A) **then**
          continue
        end if
        return false
      end if
    end for
  end function

  function ISMEMOIZABLEPOINTER(P)
    for each use I of P **do**
      if I ≠ LoadInstruction AND I ≠ StoreInstruction **then**
        return false
      end if
      return true
    end for
  end function

Now, after sorting, suppose their order becomes `arg2`, `arg3`, `arg1`. Our generated prototype for function `foo` will be:

```
_memoized_foo(int arg1, float arg2, int* arg3)
```

which, in turn, calls:

```
foo(arg2, arg3, arg1);
```

i.e., the call string is generated based on the relative ordering of the parameters before sorting and there is no impact on the parameters of the actual function. Algorithm 6 details this mechanism.

4.2 Handling Pointers

Pointers introduce a complication for memoization. A pointer can be used to read memory (`READ` case), but the value stored at that location can change from one call to another. A pointer can also be used to write memory (`WRITE` case), essentially producing a return value. In either case, we are interested in the pointed-to value, not the address.

So, we have taken a conservative approach and handle only one `READ Only` or `READ/WRITE` pointer while allowing up to

---

Algorithm 5 Checking if function is Global Variable safe

Require: Function F
Ensure: If input function is safe from global variable usage for memoization
  function ISGLOBALSAFE(f)
    count ← 0
    global ← Null
    for each instruction I in f **do**
      for each operand O of I **do**
        if ISGLOBALVARIABLE(O) **then**   ▷ All global variables are pointers in LLVM
          ptype = GETTYPE(O)
          optype ← GETTYPE(ptype)
          if optype = Integer OR optype = float OR optype = Double **then**
            if global = Null **then**
              global ← O
              continue
            end if
            if global = O **then**   ▷ Only 1 global variable is considered to be useful for memoization
              continue
            end if
            return false;
          end if
          return false
        end if
      end for
    end for
  end function

Algorithm 6 Replacing Memoizable Function Calls

Require: Function F
Ensure: Calls to Input function is replaced by calls to its memoized version
  globals ← GETGLOBALS(F)
  proto ← GETPROTOTYPE(F)
  SORT(proto)
  for each use cs of F **do**
    if ISCALLSITE(cs) **then**
      args ← GETARGS(cs)
      ARGS.ADD(globals)
      argsnew←SORT(args)
      for each argument x in args **do**
        if isPresent **then**globals,x
          continue
        end if
        index← ARGSNEW.FINDI(x)
        if ISCONSTANT(x) **then**
          PROTO.REMOVEFROMINDEX(index)
          ARGSNEW.REMOVEFROMINDEX(index)
          CALLSTRING.ADD(x.value)
        end if
      end for
      if HASUSE(F) **then**
        newF ← MAKENEWFUNCTION(F.returntype, argsnew,F.parent)
        REPLACEALLUSESWITH(F, newF)
        PRINTTOFILE(callString)
      end if
    end if
  end for

two `WRITE Only` pointers. Also we only handle cases where the pointed value is of type `int`, `float` or `double` – arrays, structures and pointer to pointers are skipped and not considered for memoization.

The `READ/WRITE` property of a parameter is analyzed by going through the use chain of the parameter and analyzing each instruction. If only `STORE` instructions use a parameter it is marked as `WRITE Only`. If no instruction with memory write capability uses a parameter, it is marked as `READ Only`. Otherwise, the parameter is marked as `READ/WRITE`.

4.2.1 Read Only Pointers

`READ Only` pointers are those which are used in a function but the pointed value is not modified. Hence, we handle them like a normal argument. The pointed value is used for indexing the memoization table and also as tag.

4.2.2 Write Only Pointers

These parameters are not used for indexing into the memoization table, but only for passing back the return value. They require storage space in the memoization table, but cause no direct overhead for indexing into the table. A typical case for this is the *libm* function *sincos*, where two `WRITE Only` pointers are used for returning simultaneously the sin and cos values for the input argument.

4.2.3 Read/Write pointers

`READ/WRITE` pointers are the most difficult to handle. They require both storage space and also cause extra lookup overhead, also reducing the chances of repetition.

4.3 Global Variables

Global variable usage can make an otherwise pure function impure because they can change across function invocation and thus affect the result of the function call (when it is used in function computation). Global variables can also be used to store a state of the program by the function. We handle global variable like a normal argument – in our memoized wrapper we add one extra argument for each global variable in use in the function. Global variables have *pointer* type in LLVM and hence our technique for handling pointers is re-used. We have considered a maximum of one global variable use and correspondingly only one extra parameter in the memoized wrapper. Global variables are not considered while generating the call string and hence this has no effect in the call to the original function. The algorithm for detecting global variables is given in Algorithm 5 and the procedure for adding it in the memoized wrapper function is given in Algorithm 6.

4.4 Constants

There are cases where one or more parameters to a function call might be constants. In such cases, it is unnecessary to consider these as regular parameters since, by definition, they are guaranteed to repeat and hence no need to be saved in the memoized table. Further, we can avoid passing these constants to the actual call, as long as there is a hit in the memoization table. This is made possible by passing the constants directly to the call string and omitting them from the memoization wrapper. The disadvantage of this approach is that the memoization wrapper becomes call site specific (a separate memoization table is needed per constant, though different call sites using the same constant use the same table) but our results show that this is not a problem. Refer to Section 5.3.2 for an example in the `histo` benchmark.

4.5 Memoization Table and Indexing

For comparison purposes, we use the same memoization table implementation as Suresh et al. [27] where a direct mapped table is being used, indexed using XOR operator on the arguments (repeated folding of argument bits to get the required number of bits for index) and tagged using the arguments.

5. Experimental Results

5.1 System Configuration

We carry out our experiments on an Intel Core i7 Ivy Bridge clocked at 2.3 GHz, equipped with 8 GB of RAM and 8 MB of L3 cache, and running Linux 3.19. Benchmarks are compiled at the `-O3` optimization level. We use LLVM version 3.19. We made sure that Turbo Boost is turned off and CPU clock frequency was set at 2.3 GHz to ensure reproducible measurements. For our experiments we have used a 64 k-entry memoization table per function. This size is empirically determined to be the best on our experimental machine.

5.2 Benchmarks

Based on our criteria for memoization – pure critical functions, we selected a variety of applications both from real life as well as standard benchmark suites that extensively call transcendental functions. Applications with no, or few calls to these functions are not impacted (neither speedup nor slowdown), they are not reported here. Our experimental benchmark applications include:

`SPEC CPU 2006`. We chose five benchmarks: `bwaves`, `povray`, GemsFDTD, `tonto` and `wrf` from `SPEC CPU 2006` suite [15] with *ref* inputs, which have a reasonable number of calls to transcendental functions. `wrf` and `bwaves` extensively use *powf* and *pow* functions respectively. `tonto` makes extensive use of *exp* and *sincos* calls, while `povray` has many calls to *sin* and *cos*.

`SPEC OMP 2001` [3]. Two applications match the criteria for memoization – `gafort` and `equake`. `gafort` has calls to *sin* in its critical region and `equake` has both *sin* as well as *cos* calls in its critical region.

`ATMI` [20] is a C library for modeling steady-state and time-varying temperature in microprocessors. It is provided with a set of examples and all of them have a large number of calls to Bessel functions *j0* and *j1*.

`Population Dynamics` [8] is a model of aphid population dynamics at the scale of a whole country. It is based on convection-diffusion-reaction equations. The reaction function makes heavy use of *exp* and *log* and uses armadillo library.

`Barsky` [5] is a partial differential equation solver, it contains many calls to *sin*.

`Splash2x`. `water_spatial` [14] of `Parsec` benchmark suite [6] can benefit from memoization as it makes extensive use of *exp* and *sqrt* functions.

`blacksholes` [7] is also taken from `Parsec` [6] benchmark suite. This involves option pricing with Black-Scholes partial differential equation. Black-Scholes equation does not have a closed form solution and hence must be computed numerically.

`histo` is a histogram application taken from Parboil benchmark suite [26] which accumulates the number of occurrences of each output value in the input set.

5.3 Discussion of Results

The result of memoization on the selected benchmark applications is shown in Table 1. We also reproduced the load-time approach [27] for comparison purposes. We obtained access to their code to guarantee fairness of comparison. The speedups are shown in Figures 4 and 5.

First of all, thanks to the inlining of the wrapper, more (math) library functions benefit from memoization, compared to the plain load-time approach.

Compile-time memoization delivers much better speedup compared to the load-time approach. A major part of this is coming from the memoization wrapper getting inlined, thus providing more optimization opportunities. This cannot be achieved by the dynamically linker.

We are also able to catch more memoization opportunities as marked by boxed functions in Table 1. Note that `histo` and `blackscholes` do not benefit from load-time memoization (speedup 1.0). The case of `equake` is detailed below in Section 5.3.1.

In `equake` we capture *phi0*, *phi1* and *phi2* functions where a global variable is being used. With our scheme, we capture them as extra argument and hence these functions are getting memoized. In `histo`, HSVtoRGB is a user function taking in three arguments of type `float` and returning a structure to three variables of type `char`, as shown below. But this is converted to an `int` return type by LLVM as per the x64 Linux ABI [12] and in the call site we avoid the constants and replace the call ignoring the constants to the memoization wrapper. Now in the call to the original function inside our memoization wrapper we include these constants as arguments.

```
// Source Code
typedef struct {
  unsigned char B;
  unsigned char G;
  unsigned char R;
} RGB;
RGB HSVtoRGB(float h, float s, float v)

// LLVM IR
define i32 @HSVtoRGB(float %h, float %s, float %v)
```

5.3.1 equake

In `equake`, we have three critical user defined functions – *phi0*, *phi1* and *phi2* – all of which have similar code structures. The code of *phi0* is given below:

```
double phi0(double t) {
  if (t <= Exc.t0) {
    return value = 0.5 / PI * (2.0 * PI * t / Exc.t0
          - sin(2.0 * PI * t / Exc.t0));
  }
  else
    return 1.0;
}
```

There is only one input argument, and the global variable `Exc.t0` is read inside the function. So, we include the global variable as an extra argument and generate our memoization wrapper as follows:

```
double _memoized_phi0(double arg1, double* arg2)
```

The second argument is used for passing the global variable, and global variables have pointer type in `LLVM IR`. Now, this function provides a challenging case for memoization. There is just an *if-then-else* block inside the function with the *then* block being expensive and the *else* block being very simple (just a return statement). During execution, we found that the *else* block has taken

90 % of the total time – 284 million calls to *then* block compared to a total number of calls of 3.7 billion – and hence memoization is getting a benefit of only 10 % of the time, causing slight overhead during every execution of *else* case. So, this causes the speedup to come down. In order to avoid this, we tried applying memoization only for the *then* part of the function, resulting in a performance gain of 3.5 % over the normal compile-time approach, showing a further enhancement which could be applied for our approach.

The *phi* functions could not be memoized by the load-time approach, but the *sin* could. This explains why load-time produces a speedup.

5.3.2 histo

In `histo`, we have a call to `HSVtoRGB`:

```
HSVtoRGB(0.0, 1.0,
        cbrt(1 + 63.0 * ((float)value)/((float)UINT8_MAX))/4);
```

The first two arguments are constants, hence removed. We modify the call site to `__memoized_HSVtoRGB_0_1(double)` keeping a single argument:

```
__memoized_HSVtoRGB_0_1(cbrt(1+
        63.0 * ((float)value)/((float)UINT8_MAX))/4);
```

and we call `HSVtoRGB` from inside `__memoized_HSVtoRGB_0_1` as follows:

```
HSVtoRGB(0.0, 1.0, arg1)
```

thus we have saved the passing of first two parameters and avoided their use in indexing the memoization table.

5.3.3 blackscholes

In `blackscholes`, the function *CNDF* (Cumulative Normal Distribution Function) performs numerical computation on an input floating point data and is expensive as well as critical. The function is using a macro defined type to handle both double as well as float types as per definition during compile-time. Our experiments were run using double type. The code involves numerous floating-point computations in addition to a call to *exp* function making it expensive.

5.3.4 wrf

In `wrf` we observe a performance loss due to a low repetition rate for *powf*. The performance achieved by Suresh et al. is reported to be caused by a performance bug in the *libm* implementation of *powf*. This bug is partially fixed in the version of *libm* we used for our current experiments. We discuss in Section 5.4 a turn-off mechanism to disable memoization in case of slowdowns.

5.3.5 water_spatial

Our compile-time approach produces a significant speedup of 8 % in `water_spatial`. This is however less than the load-time approach. On a machine with a larger instruction cache size (64 KB vs 32 KB for our experimental machine), compile-time is slightly better than load-time, as for other benchmarks. We hence attribute the slowdown to an increase in miss rate for L1 instruction cache caused by inlining which increases the code size from 36 KB to 38 KB.

5.3.6 Transcendental Functions

For transcendental functions in all other applications, we obtain better performance than the load-time approach, (shown in Figures 4 and 5). This increase in speedup comes from saving a call in case

Table 1. Memoization results. Boxed names indicate functions that cannot be caught by a load-time technique. The first two columns report benchmarks names and memoizable functions. For each function, we report the number of calls. The following two columns report the total execution time, and fraction of the time spent in the memoizable functions. The last three groups report results. First, we reproduce Suresh et al.'s results with a load-time approach. We then cover the pure compile-time approach, reporting the percentage of lookup hits per function and the overall speedup. Finally, we describe the results of the hardware-assisted memoization, showing percentage of hits in the hardware table, misprediction rates, and overall speedup.

Applications	Memoized Functions	Num calls	Exec. Time (s)	Mem. Fraction (%)	Speedup load-time cf. [27]	S/w hits (%)	Speedup compile cf. Section 4	H/w hits (%)	Mispred. (%) cf. Section 6	Speedup hardware
bwaves	pow	216M	550.0	39.2	1.14	60.1	1.29	51.8	28.8	1.25
	sqrt	54M				43.3		35.8	23.2	
gems.FDTD	exp	711M	495.0	6.7	1.04	99.9	1.04	99.7	0.3	1.07
equake	phi0	3.7G	227.7	96.1	1.07	100.0	1.08	100.0	0.0	1.16
	phi1	3.7G				100.0		99.9	0.0	
	phi2	3.7G				100.0		99.9	0.0	
tonto	exp	569M	777.0	35.3	1.25	80.02	1.26	13.8	9.5	1.31
	sincos	452M				100		99.7	0.3	
	sqrt	1.3G				73.2		15.8	13.8	
wrf	log10f	33M	467.0	37.6	1.0	16	0.99	15.9	2.2	1.04
	logf	80M				45		22.0	15.3	
	powf	1.5G				14.7		11.2	4.9	
povray	sin	5.2M	203.0	6.1	1.0	0.0	1.0	0.0	0.0	1.01
	sqrt	212.8M				13.6		12.5	0.1	
	pow	30.7M				23.7		14	0.0	
gafort	gfortran_pow	36G	1653.0	60.4	1.05	100.0	1.20	100.0	0.0	2.82
	exp	2.2G				97.2		53.4	14.8	
	sin	2.2G				97.1		54.8	14.1	
barsky	sin	26M	21.2	6.1	1.03	97.7	1.04	96.1	3.9	1.06
ATMI	exp	140M	36.0	99.1	1.06	71.2	1.06	19.2	5.4	1.31
	j0	69M				20.3		4.9	4.9	
	j1	135M				84.4		40.1	15.6	
	log	127M				77.4		47.0	15.0	
	sqrt	35M				54.2		20.7	9.8	
Population dynamics	exp	28M	8.8	50.6	1.20	100.0	1.24	100.0	0.0	1.96
	log	28M				100.0		100.0	0.0	
histo	HSVtoRGB	5M	4.7	13.2	1.0	100.0	1.12	100.0	0.0	1.48
blackscholes	CNDF	2G	151.0	51.4	1.0	94.1	1.28	76.2	20.6	1.39
water spatial	acos	5M	245.0	47.4	1.13	77.2	1.08	73.5	4.0	1.40
	exp	1G				98.2		72.2	21.8	
	pow	9M				98.5		98.5	0.2	
	sqrt	1.9G				96.7		66.3	25.0	

of hit in the lookup table (Section 3) as well as potential further compiler optimization after inlining.

5.4 Turning off Memoization

With a load-time scheme, turning off memoization amount to overwriting the address of a function in the process' GOT (global offset table). In our compile-time scheme, the wrapper is inlined in the caller and its code is likely to be optimized by the compiler and mixed up with the surrounding code, making it impossible to disable in a simple manner. Instead, we can keep an original (i.e. without memoization) version of the *caller* of the memoized function, and substitute versions at run-time whenever memoization proves to be counterproductive. As in the load-time case, the monitoring is performed by a lightweight helper thread that periodically checks statistics and makes decisions based on repetition rates and execution times. Substituting code at the granularity of a function is

easily accomplished by dynamic binary optimizers, such as DynamoRIO [4] or Padrone [24]. To avoid the impact on code size, the backup versions can be stored in a special section of the ELF file, possibly compressed, to be retrieved only as needed.

5.5 Memory Overhead

In our experiments, we used a 64 k-entry table size for each function. This requires 64 tag bits and 64 data bits for a function taking a `double` argument and returning a `double`. It amounts to $128 \times 64k = 8$ Mbits = 1 MB memory per function. For functions taking two `double` arguments, we need 1.5 MB. And for 32-bit arguments (i.e. `int` or `float`) this corresponds to 512 KB and 768 KB respectively.

Note that these tables are one-time allocated and initialized, but they cause contention in the memory hierarchy only when their content is actively used (hence useful). The initial overhead consists

Figure 3. Code size change due to Memoization
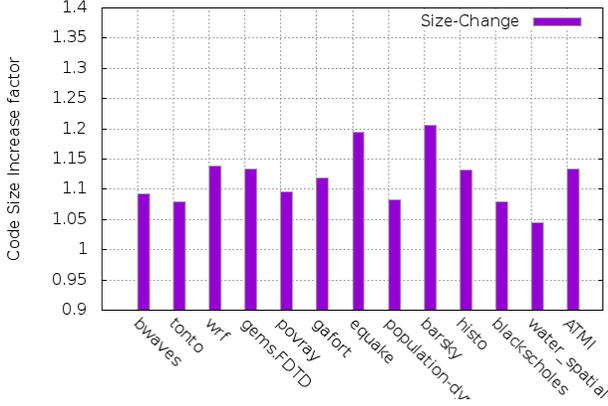
in calling `malloc` and writing the data. It is in the order of tens of microseconds per table. As shown in the results, there are a few tables per benchmark.

In addition to memory overhead for the memoization table, inlining can also cause an increase in code size. Figure 3 shows the percentage change in the code size (`text` segment) due to memoization on our selected applications. The variabtion really depends on the number of memoized call sites in a function and the size of the function. Note that inlining is a decision taken by LLVM based on its standard heuristics. We have not tried to change them.

6. Direct Hardware Memoization

This section evaluates a proposition to implement memoization directly through the hardware. Such a hardware implementation requires some ISA modification described in Section 6.1 and the use of an extra hardware table described in Section 6.3. Hardware can enable very fast hit/miss time, hence improving the profitability of memoization.

The software implementation evaluated in the previous sections necessitates to compute hash function, then to read the table stored in the global memory and finally to check the tags. When the table read misses through the whole memory hierarchy, the total access time can reach hundred cycles. If dedicated hardware table is used for memoization then one can get fixed hit/miss time on the memoization table including hashing the index, effective table reads and tag check. Moreover using a set-associative hardware memoization table can be considered since this would only marginally increase the hit/miss time.

The compile-time approach developed in the previous sections is adapted to exploit this hardware memoization.

6.1 ISA Support for Hardware Memoization

Hardware memoization was discussed by Citron et al. [10]. For hardware memoization of user defined functions they proposed two new instructions LUPM and UPDM which respectively does the memoization table lookup and memoization table update. Both instructions had two variants: one for single input functions and other for double input functions. We consider a very similar ISA support for hardware memoization that fits our compile-time approach. Our proposal has a new `CALL` instruction, specific to each function prototype supported by our compile-time approach.

We introduce the following two instructions.

MSCALL is used to call a memoized function. On such a call, the hardware memoization table is read using a hardwired hash function and on a success returns the value from the table. If the lookup fails, a normal `CALL` instruction is invoked to execute the original function. We assume different variants of `MSCALL` to handle the different possible function templates. For pointer arguments, we assume that they are `WRITE Only` since `READ ONLY` pointers can be converted to corresponding type using a `LOAD` instruction by the compiler. Similarly, `READ/WRITE` pointer arguments can be converted to two arguments: one being the data, and another `WRITE Only` pointer. Thus, we can restrict the required instruction variants as follows:

- 1 data argument, 1 return value;
- 2 data arguments, 1 return value;
- 1 data argument, 1 pointer argument, no return value;
- 1 data argument, 1 pointer argument, 1 return value;
- 1 data argument, 2 pointer arguments, no return value.

Moreover, to support multiple functions, the `MSCALL` instruction features an 8-bit function identifier.

MSUPDATE performs an update to the memoization table with the function result, therefore the compiler must allocate it immediately after `MSCALL`. In case of a hit of the `MSCALL` instruction in the memoization table, the `MSUPDATE` must be skipped to avoid a rewrite on the hardware memoization table. This is achieved by `MSCALL` incrementing the program counter.

`MSCALL` is tightly integrated with the branch predictor. At instruction fetch time, the hit/miss in the hardware memoization table is predicted so that the speculative execution engine is not impacted by our additions.

6.2 Benefit of Reduced Latency

For the sake of simplicity, in the remainder of the paper, we will assume that the overall lookup time t_{lo} for hardware memoization table is 5 cycles approximately corresponding to 1 cycle hash computation, 3 cycles table read, and 1 cycle tag check and the branch misprediction penalty t_{bmp} is 20 cycles. These values are consistent with current technology.

The overall execution cost of a call to a hardware memoized function depends on 1) hit/miss on the hardware memoization table, 2) correct/wrong branch prediction on the `MSCALL` instruction and 3) execution time of the normal call.

The average execution cost of a call to a memoized function is detailed in the following equation 2. Note that, if the memoization table misses and the branch prediction is correct, there is no significant execution overhead on a modern superscalar processor over the normal execution overhead. This explains why the term $(1 - H)$ is multiplied by t_f, not $t_f + t_{lo}$.

$$H \times t_{lo} + (1 - H) \times t_f + M_{pred} \times t_{bmp} \qquad (2)$$

M_{pred} is the misprediction ratio of the `MSCALL` branch.

A dynamic predictor built with a saturating counter (e.g. 4 bits) associated with each memoized function easily achieves a misprediction rate in the same range or lower than $min(H, 1-H)$. Therefore, taking H (respectively 1-H) as a proxy for M_{pred} when $H < 0.5$ (resp. $H > 0.5$), we can derive the condition when memoization is beneficial.

- if $H < 0.5$, we need $t_{lo} + t_{bmp} < T_f$, i.e the average execution time of the function is higher than 25 cycles.

- if $H > 0.5$, we need $\frac{H \times t_{lo} + (1-H) \times t_{bmp}}{H} < T_f$. As an example, if H=0.75, it is enough that $T_f > 12$ cycles.

As a result, memoization of any function with average execution time over 25 cycles is potentially beneficial even if its hit rate on the hardware memoization table is marginal. In particular, all the functions considered in the previous sections fit in this category.

6.3 Hardware Memoization Table

In this paper, we assume a hardware memoization table with approximately the same size as the L1 data cache. This table can be set-associative to allow higher hit rate. A single table is shared by all memoized functions.

Individually each entry has a valid bit, a process identifier field (ASID) to support multi-process environments, a function identifier field of 8 bits, two input arguments of 64 bits and two possible results also of 64 bits.

In the evaluation, we use a 256 sets, 4-way associative (1024 entries) table, with pseudo LRU replacement. An entry corresponds to 2 8-byte results, 2 8-byte tags, one 8-bit function identifier, an ASID, a valid tag and a few tag bits for replacement. The total storage is around 32 KB, i.e., approximately on-par with the L1D cache.

6.4 Software Support

Function memoization necessitates software support. For our experiments, we use the same scheme as for software memoization. At link time, we no longer need to link to the memoized wrapper functions. Instead, the LLVM compiler generates the `MSCALL` instruction variant as per the required prototype. Further, the functions must now be numbered with its corresponding identifier which are generated sequentially starting from 0. This identifier is passed as the last parameter to the `MSCALL` instruction. If any function is having more than two input parameters, the compiler enables only software memoization as described previously.

6.5 Performance Evaluation

Due to the novel hardware, evaluating this proposal relies on models and simulation. We used the same set of benchmarks and the same experimental setup as in our compile-time technique. The methodology is the following.

We first instrument all memoized functions to obtain their execution times. This is achieved with Intel's `RDTSCP` instruction that returns the value of the time-stamp counter (monotonically incremented at each clock cycle). We subtract the time spent in memoized functions from the total execution time. We then estimate the execution time of these functions in the presence of hardware memoization, thanks to Equation 2. We obtain the hit rate (H) and misprediction (M_{pred}) through simulation of the lookup table and 4-bit branch predictor (Table 1). Adding this time to the previous intermediate result produces our estimate execution time. Speedups are reported in Table 1. Figures 4 and 5 shows the comparative performance of hardware and software memoization approaches on our set of benchmarks.

As expected, substantial performance gains are encountered on most benchmarks, with up to 2.82× speedup for `gafort`, and 1.96× for `populationdynamics`. Both have perfect hit rates in their critical function, making efficient table lookup extremely effective.

The majority of benchmarks achieve speedups in the range 1.16× − 1.48×. These figures derive from the combination of several factors. `ATMI` spends practically all its time in memoized functions, but achieves only modest hit rates. `equake` has both excellent coverage and hit rates, unfortunately the *phi* functions have short execution time, thus reducing the impact of memoization. In
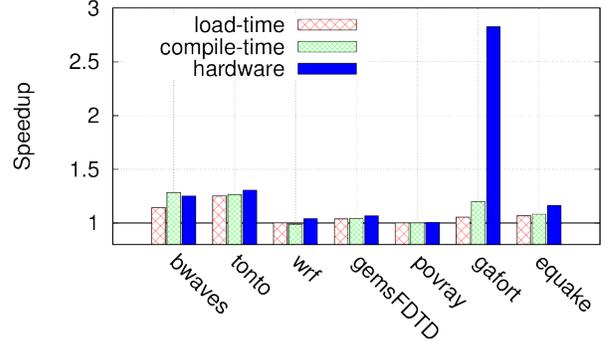

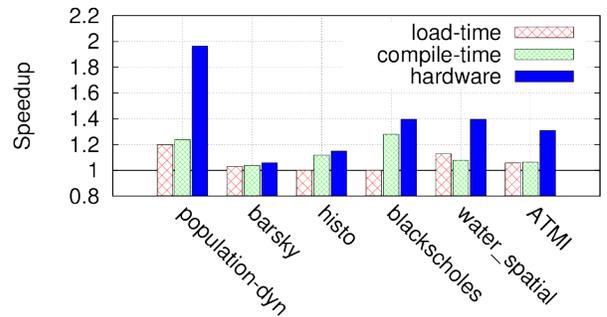**Figure 4.** Memoization speedups for SPEC

Figure 5. Memoization speedups for selected applications

the case of `histo`, only modest amount of time is spent in the user function *HSVtoRGB*, but the hit rate is perfect. We also benefit from two additional effects which explain the 1.48× speedup: two parameters are constant, hence optimized out by memoization, and the ABI convert the structure of char into an integer (see Section 5.3). `bwaves`, `tonto`, `blackscholes`, and `water spatial` combine fair amount of time spent in memoized functions and good hit rates, resulting in speedups in the upper range: 1.25× to 1.40×.

The least performing benchmarks, (speedups in the range 1.01× − 1.07×) include `barsky`, `gems.FDTD`, `povray` and `wrf`. Barsky and `gems.FDTD` have excellent hit rate, but suffer from insufficient time spent in the memoized functions, in the order of 6 %. Conversely, `wrf` spends a significant amount of time in memoized functions, but has bad hit rate, and to some extent, poor misprediction rates. `povray`, impacted by both factors, has the worst speedup: 1.01× only.

Only `bwaves` experiences a smaller speedup than with the software memoization. This is due to the relatively small hardware table considered in the experiments resulting in a reduced hit rate, combined with a high misprediction rate.

6.6 Energy Consideration

A hardware memoization table is not useful for every application. To avoid static energy consumption on applications that do not use it, power gating the table can be considered.

7. Summary and Further Extensions

In this work, we have first proposed a compile-time approach for memoization as compared to [27] where a pure link time approach was proposed. Doing memoization at compile-time captures more candidate functions for memoization and enables memoization opportunity for user defined as well as library functions. With the help of LLVM framework we are able to support pointers and global variables for the functions to be memoized. Simple *struct* objects can also be handled as they are converted to int/double type and registers are used to pass them up to size 128 bytes as per $x64$ Linux ABI [12]. Further we are able to optimize the function calls in case of constants being passed, by enabling function memoization per call site and passing constants directly to the original function call – constant parameters are avoided in the calls to memoized wrapper. The results of function memoization at compile-time is better compared to a load-time approach as shown in Figures 4 and 5. By making a memoization wrapper at compile-time we provide inlining facility for the wrapper which improves the performance by saving the cost of a call as well as by enabling other compiler optimization. Inlining causes a slight increase in code size but still is good for performance. We used a $64k$ entry table for memoization but these entries are used only when indexing happens to them and hence in case of function being non-critical we would not be losing any performance.

As a second contribution, we showed that hardware memoization is also a valid approach. It necessitates minor additions in the ISA and further increases the scope and efficiency of memoization. This offers good benefit as it lowers the profitable minimum function execution time

Acknowledgments

References

[1] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE TC*, 54(7):922–927, July 2005.

[2] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. In *ISCA*, June 2014.

[3] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *Workshop on OpenMP Applications and Tools*, 2001.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI*, 2000.

[5] Sandra Barsky. This is a program to solve nonlinear 2-D PDE using one-step linearization.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, 2008.

[8] M. Ciss, N. Parisey, F. Moreau, Ch.-A. Dedryver, and J.-S. Pierre. A spatiotemporal model for predicting grain aphid population dynamics and optimizing insecticide sprays at the scale of continental France. *Environmental Science and Pollution Research*, 2013.

[9] Daniel Citron, Dror Feitelson, and Larry Rudolph. Accelerating multimedia processing by implementing memoing in multiplication and division units. In *ASPLOS*, 1998.

[10] Daniel Citron and Dror G. Feitelson. Hardware memoization of mathematical and trigonometric functions. Technical report, 2000.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[12] Agner Fox. Calling conventions, 2015.

[13] Antonio González, Jordi Tubella, and Carlos Molina. Trace-level reuse. In *ICPP*, pages 30–37, 1999.

[14] PARSEC Group et al. A memo on exploration of SPLASH-2 input sets. *Princeton Univ*, 2011.

[15] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[16] Jian Huang and David J Lilja. Exploiting basic block value locality with block reuse. In *HPCA*, 1999.

[17] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[18] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong. Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors. In *MICRO*, 2010.

[19] Paul McNamee and Marty Hall. Developing a tool for memoizing functions in C++. *SIGPLAN Not.*, 33(8):17–22, 1998.

[20] Pierre Michaud, André Seznec, Damien Fetis, Yiannakis Sazeides, and Theofanis Constantinou. A study of thread migration in temperature-constrained multicores. *TACO*, 4(2), June 2007.

[21] Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[22] Uday Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Conf. on LISP and Functional Programming*, 1988.

[23] Stephen E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical report, 1992.

[24] E. Riou, E. Rohou, Ph. Clauss, N. Hallou, and A. Ketterlin. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. In *Intl Workshop on Dynamic Compilation Everywhere*, 2014.

[25] Hugo Rito and João Cachopo. Memoization of methods using software transactional memory to track internal state dependencies. In *PPPJ*, 2010.

[26] John Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[27] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. Intercepting functions for memoization: A case study using transcendental functions. *TACO*, 12(2), June 2015.

[28] James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. Softsig: Software-exposed hardware signatures for code analysis and optimization. In *ASPLOS*, 2008.