Intercepting Functions for Memoization: A Case Study Using Transcendental Functions

Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, André Seznec

Intercepting Functions for Memoization- A case study using transcendental functions

Arjun Suresh, INRIA/IRISA
Bharath Narasimha Swamy, INRIA/IRISA
Erven Rohou, INRIA/IRISA
André Seznec, INRIA/IRISA

Memoization is the technique of saving result of executions so that future executions can be omitted when the input set repeats. Memoization has been proposed in previous literature at the instruction level, basic block level and function level using hardware as well as pure software level approaches including changes to programming language.

In this paper, we focus on software memoization for procedural languages like C and Fortran, at the granularity of a function. We propose a simple linker based technique for enabling software memoization of any dynamically linked pure function by function interception and we illustrate our framework using a set of computationally expensive pure functions – the transcendental functions. The transcendental functions are those which cannot be expressed in terms of a finite sequence of algebraic operations (like trigonometric functions, exponential function etc.) and hence are computationally expensive.

Our technique does not need the availability of source code and thus can even be applied to commercial applications as well as applications with legacy codes. As far as users are concerned, enabling memoization is as simple as setting an environment variable. Our framework does not make any specific assumptions about the underlying architecture or compiler tool-chains, and can work with a variety of current architectures. We present experimental results for x86-64 platform using both gcc and icc compiler tool-chains, and for ARM cortex-A9 platform using gcc. Our experiments include a mix of real world programs and standard benchmark suites: SPEC and Splash2x. On standard benchmark applications that extensively call the transcendental functions we report memoization benefits of upto 50% on Intel Ivy bridge and upto 10% on ARM Cortex-A9.

Memoization was also able to regain a performance loss of 76% in *bwaves* due to a known performance bug in the GNU implementation of *pow* function. The same benchmark on ARM Cortex-A9 benefited by more than 200%.

Categories and Subject Descriptors: C.2.2 [**Programming languages**]: Optimization

General Terms: Memoization, Transcendental Functions

Additional Key Words and Phrases: Link time optimization, function interception

ACM Reference Format:
Arjun Suresh., Bharath Narasimha Swamy., Erven Rohou. and André Seznec., 2015, Intercepting Functions for Memoization- A case study using transcendental functions. *ACM* 0, 0, Article 0 (June 2015), 23 pages.

1. INTRODUCTION

Memoization is a well known technique to improve program execution time by exploiting the redundancy in program execution [Connors and Hwu 1999; Richardson 1992]. The main idea is to save the result of execution of a section of program so that future execution of the same section with same input set can benefit from the saved result.

In memoization schemes, two conditions are important:

(1) memoized code should not cause any side-effects;
(2) memoization should always produce the same result for the same input.

At function level by applying these two conditions, pure functions can be defined as:

This work was partially supported by the European Research Council Advanced Grant DAL No 267175.
Author's addresses: A. Suresh, B. N. Swamy, E. Rohou and A. Seznec, INRIA/IRISA, Campus de Beaulieu, 35042 Rennes, France.

Pure Function. A pure function always returns the same result for the same input set and does not modify the global state of the program

Implementing memoization in software at function level is relatively easy in pure functional programming languages, because procedures have no side-effects and the parameters to the procedure completely determine the result of procedure computation. A hit in the memoization table can be used to replace the execution of the procedure, and yet semantic correctness is maintained.

Correspondingly, for procedural languages there are two reasons why plain memoization can lead to incorrect program execution. The result of procedure execution can depend on the internal state of execution, such as when the procedure references a pointer or reads a global value. Also, procedures may have side effects such as I/O activity or an update to the internal program state through a memory write.

Consequently, memoization efforts in procedural languages are either to be targeted at pure functions, or special techniques are needed to handle procedures with side effects [Rito and Cachopo 2010; Tuck et al. 2008]. In this work we focus on memoization opportunities in software at function level for procedural languages such as C, Fortran, etc. and target a class of commonly occurring pure functions: the transcendental functions. We present a simple, yet low over-head scheme to memoize calls to transcendental functions by using function interception. Our proposal has the following characteristics.

(1) Dynamic linking: deployed at library procedure interfaces without need of source code, that allows approach to be applied to legacy codes and commercial applications.
(2) Independent: optimization potential to multiple software language and hardware processor targets
(3) Run-time monitoring: optimization mechanism for tracking effectiveness of memoization within a specific system to only deploy profitable instances of memoization
(4) Environmental controls: framework allows flexible deployment that can be enabled and tailored for individual application invocations.
(5) Simple: our framework is simple to enable from a user perspective, and requires just a change to an environment variable for employing.

2. MOTIVATION

Our memoization framework is meant for any pure function which is dynamically linked, however for the purpose of this study we have chosen long latency transcendental functions to demonstrate the benefits of memoization. We present our analysis of the suitability of transcendental functions for memoization, and our methodology to select target functions to memoize.

Transcendental functions satisfy the two criteria needed for a function to be safely memoized. They do not modify global program state for normal inputs (boundary cases are discussed in Section 4.4), and the result of the computation is entirely dependent on the input arguments to the function. Thus doing a lookup for the memoized result will result in the same output as the one with actual execution, and the execution result can be safely substituted without loss of program correctness.

While transcendental functions satisfy the safeness criteria, memoization can be productive and deliver an overall savings in execution time only when the following conditions are satisfied:

(1) arguments of the target function have sufficient repeatability, and their locality can be captured using a reasonably sized look-up table;
(2) total time of table look-up is shorter than a repeat execution of the target transcendental function.

2.1. Repeatability of arguments in transcendental functions

Repeatability in arguments is key to benefit from memoization. We chose a set of commonly occurring computationally expensive transcendental functions from our benchmark set and analysed them for repeatability. From our profile analysis, we find that arguments exhibit three types of repetition behaviour.

(1) There are only a limited number of unique arguments. This is just the behavioural characteristic of the application. Our programs were compiled with the `-O3` flag, and we hypothesize that the redundancy in arguments values that we observe is largely due to redundancy in input that could not be caught by static compiler optimizations such as loop invariant code motion. In this case memoization can be expected to catch most of the repetitive calls even with a small look-up table.

(2) There are a large number of unique arguments but they do exhibit repetitive behaviour. Repetitive behaviour does not necessarily mean arguments are repeating at regular intervals of time. In regular intervals, arguments are coming only from a small range of values as shown in Figure 1 for one of our selected application for memoization-ATMI_Goh for the Bessel function *j0*. In the figure, we can see that there is a pattern getting repeated at a frequency of around 700k calls which is quite a high number for memoization table size. But even inside this regular interval, there are quite a few patterns that get repeated and more of them can be captured with an increased table size. Figure 1 b shows the cumulative count of the number of calls to the *j0* function versus the number of unique arguments considered, in the decreasing order of the frequency of their repetition (without considering their actual call order). We used a sliding window for our memoization table (sliding window of size n means at any instant the previous n unique values are in the table) to study the locality of arguments to *j0* in ATMI_Goh. The percentage captures of repeated arguments for various sliding window sizes are as shown in Table I. There were a total of 21.9 million calls to *j0* and as expected, the percentage capture increases with an increase in the sliding window

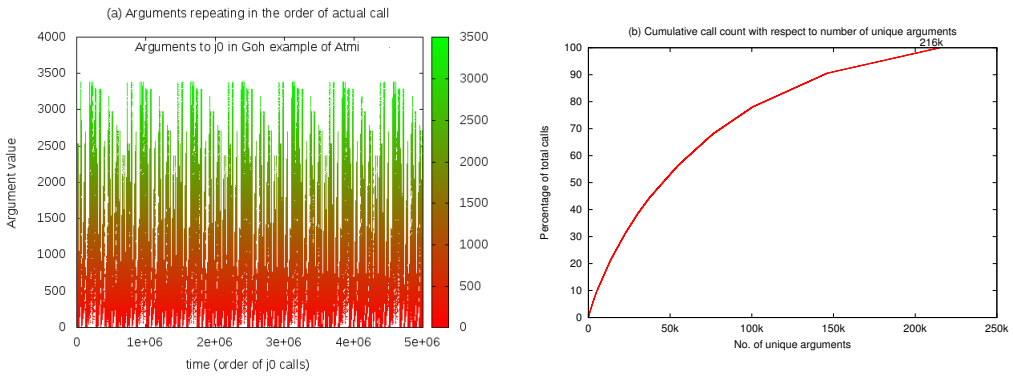

Fig. 1. Repetitive behaviour of arguments to j0 in ATMI_Goh l

Table I. Repetition of j0 args in ATMI captured with various sliding window sizes

Size of sliding window	Captures
4k	8%
16k	13%
64k	28%
256k	100%

size. Out of this 21.9M calls, less than 216k calls were actually having unique arguments (Table V), which means with a sliding window of 216k size we could capture 100% of repetitions.

(3) There are a large number of unique values for arguments. This is a difficult case for memoization. Fortunately this kind of behaviour is very rare among real applications.

2.2. Potential Latency Savings from Memoization

Transcendental functions are computationally intensive, with many functions requiring several hundreds of cycles to execute. A successful look-up in the memoization table can avoid expensive compute cycles and speed up execution time of the application. However the potential for latency saving has to be traded-off against the overheads of memoization.

An analysis to estimate the potential benefits of memoizing transcendental functions is given below. Let T_f be the execution time of the memoized function, t_h be the time when there is a hit in the memoized table and t_{mo} be the overhead of memoization when there is a miss in the memoized table. We derive an expression for the fraction of calls that should have arguments repeated (H), for memoization to be effective.

$$\begin{aligned} H \times t_h + (1-H)(T_f + t_{mo}) &< T_f \\ H \times t_h + (X + t_{mo}) - H(X + t_{mo}) &< T_f \\ t_{mo} &< H(T_f + t_{mo} - t_h) \\ \implies H &> \frac{t_{mo}}{T_f + t_{mo} - t_h} \end{aligned}$$

We designed our memoization framework for fast table look-up time and low miss overhead. Table II presents an empirical analysis (using a handwritten micro benchmark) on the performance potential of memoization on a set of long latency transcendental functions that frequently occur in our benchmark applications. Results presented are for an Intel Ivy Bridge processor running at 2 GHz (fully described in Table III). Table hit time for memoization is measured to be approximately 30 nanoseconds and miss overhead is nearly 55 nanoseconds. Though we expect miss overhead to be the same as hit time, as both are executing the same instructions for table lookup, we attribute this time difference to L2 cache latency. While hit would mean that the same entry was accessed before and hence more likely to be in the cache, a miss would mean that the entry might not have been accessed before (and hence more likely not to be in cache). We find that even with a moderate hit rate in the look-up table, these functions have the potential to give performance benefit.

To summarize our analysis, we find that commonly occurring transcendental functions having an execution time of at least 100 clock cycles (about 50 nanoseconds) are good target for software memoization in our framework. In many cases these functions exhibit repetition in their input values and can benefit from memoization even for moderate hit rates in the look up table.

3. 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 the hardware level, there are previous works on memoization of instructions where a hardware lookup table avoids the repeated execution of instructions with the same operands. Citron et al. [Citron et al. 1998] have used the term memoing – which essentially is memoization – in their work for performing multi-cycle floating point calculations in a single cycle. They assumed a hardware table for memoization and showed that the technique works especially well for multi-media processing. The work of Connors et al. in 1999 [Connors and Hwu 1999], focused on the performance of memoization of trigonometric functions using hardware memoization. In his work [Richardson 1992], Richardson has proposed a hardware

Table II. Profitability analysis of memoizing transcendental functions – 2 GHz Ivy Bridge, GNU lib

	Function	Hit Time (ns) t_h	Miss Overhead (ns) t_{mo}	Avg. Time (ns) T_f	Repetition Needed H
double	exp	30	55	90	48%
	log			92	47%
	sin			110	41%
	cos			123	37%
	j0			395	13%
	j1			325	16%
	pow			180	27%
	sincos			236	21%
float	expf			60	66%
	logf			62	63%
	sinf			59	65%
	cosf			62	63%
	j0f			182	27%
	j1f			170	28%
	powf			190	26%
	sincosf			63	63%

cache called the *result cache* to implement memoization for a set of targeted arithmetic computations (multiply, divide, square root) directly in hardware, without compiler or programmer intervention. Then there is memoization at block level [Huang and Lilja 1999] and trace level [Da Costa et al. 2000][González et al. 1999] where the result of a portion of code, either the basic block or trace portion, is memoized so that if the input values used in that code are repeated, result is taken from the lookup table.

At the software level, memoization technique has been used in languages such as Haskell [1] and Perl [2] that support functional programming to save the execution time of functions at runtime. The presence of closure [Reddy 1988] in functional programming languages gives a ready-to-use mechanism for the programmer to write memoization code.

When memoization is used in procedural languages, special techniques are needed to handle procedures with side effects. Rito et al. [Rito and Cachopo 2010] use Software Transactional Memory (STM) for function memoization, including impure functions. When memoization is found to violate semantic correctness, STM is used to roll-back the program state to the previous correct state. Tuck et al. [Tuck et al. 2008] used software-exposed hardware signatures [3] to memoize functions with implicit arguments (such as a variable that the function reads from memory). They are able to track changes to implicit arguments, and safely use the memoized result when all explicit arguments to the function repeat, and no implicit arguments have changed since the last function invocation. McNamee and Hall [McNamee and Hall 1998] in their work shows how memoization can be applied to C++ using a language level tool.

An extension of simple memoization is used by Alvarez et al. [Alvarez et al. 2005] in their work of 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 same (instead of a single input, now we have a set of inputs which have the same result). This result in improved performance as well reduction in power consumption without a significant change in the quality of the output. A similar technique is also used by Esmaeilzadeh et al. [Esmaeilzadeh et al. 2012] in their work of using neural accelerators

[1] $http://www.haskell.org/haskellwiki/Memoization$
[2] $http://perldoc.perl.org/Memoize.html$
[3] a signature is a hardware register that can represent a set of addresses

to accelerate programs using 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 , memoization gave them very good results.

Hardware techniques for memoization typically capture short sequences of instructions with a small table, while purely software techniques require the programmer to write a custom implementation of the function for memoization. In this work, we demonstrate a framework that uses large memoization tables to capture longer intervals of repetition to benefit computationally intensive pure functions at the software level without programmer intervention. Previous approaches have been either at language level or at hardware level, while we are working at an intermediate level, thereby making it both language- and hardware-independent. Using tools like Pin [Luk et al. 2005] or DynamoRIO [Bruening et al. 2003], we can achieve memoization by function interception at binary level. But such binary level approaches have the following restrictions: is limited to supported architectures, a bit slower compared to native execution and is complex to try for a naive user. By using our link time approach for function interception we overcome these problems of binary level approaches and still get the benefit of memoization.

4. MEMOIZATION APPROACH

We implement memoization in a transparent manner by intercepting calls to the dynamically linked math library (libm). We leverage the library pre-loading feature in Linux (using environment variable `LD_PRELOAD`) to redirect calls to the targeted transcendental functions to a memoized implementation of those functions. The same mechanism exists in Solaris and Mac OS X (for the latter, the variable is `DYLD_INSERT_LIBRARIES`). Windows also support DLL injection, even though the mechanism is different.

In our technique there is no need for code modification or recompilation – memoization works with legacy binaries that are compiled to link dynamically with the math library as well as commercial applications. We now discuss the design of the look-up table, and provide details on the hash function we use and the overhead involved in memoization.

4.1. Intercepting Loader

An essential part of our implementation in memoization is to intercept the dynamic function calls. Transcendental function calls which are dynamically linked are resolved to the corresponding function in the dynamically loaded math library. Working of our interception of these calls is shown in Figure 2. We preload memoized version of the target transcendental functions using `LD_PRELOAD` environment variable to contain the path to our memoized library, so that memoized implementation takes precedence over standard math library implementation[4]. Thus during the first call to these functions, the address of the corresponding function in Global Offset Table (GOT) gets filled from our memoized library instead of getting filled from actual libm. For other functions, the call is resolved to the standard math library implementation and incurs no runtime overhead.

Algorithm 1 provides a pseudo-code listing our implementation of the memoized function. A fixed size direct mapped hash-table is used as our memoization table. First, a hash-key is generated using the hash function and a table lookup is performed to check for a hit in the table. On a hit, the cached result is read out of the table and returned as the result of the function. On a miss, the call is redirected to the function in the libm library and the result ($table[index][1]$) is also cached in the memoization table with the argument being the tag ($table[index][0]$). The layout of our table guarantees that both tag and value are in the same line of the L1 data cache. We also evaluate an associative implementation for our hash-table which is discussed in Section 5.4

[4]Enabling memoization is as simple as typing: `export LD_PRELOAD=/path/to/new/libm.so`.

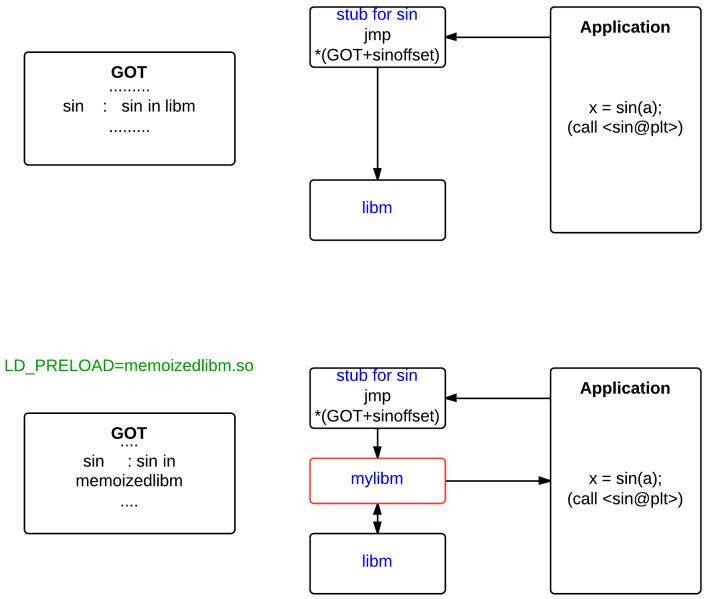

Fig. 2. Intercepting dynamic call to *sin*

ALGORITHM 1: Pseudo code for memoization

```
index = hash(argument)
if table[index][0] ≠ argument then
   table[index][0] = argument
   table[index][1] = sin_from_libm(argument)
end if
return table[index][1]
```

4.2. Hash Function

A fast hash function is needed since the evaluation of hash function is on the critical path to table look-up. We designed a simple hash function using the XOR function by repeatedly XORing the bits of the arguments to get the table index. Thus for a double-precision argument with 64 bits, we XOR the higher 32 bits with the lower 32 bits. The same procedure is repeated for the higher 16 bits of the result and the lower 16 bits, and obtain a 16-bit hash-key which we use directly to index into a hash-table of size 2^{16} entries. This mechanism is shown in Figure 3. For smaller sized tables we mask off the higher bits to get the required bits. For a function with multiple arguments, for example the *pow* function with two arguments, we first XOR the arguments and then repeat the same procedure as for functions with single argument.

The value of input arguments is stored in the table entry as a tag to the memoized function result. At the time of look-up, the stored tag values are checked against the incoming input arguments before using the result value. Collisions can occur when different input arguments hash to the same table entry. To evaluate the effectiveness of our hash function, we measured the number of collisions that occurred during the memoized run of our applications (Table V). The simple hash function we designed is fast to compute, as few as five instructions on x86 using xmm registers and the SSE SIMD extension. Yet, it results in a low percentage of collisions as compared to the number of unique values for input arguments.

4.3. Memoization Overhead

Memoization overhead includes the time needed to calculate the hash function and the time spent in table lookup. When table entries are located in a cache closer to the processor, less time is spent in reading out the table entries. On a miss, this additional overhead is added to the function execution time. For the hash function we designed, we experimentally measured the time needed for hash table lookups on an Intel Ivy Bridge processor clocked at 2 GHz (see also Table III for details). We measured successful table lookup time of around 60 clock cycles (approximately 30 nanoseconds) on our benchmark applications. During a miss in the table, table look-up overhead[5] was measured to be nearly 110 clock cycles (55 nanoseconds) which we attribute to the fact that during a miss the table entry is more likely not present in the caches closer to the processor. Both our hit time and miss time compares favourably with the 90–800 clock cycles average execution time of our target transcendental functions.

4.4. Error Handling

For normal inputs (inputs within the domain of the function) transcendental functions do not modify any global variables. But on boundary conditions they do set global error flags and throw exceptions. In order to ensure that program behaviour is not altered by memoization, we only memoize for non-boundary input values. For boundary cases which do set the global error states, results are not memoized. These cases occur for the following functions:

exp. Both overflow (`FE_OVERFLOW` exception is raised) and underflow (`FE_UNDERFLOW` is raised) may occur and in both cases `ERANGE` flag is set.
log. For log (x), `ERANGE` is set if x is 0 (`FE_DIVBYZERO` is raised) and `EDOM` is set if x is negative (`FE_INVALID` is raised)

[5]Miss overhead adds to the function execution time when there is a miss in the memoization table.

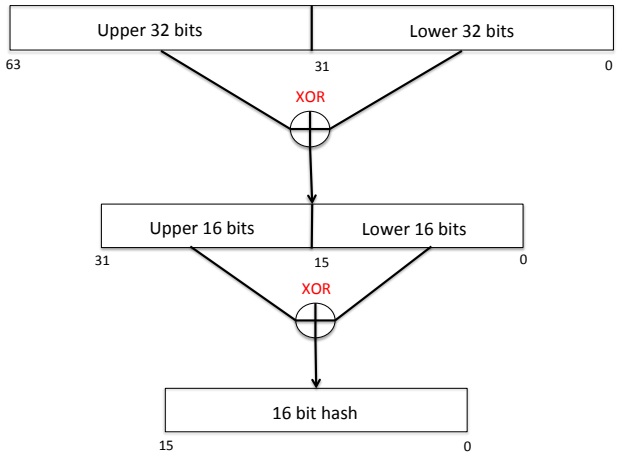

Fig. 3. Implementation of Hash Function

pow. For pow(x,y), `EDOM` is set if x is negative and y is a finite noninteger (`FE_INVALID` is raised). If x is 0 and y is negative `ERANGE` is set (`FE_DIVBYZERO` is raised). `ERANGE` is also set when the result underflows or overflows.

4.5. Rounding Modes

Math library has a method – `fesetround` – which can be used to specify the rounding mode to be used. If this method is called in the application, the stored memoized results are no longer valid. Thus, we have to intercept `fesetround` function and if there is any change in the rounding mode, we have to flush the contents of memoization table.

4.6. Handling Negative Effects

On applications that have a poor locality in the arguments to targeted transcendental functions, memoization may slow down execution time. Due to poor hit rate in the table, function call executions pay the table look-up overhead without benefiting from the use of stored result. To handle this problem, we have implemented a mechanism which can disable memoization in the event of bad effect.

We periodically monitor the number of calls and the number of hits in the memoized table. Once it is found that the hit rate is below the required threshold (as mentioned in Table II) and it is not increasing, we change the jump address in the Global Offset Table (GOT) to that of the original dynamically linked library entry so that future calls to the function goes directly to the original dynamically linked library and there is no interference from the memoized library for any of the future calls to that function. In our implementation, we chose to disable memoization once the number of hits in the memoized table falls below the required threshold. However, to account for different phases of program execution, memoization can also be re-enabled after some time by using a time triggered signal.

5. EXPERIMENTAL RESULTS

5.1. System Set-up

In order to analyse the performance of memoization across compilers and architectures we did our experiments on the following configurations:

(1) Intel Ivy Bridge (GNU compiler);
(2) Intel Ivy Bridge (Intel compiler);
(3) ARM Cortex-A9 (GNU compiler).

The characteristics of Ivy Bridge and Cortex-A9 are presented in Tables III and IV respectively. The Intel compiler also provides its own implementation of the math library.

Based on our criteria for memoization, we picked up 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 speed-up nor slowdown), they are not reported here.

For the other applications, memoization has no impact as there are very few calls to memoized libm from them. Our experimental benchmark applications include:

(1) SPEC CPU 2006
We chose six benchmarks: bwaves, gamess, povray, GemsFDTD, tonto and wrf from SPEC CPU 2006 benchmark suite [Henning 2006] with *ref* inputs, which have a reasonable number of calls to transcendental functions. *wrf* and *bwaves* are critical in *pow* function. *gamess*, and *tonto* are expensive in *exp* and *sincos* calls, while *povray* has many calls to *sin* and *cos*.
(2) SPEC OMP 2001
In SPEC OMP 2001 [Aslot et al. 2001], two applications match the criteria for memo-

ization – 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.

(3) ATMI
ATMI[Michaud et al. 2007] is a C library for modelling 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*.

(4) Population Dynamics
Population Dynamics [Ciss et al. 2013] 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 make heavy use of *exp* and *log* using the armadillo library.

(5) Barsky
Barsky [Barsky] is a Partial Differential Equation solver and contains many calls to *sin*.

(6) Splash2x
Three applications from Splash 2x of Parsec benchmark suite [Bienia et al. 2008] can benefit from memoization: fmm, ocean_cp, and water_spatial. fmm is critical in *log* function, ocean_cp in *sin* and water_spatial in *exp* functions.

5.2. System Configuration

5.2.1. Intel Ivy Bridge. We ran our experiments on an Intel i7 machine under the set-up as described in Table III. We made sure that the Turbo Boost feature [Intel Corporation] was turned off and CPU clock frequency was set at 2 GHz to ensure reproducible time measurements. We used GNU compiler+library as well as Intel compiler+library combinations and ran experiments using different table sizes. The results of the benchmark run under the GNU setup is illustrated in Figures 4 and 5. Figure 6 shows the results for the same configuration but the applications compiled using Intel compiler.

Table III. Experimental set-up on Intel Ivy Bridge

Processor	Intel Core i7
L3 cache	8 MB
Clock speed	2 GHz
RAM	8 GB
Linux version	3.11
gcc version	4.8
icc version	9
optimization flag	O3
libm version	2.2

5.2.2. ARM Cortex 9. We also ran our experiments on Pandaboard running ARM Cortex A9 processor under the set-up as described in Table IV. The results of the benchmark run under this set-up are given in Figure 9. We could not run splash 2x of PARSEC benchmark suite on ARM because this architecture is not supported. Also, due to memory limitations, all the SPEC benchmarks were run with train inputs on ARM while we were using ref inputs for Intel Ivybridge.

5.3. Discussion of Results

5.3.1. Intel Ivy Bridge (GNU compiler). Table V summarizes the characteristics of applications relevant to memoization, produced on Ivy Bridge with GNU compiler. The columns first report, for each benchmark, the involved transcendental functions, the number of calls to

Table IV. Experimental set-up on ARM Cortex

Processor	ARM Cortex A9
L2 cache	1 MB
Clock speed	1.2 GHz
RAM	1 GB
Linux version	3.11
gcc version	4.7
libm version	2.2

Table V. Function call analysis on Intel Ivy Bridge with GNU compiler and 64k table

Application	**Fun**	**No. of calls**	**Unique values**	**Hits**	**Evictions**	**Exec. Time (s)**	**Speed-up**	**Modelled Speed-up**
bwaves	pow	216,320,000	18,329,205	75.5%	24.5%	803	1.76	1.02
gamess	exp	1,066,610,840	24,761,242	48.2%	51.8%	1134	1.01	0.99
povray	sin	5,241,639	5,241,639	0.0%	98.7%	249	1.00	1.00
	pow	30,664,910	23,001,147	23.7%	76.0%			
gemsFDTD	exp	711,400,841	621,783	99.9%	0.1%	455	1.04	1.07
tonto	exp	569,927,519	4,767,144	80.0%	20.0%	1210	1.24	1.09
	sincos	453,108,244	305	100%	0%			
wrf	expf	331,317,492	10,144,648	14.3%	85.7%	584	1.02	0.96
	powf	1,675,704,575	80,555,848	23.5%	76.5%			
	logf	80,425,116	16,075,039	45.0%	54.9%			
equake	sin	568,120,502	252	100.0%	0.0%	275	1.09	1.25
	cos	284,060,252	251	100.0%	0.0%			
gafort	sin	2,200,000,000	32,768	97.1%	2.8%	2420	1.07	1.09
	exp	2,200,000,000	32,768	97.2%	2.9%			
ATMI	exp	129,606,540	8,826,650	84.9%	17.5%	60.02	1.27	1.20
	j0	53,110,575	3,357,377	23.1%	75.9%			
	j1	33,095,178	73,542	43.5%	55.0%			
	log	35,156,814	9,569,289	100.0%	0.0%			
	pow	530,721	445,140	14.0%	58.9%			
barsky	sin	26,148,922	455	97.7%	2.3%	23.4	1.03	1.07
population dynamics	exp	28,744,800	589	99.9%	0.0%	5.3	1.52	1.81
	log	28,744,800	589	99.9%	0.0%			
splash2x.fmm	log	37,717,281	78	99.9%	0.0%	147.7	1.02	1.01
	pow	9,109,011	9,101,294	0.0%	99.3%			
splash2x. ocean_cp	sin	16,785,411	4,097	94.5%	5.5%	123.4	1.00	1.01
splash2x. water_spatial	exp	1,039,992,980	12,538,545	98.2%	1.8%	241.2	1.16	1.20

each of them, and the number of unique values. The next two columns report, for a 64k-entry table, the number of hits and evictions due to collisions in the hash table. These two columns do not add up to exactly 100 %: the rest is due to cold misses, i.e. first access to a value. Actual original execution time (in seconds), without memoization is shown is the next column. Speed-up is defined as the ratio of the measured running times: original vs. memoized.

Finally, the modelled Speed-up is computed based on the collected statistics and the estimated function execution time (as shown in Table II), assuming an otherwise ideal scenario, as follows:

$$S = \frac{\text{Actual Runtime}}{\text{Modelled runtime}}$$

For example, considering gafort, the modelled Speed-up is:

$$\begin{aligned}\text{Modelled runtime} = \text{Total execution time} \\ &- (Ncalls_{sin} \times T_{sin} + Ncalls_{exp} \times T_{exp}) \\ &+ Nmisses_{sin} \times (T_{sin} + t_{mo}) + Nhits_{sin} \times t_h \\ &+ Nmisses_{exp} \times (T_{exp} + t_{mo}) + Nhits_{exp} \times t_h\end{aligned}$$

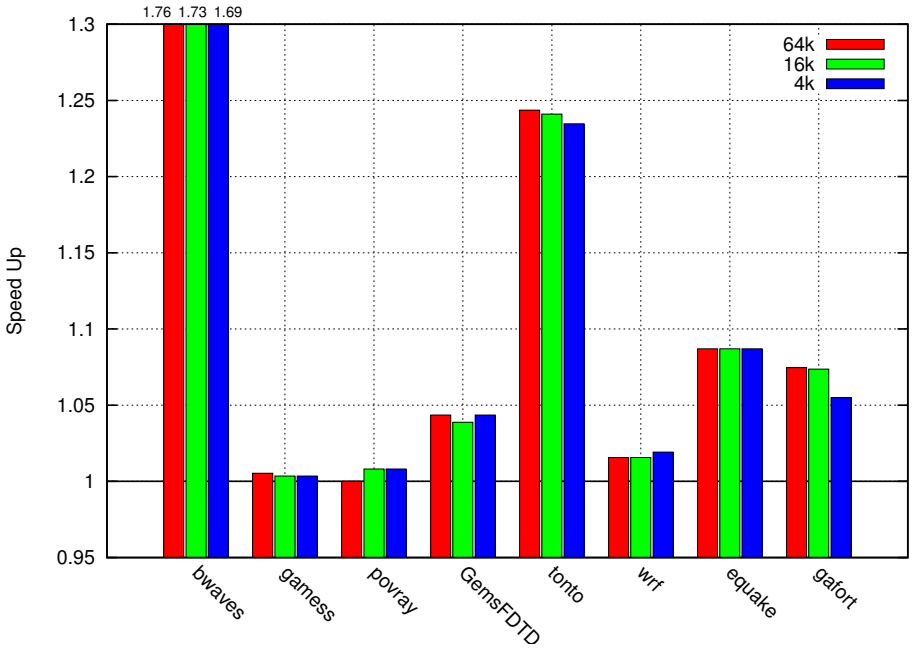

Fig. 4. Memoization of SPEC benchmarks on Intel Ivy Bridge (GNU compiler) for different table sizes

On Intel Ivy Bridge machine using the GNU compiler (Table III), SPEC CPU benchmarks gave benefit ranging between 1% and 24% (see Figure 4). Only bwaves produces an outstanding and unexpected speed-up of 1.76. The large benefit for memoization for bwaves is surprising considering pow on an average takes around 300 clock cycles only. We found out that this is due to a performance bug in the GNU libm for *pow* for some inputs[6] which causes a slow down even up to 10,000× compared to a normal call. In the case of bwaves this happens when m is very close to 1 and n is very close to 0.75 for m^n. For these input values, the current implementation is found to take more than 1000× the normal execution time. On memoization this long latency is saved and thus we get high benefit. To a smaller extent, this performance bug (in *powf* function) has an impact on the speed-up of wrf as well.

All benchmarks use double precision floating point numbers, with the exception of wrf. Single precision functions are faster, and require a higher repetition ratio to be profitable. This is the reason the modelled speed-up drops below 1.0 for wrf.

[6]https://sourceware.org/bugzilla/show_bug.cgi?id=13932

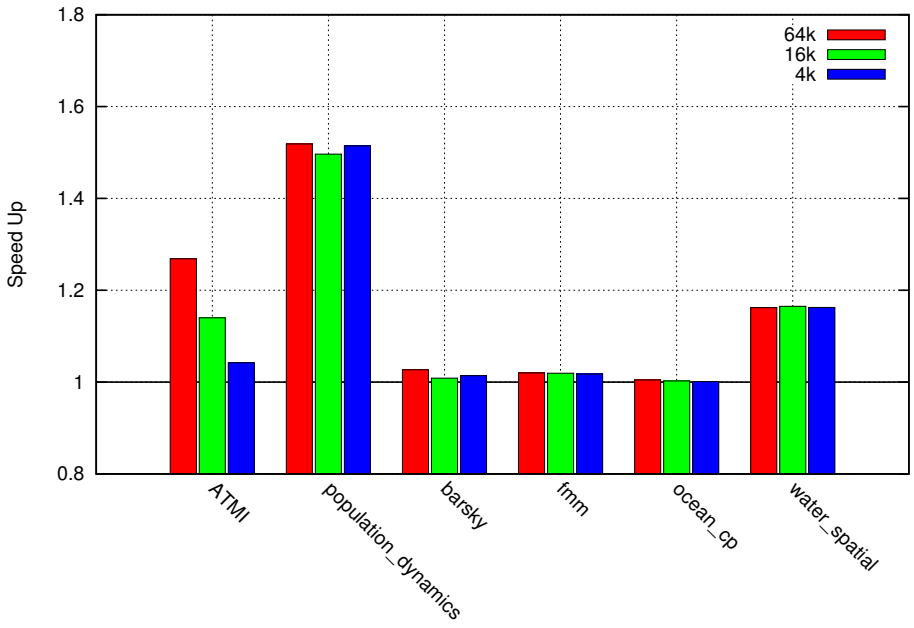

Fig. 5. Memoization of selected applications on Intel Ivy Bridge (GNU compiler) for different table sizes

Compared to SPEC benchmarks, other applications give much higher benefit for memoization as shown in Figure 5. We get a speed-up of 27% for ATMI (average value for all provided example inputs). This is due to the fact that very expensive Bessel functions – *j0* and *j1* – are used a lot in these programs. Population dynamics gives 52% speed-up as the memoized functions– *log* and *exp* – cover most of the program execution time. Barsky improves by 3% as the memoized function *sin* covers only a part of the program execution. From the Splash 2x benchmarks, water_spatial gave 16%, fmm 2% and ocean_cp less than 1% speed up.

Our actual speed-up correlates with the modelled speed-up in most cases. The variation for bwaves and wrf of SPEC 2006 are already discussed and attributed to the performance bug of *pow* implementation. The other significant variation happened with tonto where the modelled speed-up is 10% and we got 24% actual speed-up. We found that in tonto memoization was causing 13% reduction in instructions retired while there weren't any significant change in L1 or L2 cache misses due to memoization. Thus, we assume that the main reason for the variation in the modelled and the actual speed-up is due to the variation in the estimated running time of sincos function as shown in Table II and the actual running time of sincos in tonto, as arguments to sincos were different in both cases. The small variation we see in other benchmarks are also attributed to the variation between the actual runtime of transcendental functions and the empirical runtime we used for modelling (shown in Table II).

Memoization substitutes execution of code by a table lookup. Given the size of the table, it might interfere with application's data in lower level caches. We measured the L1 and L2 miss rates on all the applications. In most of the applications L1 misses got reduced due to memoization but the reduction was less than 1%.

For gafort the L1 miss was increased by 4% and for gamess it was increased by 2%. For ATMI only, increase in L1 miss was abnormally large at 40%. L2 misses were also reduced

for some applications with the maximum being for barsky at 2%. Increase in L2 misses were within 1% for all applications except population dynamics, gamess, povray and ATMI. For population dynamics L2 miss increased by 4.6%, for gamess by 3.1% and for povray by 3.6%. For ATMI L2 miss was very high at 46%. The large cache misses for ATMI is expected as it is very critical in Bessel functions (more than 75% of the execution time is spend in Bessel functions) and so with memoization, the table is intensively used causing a lot more cache misses. This fact is proved by the use of an associative table as discussed in section 5.4.

5.3.2. Intel Ivy Bridge (Intel compiler). Figure 6 shows the speed-up with memoization for SPEC benchmarks compiled with the Intel compiler icc. With icc, benefits were reduced to upto 3%. Bwaves is no longer outstanding, as the performance bug is not present in Intel's implementation. The benefit is reduced to only 3%. The lesser impact of memoization is due to a faster implementation of transcendental functions in Intel's math library. Table VI reports on the performance of Intel's implementation of transcendental functions. Runtimes were measured for the same test suite as used for the GNU library in Table II and the runtime difference shows that icc implementations are much superior on Intel Ivy Bridge. Figure 8 plots the profitability curve, i.e. the percentage of repetition needed for a given average execution time. We also identified the location of the considered functions for each compiler. Many functions in Intel's library have running times close to the break-even point of our memoization implementation: 30 ns require a 100 % repetition.

Our other selected applications also gave a low benefit with icc (Figure 7 as the maximum performance benefit went to less than 15% for population dynamics. ATMI gave a performance gain of upto 7% while barsky and fmm gave only 1%. ocean_cp and water_spatial did not give any performance gain. We could not compile gafort as the code was not compatible. In the case of tonto and equake we found that Intel compiler was using its own custom implementation for sincos called *__libm_sse2_sincos*(for equake calls to sin and cos were replaced by calls to sincos by icc) and hence our memoization scheme could not capture the sincos calls.

Table VI. Profitability analysis of memoizing transcendental functions – 2 GHz Ivy Bridge, Intel lib

	Function	Hit Time (ns)	Miss Overhead (ns)	Avg. Time (ns)	Repetition Needed
double	exp	30	55	54	70%
	log			53	71%
	sin			56	68%
	cos			55	69%
	j0			115	39%
	j1			121	38%
	pow			88	49%
float	expf			50	73%
	logf			40	85%
	sinf			45	79%
	cosf			41	83%
	j0f			126	36%
	j1f			67	60%
	powf			67	60%

5.3.3. ARM Cortex-A9. On ARM Cortex-A9, the memoization benefit varied compared to that in Intel Ivy Bridge as shown in Figure 9. We found that transcendental functions take more execution time on ARM Cortex-A9, but memoization overhead is also higher as shown in Table VII. A comparison of the profitability curve for ARM Cortex A-9 and Intel Ivy bridge is shown in Figure 10.

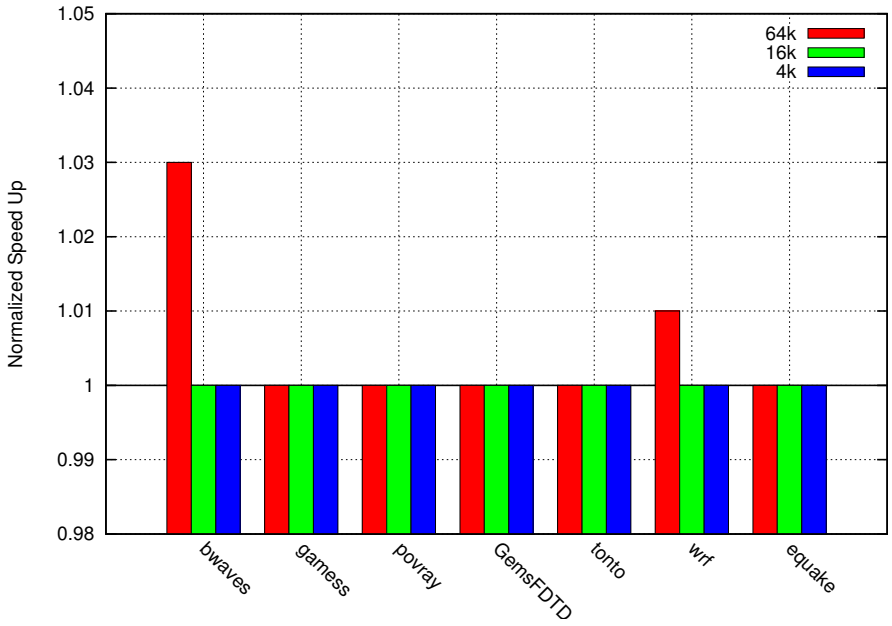

Fig. 6. Memoization of SPEC benchmarks on Intel Ivy Bridge for different table sizes (icc)

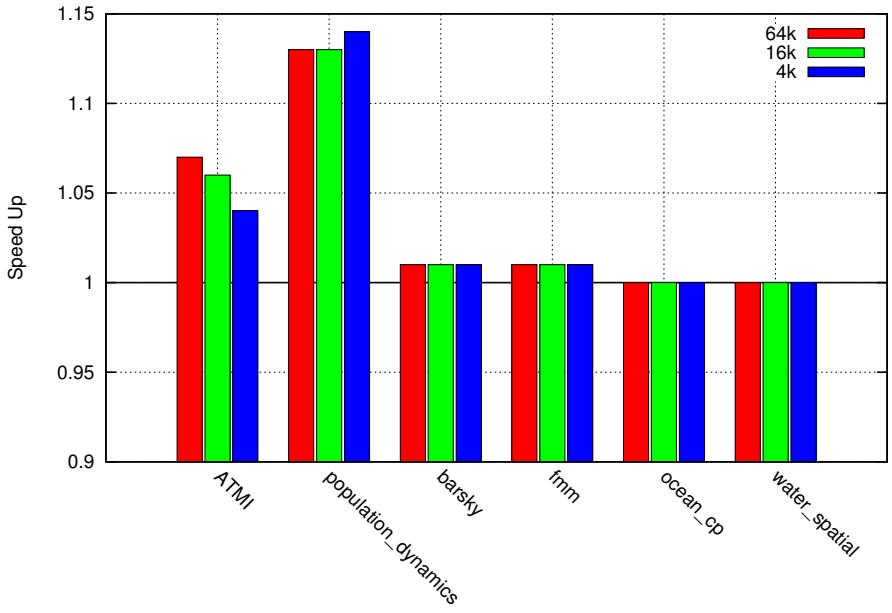

Fig. 7. Memoization of selected applications on Intel Ivy Bridge for different table sizes (icc)

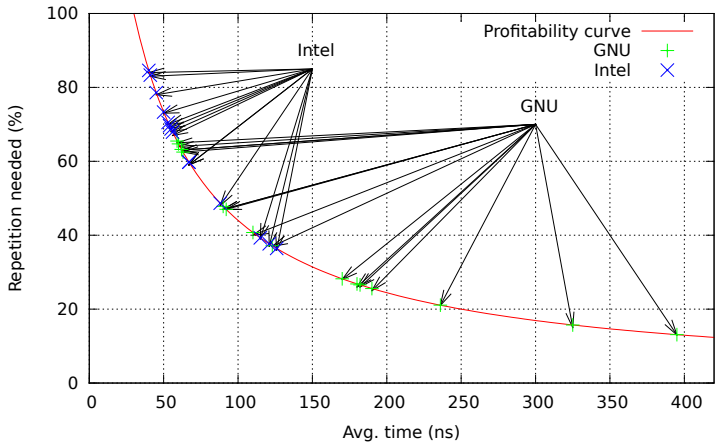

Fig. 8. Profitability curve of memoizing transcendental functions

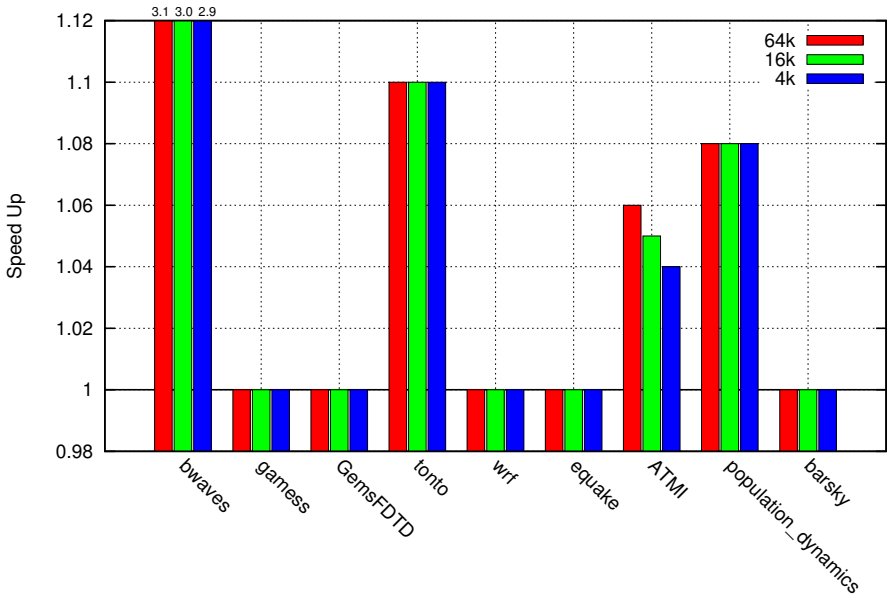

Fig. 9. Memoization of SPEC benchmarks and selected applications on ARM (GNU compiler) for different table sizes

From the SPEC benchmarks, bwaves and gafort gave gain on memoization with bwaves giving a speed-up upto 3.1 and tonto giving upto 1.1. Compared to on Intel Ivy Bridge, gain for bwaves was more while that on tonto was less. We could not run gafort on ARM due to memory limitation.

Among the other applications, we got performance benefit for ATMI and population dynamics but the gains were lesser as compared to that on Intel Ivy Bridge. Population dynamics gave the same benefit on memoization for all considered table sizes while the

Table VII. Profitability analysis of memoizing transcendental functions – 1.2 GHz ARM Cortex-A9, GNU lib

	Function	Hit Time (ns) t_h	Miss Overhead (ns) t_{mo}	Avg. Time (ns) T_f	Repetition Needed H
double	exp	44	180	199	54%
	log			300	41%
	sin			300	41%
	cos			300	41%
	j0			1380	12%
	j1			1362	12%
	pow			402	33%
	sincos			330	39%
float	expf			100	76%
	logf			140	65%
	sinf			95	78%
	cosf			117	71%
	j0f			720	21%
	j1f			800	19%
	powf			212	52%
	sincosf			170	59%

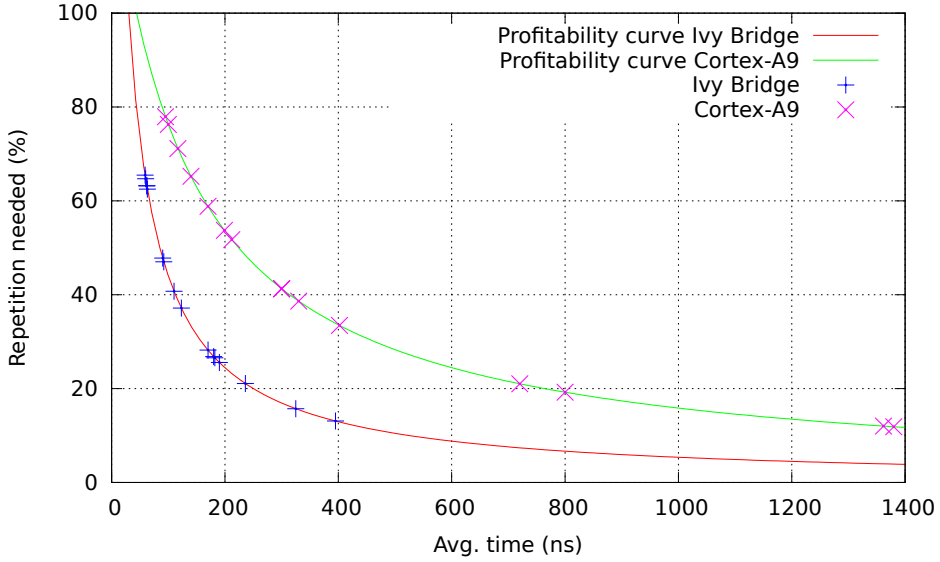

Fig. 10. Profitability curve of memoizing transcendental functions (Intel Ivy bridge & ARM Cortex-A9 with GNU library

benefit for ATMI got reduced with the reduction in size of the memoization table as was the case in Intel Ivybridge.

5.4. Associativity

In an associative hash-table, more entries that can be stored for each hash location, minimizing the chance of collisions, but at the same time increasing the overhead of table look-up. We found that associative hash-tables are beneficial to long latency transcendental functions – specifically the Bessel functions. For other transcendental functions which are relatively

low latency, we found that associativity was causing a negative impact due to the extra overhead of associative table look-up. In this section, we present results for memoization using an associative hash table implementation for Bessel functions and a non-associative implementation for other functions.

The associativity we have chosen is such that for each function call, only one hardware cache line is fetched. To do this we aligned the table storage on a 64 bytes boundary for Intel Ivy Bridge. Now, for double precision functions we used a 4-way associative table as both the 8 byte argument as well as result requires 16 bytes, and four such entries will fit in a cache line. Figure 11 shows the effect of an associative hash table for the Bessel functions in ATMI runs on Intel Ivy Bridge. With a 256k 4-way associative table we got the best performance as we could capture most of the repetitions. A 256k non associative table actually gave worse performance than a 64k non associative table as the hash function worked better for 64k table and a 256k non-associative table polluted the cache a lot more. With associativity we could use the same hash-function for the 256k 4-way associative table and we could maintain the same number of cache line accesses.

Table VIII. Function call analysis for ATMI on Intel Ivy Bridge with GNU compiler and 64k table

Application	Func	No. of calls	Unique values	Hits	Evictions	Exec. Time (s)	speed-up	Modelled speed-up
ATMI.chipedges	exp	3,321,150	6,825	96.7%	3.1%	2.29	1.55	1.38
	j0	2,757,132	79,941	43.2%	55.1%			
	j1	1,562,103	73,542	50.5%	46.7%			
ATMI.Fisher	exp	3,099,516	97,966	90.0%	8.4%	0.964	1.27	1.67
	j0	600,831	6,537	91.8%	7.2%			
	j1	137,928	10,983	89.7%	3.0%			
	log	1,657,656	1	100.0%	0.0%			
ATMI.migration	exp	62,309,016	8,613,241	68.3%	31.6%	22.216	1.22	1.14
	j0	20,475,609	387,220	12.8%	86.9%			
	j1	2,867,004	127,054	83.4%	14.6%			
	log	33,499,158	1	100.0%	0.0%			
ATMI.onoff	exp	347,088	15,228	91.5%	4.6%	2.21	1.00	1.01
	j0	181,944	145,863	14.8%	53.0%			
	j1	165,942	78,411	49.2%	23.2%			
ATMI.Xu	exp	12,151,692	5,707	99.8%	0.2%	4.074	2.13	1.81
	j0	22,071	7,035	66.7%	2.8%			
	j1	6,063,225	48,822	83.3%	16.2%			
ATMI.Goh	exp	24,240,888	19,949	93.0%	6.9%	17.145	1.36	1.25
	j0	21,903,210	215,902	31.2%	68.6%			
	j1	11,179,854	126,075	41.9%	57.6%			
ATMI.pentium	exp	4,744,866	16,380	98.2%	1.5%	2.059	1.08	1.10
	j0	1,425,816	1,051,150	5.5%	89.9%			
	j1	1,186,437	685,119	26.8%	67.6%			
ATMI.ppc1	exp	15,437,310	17,118	97.3%	2.6%	3.741	1.14	1.11
	j0	235,788	180,084	16.5%	57.5%			
	j1	7,710,213	6,916,848	6.9%	92.3%			
	pow	358,842	315,934	10.0%	71.8%			
ATMI.ppc2	exp	4,928,910	17,118	94.8%	4.9%	2.549	1.02	1.07
	j0	1,499,778	1,103,571	5.0%	90.6%			
	j1	1,192,023	645,420	28.3%	66.2%			
	pow	114,665	89,224	21.0%	36.3%			
ATMI.ppc3	exp	2,347,254	17,118	92.8%	6.6%	2.772	1.14	1.08
	j0	4,008,396	180,074	21.8%	76.7%			
	j1	1,030,449	857,015	9.1%	84.5%			
	pow	57,214	39,982	24.5%	22.9%			

On ARM platform also we tried the same mechanism for associativity. Since, the cache line size is 32 bytes, we used a 2-way associative table. Figure 12 shows the effect of an associative hash table for the Bessel functions in ATMI on ARM plaform. As in the case of Intel Ivy Bridge, associativity gives very good performance on ARM platform also. Again, a 2-way 32k table performs better than a 128k table.

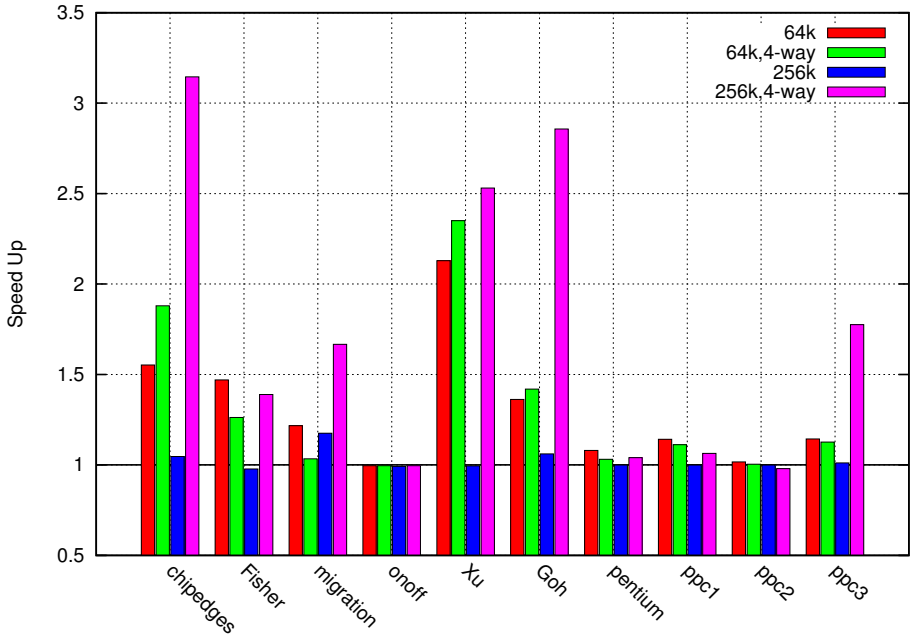

Fig. 11. Associative table for ATMI (GNU compiler) on Intel Ivy Bridge

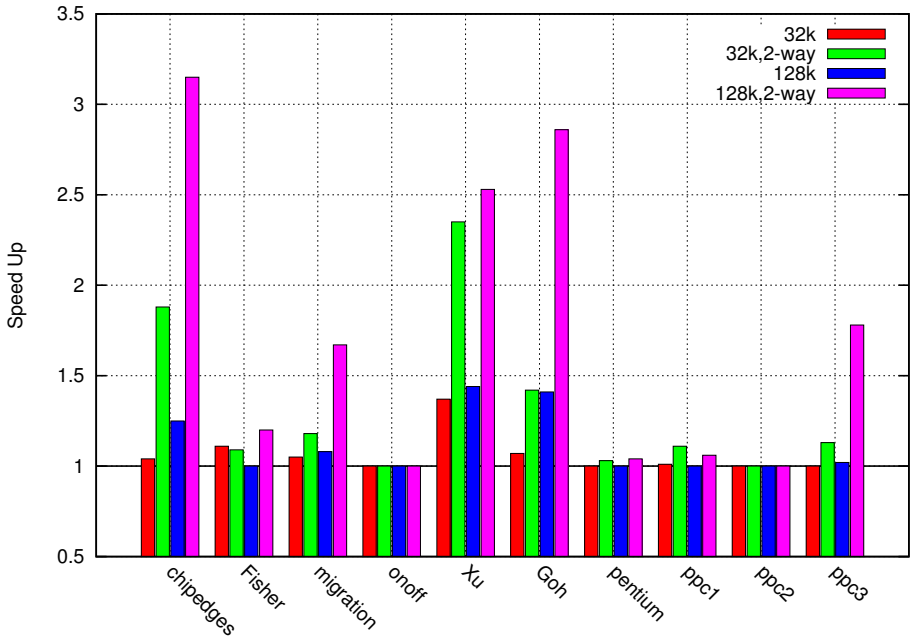

Fig. 12. Associative table for ATMI (GNU compiler) on ARM Cortex-A9

5.5. Call site Analysis

In order to find the importance of call sites for function memoization, we did an analysis per call site and the result is shown in Table IX. We classified the results into three cases:

— Single call site: All calls to a memoized function comes from a single call site
— Multiple call sites similar behavior: There are multiple call sites for a memoized function but all of them shows similar behaviour– either repeating or non-repeating– for the input arguments
— Multiple call sites differential behaviour: There are multiple call sites for a memoized function and some of them have repeating behaviour while others have non-repeating behaviour with respect to their input arguments

In the first two cases effectiveness of memoization is independent of the call site, while in the third case doing a call site specific memoization might result in a better memoization result. In our considered benchmarks only wrf and gamess falls into the third category.

Table IX. Function call-site analysis on Intel Ivy Bridge with GNU compiler and 64k table

Application	Function	Behaviour
bwaves	pow	Multiple call-sites similar behaviour
povray	sin pow	
tonto	exp sincos	
equake	sin	
gafort	sin exp	
ATMI	exp j0 j1 log pow	
barsky	sin	
population dynamics	exp log	
splash2x.fmm	pow	
splash2x.ocean_cp	sin	
splash2x.water_spatial	exp	
GemsFDTD	exp	Single call-site
equake	cos	
splash2x.fmm	log	
gamess	exp	Multiple call-sites differential behaviour
wrf	expf powf logf	

5.6. Hash-table Performance

We compared the performance of our hash function with a simpler approach of considering only the first 16 mantissa bits for indexing and the result is shown in Table X. The result shows that XOR hash function is much better performing than the simpler method while adding two extra XOR operations.

Table X. Masking Hash-function performance analysis on Intel Ivy Bridge with GNU compiler and 64k table

Application	Function	No. of calls	No. of unique values	Hit % with XOR	Hit % with 16 bits
bwaves	pow	216,320,000	18,329,205	75.5	75.3
gamess	exp	1,066,610,840	24,761,242	48.2	48.1
povray	sin	5,241,639	5,241,639	0	0
	pow	30,664,910	23,001,147	23.7	0.1
GemsFDTD	exp	711,400,840	621,783	99.9	99.9
tonto	exp	567,927,519	4,767,144	80	70.2
	sincos	453,108,244	305	100	93.8
wrf	expf	331,317,492	10,144,648	14.3	14.2
	powf	1,675,704,575	80,555,848	23.5	22.9
	logf	80,425,116	16,075,039	45	39.7
equake	sin	568,120,502	252	100	100
	cos	284,060,252	251	100	100
gafort	sin	2,200,000,000	32,768	97.1	90
	exp	2,200,000,000	32,768	97.2	97.2
ATMI	exp	132,927,690	8,826,650	82.8	27.5
	j0	53,110,575	3,357,377	23.1	9.2
	j1	33,095,178	73,542	43.5	4.5
	log	35,156,814	9,569,289	100	100
	pow	530,721	445,140	14	13.9
barsky	sin	26,148,922	455	97.7	20.6
population dynamics	exp	28,744,800	589	99.9	99.9
	log	28,744,800	589	99.9	99.9
splash2x.fmm	log	37,717,281	78	99.9	85.5
	pow	9,109,011	9,101,294	0	0
splash2x.ocean_cp	sin	16,785,411	4,097	94.5	24.5
splash2x.water_spatial	exp	1,039,992,980	12,538,545	98.2	94.5

6. CONCLUSION

We have experimentally demonstrated that software memoization is applicable to many CPU intensive scientific codes written in C or Fortran. Without any modification to the source code we can benefit from the repeated calls to the same function with the same input set. Most of the math intensive programs we tried gave good benefit for memoization. And there is negligible slow down which is guaranteed to be below a few hundred milliseconds as we have a mechanism to disable memoization during such instances.

Even for double precision values, we have obtained reasonable or very good hit rate in the memoization table for a variety of benchmark applications. We also found that a memoization table of upto 64k size worked well on Intel Ivy bridge as well as ARM Cortex A-9.

In conclusion, we benefit from memoizing any function with the following characteristics:

(1) Expensive: execution time in the order of 100 clock cycles on average, or more.
(2) Side-effect free: always returns the same result for the same input set, and does not modify the global program state.
(3) Repeated arguments: the function must be called repeatedly with the same arguments.
(4) Critical: the function to be memoized must be critical with respect to the overall program run. Otherwise, the benefit of memoization will be negligible.

We have demonstrated memoization for transcendental functions, but the same strategy can be used for any dynamically linked function and this will be investigated in our future work. Memoization is guaranteed to give a performance boost if the program is intensive in the memoized function calls and arguments are repetitive. Even in the case of non repetitive arguments, due to the disable mechanism, memoization would not cause any slow down.

Our memoization approach is architecture neutral and works for any executable where the function to be memoized is dynamically linked. The only requirement of memoization approach is to store the hash-table in memory which requires 2^{16} double-precision entries in the hash-table amounting to 1 MB of physical memory for each memoized function, which is very much acceptable for a modern computer. Being done during execution stage, our technique does not need the availability of any source code and hence can be applied

to commercial applications as well as legacy codes. Moreover, using this mechanism is as simple as setting an environmental variable.

7. FUTURE WORK

We have got reasonable amount of performance improvement through memoization of transcendental functions in a variety of programs. To improve the effectiveness of memoization, we plan to extend our framework as follows:

(1) Use of approximation: Currently we are memoizing the exact value of arguments/results. But in many domains like media streaming, we require only an approximate value. So, we can do an approximation on the memoized values and this can improve our hit rate in the memoized table
(2) Use of runtime support for memoization: Currently our memoization scheme does not do any work at runtime other than to turn off memoization if there is a negative impact. We plan to do more work at runtime like finding user functions which are memoizable at run time, doing speculative memoization etc. With the support of a runtime code replacement framework like PADRONE [Riou et al. 2014] we can do memoization of a function at runtime even without the need of `LD_PRELOAD`.

REFERENCES

C. Alvarez, J. Corbal, and M. Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *Computers, IEEE Transactions on* 54, 7 (July 2005), 922–927. DOI:http://dx.doi.org/10.1109/TC.2005.119

Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools*. 1–10.

Sandra Barsky. This is a program to solve nonlinear 2-D PDE using one-step linearization. http://www.mgnet.org/mgnet/Codes/barsky/nl.c

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. DOI:http://dx.doi.org/10.1145/1454115.1454128

D. Bruening, T. Garnett, and S. Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. 265–275. DOI:http://dx.doi.org/10.1109/CGO.2003.1191551

Mamadou Ciss, Nicolas Parisey, Fabrice Moreau, Charles-Antoine Dedryver, and Jean-Sébastien Pierre. 2013. 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), 1–9.

Daniel Citron, Dror Feitelson, and Larry Rudolph. 1998. Accelerating Multi-media Processing by Implementing Memoing in Multiplication and Division Units. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, New York, NY, USA, 252–261. DOI:http://dx.doi.org/10.1145/291069.291056

Daniel A. Connors and Wen-Mei W. Hwu. 1999. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO)*.

Amarildo T Da Costa, Felipe MG França, and MC Eliseu Filho. 2000. The Dynamic Trace Memorization Reuse Technique. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 92–92. http://www.computer.org/csdl/proceedings/pact/2000/0622/00/06220092.pdf

H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. 449–460. DOI:http://dx.doi.org/10.1109/MICRO.2012.48

Antonio González, Jordi Tubella, and Carlos Molina. 1999. Trace-level reuse. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*. IEEE, 30–37. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=797385

John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. DOI:http://dx.doi.org/10.1145/1186736.1186737

Jian Huang and David J Lilja. 1999. Exploiting basic block value locality with block reuse. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*. IEEE, 106–114. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=744342

Intel Corporation. Intel Turbo Boost technology. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *In PLDI 05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, 190–200.

Paul McNamee and Marty Hall. 1998. Developing a Tool for Memoizing Functions in C++. *SIGPLAN Not.* 33, 8 (Aug. 1998), 17–22. DOI:http://dx.doi.org/10.1145/286385.286386

Pierre Michaud, André Seznec, Damien Fetis, Yiannakis Sazeides, and Theofanis Constantinou. 2007. A Study of Thread Migration in Temperature-constrained Multicores. *ACM Trans. Archit. Code Optim.* 4, 2, Article 9 (June 2007). DOI:http://dx.doi.org/10.1145/1250727.1250729

Uday Reddy. 1988. Objects As Closures: Abstract Semantics of Object-oriented Languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP '88)*. ACM, New York, NY, USA, 289–297. DOI:http://dx.doi.org/10.1145/62678.62721

Stephen E. Richardson. 1992. *Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation*. Technical Report. Sun Microsystems, Inc. Mountain View, CA, USA.

Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. 2014. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. In *DCE 2014 - International workshop on Dynamic Compilation Everywhere*. Vienna, Austria. http://hal.inria.fr/hal-00917950

Hugo Rito and João Cachopo. 2010. Memoization of Methods Using Software Transactional Memory to Track Internal State Dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*. ACM, New York, NY, USA, 89–98. DOI:http://dx.doi.org/10.1145/1852761.1852775

James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. 2008. SoftSig: Software-exposed Hardware Signatures for Code Analysis and Optimization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 145–156. DOI:http://dx.doi.org/10.1145/1346281.1346300