ANNÉE 2016





THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

En Cotutelle Internationale avec

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Arjun SURESH

préparée à l'unité de recherche INRIA
Institut National de Recherche en Informatique et Automatique
Université de Rennes 1

Intercepting Functions for Memoization

Thèse soutenue à Rennes
le 10 Mai, 2016

devant le jury composé de :

Fabrice RASTELLO
Chargé de recherche Inria / *Rapporteur*
Jean-Michel MULLER
Directeur de recherche CNRS / *Rapporteur*
Sandrine BLAZY
Professeur à l'Université de Rennes 1 / *Examinateur*
Vincent LOECHNER
Maître de conférences, Université Louis Pasteur, Strasbourg / *Examinateur*
Erven ROHOU
Directeur de recherche INRIA / *Directeur de thèse*
André SEZNEC
Directeur de recherche INRIA / *Co-directeur de thèse*

If you save now you might benefit later.

Acknowledgement

I thank Erven Rohou for advising me for my thesis and for his continuous support.

I thank André Seznec for providing me the opportunity for working in the ALF team.

I thank Bharath Narasimha Swamy for helping me during the early part of my thesis.

I thank all the members of ALF team and all INRIA/IRISA employee for providing me a good environment for completing my thesis. I especially thank Emmanuel Riou for helping me write my thesis summary in French and for all other help.

I also thank my wife Resmi, my parents and my friends who have helped me whenever I had any difficulty.

Contents

Table of Contents 1

Résumé 7

0.1 Les gains potentiels de la Memoization 8

0.2 L'implémentation 8

Introduction 13

0.3 Context 13

0.4 Problem Definition 14

0.5 Proposed Solution 15

0.6 Motivation 16

0.6.1 Repetition of arguments in transcendental functions 16

0.6.2 Potential Latency Savings from Memoization 18

0.7 Organization of Thesis 19

1 Memoization 21

1.1 State of the Art 23

1.1.1 Instruction level Memoization 23

1.1.2 Block-level Memoization 26

1.1.3 Region-level Memoization 27

1.1.4 Memoization in Programming Languages 28

1.1.5 Memoization for Impure Code . . . 29
1.1.6 More Applications of Memoization . . . 31
1.1.6.1 Fuzzy Memoization . . . 31
1.1.6.2 Memoization in GPUs . . . 31
1.1.6.3 Memoization for Multi-threaded Architectures . . . 31
1.1.6.4 Memoization for Symbolic Execution . . . 32
1.2 Our Work . . . 32

2 Intercepting Functions for Memoization at Load Time 35
2.1 Introduction . . . 35
2.2 Implementation . . . 37
2.2.1 Intercepting Loader . . . 37
2.2.2 Hash Function . . . 38
2.2.3 Memoization Overhead . . . 40
2.2.4 Error Handling . . . 40
2.2.5 Rounding Modes . . . 41
2.2.6 Handling Negative Effects . . . 41
2.2.7 If-memo tool . . . 41
2.3 Experimental Results . . . 42
2.3.1 System Set-up . . . 42
2.3.2 System Configuration . . . 43
2.3.2.1 Intel Ivy Bridge . . . 43
2.3.2.2 ARM Cortex 9 . . . 44
2.3.3 Discussion of Results . . . 44
2.3.3.1 Intel Ivy Bridge (GNU compiler) . . . 44
2.3.3.2 Intel Ivy Bridge (Intel compiler) . . . 49
2.3.3.3 ARM Cortex-A9 . . . 50
2.3.4 Associativity . . . 53

2.3.5 Call site Analysis . . . 55
2.3.6 Hash-table Performance . . . 56
2.4 Summary . . . 58

3 Function Memoization - A Generalized Approach 61
3.1 Introduction . . . 61
3.1.1 Advantages of Compile-time Approach . . . 62
3.2 Implementation . . . 63
3.2.1 Function Templates and Parameter Ordering . . . 67
3.2.2 Handling Pointers . . . 70
3.2.2.1 Read Only Pointers . . . 71
3.2.2.2 Write Only Pointers . . . 71
3.2.2.3 Read/Write Pointers . . . 71
3.2.3 Global Variables . . . 71
3.2.4 Constants . . . 72
3.3 Experimental Results . . . 74
3.3.1 System Configuration . . . 74
3.3.2 Discussion of Results . . . 75
3.3.2.1 Speed-up in equake . . . 76
3.3.2.2 histo . . . 77
3.3.2.3 blackscholes . . . 77
3.3.2.4 wrf . . . 78
3.3.2.5 water_spatial . . . 78
3.3.2.6 Transcendental Functions . . . 78
3.3.3 Turning off Memoization . . . 78
3.3.4 Memory Overhead . . . 80
3.4 Summary . . . 81

4 Memoization with Hardware Support 83

4.1 Introduction . . . 83
4.1.0.1 Reduced latency . . . 84
4.2 Implementation Details . . . 86
4.2.1 New Instructions . . . 86
4.2.2 Hardware Table . . . 87
4.2.3 Associative Storage . . . 88
4.2.4 Multi-processing Environment . . . 89
4.2.5 Software Support . . . 89
4.3 Analysis of Potential Benefit . . . 90
4.3.1 Hit/Miss-prediction . . . 90
4.3.2 Evaluation . . . 90
4.4 Conclusion and Future Work . . . 91
4.4.1 A Hybrid Approach . . . 92

5 Conclusion 95
5.1 Further Extensions . . . 97
5.1.1 Encapsulating Function Use . . . 97
5.1.2 Capturing Expressions Involving Arguments . . . 97
5.1.3 Pre-filling Memoization-table . . . 98
5.1.4 Invariant Analysis . . . 98
5.1.5 Approximation and Prediction . . . 99

Appendix 101

Glossaire 103

Bibliography 103

List of Figures 111

List of Tables 113

List of Algorithms 115

Index 117

Résumé

Une façon d'améliorer les performances d'un code séquentiel est de faire disparaître l'exécution de code indésirable. La Memoization - sauvegarder les résultats d'une séquence d'exécution de code pour une réutilisation future - est une technique ancienne pour faire ceci. Historiquement la memoization a été proposée à différents niveaux de mise en œuvre allant de la fonction à l'instruction matérielle. La technique de memoization au niveau de la fonction - où la granularité de la memoization est la fonction - vise à réduire des exécutions répétées d'une même fonction, en éliminant ainsi des cycles CPU coûteux. Comme avec d'autres techniques de memoization, il y a une surcharge mémoire et CPU pour enregistrer l'état de la fonction et récupérer ces valeurs. Le défi est de minimiser cette surcharge pour obtenir des performances maximales. Plus la quantité de code sauvegardée par memoization est petite, plus la surcharge doit également être faible. Cela implique de mettre en place un mécanisme très efficace de récupération des données dans la table de memoization, et qui ne doit pas gêner l'exécution normale du programme. En outre, l'identification des fonctions à mémoizer et la technique de mémoization utilisée sont deux choses importantes.

Cette thèse est centrée autour de la notion de mémoization de fonctions. Pour une fonction donnée, cela consiste essentiellement à enregistrer ses entrées et sorties (les résultats) de sorte que lorsque une entrée se répète, le résultat de la fonction est tiré d'une table de hachage, économisant ainsi des cycles CPU. Comme avec tout autre système de memoization, c'est un compromis à trouver entre l'espace mémoire nécessaire au stockage des résultats et le temps CPU que l'on économise lorsqu'une entrée se répète. Nous considérons dans cette thèse uniquement des fonctions pures - qui renvoient toujours le même résultat pour une même entrée et ne modifient pas l'état global du programme.

Tout d'abord, nous avons évalué le bénéfice de notre technique de memoization sur une architecture Intel Ivybridge. Nous avons sélectionné un ensemble de fonctions transcendantes qui sont couramment appelées dans des applications de référence et analysé la manière dont leurs arguments se repètent. Nous avons constaté ainsi trois

cas de figure:

1. Il y a seulement un nombre limité d'arguments uniques.
2. Il existe un grand nombre d'arguments uniques mais avec beaucoup de répétition.
3. Il existe un grand nombre de valeurs uniques d'arguments sans répétition. Ceci est un cas difficile pour memoization.

0.1 Les gains potentiels de la Memoization

Les fonctions transcendantes sont des fonctions de calcul intensif, nécessitant souvent plusieurs centaines de cycles CPU. Récupérer les résultats d'une fonction transcendante dans notre table de memoization nous évite de répéter ces cycles de calcul coûteux et d'optimiser ainsi l'application. Cependant, il faut mesurer le gain potentiel au regard de la surcharge mémoire et CPU nécessaire à la memoization. Une estimation des gains potentiels pour des fonctions transcendantes est donnée ci-dessous: Soit T_f le temps d'exécution de la fonction memoizée, t_h le temps d'execution lorque le résultat est récupéré dans la table de memoization, et t_{mo} le temps perdu de recherche dans la table quand le résultat n'est pas présent. Nous obtenons une expression pour la fraction des appels qui doivent avoir des arguments répétés (H), pour que la memoization soit efficace.

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

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

0.2 L'implémentation

Dans la première partie, nous avons utilisé une technique load-time pour un ensemble de fonctions transcendantes pures de la bibliothèque libm. La technique de memoization est activée au moment du chargement de l'application (`LD_PRELOAD` Sur Linux) et il n'y a donc pas besoin de toucher au code source ni au matériel. Nous avons utilisé une table de hachage en correspondance directe indexée par une fonction de hachage de type XOR et nous avons évalué la performance sur des architectures verb!

Intel Ivy-pont! et `ARM Cortex A9`. Pour `Intel Ivy-bridge`, nous avons également évalué la performance d'une table de hachage associative pour des benchmarks où une table de memoization de grande taille était nécessaire. A des fins d'évaluation, une analyse de memoization spécifique des call sites a aussi été faite. Nous avons de plus développé un outil Ifmemo [SR] pour faire du load-time memoization pour un ensemble de fonctions faisant partie de bibliothèques chargées dynamiquement. L'outil fournit des informations de profiling pour les fonctions memoizées.

Dans la deuxième partie, les fonctions sont interceptées directement dans le code source, c'est ainsi une approche compile-time. Comme la mémoization se fait au moment de la compilation, cette technique est plus générale et peut être appliquée à toute fonction définie par l'utilisateur du moment qu'elle est pure. La Memoization au moment de la compilation fournit également un avantage supplémentaire, elle permet d'inliner les wrappers de memoization pour les fonctions linkées dynamiquement. Cela garantit qu'il n'y a pas d'appel de fonction effectué tant que les résultats sont récupérés depuis la table de memoization. Nous avons utilisé `LLVM`[LA04] comme framework de compilateur et implémenté notre technique de memoization comme une "pass d'optimisation".

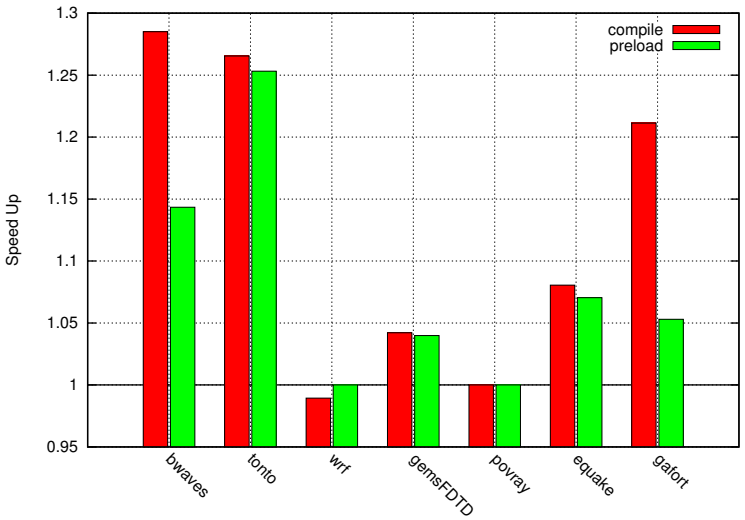


Figure 1: Speed-Up par Memoization pour les benchmarks SPEC
64k table d'entrée, Intel Ivy bridge 2.3 GHz.

Notre approche load-time est applicable uniquement pour les fonctions linkées dynamiquement tandis que la technique compile-time est applicable également aux fonctions linkés statiquement. En outre, la technique compile time offre la possibilité de

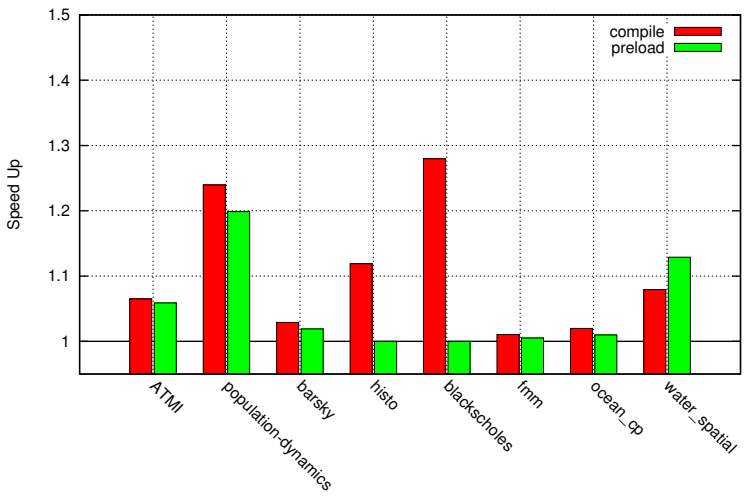


Figure 2: Speed-Up par Memoization pour les applications sélectionnées
64k table d'entrée, Intel Ivy bridge 2.3 GHz.

gérer des fonctions qui ne sont pas pures (pointeurs comme arguments, variables globales, etc.). Nos deux approches utilisent le même mécanisme de table de memoization, mais l'approche compile-time permet également de remplacer directement l'appel de fonction de la fonction d'origine (le call site) ce qui est particulièrement utile lorsque des constantes sont passées à la fonction. Cette technique compile time permet également de faire de la memoization pour une partie d'une fonction seulement. Les Figures 1 et 2 comparent les performance de nos deux approches (load-time et compile-time) pour une sélection de benchmarks (SPEC benchmarks) ainsi que pour d'autres applications.

Nous proposons également une solution basée sur une instruction matérielle et discutons des avantages et difficultés liées à cette technique. La memoization matérielle nécessite une table globale de memoization, une table de hachage associative est plus pratique ici. La taille de la table de memoization doit être beaucoup plus petite pour qu'lle soit le plus "proche" possible du CPU et assurer ainsi un accès rapide. L'évaluation des gains de cette méthode est effectuée en utilisant une technique d'estimation du temps d'exécution (détaillée dans la section 4.3.2) et pour la plupart des applications, cette technique permet de meilleures performances. Les figures 3 et 4 montrent les gains de performance avec cette technique de memoization matérielle pour des benchmarks SPEC ainsi que d'autres applications sélectionnées. La memoization matérielle diminue le surcoût logiciel des techniques précédentes et permet ainsi à plus de fonctions de bénéficier de cette technique.

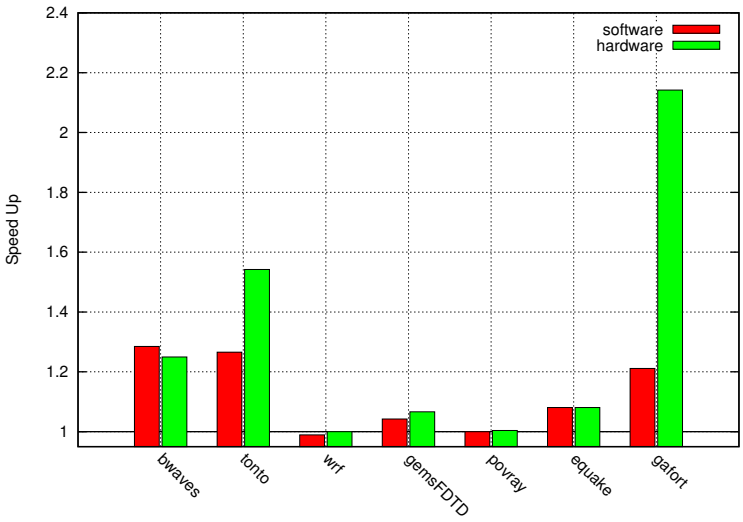


Figure 3: Speed-Up de la Memoization matérielle pour les Benchmarks SPEC
64k table d'entrée, Intel Ivy bridge 2.3 GHz.

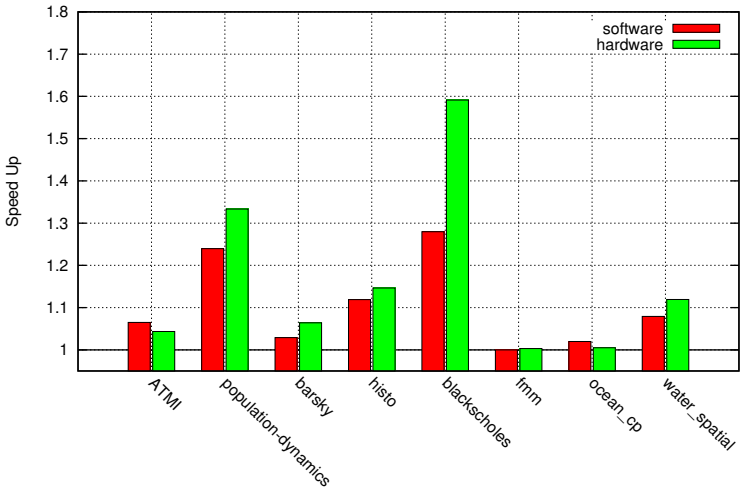


Figure 4: Speed-Up de la Memoization matérielle pour les applications sélectionnées
64k table d'entrée, Intel Ivy bridge 2.3 GHz.

Introduction

0.3 Context

From the very beginning of the modern computer life cycle, increase in performance has been a basic need. Up till the end of the 2010's decade, this demand was met by hardware changes – by increasing the clock speed of processors. But by then hardware had reached its limit. Clock speed could not be increased further as the separation between transistors had reached its physical limit, meaning their speed could not be increased further. But transistor size has been going down steadily even today obeying Moore's law, making room for more components on the chip. As a result, we get more number of cores on a chip and efforts were made to utilize these multiple cores better. These efforts lead to the era of parallelization.

But as Amdahl's law [Rod85] famously put it, the running time of any program is limited by its sequential section, irrespective of the parallelization we do. If there is x fraction of sequential code, and we have n cores, the maximum speed-up is given by

$$S = \frac{n}{(x.n) + (1 - x)}$$

The above formula is known as Amdahl's Law [Rod85]. Using this formula we can derive the maximum speed-up on any n core machine where the fraction of sequential code varies from 0 to 1 as shown in Figure 5, where n is taken as 16. The figure shows that if we have no sequential component in code, we get ideal speed-up of 16 and when there is 100% sequential component in code, we get 0 speed-up. Even if we have 10% of sequential component in code, the speed-up drops by more than 50%. Thus we can see that even when the sequential code fraction is very less, improving it can result in significant overall speed-up and this shows the significance of improving the sequential code performance in the multi-core era.

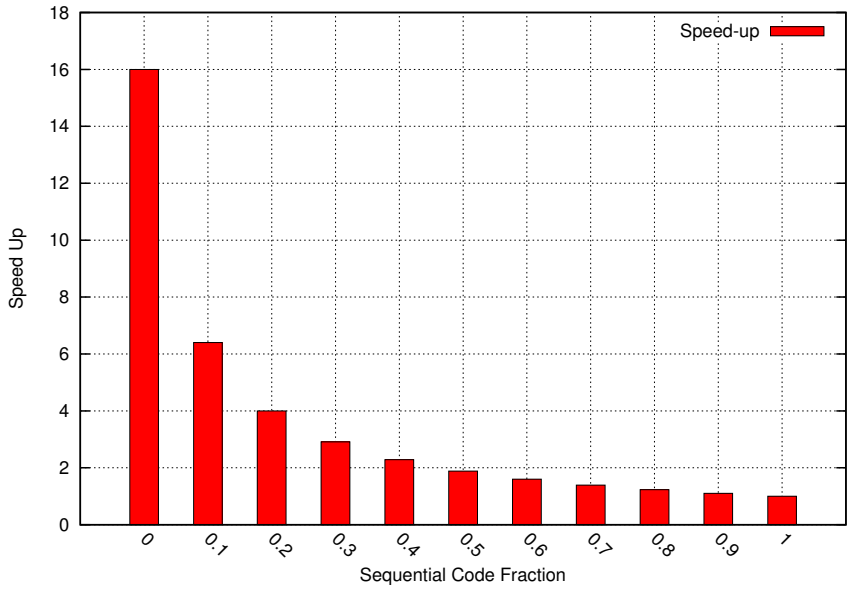


Figure 5: Importance of Amdahl's law

0.4 Problem Definition

One way to improve the sequential code performance is to do-away with unwanted code execution. Memoization – saving the results of a sequence of code execution for future reuse – is a time old technique for doing the same. 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 granularity of memoization is a function – aims at reducing repeated function executions there-by saving expensive CPU cycles. As with other memoization schemes, there is a memory overhead for saving function state and also the CPU overhead for look-up. 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 look up 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 memoization technique is also important.

0.5 Proposed Solution

This thesis is centered around the concept of function memoization which essentially involves saving the results of functions together with their inputs so that when the input repeats, the result is taken from a look-up table, with the objective of saving CPU cycles. Function memoization is implemented on a series of benchmarks and the performance impact of various function memoization techniques on the current architectures is evaluated. As with any other memoization scheme, here we are trading memory space for CPU time, by storing result in a software table and reusing when an input repeats.

In the first part of our approach to function memoization, we did a load-time approach which enables memoization for a select set of pure transcendental functions from the libm library. Here memoization was enabled by a load time technique (`LD_PRELOAD` on Linux) and hence there was no need of a source code or hardware change. We used a direct-mapped hash table indexed by an XOR hash function for look up and evaluated the performance on `Intel Ivy-bridge` and `ARM Cortex A9` architectures. For `Intel Ivy-bridge`, we also evaluated the performance of an associative hash table on benchmarks where a large memoization table size was needed. For evaluation purpose, an analysis for call site specific memoization was also done. We also developed a tool Ifmemo [SR] which enables load-time memoization for a select set of dynamically linked functions and also provides profiling details for functions which are memoized.

In the second part, functions are intercepted at source code level thus making it a compile-time approach. As the memoization is done at compile time, this scheme is more general and can be applied to any user defined function as long as it is pure – not modifying the global state and always returning the same value(s) for same input(s) – and so can benefit from memoization. Doing memoization at compile time also provides an added advantage of negating the function call overhead for dynamically linked functions by inlining their memoization wrapper. This ensures that there would not be any function call as long as there is no miss in the memoization table. We used `LLVM`[LA04] compiler framework and implemented our memoization mechanism as an optimizer pass. We also propose a hardware model which can lower the threshold for function execution times for effective memoizaton and also increase the performance benefit of memoization.

Our load-time approach is applicable only for dynamically linked functions while the compile time technique is applicable for both statically linked as well as dynamically linked functions. Further it provides opportunities for handling none pure functions such as pointer arguments, global variables etc. as detailed in later part of the thesis. Both our approaches use the same software table mechanism for memoization table but

the compile time approach has an option to do call site specific memoization which is useful especially when constants are being passed to function. Compile time technique also allows to do block level memoization where if only one block of a pure function is expensive, memoization can be enabled only for that part.

0.6 Motivation

There are many codes written in procedural languages like C, FORTRAN where memoization can be applicable. So, we conducted a few experiments to analyze the potential of memoization. The most important experiment was to analyze the repetition behaviour of the arguments in a real code. This is critical for benefiting from memoization.

0.6.1 Repetition of arguments in transcendental functions

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

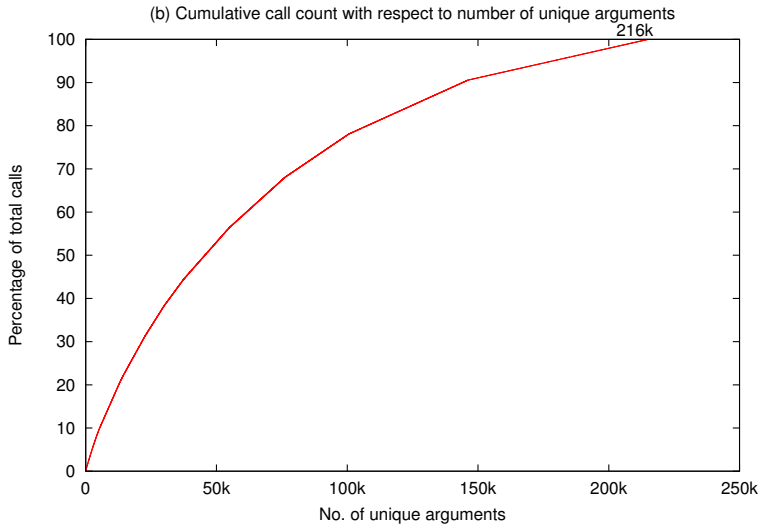


Figure 6: Reuse distance of arguments to j0 function in ATMI_Goh

Table 1: Behaviour of j0 args in ATMI

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

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 hypothesized that the redundancy in arguments values is largely due to redundancy in input that could not be caught by static compiler optimization (such as loop invariant code motion). Memoization, if applied in this case, 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 that the arguments

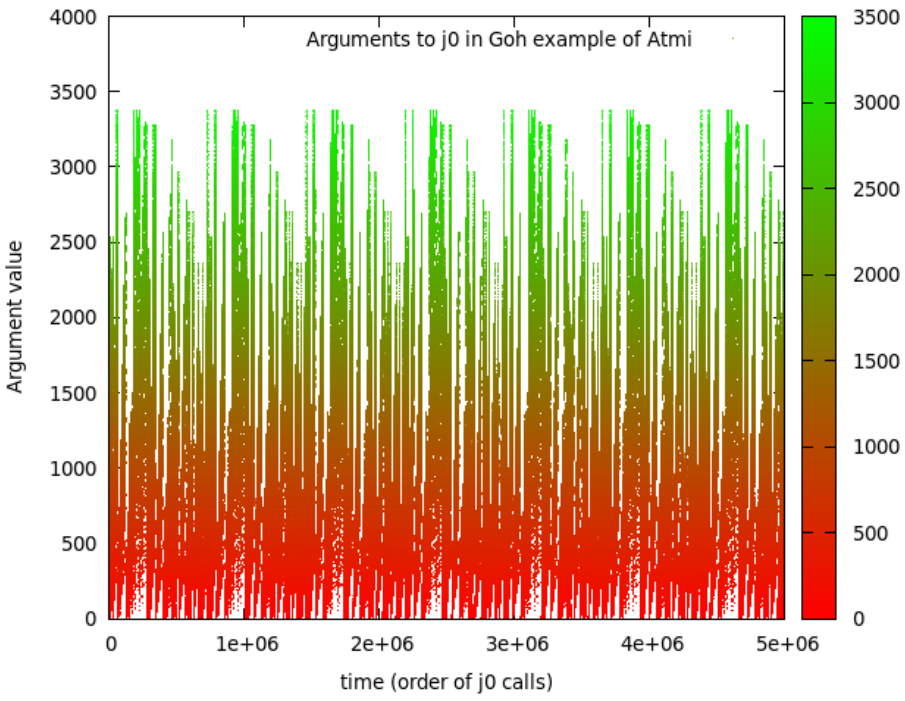


Figure 7: Repetitive behaviour of arguments to j0 function in ATMI_Goh

repeat at regular intervals of time. In regular intervals, arguments come only from a small range of values as shown in Figure 7 for one of our selected applications for memoization- ATMI_Goh for the Bessel function[Wat95] j0. In the same figure, we can see that there is a pattern getting repeated at a frequency of around 700k calls which is quite high a number for memoization table size. But even inside this regular interval, the patterns get repeated and more of them can be captured with an increased table size. F*igure* 7 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). In our experiment, 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 [Mic08]. The percentage captures of repeated arguments for various sliding window sizes are as shown in Table 1. There were a total of 21.9 million calls to j0. As expected, the percentage capture increased with an increase in the sliding window size. Out of the 21.9 M calls, less than 216 k calls actually had unique arguments (Table 2.3). This means that with a sliding window of 216 k size, we were able to capture 100% of repetitions.

3. There are a large number of unique values for arguments. This is a difficult case for memoization as without argument value being repeated we can do little about benefiting from reuse. Fortunately this kind of behaviour is very rare among real applications.

0.6.2 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.

$$\begin{aligned} H \times t_h + (1-H)(T_f + t_{mo}) &< T_f \\ H \times t_h + (T_f + t_{mo}) - H(T_f + t_{mo}) &< T_f \\ t_{mo} &< H(T_f + t_{mo} - t_h) \\ \implies H &> \frac{t_{mo}}{T_f + t_{mo} - t_h} \end{aligned} \tag{2}$$

We designed our memoization framework for fast table look-up time and low miss overhead. Table 2 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 2.1). 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 look-up, 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 enhance performance.

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

0.7 Organization of Thesis

Chapter 1 of the thesis gives a brief history of Memoization and discusses the major works in this area both old and recent. In Chapter 2 we detail our load-time approach to intercept functions and enable memoization. We also discusses the evaluation of our method on two different but current architectures and compiler tool chains and also discusses our load time tool which enables this memoization technique. In Chapter 3 we detail our compile time approach to enable memoization which enables memoization for user defined functions and works transparently for the programmer. Here, we also discuss how we handle some cases of impure functions for memoization. We propose a hardware model and its estimated performance evaluation in Chapter 4 and this

Table 2: Profitability analysis
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%

also includes a performance comparison with a pure software approach. Chapter 5 summarizes our work and also details further extensions which can be done to our work.

Chapter 1

Memoization

The term 'memoization' was coined by Donald Michie in his paper "Memo" Functions and Machine Learning [Mic68] in 1968. His work is the first one to show the benefit of saving the result of a function and hence Michie is regarded as the father of memoization. The following passage is taken from his paper:

> **"The present proposals involve a particular way of looking at functions. This point of view asserts that a function is not to be identified with the operation by which it is evaluated (the rule for finding the factorial, for instance) nor with its representation in the form of a table or other look-up medium (such as a table of factorials). A function is a function, and the means chosen to find its value in a given case is independent of the function's intrinsic meaning. This is no more than a restatement of a mathematical truism, but it is one which has been lost sight of by the designers of our programming languages. By resurrecting this truism we become free to assert: (1) that the apparatus of evaluation associated with any given function shall consist of a "rule part" (computational procedure) and a "rote part" (lookup table); (2) that evaluation in the computer shall on each given occasion proceed whether by rule, or by rote, or by a blend of the two, solely as dictated by the expediency of the moment; (3) that the rule versus rote decisions shall be handled by the machine behind the scenes; and (4) that various kinds of interaction be permitted to occur between the rule part and the rote part."**

What Michie states is that designers of programming language have fixed a procedure for most functions. For example, for factorial function we do have a recursive definition $\text{fact}(n) = n \times \text{fact}(n-1), n > 1$. But a function is a mapping from input to output, and there is no need to tie a particular method to do this. Thus Michie proposed a new apparatus for function evaluation which consists of a rule part and a rote part. The "rule part" is the conventional computational procedure and the "rote part" is the lookup of result. Second, he states that a function might be evaluated just by rule part or just by rote part or even by a mix of the two. This can mean that for particular inputs to function, it might trigger only the normal computation, and for particular inputs, it triggers the lookup and in case the lookup fails we do have a mix of the two. His third point states that the interaction between the rule and rote part must be handled behind the scenes which basically means it must not be handled by a programmer but by the programming language or by the computer hardware. His final point provides interaction between the rule part and rote part which provides self improving capability for a memo function. That is, memo functions must be capable of adapting from the previous evaluations. For example, if the inputs can be predicted, they might be evaluated ahead and thus rote part can always be triggered for future calls. Similarly, if inputs are coming randomly, the rote part can be completely avoided and rule part can take prominence.

Thus we can say that as a result of memoization, a part of program execution is saved, so that, when the same execution is needed in future, the result is obtained without a repeat execution. The trade-off is space for time, and a good repetition means good scope for memoization.

During memoization, the result of an execution needs to be stored, usually in a table. At each concerned stage, the situation also needs to be checked if it is getting repeated. This lookup time is the overhead of memoization. Memoization yields benefits if the saved execution time during a successful lookup is more than the overhead. Further, one must ensure that the memoized code does not cause any side effects. In producing a result, the code should not change the global state of the process, and memoization should always produce the same result for the same input.

At a function level, we can say that a memoizable function is the same as a pure function.

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.

However, for procedural languages, there are two reasons why plain memoization can lead to incorrect program execution: First, 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. Second, the 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 are either targeted at pure functions, or special techniques are needed to handle procedures with side effects [RC10, TACT08, LGNG14].

1.1 State of the Art

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 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.

1.1.1 Instruction level Memoization

In his work [Ric92, Ric93], Richardson shows the scope of avoiding trivial computations which includes multiplication by 0, 1, power to 1 etc. which he showed are not uncommon in real world programs. These are happening mainly due to the fact that the program might be written for a more general case while the input data is rather specific – for example program being for a 3-dimensional data while input being 2 dimensional which can trivialize one input to many functions. He also demonstrated that there are a lot of redundant function calls in real world programs and by using a suitable memoization scheme these can be benefited. The last part of the work also proposes a hardware cache called the *result cache* to implement memoization for a set of targeted arithmetic computations (multiply, divide, square root) directly in the hardware, without compiler or programmer intervention. Access to this result cache could be initiated at the same time as, for instance, a floating point divide operation. If the cache access results in a hit, the answer appears quickly and the floating point operation can be halted. On a miss, the divide unit can write the result into the cache at the same time as it can send the result on to the next pipeline stage thus minimizing the memoization overhead. For the experiments a direct-mapped result cache is used. A filter is also

used to detect and handle trivial operations, sending only non-trivial operations on to the result cache.

Citron et al. [CFR98] proposes a hardware based technique that enables multi-cycle computation in a single-cycle. The motivation behind the technique is that most of the arithmetic operations are done in the hardware using iterative algorithms. On most of the processors the multiplication operations involving floating point are already pipelined. However, the division operations for floating point operands are not pipelined; as a result the latency time for floating point division is way higher than floating multiplication. Citron et al. propose to mitigate the effect of floating point divisions by reducing their frequency via memoing – essentially a memoization technique. The input and output of assembly level operations are stored in a cache like lookup table(as the values in lookup table changes like a cache) and accessed in parallel to the conventional computation. A successful lookup gives the result of a multi-cycle computation in a single cycle, and a failed lookup does not necessitate a penalty in computation time. In their technique the input is forwarded in parallel both to a division unit and its adjacent memo-table. In short when the proper conditions are met the technique enables to bypass the processor altogether.

A memo-table is added to each computation unit that takes multiple cycles to complete. The Instruction Decode (ID) stage forwards the operands to the appropriate computation unit and in parallel, to the corresponding memo table. If there is a hit in the memo-table, its value is forwarded to the next pipeline stage (the write back (WB) stage), the computation unit is aborted and signals it is free to receive the next set of operands. If there is a miss from the memo-table, the computation is allowed to complete, the result obtained forwarded to WB stage and in parallel entered into the memo-table. This is similar to the `result-cache` introduced by Richardson and is shown in Figure 1.1 which is taken from their work.

The memo-table does not increase latency along the critical path: The calculation and the lookup are performed in parallel. Updating the memo-table with the new result in the case of a miss is also done in parallel with the forwarding of it to the WB stage. Thus on the next cycle, a new lookup/computation can be performed.

In a follow up paper on memoization Citron et. al. [CF00] 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 of 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%.

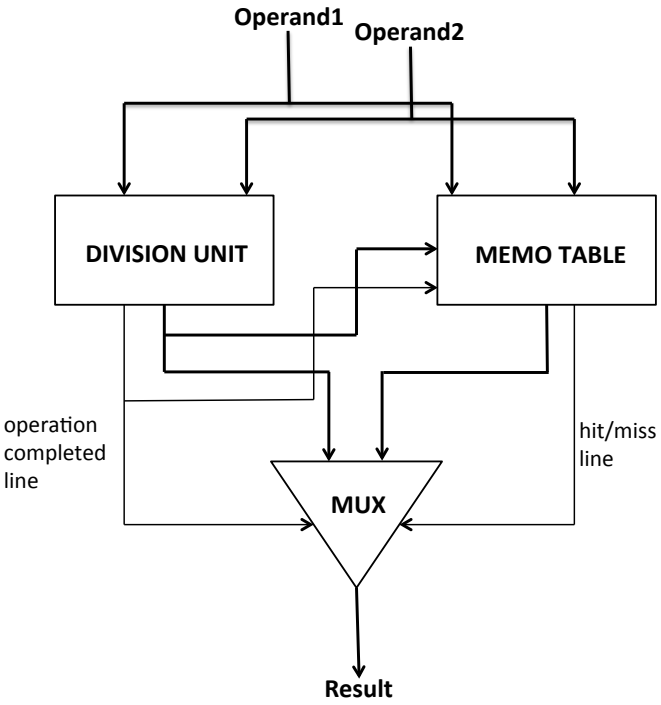


Figure 1.1: Division Unit using a MEMO Table

In instruction-level memoization a single instruction is reused while in trace-level memoization a dynamic sequence of instructions are reused. In their work [GTM99], Gonzalez et.al. explore hardware techniques for trace-level reuse. Exploiting reuse at trace-level implies that a single reuse operation can skip the execution of a potentially large number of instructions. More importantly, these instructions do not need to be fetched and thus they do not consume fetch bandwidth. Finally, since these instructions are not placed in the reorder buffer, they do not occupy any slot of the instruction window and thus, the effective instruction window size is increased as a side effect.

The reuse trace memory (RTM) is a memory that stores previous traces that are candidates to be reused. From the point of view of reuse, a trace is identified by its input and its output. The input of a trace is defined by: (i) the starting address, i.e. initial program counter (PC), and (ii) the set of register identifiers and memory locations that are live, and their contents before the trace is executed. The output of a trace consists of: (i) the set of registers and memory locations that the trace writes and their contents after the trace is executed, and (ii) the address of the next instruction to be executed after the trace. The RTM is indexed by PC and its working

is shown in Figure 1.2 which is taken from their paper. The processor dynamically decides which traces of the dynamic stream are candidates to be reused. They showed that a convenient criterion to decide the starting and ending points of a trace could be to start a new trace when a reusable instruction is encountered and to terminate the trace just before the first non reusable instruction is found. Another possibility they evaluated is to consider fixed-length traces which can be dynamically expanded once they are reused.

1.1.2 Block-level Memoization

Huang et.al. [HL99] 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. Their experiments showed that, for the SPEC 95 benchmark programs, 90% of the basic blocks have fewer than 4 register inputs, 5 live register outputs, 4 memory inputs and 2 memory outputs. About 16% to 41% of all the basic blocks were simply repeating earlier calculations when the programs were compiled with the -O2 optimization level in the GCC compiler. They evaluated the potential benefit of basic block reuse using a mechanism called a block history buffer. This mechanism records input and live output values of basic blocks to provide value prediction and reuse at the basic block level.

Ding et.al. in their work [DL04] present a software scheme based on compiler analysis, profiling information and program transformation. The scheme identifies code

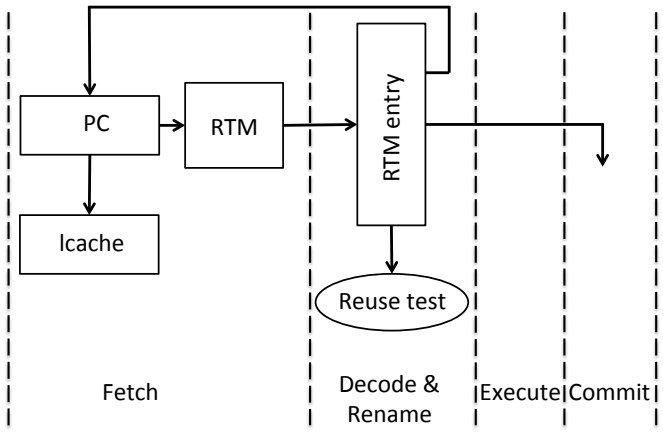


Figure 1.2: Implementation of Trace Level Memoization

segments which are good candidates for computation reuse. For each selected code segment, the scheme creates a hash table to continuously record the inputs and the matching outputs of the code segment. Based on factors such as the value repetition rate, the computation granularity, and the hashing complexity, a formula is developed to estimate whether the table look-up will cost less than repeating the execution. The hashing complexity depends on the hash function and the size of the input and the output. Unlike the special hardware reuse buffer, the hash table is very flexible in its size. It can be as large as the number of different input patterns. This offers opportunities to reuse computation whose inputs and outputs do not fit in a special hardware buffer. On the other hand, the overhead of accessing and updating the hash table is higher than the overhead for the hardware buffer.

1.1.3 Region-level Memoization

In region level memoization a region of code– can be larger than a block – is memoized to make use of potential reuse. Connors et. al. [CH99] proposes an approach that integrates architectural and compiler techniques to exploit value locality for large regions of code. The approach strives to eliminate redundant processor execution created by both instruction-level input repetition and recurrence of input data within high-level computations. In this approach, the compiler performs analysis to identify code regions whose computation can be reused during dynamic execution. The instruction set architecture provides a simple interface for the compiler to communicate the scope of each reuse region and its live-out register information to the hardware. During run time, the execution results of these reusable computation regions are recorded into hardware buffers for potential reuse. Each reuse can eliminate the execution of a large number of dynamic instructions. Furthermore, the actions needed to update the live-out registers can be performed at a higher degree of parallelism than the original code, breaking intrinsic dataflow dependence constraints. Initial results show that the compiler analysis can indeed identify large reuse regions. Overall, the approach was shown to improve the performance of a 6 issue micro-architecture by an average of 30% for a collection of SPEC and integer benchmarks.

Potential of region-level memoization is shown in Figure 1.3 which is taken from [CH99]. The code region is computing the sum of elements of an array A. At times A_τ and $A_\tau + \delta$, the code region is executed without possible change in array A. Thus for the second execution, the result of the summation could be taken from the previous saved result avoiding a repeat execution.

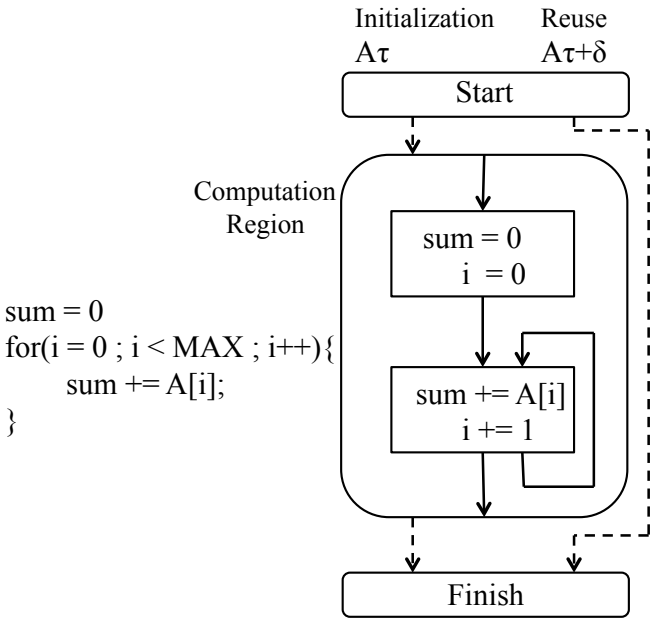


Figure 1.3: Scope of Region Based Memoization

1.1.4 Memoization in Programming Languages

Memoization technique has been used in programming languages such as Haskell [1] and Perl [2]. The presence of closure [Red88] in functional programming languages like Lisp, Javascript etc. give a ready-to-use mechanism for the programmer to write the memoization code.

Memoization is also used in logic programming languages. Tabled Logic Programming (TLP) [SW12], which is essentially memoization, is used in a number of Prolog systems including XSB, YAP Prolog, B-Prolog, Mercury, ALS, and Ciao. Tabling ensures termination and hence provides optimal known complexity for queries to a large class of programs. TLP also allows sophisticated programs to be written concisely and efficiently, especially when mechanisms such as tabled negation and call and answer subsumption are supported. As a result, TLP has now been used in a variety of applications from program analysis to querying over the semantic web.

McNamee and Hall [MH98], in their work, study how memoization can be applied to C++.While most functional languages provide the necessary facilities for memoization

[1] http://www.haskell.org/haskellwiki/Memoization
[2] http://perldoc.perl.org/Memoize.html

like (1) first-class function objects, (2) run time type information, (3) facility to modify argument list at run time and (4) built-in polymorphic collections. These features are absent in C++ and further it is not easy to modify the call to functions at run time from all call sites. Hence, they proposed two methods for memoization – (1) using function objects and (2) source code rewriting. Function objects are C++ objects that encapsulate a function pointer and overload the function call operator, operator (), to call the captured function. To facilitate memoization, the corresponding function must access arguments from `stdarg.h` library and the class must also be declared as a template class so that the function can return variable types. The second approach involves prefixing a memoizable function with a unique comment. (e.g. "`// MEMOIZE`").

In Python which is traditionally object oriented, memoization capability is provided by packages. Jug [jug] and Joblib[job] are two Python packages used for distributed computing which also provides memoization capability.

1.1.5 Memoization for Impure Code

When memoization is used in procedural languages, special techniques are needed to handle procedures with side effects.

Rito et al. [RC10] 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. When memoization is found to violate semantic correctness, STM is used to roll-back the program state to the previous correct state. Their proposal was implemented as the Automatic Transaction Oriented Memoization (ATOM) system, a Java implementation of their memoization model that extends the Java Versioned STM (JVSTM).(1) ATOM allows memoizing methods that depend on the internal state of objects (rather than only pure functions that depend only on their arguments), (2)It prevents errors in memoizing unintended methods, (3)It allows memoizing methods that have side-effects, and (4)It offers a memoization advisory tool that aids programmers in choosing which methods are profitable to memoize.

Tuck et al. [TACT08] used software-exposed hardware signatures [3] to memoize functions with implicit arguments (such as a variable that the function reads from memory). To enable flexible use of signatures, they proposes to expose a Signature Register File to the software through a rich `ISA`. The software has great flexibility to decide, for each signature, which addresses to collect and which addresses to disam-

[3] a signature is a hardware register that can represent a set of addresses

biguate against. They call this architecture *SoftSig*. As an example of *SofSig* use, they show how to detect redundant function calls efficiently and eliminate them dynamically using their algorithm called *MemoiSE*. On average for five popular applications, *MemoiSE* reduced the number of dynamic instructions by 9.3%, thereby reducing the execution time of the applications by 9%.

In a recent patent filed by Google [LGNG14], a technique is proposed to use memoization for impure functions. The technique involves analyzing several different sets of input parameters which are treated as input vectors, preforming clustering to create areas of known input vectors for which memoization may be performed and areas of known input vectors for which memoization may not be performed. The areas are defined by clustering analysis performed within the n-dimensional space defined by the input vectors. The clustering analysis is used to create confidence boundaries within the input space which is used to estimate whether an input vector of arguments may be memoized. When a new input vector lands within a confidence boundary, the input vector is treated as a memoizable or not memoizable function without performing a memoization analysis on the input vector.

The purity analysis use a control flow graph, call trace analysis, or other flow representation of an application to identify potential functions for analysis, as well as to evaluate the downstream connections of a given function to determine when and if any side effects occur. To select a function for purity analysis, the control flow graph is traversed to classify functions regarding their side effects. Some side effects, such as input from outside devices, are side effects that prohibit memoization. Other side effects, such as writing to a log file, prohibit memoization when logging is requested, but not prohibit memoization when logging is not requested. The purity of a function may also be defined on a conditional basis. The conditions may be, for example, certain sets of input parameters, specific set of side effects, or other conditions. When such a condition occurs, the function may be memoized for one set of conditions and not memoized for other sets of conditions.

In some embodiments, the output of a side effect may be captured and treated as a function input or result. In some such embodiments, the function may be considered pure when the side effect behavior is consistent and repeatable. Further, the global state of the application or device executing the application may be considered as an input to an impure function. In cases where the global state may play a role in the consistent behavior of a function, the function may be declared pure for the specific cases corresponding to a global state.

The purity of a function is determined using a statistical confidence. For example, the operations of a function may be gathered over many instances, and many devices. These data are analyzed to identify functions that behave as pure functions that may

otherwise be classified as impure functions using static analysis. In some cases, a function may be considered pure when the behavior is predictable with a high confidence, such as when the behavior may be consistent with a confidence of 0.99 or better.

1.1.6 More Applications of Memoization

1.1.6.1 Fuzzy Memoization

An extension of simple memoization is used by Alvarez et al. [ACV05] 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, there is now a set of inputs which have the same result). As a result, there is improved performance as well as reduction in power consumption without a significant change in the quality of the output. A similar technique is also used by Esmaeilzadeh et al. [ESCB12] 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.

1.1.6.2 Memoization in GPUs

Memoization is also used in graphical domains and in a recent work [APX14] Arnau et.al., proposed a hardware scheme in GPUs for saving redundant computations across successive rendering of frames. They have showed that about 40% of a frame is already processed as part of previous frames and their proposed scheme could capture nearly 60% of these repetitions.

1.1.6.3 Memoization for Multi-threaded Architectures

Long et.al. in their work [LFB+10] 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 SPMD applications. The main idea in MMT is to fetch instructions from different threads together and only splitting them if the computation is unique. For this they provide two contributions: (1) a mechanism that facilitates the synchronization of multiple threads without software hints to increase the time they spend fetching and executing identical instructions. (2) Propose changes to the decode and register renaming stages

to identify and manage instructions with the same inputs, executing and committing those instructions once for all threads. Their technique achieved speedups of 1.5 and 1.84 over a traditional out-of-order processor. Moreover, performance increase in most applications happens with no power increase because the increase in overhead is countered with a decrease in cache accesses, leading to a decrease in energy consumption for all applications.

1.1.6.4 Memoization for Symbolic Execution

In [YPK12], Yang et al. present memoized symbolic execution of source code - a technique that stores the results of symbolic execution from a previous run and reuses them as a starting point for the next run of the technique to avoid the re-execution of common paths between the new version of a program and its previous one, and to speed up the symbolic execution for the new version. They introduce a trie (an efficient tree-based data structure) [Fre60, Wil84] – based data structure that stores the key elements of a run of symbolic execution. Maintenance of the trie during successive runs allows re-use of previously computed results of symbolic execution without the need for recomputing them as is traditionally done.

In a recent work Amal Khalil and Juergen [KD15]also have proposed a pure software artifact based memorization approach in which symbolic execution technique is being optimized. Two optimization techniques for symbolic execution of evolving machines are introduced in their paper, one is based on memoization and other is based on dependency analysis. The memoized based approach reuses the Symbolic Exeution Tree (SET) generated from a previous analysis and incrementally updates the relevant parts corresponding to the changes made to the new version.

1.2 Our Work

Most of the memoization techniques require either hardware modification or have been application/language specific. Hardware techniques for memoization typically capture short sequences of instruction within a small table, while pure software techniques require the programmer to write a custom implementation of the function for memoization. A language independent software scheme for memoization was suggested by Citron et. al. [CF00] but their evaluation nearly 15 years ago, resulted in only a slowdown compared to normal execution.

In this thesis, we work on function memoization at software level which can be applied across different languages. Our framework uses large memoization tables to cap-

ture longer intervals of repetition that benefit computationally intensive pure functions at the software level without programmer intervention and we evaluate the potential memoization benefit on the current architectures. Our experimental results show that software memoization is capable of producing speed-up in current architectures over current compiler techniques. In Chapter 2 we detail our load-time approach to intercept functions and to enable memoization. In Chapter 3 we detail our compile time approach to enable memoization for user defined functions and which works transparently to the programmer. Some cases of impure functions are also handled in this work. Chapter 4 details a hardware model which can improve the benefit achieved by software memoization.

Chapter 2

Intercepting Functions for Memoization at Load Time

2.1 Introduction

In this work, we focus on opportunities for memoization 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 overhead scheme to memoize calls to transcendental functions by using function interception. While previous memoization approaches have been either at the language level or at hardware level, we are working at an intermediate level, thereby making it both language and hardware independent. Using tools like Pin [LCM+05] or DynamoRIO [BGA03], we can achieve memoization by function interception at binary level. But such binary level approaches are limited to supported architectures, are slightly slower compared to native execution and are also complex to try for a novice user. However, by using our load-time approach for function interception, we can overcome the problems of binary level approach while still benefiting from memoization. Our proposal has the following characteristics.

1. Dynamic linking: deployed at library procedure interfaces without the need of a 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 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.

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 clearly. 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, i.e. they do not modify global program state for normal inputs (boundary cases are discussed in Section 2.2.4), and the result of the computation is entirely dependent on the input arguments to the function. Thus doing a look-up 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 safety 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 repetition, 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.

We start with a load time approach for function memoization. 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. These are then redirected 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 supports `DLL` (Dynamic Link Library) injection although the mechanism is different.

In our technique, there is no need for code modification or recompilation because memoization works with legacy binaries that are compiled to link dynamically with the math library as well as commercial applications.

The remaining of the chapter is organized as follows: Section 2.2 details the implementation techniques used in our technique which involves the function interception technique and hash function being used. This section also discusses the overhead of our approach, error handling and our implementation tool called If-memo[SR]. Section 2.3 details the experimental setup we used which includes the CPU architectures, compilers and the benchmark applications we considered along with the results of our memoization approach. As a last part of our work we also discusses the benefit of an associative implementation of our hash-table, does call-site specific analysis for memoization and compares our hash-function with a simpler hash function. Section 2.4 provides a summary and conclusion for our technique.

2.2 Implementation

2.2.1 Intercepting Loader

An essential part of our implementation of 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. The working of our interception of these calls is shown in Figure 2.1. We preload a memoized version of the target transcendental functions using `LD_PRELOAD` environment variable to contain the path to our memoized library. This ensures that memoized implementation takes precedence over standard math library implementation[1]. 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

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

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 2.3.4

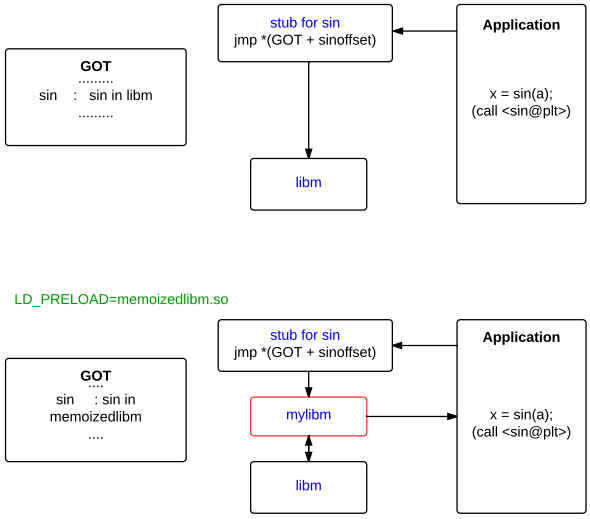


Figure 2.1: Intercepting dynamic call to *sin*

Algorithm 1 Pseudo code for memoization

```
1: index = hash(argument)
2: if table[index][0] ≠ argument then
3:     table[index][0] = argument
4:     table[index][1] = sin_from_libm(argument)
5: end if
6: return table[index][1]
```

2.2.2 Hash Function

A fast hash function is needed since the evaluation of hash function is critical to table look-up. We designed a simple hash function using the `XOR` function by repeatedly `XOR`ing the bits of the arguments to get the table index. Thus for a double-precision

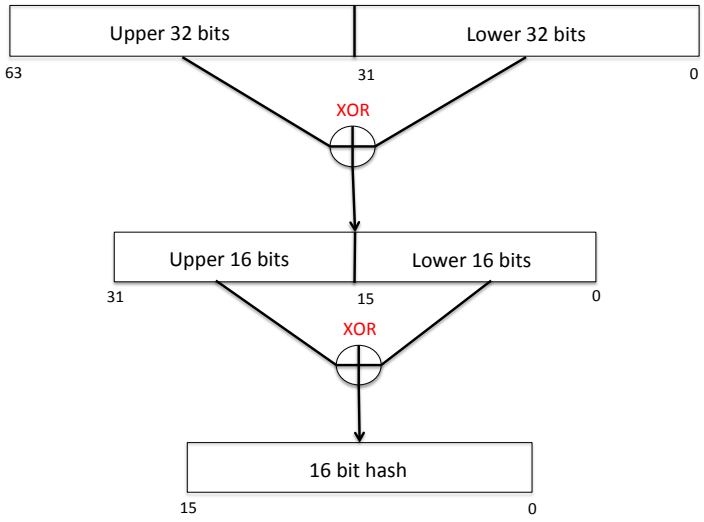


Figure 2.2: Implementation of Hash Function

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 to 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 2.2. 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 2.3). The simple hash function we designed is fast to compute and has 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. The `Intel` assembly code for doing indexing for a single double precision argument is as follows:

```
1     movq xmm0, xmm1
2     pinsrq xmm1, xmm0, 0
3     pshuflw xmm2, xmm1, 0x1b
4  ; byte-wise reverses the contents of xmm1 and stores in xmm2
5     pxor xmm2, xmm1
6     pshuflw xmm1, xmm2, 0x41
7  ;ABCD in xmm2 is made to CDDC in xmm1
8     pxor xmm1, xmm2
9     pextrw xmm0, xmm1, 0
```

2.2.3 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 2.1 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[2] 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.

2.2.4 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)

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

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.

2.2.5 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 the memoization table.

2.2.6 Handling Negative Effects

On applications that have poor locality in the arguments to target transcendental functions, memoization may slow down execution time. Due to a poor hit rate in the table, function call executions pay the price of table lookup overhead without benefiting from the use of stored results. 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 2) 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.

2.2.7 If-memo tool

If-memo [SR] [3] is our tool which implements the memoization for our selected transcendental functions. The tool provides a template like mechanism which allows memoization for the following types of function prototypes.

1. `int fun(int);`

[3] APP number: IDDN.FR.001.250013.000.S.P.2015.000.10800

2. `float fun(float);`
3. `int fun(int, int);`
4. `float fun(float, float);`
5. `double fun(double, double);`
6. `void fun(double, double *, double *);`

These function prototypes were decided so as to capture the most commonly used transcendental functions (the last one is added for capturing the *sincos* function). For each of these function types, the tool initializes the memoization table and adds the corresponding hashing code as well as the look-up and update code. Further, profiling facility like measuring hit-time (when a look-up hits in the memoization table), miss-time (when a look-up fails and original function is called), amount of collisions in memoization table etc. are provided by the tool. A helper thread is also present which can monitor the memoization efficiency and turn of memoization in case of a negative effect.

2.3 Experimental Results

2.3.1 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 2.1 and 2.2 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 effected (neither speed-up nor slowdown), and so 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 choose six benchmarks: `bwaves`, `gamess`, `povray`, GemsFDTD, `tonto` and `wrf` from `SPEC CPU 2006` benchmark suite [Hen06] with *ref* inputs, which have a reasonable number of calls to transcendental functions. `wrf` and `bwaves` extensively use *pow* function. `gamess`, and `tonto` make extensive use of *exp* and *sincos* calls, while `povray` has many calls to *sin* and *cos*.

2. `SPEC OMP 2001`
   In `SPEC OMP 2001` [ADE+01], 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.

3. `ATMI`
   `ATMI`[MSF+07] 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` [CPM+13] 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.

5. `Barsky`
   `barsky` [Bar] is a Partial Differential Equation solver and contains many calls to *sin*.

6. `Splash2x`
   `fmm`, `ocean_cp`, and `water_spatial` from Splash 2x [G+11] of `Parsec` benchmark suite [BKSL08a] can benefit from memoization. `fmm` makes extensive use of *log* function, `ocean_cp` in *sin* and `water_spatial` in *exp* functions.

2.3.2 System Configuration

2.3.2.1 Intel Ivy Bridge

We run our experiments on an `Intel i7` machine under the set-up described in Table 2.1. We make sure that the Turbo Boost feature [Int] is turned off and CPU clock

frequency is set at $2\,GHz$ to ensure reproducible time measurements. We use GNU compiler+library as well as Intel compiler+library combinations and run experiments using different table sizes. The results of the benchmark run under the GNU setup is illustrated in Figures 2.3 and 2.4. Figure 2.5 shows the results for the same configuration but the applications are compiled using `Intel` compiler. Table 2.3 lists the applications used, functions being memoized their hit rates on Intel Ivybridge with `GNU` compiler and $64k$ table size and the memoization speed up.

Table 2.1: 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

2.3.2.2 ARM Cortex 9

Next we run our experiments on Pandaboard,that has `ARM Cortex A9` processor under the set-up as described in Table 2.2. The results of the benchmark run under this set-up are given in Figure 2.8. We can not run splash $2x$ of `PARSEC` benchmark suite on `ARM` because this architecture is not supported. Due to memory limitations, all the `SPEC` benchmarks are run with train inputs on `ARM` while we use ref inputs for `Intel Ivybridge`.

2.3.3 Discussion of Results

2.3.3.1 Intel Ivy Bridge (GNU compiler)

Table 2.3 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 each of them, and the number of unique values. The next two columns report, for a 64k-entry table, the

Table 2.2: 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

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 2), assuming an otherwise ideal scenario, as follows:

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

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

$$\begin{aligned}\text{Modelled runtime} = \text{Total execution time} \\ &- (N_{\text{calls}}(\sin) \times T_{\sin} + N_{\text{calls}}(\exp) \times T_{\exp}) \\ &+ N_{\text{misses}}(\sin) \times (T_{\sin} + t_{mo}) + N_{\text{hits}}(\sin) \times t_h \\ &+ N_{\text{misses}}(\exp) \times (T_{\exp} + t_{mo}) + N_{\text{hits}}(\exp) \times t_h\end{aligned}$$

On `Intel Ivy Bridge` machine using the `GNU` compiler (Table 2.1), `SPEC` CPU benchmarks gain benefit ranging between 1% and 24% (see Figure 2.3). Only `bwaves` produces an outstanding and unexpected speed-up of 1.76. The hugh benefit we get from memoization of `bwaves` is both accidental and surprising .Surpurising in the sense that *pow* on an average takes around 300 clock cycles. Our findings show that this is due to a performance bug in the `GNU` libm for *pow* for some inputs[4] which causes

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

Table 2.3: Function call analysis
Intel Ivy Bridge with GNU compiler and 64k table

Application	**Func.**	**No. of calls**	**Unique**	**Hits**	**Evictions**	**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%			
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%			
ocean_cp	sin	16,785,411	4,097	94.5%	5.5%	123.4	1.00	1.01
water_spatial	exp	1,039,992,980	12,538,545	98.2%	1.8%	241.2	1.16	1.20

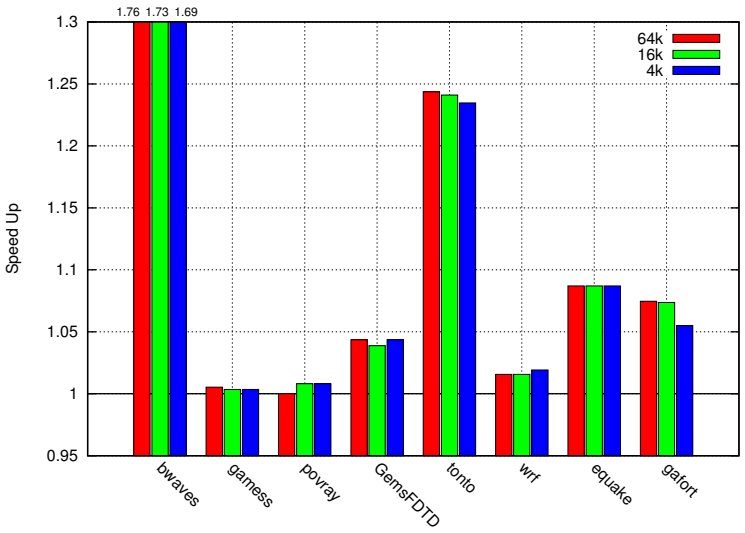


Figure 2.3: Memoization of SPEC benchmarks on Intel Ivy Bridge (GNU compiler) for different table sizes

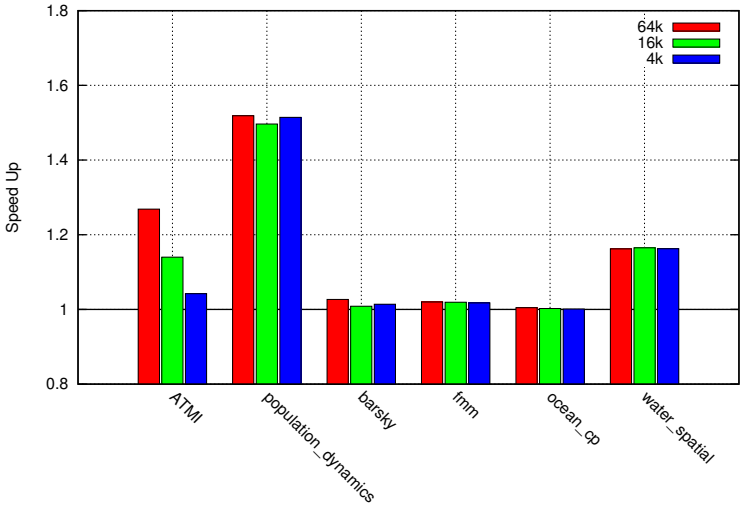


Figure 2.4: Memoization of selected applications on Intel Ivy Bridge (GNU compiler) for different table sizes

a slow down of up to $10,000\times$ compare 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 takes more than 1000 × the normal execution time.On memoization, this long latency is saved and thus we get increased benefits. 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`.

As compare to `SPEC` benchmarks, other applications gain much higher benefit from memoization as shown in Figure 2.4. 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 gain 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 Splash 2x benchmarks, `water_spatial` gain 16%, `fmm` gain 2% and `ocean_cp` gain less than 1% speed up.

Our actual speed-up correlates with the modelled speed-up in most cases. The observed variation for `bwaves` and `wrf` of `SPEC 2006` are already discussed and attributed to the performance bug of *pow* implementation. The other significant variation was with `tonto` where the modelled speed-up is 10% and we get 24% of actual speed-up.Our finding shows that in `tonto`, memoization was causing 13% reduction in retired instructions while there was not 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 2 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 2).

Memoization substitutes execution of code by a table lookup. Given the size of the table, it might interfere with an 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 reduced due to memoization but the reduction was less than 1%.

For `gafort` the *L1* misses increase by 4% and for `gamess`, by 2%. For `ATMI`, increase in *L1* miss was abnormally large at 40%. *L2* misses reduce for some applications with the maximum being for `barsky` at 2%. Increase in *L2* misses is within 1% for all applications except `Population Dynamics`, `gamess`, `povray` and `ATMI`. For `Population Dynamics` *L2* miss increase 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 makes extensive use of Bessel functions (more than 75% of the execution time is taken by Bessel functions) and so with memoization, the table is intensively used causing a lot more cache misses. This fact is further substantiated by the performance results of using an associative memoization table as discussed in section 2.3.4.

2.3.3.2 Intel Ivy Bridge (Intel compiler)

Figure 2.5 shows the speed-up with memoization for `SPEC` benchmarks compiled with the Intel compiler icc. With icc, overall speed up gains are reduced upto 3%. `bwaves` is no longer outstanding, as the performance bug is not present in Intel compiler's implementation, and so speedup gain is reduced to 3%. The lesser impact of memoization is due to a faster implementation of transcendental functions in Intel's math library. Table 2.4 reports on the performance of Intel's implementation of transcendental functions. Runtimes are measured for the same test suite as used for the GNU library in Table 2 and the runtime difference shows that icc implementations are much superior on `Intel Ivy Bridge`.

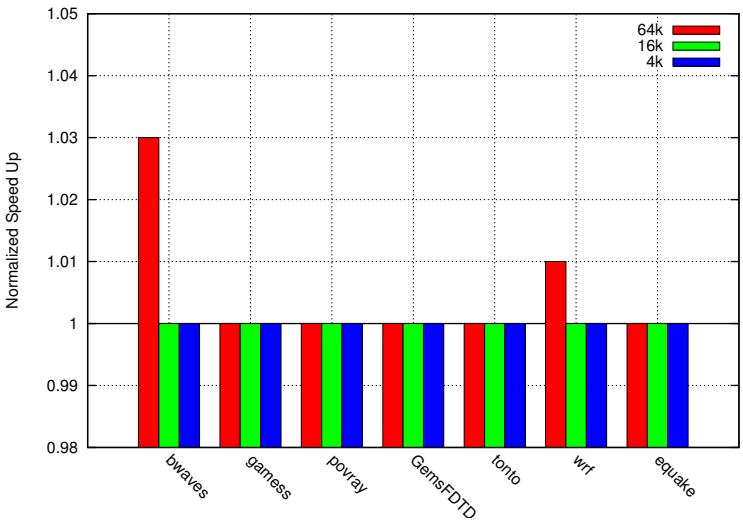


Figure 2.5: Memoization of SPEC benchmarks on Intel Ivy Bridge for different table sizes (icc)

Figure 2.6 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

Table 2.4: Profitability Analysis
2 GHz Intel 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%

to the break-even point of our memoization implementation: 30 ns requires a 100 % repetition.

Our other selected applications gain relatively low benefit with icc Figure 2.7 shows that the maximum performance gain went down to less than 15% for `population dynamics`. `ATMI` achieves a performance gain of upto 7% while `barsky` and `fmm` gain only 1% speed up. `ocean_cp` and `water_spatial` did not achieve any performance gain. We could not compile `gafort` as the code is not compatible. In the case of `tonto` and `equake` we found that `Intel` compiler is using its own custom implementation for *sincos* called *__libm_sse2_sincos*(`equake` calls to *sin* and *cos* are replaced by calls to *sincos* using icc) and hence our memoization scheme could not capture the *sincos* calls.

2.3.3.3 ARM Cortex-A9

On `ARM Cortex-A9`, the speed-up achieved by memoization varies as compared to that on `Intel Ivy Bridge` and this is shown in Figure 2.8. Our findings show that not only transcendental functions take more execution time on `ARM Cortex-A9`, but memoization overhead is also higher as shown in Table 2.5. A comparison of the profitability curve for `ARM Cortex A-9` and `Intel Ivy bridge` is shown in Figure 2.9.

From the `SPEC` benchmarks, `bwaves` and `gafort` achieve speed up gain on mem-

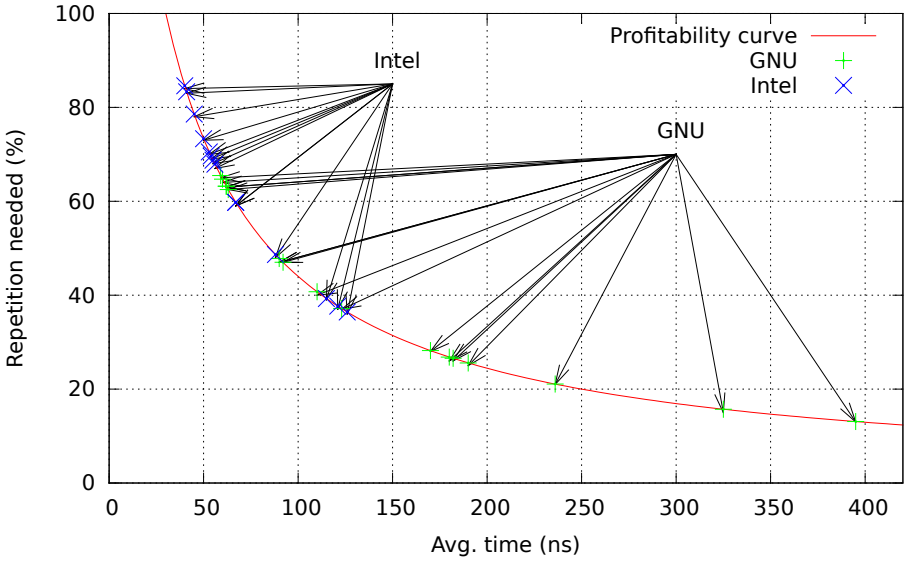


Figure 2.6: Profitability curve of memoizing transcendental functions

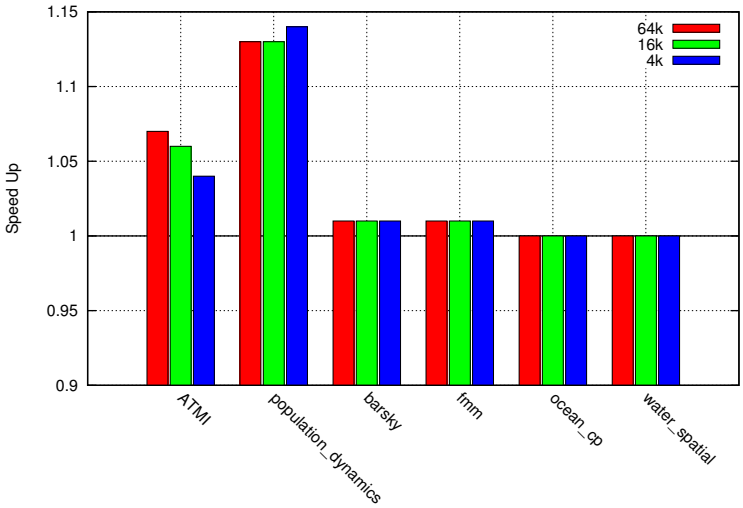


Figure 2.7: Memoization of selected applications on Intel Ivy Bridge for different table sizes (icc)

oization, with `bwaves` achieving a speed-up gain of upto 3.1 and `tonto` achieving a

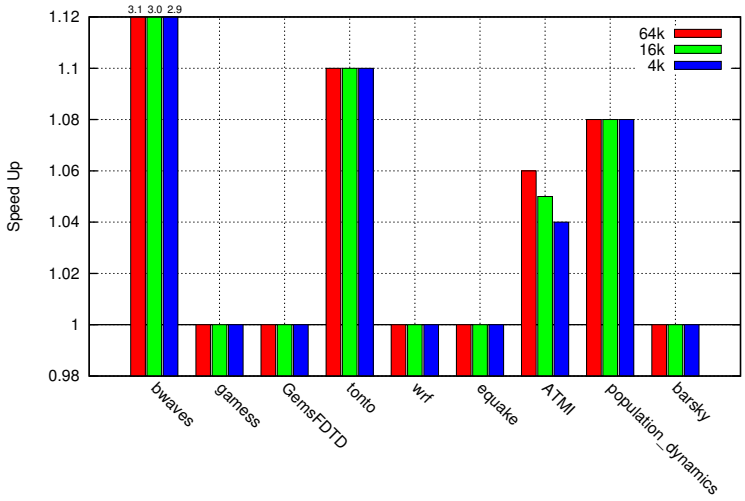


Figure 2.8: Memoization of SPEC benchmarks and selected applications on ARM (GNU compiler) for different table sizes

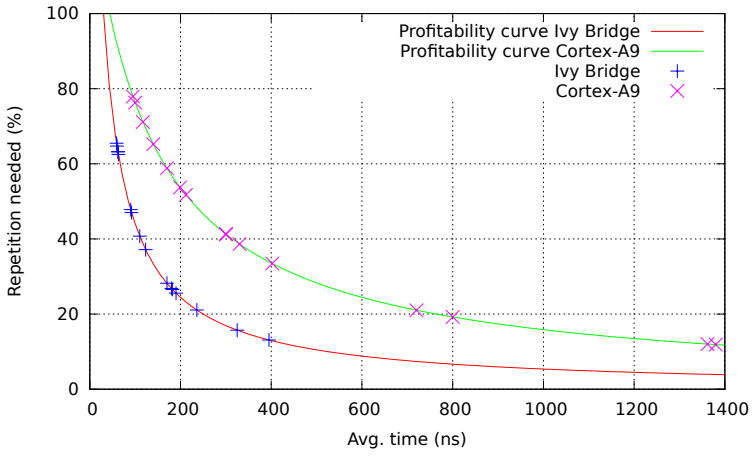


Figure 2.9: Profitability curve of memoizing transcendental functions
Intel Ivy bridge & ARM Cortex-A9 with GNU library

speed-up gain of up to 1.1. On `Intel IvyBridge`, `bwaves` has higher speed-up gains as compared to `tonto`. And as far as `gafort` is concerned, we can not run it on `ARM` due to memory limitation.

Among the other applications, `ATMI` and `population dynamics` achieve performance gain, but these gains are less as compared to those achieved on `Intel Ivy Bridge`. However apart from the observed variation in overall speed-up gain on both processors, `Population dynamics` achieves the same speed-up on memoization for all considered table sizes while the speed-up for `ATMI` reduces with the reduction in size of the memoization table. A similar behavior was observed on `Intel Ivybridge`.

2.3.4 Associativity

In an associative hash-table, we have multiple entries (n-entries for $n - way$ associative) stored at each hash location and the chance of collisions are minimized, while at the same time the overhead of table look-up is increased. Our finding shows that associative hash-tables are beneficial for long latency transcendental functions – specifically the *Bessel* functions. For other transcendental functions which have relatively low latency, our finding shows that associativity is causing a negative impact due to the extra overhead of associative table look-up. In this section, we present results for

Table 2.5: Profitability Analysis
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%

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`. 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 exactly in a cache line. Figure 2.10 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 get the best performance as we can capture most of the repetitions. A 256k non associative table actually shows worse performance as compared to a 64k non-associative table as the hash function works better for 64k table and a 256k non-associative table pollute the last-level cache a lot more. With associativity we can use the same hash-function for the 256k $, 4-$ way associative table and we can maintain the same number of cache line accesses.

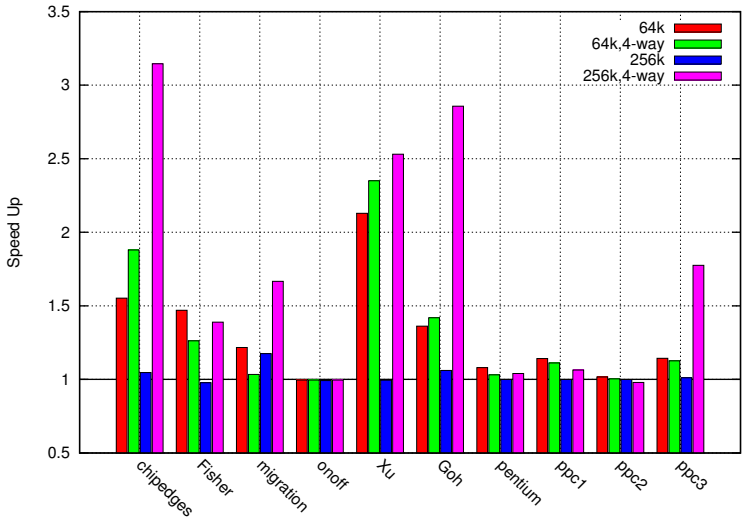


Figure 2.10: Associative table for ATMI (GNU compiler) on Intel Ivy Bridge

We apply the same mechanism of associativity on `ARM` platform also. Since the cache line size is 32 bytes, we use a 2-way associative table. Figure 2.11 shows the effect of an associative hash table for the Bessel functions in `ATMI` on `ARM` platform. As in the case of `Intel Ivy Bridge`, associativity shows very good performance on `ARM` platform. Again, a 2-way 32k table performs better than a 128k table.

Table 2.6: Function call analysis for ATMI
Intel Ivy Bridge with GNU compiler and 64k memoization table

Application	Func	No. of calls	Unique	Hits	Evictions	Time (s)	Speed-up	Modelled speed-up
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%			
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%			
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%			
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%			
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%			
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%			
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%			
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%			
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%			
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%			

2.3.5 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 2.7. We classify 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 memo-

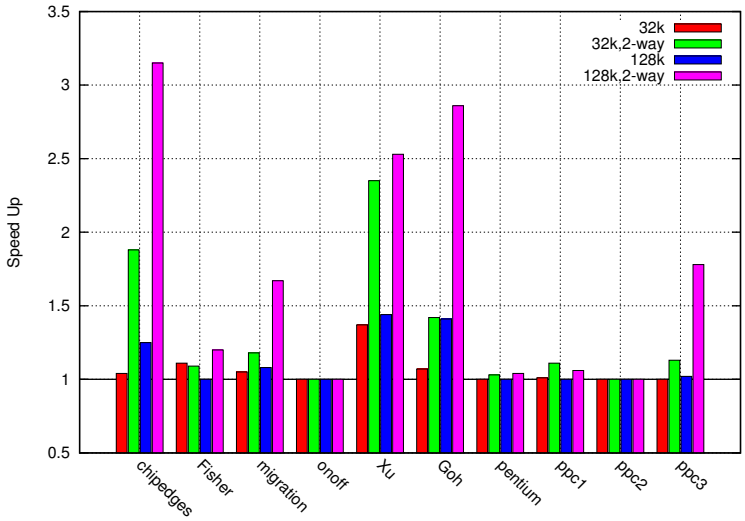


Figure 2.11: Associative table for ATMI (GNU compiler) on ARM Cortex-A9

ized 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` fall into the third category.

2.3.6 Hash-table Performance

We compare the performance of our hash function with a simpler approach by considering only the first 16 mantissa bits for indexing. The result is shown in Table 2.8. The result shows that *XOR* hash function is much better performing than the simpler method while adding two extra *XOR* operations.

Table 2.7: Function call-site analysis

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	
`fmm`	pow	
ocean_cp	sin	
`water_spatial`	exp	
GemsFDTD	exp	Single call-site
`equake`	cos	
`fmm`	log	
gamess	exp	Multiple call-sites differential behaviour
wrf	expf powf logf	

Table 2.8: Hash-function Comparison
Intel Ivy Bridge with GNU compiler and 64k table

Application	**Function**	**No. of calls**	**Unique values**	**Hit % - XOR**	**Hit % - Masking**
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
fmm	log	37,717,281	78	99.9	85.5
	pow	9,109,011	9,101,294	0	0
ocean_cp	sin	16,785,411	4,097	94.5	24.5
water_spatial	exp	1,039,992,980	12,538,545	98.2	94.5

2.4 Summary

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, showed good performance on memoization. And there is negligible slow down (expected to be below a few hundred milliseconds) but for that 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.Our findings also reveal that a memoization table of up to 64k size works well on `Intel Ivy bridge` as well as on `ARM Cortex A-9`.

In conclusion, the following characteristics are seen as benefits of memoization:

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 further. Memoization is guaranteed to achieve 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 certainly acceptable for a modern computer.Memoization 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.

Chapter 3

Function Memoization - A Generalized Approach

3.1 Introduction

In our first memoization technique [SSRS15] we intercepted functions at program start-up and provided a way for memoizing dynamically linked functions. A major restriction of the technique is that memoization is restricted only to dynamically linked functions and those functions must be determined beforehand. Here, we propose an extension for 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 library function. Our compile time technique allows static linking of memoization code and this avoids a function call overhead during a hit in memoization table, which increases the potential benefit due to memoization. We also handle some impure functions like those having pointer arguments, global variable usage etc.

In our previous work we have targeted transcendental functions for memoization. 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 applications critical to numerical computation (cases where a closed form solution is not available) and in case of recursions.

3.1.1 Advantages of Compile-time Approach

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 removed for the memoization purpose. 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 and no inter-procedural optimization etc. where inlining would not work. In order to make use of inlining for memoization, we provide an inline wrapper for memoization which in turn works like an inlined code except when memoization fails – i.e., a function call happens only during a miss in the look-up to memoization table. By doing this we increase the effect of memoization and lower the threshold needed for memoizing a function.

In addition to providing compiler optimization, inlining also saves a dynamic function call whenever there is a hit in the memoization 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. Now, the run time of the inlined memoization code made a time difference of 400 milliseconds which would mean $\frac{400.10^{-3}}{100.10^{6}} = 4$ nanoseconds on average for a single call. We ran our experiment on a CPU running at 2.3GHz and hence the run time amounts to $4 \times 2.3 \approx 9$ clock cycles which means, approximately 9 clock cycles are saved on average for a single call. So, even without considering compiler optimizations, which are possible on the memoized wrapper code, we are sure to save 9 clock cycles by inlining during a hit in the memoization table. Thus the analysis we have done for memoizing functions in 2 can be modified as shown in Table 3.1 using new *hit time* for the Equation (2) in Section 0.6. Figure 3.1 compares the profitability curve of our inline approach compared to `LD_PRELOAD` approach assuming no other compiler optimizations.

Thus in this 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.

The remaining part of the chapter is organized as follows: Section 3.2 describes our

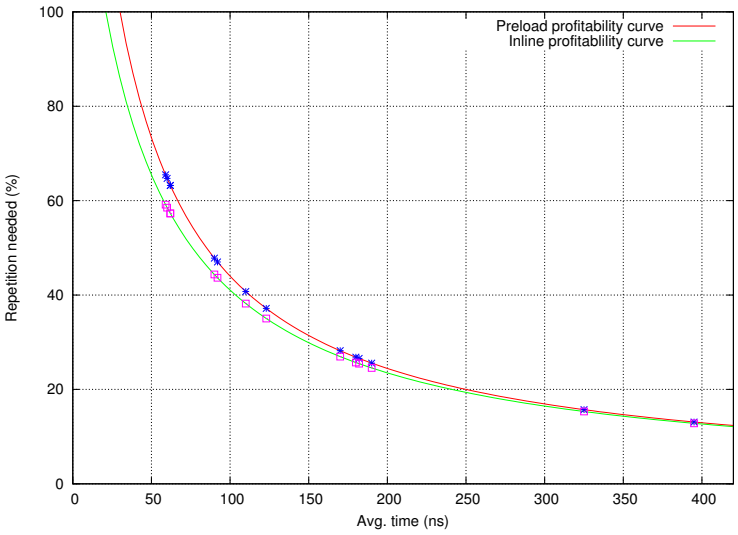


Figure 3.1: Profitability Curve - preload vs inline

implementation approach, how we are identifying potential candidates for memoization and the challenges involved here. Section 3.3 details the experimental set up used and the results of our memoization technique. We also compare the performance benefit of our approach with our previous technique in this section. Section 3.4 gives the summary of our technique and its evaluation.

3.2 Implementation

We implement our compile time memoization technique in `LLVM` [LA04]. `LLVM` is a modular compiler tool chain which provides Static Single Assignment (SSA) [CFR+91] based compilation support for both static as well as dynamic compilation of a variety of programming languages. Optimizations are enabled in `LLVM` via passes [llv] which are broadly classified into three types – Analysis Passes, Transform Passes and Utility Passes. Analysis passes compute information that other passes can use for debugging or program visualization purposes. Transform passes can use the analysis passes and they may mutate the program in some way. Utility passes provides some utility and are the ones which do not fit the other two categories. Further each pass traverse some portion of a program to either collect information or transform the program. For our technique we wrote an `LLVM` optimizer pass at module level since we would be looking at all functions in a module and possibly modify them (we can analyze functions using

Table 3.1: Profitability Analysis with Inlining
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	21	55	90	44%
	log			92	43%
	sin			110	38%
	cos			123	35%
	j0			395	13%
	j1			325	15%
	pow			190	25%
	sincos			236	21%
float	expf			60	59%
	logf			62	57%
	sinf			59	59%
	cosf			62	57%
	j0f			182	25%
	j1f			170	26%
	powf			180	27%
	sincosf			63	63%

a function level pass but modifying calls to functions is not possible here).

Since many of our benchmark applications are in Fortran, and `LLVM` does not have inbuilt front end for Fortran, we use `dragonegg` [dra] plugin of gcc for converting the source code to `LLVM IR`. `dragonegg` is a gcc plugin which replaces GCC's optimizations and code generation techniques with those from the `LLVM` project. Thus, we use gcc front-end for parsing the input source files and uses `LLVM` optimizer and code generator. We run our `LLVM` pass – a module pass – on each source module. In our module pass we iterate through each function and mark them if they are memoizable. If so, we create a template for the corresponding memoization wrapper and replace each call site to the function, with a call to the 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 being changed under the condition detailed in Section 3.2.1. The pseudo-code for our approach is given in Algorithm 2.

Memoizing small functions in software is counter-productive and so we run our `LLVM` pass at the last optimization stage. This ensures that small functions are inlined ahead and thus memoization would not be capturing them. Our pass generates the function

templates of all candidate memoizable functions. It also replaces calls to those functions with those to the memoized version at this stage. After this stage we link together all the generated `LLVM IR` files, including that of the memoized versions of functions, and make one single unit of `LLVM IR`. At this stage we run `LLVM` optimization there by providing a way for the memoized code to be inlined – the memoized part of even dynamically linked functions (memoization wrapper) can get inlined here. Finally we use `gcc` to generate the executable. A flow diagram of our approach is shown in Figure 3.2.

The generation of memoized version of the functions is done using an extended version of *if-memo* [SR] tool. The original tool was made to handle *math* functions with predefined signatures. Now, this is extended to handle more generic function signatures as detailed in Section 3.2.1. The default tool handled cases for *int, float, double* which we extended for *pointers* also. For each function prototype considered, the tool now provides a ready template for the memoized version of the function. i.e., the tool does the memoization table initialization and hashing mechanism and table access mechanism for the corresponding function prototype.

Algorithm 2 Pseudo-code for LLVM Module Pass

Require: Source Module

Ensure: Function calls are replaced by their memoized wrapper and function meta for memoization is produced

```
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)
        outputFunctionMeta(f); ▷ For input to If-memo tool
    end if
end for
```

For our work we considered functions with up to 3 parameters as with an increase in number of parameters the chance of repetition goes down and the overhead of memoization increases. Constant arguments at call sites are excluded from this limit as per the algorithm detailed in Algorithm 7. Our handling of function parameters is detailed in next section. Section 3.2.2 details our mechanism for handling pointers and Section 3.2.3 details our global variable handling mechanism.

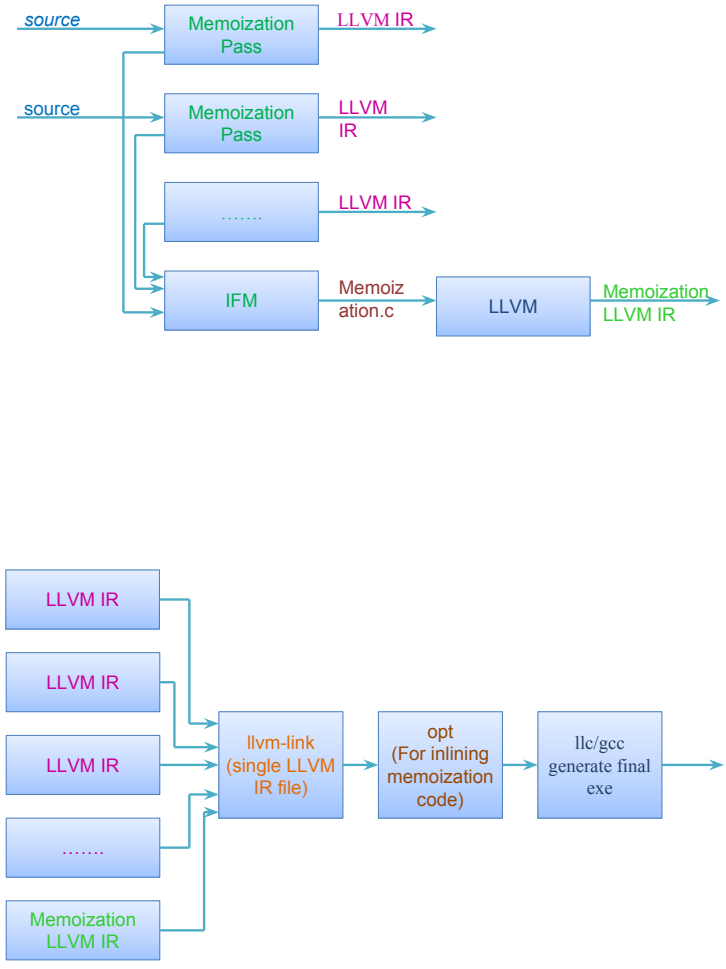


Figure 3.2: Flow Diagram of Memoization

Algorithm 3 Pseudo-code for identifying memoizable functions

Require: Function f
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, count) then
        return true;
    end if
    return false
end function
```

3.2.1 Function Templates and Parameter Ordering

In order to generalize the handling of function types for *Ifmemo* tool we make use of a set of custom function templates which are given in 3.2.1. Here, the first character represents the return data type and then each letter represents the corresponding argument type with their encoded meaning detailed as shown below:

Algorithm 4 Pseudo-code for identifying if input function is safe for memoization

```
function checkFunctionCalls(f, count)
    for each instruction I in f do
        if isCallinstruction(I) then
            calledFunction ← getCalledFunction(I)
            if isPure(f) OR calledFunction = f then
                continue
            end if
            count++
            if count = 10 then  ▷ We only check up to 10 levels in case of recursions
across different functions
                return false
            end if
            if ismemoizable(f) then
                continue
            end if
            return false
        end if
    end for
end function
```

Generalized Function Prototypes

```
ii    - int,int
ff    - float,float
dd    - double, double
iii   - int, int, int
fff   - float, float, float
ddd   - double, double, double
vij   - void, int, int*
vijj  - void, int, int*, int*
viijj - void, int, int, int*, int*
vfg   - void, float, float*
vfgg  - void, float, float*, float*
vffgg - void, float, float, float*, float*
vde   - void, double, double*
vdee  - void, double, double*, double*
vddee - void, double, double, double*, double*
```

Algorithm 5 Pseudo-code for 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 OR I ≠ StoreInstruction then
            return false
        end if
    end for
    return true
end function
```

Argument Type Encoding

```
v - void
i - integer
f - float
d - double
I - read only pointer to int
F - read only pointer to float
D - read only pointer to double
j - write only pointer to int
g - write only pointer to float
e - write only pointer to double
J - read/write pointer to int
G - read/write pointer to float
E - read/write pointer to double
```

In order to reduce the number of function types to be handled, we sort the parame-

ters to functions and our function prototypes are generated as per this new order. This ensures that for all the permutations of a given set of parameter types, we need just one memoization scheme. We generate the call string for the original function during this stage itself and hence it preserves the actual call order. This is done by the following naming mechanism:

Let the original function prototype be

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

Now, after sorting suppose their order become `arg2`, `arg3`, `arg1`. So, now our generated prototype for function `fun` will be

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

and the generated call string to original function will be

```
fun (arg2, arg3, arg1)
```

i.e., the call string is generated based on the relative ordering of the parameters before sorting to their original order after sorting as explained in detail in Algorithm 7. Thus, there would not be any effect of sorting when the call reaches the actual function after going through the memoization wrapper.

3.2.2 Handling Pointers

Pointers are always tricky to handle for memoization. A pointer could mean the memory being pointed could be modified across the function call which means we must store the pointed value in our memoization table (`READ`). The pointer could also be used to return a value (`WRITE`) from a function. For memoization purpose we are not interested in the pointer value but rather we look at the pointed value. We have taken a conservative approach and handle only up to one `READ Only` or `READ/WRITE` pointer while allowing up to two `WRITE Only` pointers. Also we only handle cases where the pointed value is an `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`.

3.2.2.1 Read Only Pointers

`READ Only` pointers are those which are used in a function but their pointed location 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. For memoization purpose the pointer value has little significance and we need to care only for the pointed value. The pointer type and Read Only information is passed using the corresponding function template as detailed in Section 3.2.1.

3.2.2.2 Write Only Pointers

For `WRITE Only` pointers we use the corresponding template as given in Section 3.2.1. Now these parameters will not be used for indexing into the memoization table but only for passing back the return value – i.e., 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 the sin and cos values for a given input argument.

3.2.2.3 Read/Write Pointers

`READ/WRITE` pointers are the most difficult to handle. They require both storage space and also causes extra overhead in look-up and further reduces the chances of repetition. Like `WRITE only` pointers these pointers are also handled using the corresponding function template with the extra condition of being part of the table indexing process.

3.2.3 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 Pointer handling are re-used here. We have considered maximum 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

Algorithm 6 Pseudo-code for checking if a function is Global Variable safe for memoization

```
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 Op of I do
              if isGlobalVariable(Op) then    ▷ All global variables are pointers in
  LLVM
                  ptype = getType(Op)
                  optype ← getType(ptype)
                  if optype = Integer OR optype = float OR optype = Double then
                      if global = Null then
                          global ← Op
                          continue
                      end if
                      if global = Op 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
```

in Algorithm 6 and the procedure for adding it in the memoized wrapper function is given in Algorithm 7.

3.2.4 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 as they are sure to repeat and hence no need to be saved in the memoized table. Further, we can avoid the passage of these constants to the actual call, as long as there is a hit in the memoization

Algorithm 7 Pseudo-code for 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(globals,x) then
                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
```

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, now the memoization wrapper becomes call site specific (a separate memoization table is used per call site in-case any argument is a constant) but our results show that this is not very problematic.

3.3 Experimental Results

3.3.1 System Configuration

We carried our experiments on an `Intel i7` machine under the set-up as described in `Table` 3.2. We made sure that the Turbo Boost feature [Int] is turned off and CPU clock frequency was set at 2.3 GHz to ensure reproducible time measurements. For our experiments we have used a 64k entry memoization table per function and the memory overhead for this is discussed later in Section 3.3.4.

Table 3.2: Experimental set-up on Intel Ivy Bridge

Processor	Intel Core i7
L3 cache	8 MB
Clock speed	2.3 GHz
RAM	8 GB
Linux version	3.19
llvm version	3.7
optimization flag	-O3

We included all the benchmarks used in our previous work as detailed in Section 2.3.1. In addition we include 2 more benchmarks which have memoizable functions but not dynamically linked and hence could not be used in our previous work:

1. blacksholes
   Blackscholes [BKSL08b] is taken from PARSEC [BKSL08a] benchmark suite. This involves option pricing with Black-Scholes Partial Differential Equation (PDE). Black-Scholes equation does not have a closed form solution and hence must be computed numerically.

2. histo
   This is a histogram application taken from Parboil benchmark suite [SRS+12] which accumulates the number of occurrences of each output value in the input set.

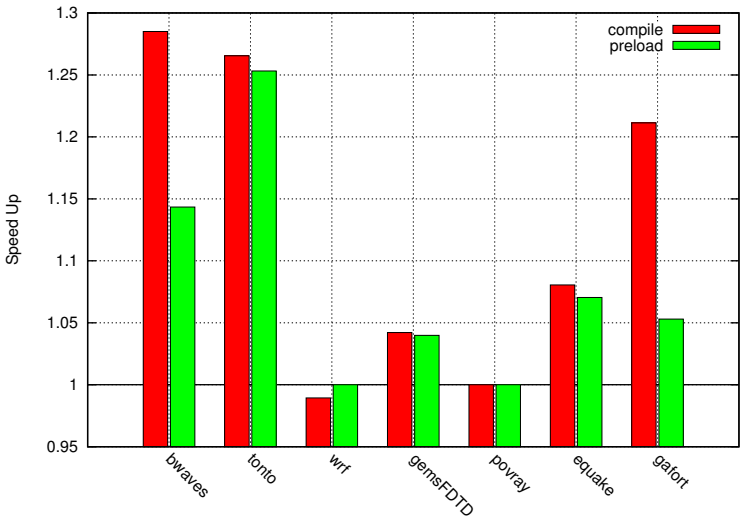


Figure 3.3: Speed-Up by Memoization for SPEC benchmarks
64k entry table, Intel Ivy bridge 2.3 GHz.

3.3.2 Discussion of Results

The result of memoization on the selected benchmark applications is shown in Table 3.3 and the speed up comparison with our previous technique is shown in Figures 3.3 and 3.4. Doing memoization at compile time, gives much better speed-up compared to our previous load-time approach. A major part of this is coming from memoization wrapper getting inlined and thus providing more optimization opportunities which can no way be done otherwise for any dynamically linked functions. We are also able to catch more memoization opportunities as marked by boxed functions in `Table` 3.3. In `equake` benchmark we are capturing *phi0*, *phi1* and *phi2* functions where a global variable is being used. With our scheme, we are capturing them as extra argument and hence these functions are getting memoized. In `histo` benchmark `HSVtoRGB` is a user function taking in 3 float type arguments and returning a structure to 3 char type variables as shown in Code 3.3.2. But this is converted to an `int` return type by `LLVM` as per `x64` Linux ABI [Fox] 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)
```

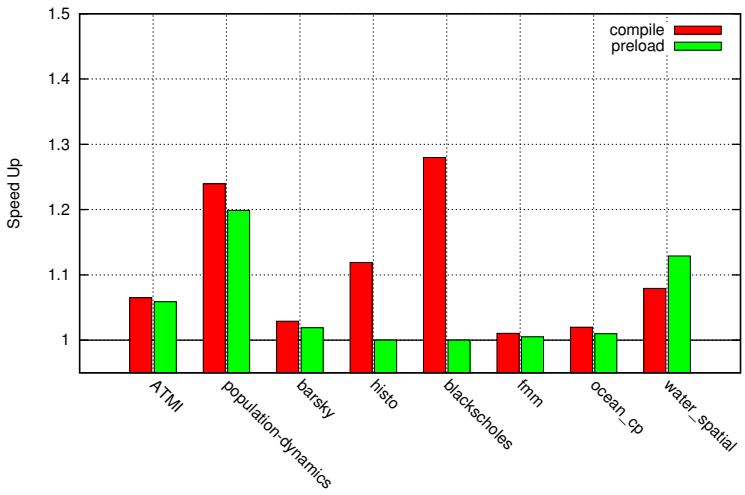


Figure 3.4: Speed-Up by Memoization for Selected Applications
64k entry table, Intel Ivy bridge 2.3 GHz.

3.3.2.1 Speed-up in equake

In `equake` benchmark of `SPEC 2000` we have three user defined functions which are critical – *phi0*, *phi1* and *phi2* – all of which have similar code structures, the code of *phi0* being given below:

```
double phi0(double t)
{
  double value;
  if (t <= Exc.t0) {

    value = 0.5 / PI * (2.0 * PI * t / Exc.t0
        - sin(2.0 * PI * t / Exc.t0));
    return value;

  }
  else
    return 1.0;
```

```
}
```

Here, we have only one input argument and the global variable is being read inside the function. So, we include the global variable as an extra argument and generate our memoization wrapper which has the signature as follows:

```
double 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 code provides a tricky case for memoization. There is just an *if-else* block inside the function with the *if* block being expensive and the *else* block being very simple (just a return statement). On execution we found that the *else* block has taken 90% of the total time – 284 million calls to *if* 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 speed up to come down. In order to avoid this we did a memoization only for the *if* part of the function and this gave a performance gain of 3.5% over the normal compile-time approach.

3.3.2.2 histo

In `histo` benchmark we have a function called *HSVtoRGB* and in the only call-site of this function we have the following call:

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

The first two arguments are constants here and hence removed from the call site and the call site is modified to call the function `__memoized_HSVtoRGB(double)` having a single argument as follows:

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

We in turn call `HSVtoRGB` from inside `__memoized_HSVtoRGB` as follows:

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

thus saving the passing of first two parameters and avoiding their usage in the table indexing process of memoization.

3.3.2.3 blackscholes

In `blackscholes` benchmark from `PARSEC` suite there is a function *CNDF* (Cumulative Normal Distribution Function) which performs numerical computation on an input

floating point data and is expensive as well as critical. The code of this function is given in Appendix 3. 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.

3.3.2.4 wrf

In `wrf` we have a performance loss as the repetition is very low for *powf* function. The extra performance gain we achieved in our previous work is due to the performance bug in *libm* implementation of *powf* function.This function is only partially visible in the latest *libm* version which we used for our current experiments. For our *preload* approach there is no slowdown as we have a turn off mechanism for memoization. A turn-off mechanism for our compile-time approach is discussed in Section 3.3.3.

3.3.2.5 water_spatial

Here also we did not get as much speed up as was achieved with our preload technique. We attribute this to the numerous call sites which calls the *exp* function and hence inlining probably is causing a relative slowdown.

3.3.2.6 Transcendental Functions

For transcendental functions in all other applications we get better performance as compared to our results in previous work and this is demonstrated in `Figures` 3.3 and 3.4. This increase in speed-up is coming from saving a call during hit-case 3.1.1 and due to possible compiler optimization after inlining.

3.3.3 Turning off Memoization

Being a compile time technique turning off memoization is not straightforward in this approach. In our previous work we used a helper thread to monitor the memoization behaviour and the thread used to turnoff memoization whenever performance deteriorates due to memoization. Using a runtime code modification tool like `PADRONE` [RRC+14] we can employ a similar technique here. PADRONE is a platform for dynamic binary code analysis and optimization. It does not require any source code and provides an API to analyze and optimize binary executable while program runs. In order to turn-off memoization we propose a helper thread to monitor the memoization

Table 3.3: Calls to memoized functions
All run times averaged for 3 runs

Application	Func	No. of calls	Hits	Normal	Memoized	Preload	Gain	Preload Gain
bwaves	pow	216,320,000	60.15%	550	428	481	22.18%	12.55%
	sqrt	54,080,000	43.32%					
gems.FDTD	exp	711,400,841	99.91%	495	475	476	4.04%	3.84%
equake	sin	530	52.26%	227.7	210	213	7.77%	6.46%
	cos	280	9.64%					
	phi0	3,788,227,494	100.00%					
	phi1	3,788,227,494	100.00%					
	phi2	3,788,227,494	100.00%					
tonto	exp	569,927,519	80.02%	777	614	620	21%	20.2%
	sincos	452,321,280	100%					
wrf	log10f	33,505,590	24.39%	467	472	467	-2.14%	0.00%
	logf	80,425,135	44.97%					
	powf	1,521,585,822	14.66%					
povray	sin	5,241,798	0.00%	203	203	203	0.00%	0.00%
	pow	30,664,910	23.75%					
gafort	gfortran_pow [a]	36,794,622,556	100.00%	1653	1377	1567	16.70%	5.20%
	exp	2,200,000,000	97.21%					
	sin	2,200,000,000	97.10%					
barsky	sin	26,251,805	98.2%	21.2	20.4	20.6	3.77%	2.83%
ATMI	exp	129,606,540	6.81%	36	33.8	34	6.1%	5.6%
	j0	53,110,575	6.32%					
	j1	33,095,178	0.22%					
	log	35,156,814	27.22%					
Population dynamics	exp	28,744,800	100.00%	8.8	7.1	7.34	19.3%	16.6%
	log	28,744,800	100.00%					
histo	HSVtoRGB	5,179,200	100.00%	4.7	4.2	4.7	10.64%	0.00%
blackscholes	CNDF	2,000,000,000	94.10%	151	118	151	21.85%	0.00%
water spatial	acos	5,000,000	77.20%	245	227	217	7.35%	11.43%
	exp	1,039,992,980	98.19%					
	pow	9,000,005	98.52%					
fmm	log	37,717,281	100.00%	192	192	191	0.00%	0.52%
	pow	9,109,011	0.00%					
ocean_cp	sin	16,785,411	94.51%	103	101	102.7	1.94%	0.29%

[a] Full Name: gfortran_pow_i4_i4

behaviour.So whenever deterioration in performance is observed due to memoization , the thread uses the function call replace API of `PADRONE` to turn off memoization. To facilitate this, we have to keep a copy of all functions which calls our memoized function before their call sites were modified. Now, to turnoff memoization, all those function bodies will be replaced with their original versions and thus they continue calling the original function and not their memoized version. This approach has the disadvantage

of further increasing the code size but is particularly useful in cases where memoization produces negative results.

3.3.4 Memory Overhead

For our experiments we have used a $64k$ entry table size for each function. This would mean 64 tag bits and 64 data bits for a function taking a double argument and returning a double and would amount to $128 \times 64k = 8Mbits = 1MB$ memory per function. For functions taking two double arguments this would be $1.5MB$ and for *integer/float* argument functions these corresponds to $512KB$ and $384KB$ respectively on Intel Ivybridge. These tables are one time initialized and only in case of continuous usage, they would be causing any overhead.

In addition to memory overhead for the memoization table, inlining can also cause an increase in code size. Figure 3.5 shows the percentage change in the code size (text segment) due to memoization on our selected applications.

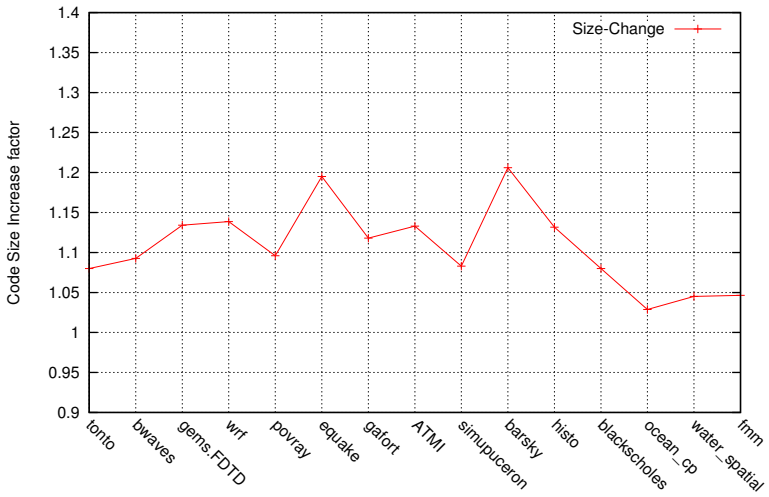


Figure 3.5: Code size change due to Memoization
64k entry table, Intel Ivy bridge 2.3 GHz.

3.4 Summary

In this work we have proposed a compile time approach for memoization as compared to our previous work where a load time approach was proposed. Doing memoization at compile time enables to capture more candidate functions for memoization and enables memoization opportunity for both statically linked as well as dynamically linked functions. With the help of `LLVM` framework we are able to support pointers and global variables for the function 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 [Fox]. 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 our load time approach as shown in Figures 3.3 and 3.4. 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 function call as well as by enabling other compiler optimization. Inlining causes a slight increase in code size but still is good for performance. We have used a $64k$ entry table for memoization but these entries are being used only when indexing happens to them and hence in case of function being non-critical we would not be losing any performance.

Chapter 4

Memoization with Hardware Support

4.1 Introduction

In the previous two chapters we concentrated on memoization assuming no help from hardware. In this chapter we discuss our memoization technique and proposes possible hardware changes that can boost memoization benefits. By doing memoization in hardware we have the following advantages:

1. **Faster Hashing**: A hardwired hash function would enable to get the hash done in a single cycle compared to about 10 clock cycles for our software implementation.

2. **Faster Table Look-up**: Our software implementation table usually hits in lower level cache and this adds tens of cycle delay in getting the required data. With a hardware table this latency can be avoided.

3. **No need of Dynamic Turn-off**: In our software implementation, table look-up and actual function call execution on look-up failure is done sequentially as doing a parallel implementation is not practical due to high software threshold. This can cause slowdown in case of low repetition rate for memoization which necessitate the need of a turn-off mechanism for memoization. But with an almost zero overhead look-up mechanism in hardware, we can practically do away with the turn off mechanism and even do a parallel look-up and subsequent function execution.

4. **Associative Table**: For software memoization, doing an associative implementation was expensive. But once done in hardware we can do the look-up in parallel for all the set entries and hence an associative implementation can be done without any significant overhead.

4.1.0.1 Reduced latency

With hardware memoization, we can effectively do a hash computation in 1 *cycle* and a table look-up in about 4 *cycles*. This greatly reduces the required repetition for achieving a potential performance gain for a memoized function as shown in Table 4.1 for *libm* functions. The hit time becomes just 2 nanoseconds (on 2 GHz machine) now as compared to 21 nanoseconds for our compile time approach. There is no miss-overhead as with a branch predictor on a successful prediction of a miss, table look-up would not cause a slowdown for actual execution. But branch predictor causes a penalty in case of mis-prediction on the newly created table look-up branch. The hit-rate required for hardware memoization to be profitable is given in `Equation` 4.1, where t_h is the hit time, T_f is the actual function execution time and t_{pen} is the effective penalty per function call due to hit/miss-prediction on the table-lookup branch .

$$\begin{aligned} H \times t_h + (1-H)(T_f) + t_{pen} &< T_f \\ H \times t_h + T_f - H \times T_f + t_{pen} &< T_f \\ t_{pen} &< H(T_f - t_h) \\ \implies H &> \frac{t_{pen}}{T_f - t_h} \end{aligned} \tag{4.1}$$

The profitability curve of our hardware approach vs our compile time approach is shown in `Figure` 4.1 assuming 80% successful hit/miss prediction and a 20 cycle mis-prediction penalty (which amounts to $0.2 \times 10 = 2ns$ per function call on a $2GHz$ machine), and we can see a great advantage for hardware memoization when the function execution time is less.

But doing memoization on hardware has the following restrictions:

1. **Smaller Table Size**: In software we could use a table as large as $64k$ entries without any significant performance slowdown. But such a table is not possible in hardware due to practical limitations and we have to restrict the table size to a few kilobytes in size.

Table 4.1: Profitability Analysis for Hardware Memoization
2 GHz Ivy Bridge, GNU lib

Function		**Hit Time (ns)** t_h	**Mis-prediction Penalty (ns)** t_{pen}	**Avg. Time (ns)** T_f	**Repetition Needed** H
double	exp	2	2	90	2%
	log			92	2%
	sin			110	2%
	cos			123	2%
	j0			395	1%
	j1			325	1%
	pow			190	1%
	sincos			236	1%
float	expf			60	3%
	logf			62	3%
	sinf			59	4%
	cosf			62	3%
	j0f			182	1%
	j1f			170	1%
	powf			180	1%
	sincosf			63	3%

2. **Global Memoization Table**: In software we were using a single table for each memoized function. But in hardware we have to use a global memoization table and all memoized functions must share it or else implementation will not be practical. Further hardware memoization table is shared by all processes on a multi-processing environment and hence we need a mechanism to ensure independent usage of memoization table by each process.

Implementing memoization in hardware was discussed by Citron et.al. in their work [CF00] done in year 2000. They argued hardware memoization is much more beneficial to software memoization as per then architectures. For hardware memoization of user defined functions they proposed two new instructions `LUPM` and `UPDM` which does the memoization table look-up and memoization table update respectively. Both these instructions had two variants - one for single input functions and other for double input functions. In the following section we propose a similar hardware mechanism for memoization to extend our compile time approach and conclude it with an empirical analysis of the potential performance benefit assuming current architectures. Our proposal has

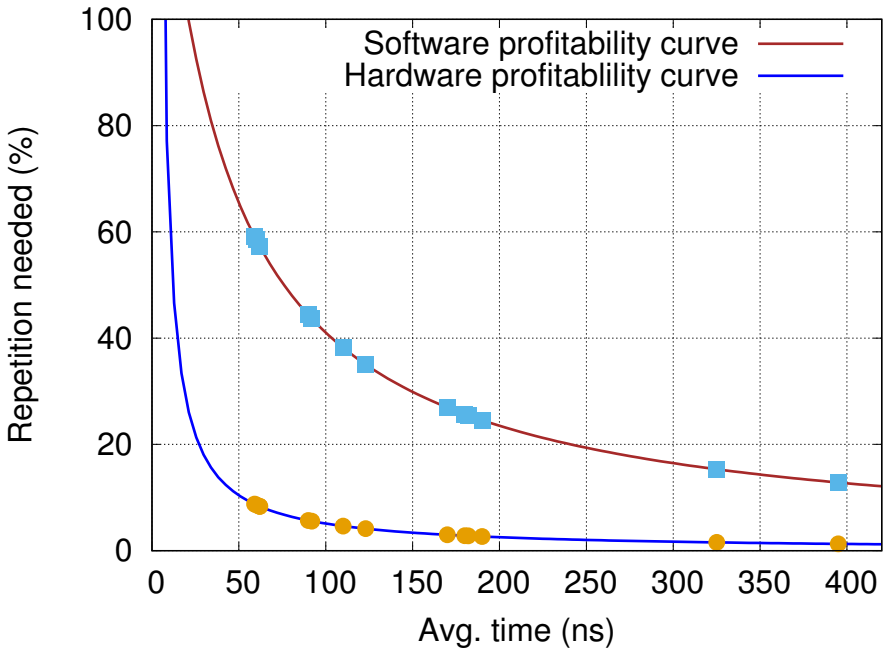


Figure 4.1: Profitability Curve- Software vs Hardware

a new `CALL` instruction specific to each function prototype we handled for our compile time approach and this allows to capture more types of user defined functions. More over we could implement the hardware approach as an extension to our compile time approach.

4.2 Implementation Details

We propose our hardware implementation as an extension to our compile time approach. To enable this we had to do the following changes to our implementation.

4.2.1 New Instructions

To facilitate memoization in hardware we propose the following two instructions:

`MSCALL`: This instruction is used for a memoized function with a single input argument and this carries one extra argument - an $8 - bit$ function identifier. During such a call, CPU does a look-up in memoization table using a hardwired hash function

and on success returns the value from table. If the look-up fails, a `JMP` instruction is invoked to execute the original function. Update to the memoization table is ensured by our next instruction `MSUPDATE`. We also need different variants of the instruction to handle the different function templates as detailed in Section 3.2.1. For pointer arguments, we assume that they are `WRITE Only` as `READ ONLY` pointers can be converted to corresponding type using a LOAD instruction by the compiler. Similarly, `READ/WRITE` pointer arguments can be changed 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

`MSUPDATE`: This instruction follows the `MSCALL` always and does an update to the memoization table at the given index location. But whenever there is a hit in the memoization table, the return address is modified by the CPU to ensure that this instruction is skipped from being executed. For each of the variant of `MSCALL`, we have a corresponding `MSUPDATE` instruction variant.

The sequence of operations initiated by `MSUPDATE` is shown in Figure 4.2. The mis-prediction penalty is associated with a hit/miss prediction on the hardware look-up table. For every mis-prediction- hit predicted and miss happens or a miss predicted and a hit happens, we have a penalty for flushing the instruction pipeline.

4.2.2 Hardware Table

We propose a centralized memoization table for memoization and fix the size of the table to 256 sets and $4-way$ associative (1024 entries) which amounts to $4\times4\times8\times256 = 32kB$ of memory as in each entry, we allow two 8-bytes arguments and two potential 8-bytes results. Further to identify a function, we need to store the function identifier for each entry in the table. Since using the function address is expensive in storage as it requires $64-bits$ on x64, we propose to number the functions from 0 and limit the memoization to up to 256 candidate functions. By doing so, we require only $8-bits$ for identifying the function in the memoization table and this only adds an overhead of 8-bits per entry and thus 1024 bytes in total for 1024 entries. Our assumption is that 256 is a

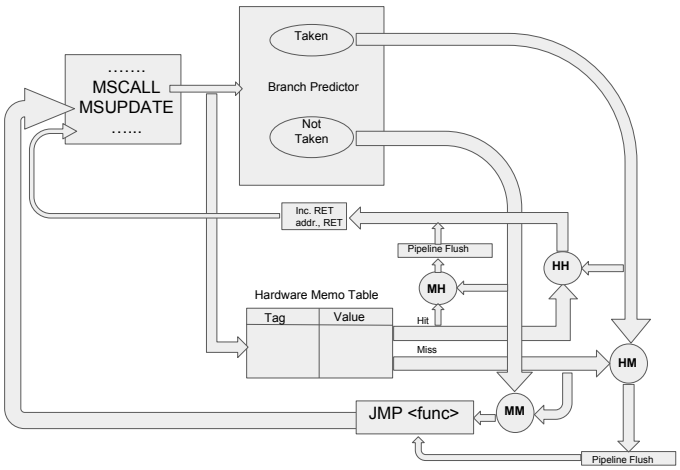


Figure 4.2: Working of MSCALL/MSUPDATE

good enough number for the amount of candidate memoizable functions and in case there are more functions in any application we skip the remaining ones. For associative implementation we require 3 more bits per set and this amounts to $3 \times 256 = 768$ bits $= 96$ bytes for the whole table. Thus, the overall storage requirement will be $32000 + 1024 + 96 = 33120$ bytes which is comparable to an $L1$ cache size on modern architectures. For hashing we use the same *XOR* function as used in our compile time approach with the only difference of adding the function identifier to the hash input.

A set entry of our memoization table is shown in Figure 4.3. Shown are 4 set entries and the upper part of each row shows the required storage in bits. The 3 LRU bits are common for the 4 set entries. Individually each entry has a valid bit, a process identifier field (ASID) which is discussed in Section 4.2.4, a function identifier field of 8 bits, 2 input arguments of 64 bits and 2 possible results also of 64 bits.

4.2.3 Associative Storage

For most part of our software implementation we have used a direct-mapped memoization table as doing an associative look-up was counter-productive unless we needed an extremely large sized table as required for Bessel functions in `ATMI`. Being done in hardware, an associative implementation makes more sense as it enables to capture more hits compared to a direct mapped table for same size. This is particularly useful as we have a small table size of 256 sets in hardware. We experiment with 4-way associativity supported by a pseudo-LRU policy as detailed in [lru] which requires 3-bits

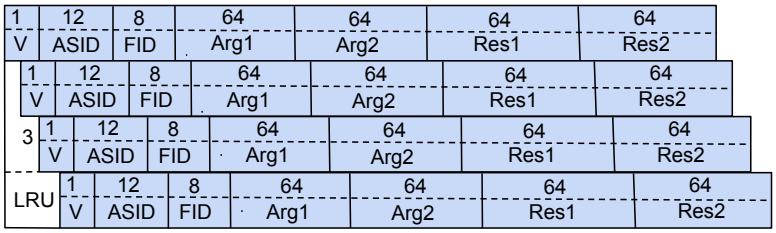


Figure 4.3: A set entry in Hardware Memoization Table

to maintain the last-use information for the 4 set entries.

4.2.4 Multi-processing Environment

With a global hardware table we do have an issue on multi-processing environment where more than one process might be using the memoization table and hence we have to ensure independent usage of memoization table for each process. The simplest way would be to flush the table on every context switch but this is expensive. Another alternative is to use `ASID` (Address Space Identifier) which is used by Translation Look-aside Buffer to avoid more or less the same issue for `TLB` access. ASID can be looked upon as a hardware variant of process id and is supported by quite a few architectures [JM98]. x64 architecture uses a different variant - Process Context Identifier (PCID) which is of 12 bits and can be used to distinguish up to 4096 processes at a time.

Using hardware process identifier is more efficient as it avoids an extra memory access each time memoization table is accessed. Also, it reduces the storage requirement on the table as assuming a 12 bit identifier, we now require $1024 \times 12 = 12288$ bits $= 1536$ *bytes* which amounts to 4.7% storage overhead.

4.2.5 Software Support

In the software part we very much use the same scheme with LLVM. During the link time we no longer need to link to the memoized wrapper functions. Instead now LLVM must generate 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 2 input parameters, the compiler enables only software memoization for it as we had done in our previous compile-time work.

4.3 Analysis of Potential Benefit

We now discuss the potential benefit of hardware memoization on SPEC benchmarks. We used the same set of benchmarks we used in our compile time technique. Lower latency functions like *sqrt* is now enabled for memoization. Our memoization table is now made a global one for all functions and restricted to 256 sets and $4-way$ pseudo-LRU associative. So, our hash index is now an 8 bit number. Rest of the experimental set up is same as we used for our compile time approach. By running the benchmarks using this global memoization table we obtained the hit rate of each functions. Using this we estimate the potential hardware gain using the following estimation method.

4.3.1 Hit/Miss-prediction

In hardware a `MSCALL` instruction causes a branch which basically does the look-up in memoization table. So, this can cause a penalty on every mis-predicted branch. To model this behaviour in our experimental set up, we counted the no. of hit/miss predictions measured using a $4-bit$ counter predictor for each memoized function. We assumed a 20 cycle penalty for mis-prediction.

4.3.2 Evaluation

- For each memoized function we obtained the actual execution time - f_{time}
- We estimated the execution time of the application excluding the time for the memoized functions.

  $$E_{time} = T_{time} - \sum_{i=1}^{n} f_{time_i}$$

  where E_{time} is the execution time of the application excluding that of the memoized functions and T_{time} is the total runtime of the application.
- Now, for each memoized function we estimate the hardware memoized time using the equation

  $$hw_{time} = N \times (h_{rate} \times ht + (1 - h_{rate}) \times f_{time}),$$

where N is the total number of calls to the memoized function, h_{rate} is the hit rate in the memoized table and ht is the hit time. We assumed $ht = 2ns$.

- The hit/miss-prediction penalty for each function is calculated as

$$P_b = N_m \times P,$$

where N_m is the total number of mis-predicted branches for the function and P is the mis-prediction penalty (we assume $P = 10ns$).

- Now, the estimated runtime assuming hardware support for memoization is derived as

$$H_{time} = E_{time} + \sum_{i=1}^{n} hw_{time_i} + P_{b_i},$$

where hw_{time_i} is the estimated hardware runtime and P_{b_i} is the average branch mis-prediction penalty for the i^{th} memoized function.

The numbers so derived are shown in Table 4.2. The speedup comparison of our compile time software approach and the hardware approach is shown in Figures 4.4 and 4.5 for SPEC benchmarks and other selected benchmarks respectively. In the figure for SPEC benchmarks we can see a reduction in performance gain for `bwaves` and this is due to the reduction in hit rate in hardware memoization table (due to reduced size) and higher mis-prediction rates for memoized functions as shown in Table 4.2. For other SPEC benchmarks, hardware approach gives better performance gain due to reduced memoization latency. `equake` has functions *phi0*, *phi1* and *phi2* all three have 3 input parameters including a global `READ Only` parameter and thus hardware memoization is not enabled (only functions with up to 2 parameters considered for hardware memoization) and we get the same software gain. For other selected benchmarks, hardware memoization gives better performance gain except for `ATMI` and `ocean_cp`, where a smaller hardware table significantly reduced the hit rate in memoization table.

4.4 Conclusion and Future Work

With hardware support we are able to bring down the required latency for memoization and this benefit many applications as shown by our estimation of performance using a modeled experimental set up. But the restricted table size on hardware brings down the memoization gain for few functions where a large memoization table is required. In order to handle such scenarios we propose a hybrid memoization approach where we can handle the miss in hardware table using a second level software table there by giving us the best of the two memoization techniques.

Table 4.2: Comparing Hardware-Software Memoization

Application	Func	Hits (Software)	Hits (Hardware)	Mis-prediction	Normal Time	Software Gain	Hardware Gain
bwaves	pow	60.1%	51.8%	28.9%	550	22.18%	20.0%
	sqrt	43.3%	35.7%	23.2%			
gems.FDTD	exp	99.9%	99.7%	0.3%	495	4.04%	6.4%
equake	phi0	100%	-	-	228	7.7%	7.7%
	phi1	100%	-	-			
	phi2	100%	-	-			
tonto	exp	80.0%	13.8%	9.5%	777	21%	35.1%
	sincos	100%	99.7%	0.3%			
	sqrt	73.1%	15.8%	13.8%			
wrf	log10f	24.4%	16.4%	2.2%	467	-2.14%	0.00%
	logf	44.9%	22.3%	15.3%			
	powf	14.7%	0.7%	4.9%			
povray	sin	0.00%	0.00%	0.00%	203	0.0%	0.4%
	pow	23.7%	14%	9.4%			
	sqrt	13.6%	12.5%	12.5%			
gafort	gfortran_pow [a]	100%	100%	0	1653	16.70%	52.9%
	exp	97.21%	53.3%	14.8%			
	sin	97.10%	54.8%	14.1%			
barsky	sin	98.2%	96.1%	3.9%	21.2	3.77%	5.7%
ATMI	exp	71.2%	18.8%	5.4%	36	6.1%	4.1%
	j0	20.3%	4.9%	4.9%			
	j1	84.4%	40.0%	15,6%			
	log	77.4%	46.7%	15.0%			
	sqrt	54.2%	20.9%	9.8%			
Population dynamics	exp	100%	99.9%	0.01%	8.8	19.3%	32.7%
	log	100%	99.9%	0.01%			
histo	HSVtoRGB	100%	100%	0	4.7	10.64%	14.2%
blackscholes	CNDF	94.10%	76.25%	20.6%	151	21.85%	59.1%
water spatial	acos	77.20%	73.5%	4.0%	245	7.3%	11.9%
	exp	98.19%	72.2%	21.8%			
	pow	98.52%	98.5%	0.2%			
	sqrt	96.7%	66.3%	25.1%			
fmm	log	100%	99.2%	0.8%	192	0.0%	0.3%
	pow	0.00%	0.00%	0			
ocean_cp	sin	94.51%	18.6%	7.7%	103	1.94%	0.4%

[a] Full Name: gfortran_pow_i4_i4

4.4.1 A Hybrid Approach

Doing memoization in hardware offers good performance benefit as it lowers the minimum time needed for memoization to be effective as shown in Figure 4.1. But it has its own challenges and the potential bottleneck of a smaller global hardware table can bring down the performance gain as compared to software memoization as shown in Figures 4.4 and 4.5. To do away with this, we propose a hybrid technique where we use both the hardware table as well as a software table. Hardware table works exactly as described here. The software table is just an enlarged version of the hardware table-$64k$ sets compared to 256 sets on hardware table and we propose the same $4 - way$

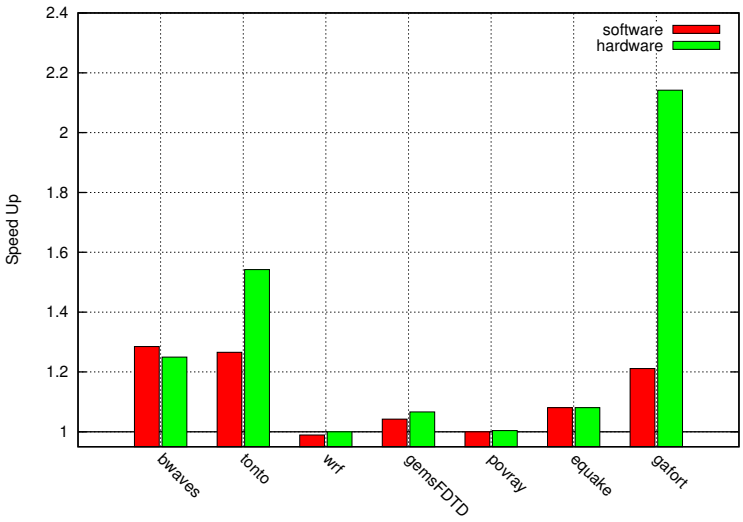


Figure 4.4: Speed-Up by Hardware Memoization for SPEC Benchmarks

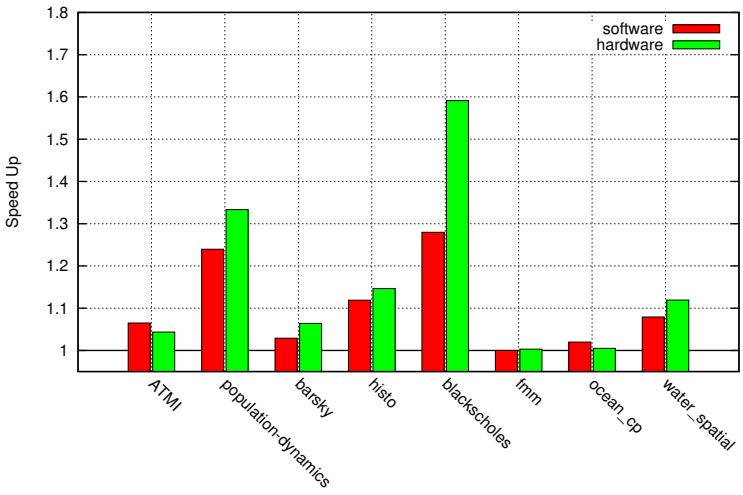


Figure 4.5: Speed-Up by Hardware Memoization for Selected Applications

pseudo-LRU associative implementation. Now, whenever we have a miss in hardware table, we do a look-up in software table. Software table can be filled in two ways:

1. Whenever a new value is being computed fill both the hardware as well as software

table (ensures inclusion).

2. Whenever an entry is being removed from hardware table- move it to software table (does not ensure inclusion).

For memoization, the first approach is more reasonable as it causes a memory write operation whenever we have a miss in the software table while the second causes a memory write operation whenever we have a miss in the hardware table. Since, miss rate is more for hardware table, first approach is better in reducing the amount of memory writes. So, on a miss in the hardware table and subsequently in the software table also, we initiate the actual function call, returns the result and simultaneously stores the result in both the hardware table as well as software table.

Using this hybrid approach we basically ensures a backup software memoization scheme in case there is a miss in the hardware memoization table thus achieving the benefit of both software as well as hardware memoization.

Chapter 5

Conclusion

As part of our effort to improve the sequential code performance, we have analyzed the applicability of function memoization in present architecture/compilers and provided three implementation schemes for facilitating memoization for functions in procedural languages like C, FORTRAN etc. First one is a load time approach and does not require the availability of source code and is feasible for any dynamically linked function. Our second one is a compile time approach and extends the scope of memoization to user written functions as well. Finally we propose a hardware scheme where by we can lower the threshold required for effective memoization and also increase the efficiency of memoization.

Our load time approach has the following advantages:

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.

We demonstrated the potential of this technique using a selected set of transcendental functions from *libm* library and the results were good on `Intel Ivybridge` as well as `ARM Cortex A9` architectures.

We also developed a tool *Ifmemo* [SR] which enables link-time memoization for a select set of dynamically linked functions and also provides profiling details for functions which are memoized.

But our load-time approach has the restriction of being applicable only for dynamically linked functions and those functions must be decided ahead.

Our second approach is a compile time technique using `LLVM` framework. It has the following characteristics:

1. It is applicable to generic functions– both user-defined as well as library functions including those which are dynamically linked.
2. Provides the performance benefit of inlining for dynamically linked functions when there is a good repetition.
3. Can handle some cases of non pure functions like functions with pointer and structure arguments, and those with global variable usage.
4. Requires recompilation of the code and could potentially increase the code size.
5. Turning off memoization can be done but requires the support of a run time code modification framework like PADRONE. [RRC+14].

These two schemes show how function memoization can be used on procedural languages in current architectures. Function memoization is not a technique which can be applied to any generic function but is useful only for pure and expensive functions like transcendental functions which are usually made in to libraries. It is also applicable for numerical computation where a closed form solution is not available.

Our hardware proposal shows the increasing performance benefit we can get if we use a hardware memoization table. This can enable memoization for low latency functions as well.

Another contribution of this thesis is in showing the repetition of arguments in real applications. Even for double precision arguments, most applications exhibited considerable repetition which could be easily exploited using a software memoization table. We also evaluated the memoization scheme across different compilers - *gcc* and *icc* in our first work and *llvm* in second and also across Intel Ivy bridge and ARM Cortex A9 architectures.

5.1 Further Extensions

To further our contribution we would like to propose the following additions:

5.1.1 Encapsulating Function Use

In many cases a function call is used in an expression as shown in Listing 5.1 and we can actually memoize more than just the function call. i.e., we store the function arguments like before but instead of returning the normal function result we return the result of the expression where the function call is used paving way for removing that expression. For example the below code is taken from an example benchmark.

Listing 5.1: Before Memoization

```
sagitta = sin((old_axis_slope_angle + iang) / 2.0);
sagitta = 2.0 *sagitta*sagitta;
```

Here, it is possible to replace the code as

Listing 5.2: After Memoization

```
sagitta = memoized_sin1(old_axis_slope_angle + iang);
```

and now the memoized function doing the original call as

```
memoized_sin1(x){
...
ret = sin((old_axis_slope_angle + iang) / 2.0);
return 2.0 * ret * ret;
}
```

which essentially saves 2 multiplication and 1 division instructions whenever we have a hit in memoization table. This also helps to reduce some temporary computations which could reduce the register pressure.

5.1.2 Capturing Expressions Involving Arguments

In many cases the argument to a memoizable function is obtained as a result of an expression which in turn depends on some input value. For example, a memoized function call happens like

```
x = sin(exp(angle * pi) / 2.0));
```

where angle is the only variable. Now we can extend the normal memoization technique to handle this expression inside a single function call as follows:

```
memoized_sin2(x){
...
ret = sin(exp(x * pi) / 2.0));
}
```

and the call being replaced as

```
x = memoized_sin2(angle);
```

5.1.3 Pre-filling Memoization-table

This technique involves a helper thread which essentially would run ahead of main thread of execution by executing only the instructions needed for evaluating the argument(s) to a function to be memoized. Now this helper thread and the main thread can share the memoization table with helper doing the filling job there by making the main thread run faster.

Such a helper thread technique is presented by Swamy et.al. in [NSKS14] where they propose a hardware/software framework called core-tethering to support efficient helper threading on heterogeneous many-cores. Core-tethering provides a co-processor like interface to the small cores that (a) enables a large core to directly initiate and control helper execution on the helper core and (b) allows efficient transfer of execution context between the cores, thereby reducing the performance overhead of accessing small cores for helper execution. For memoization we would need the helper thread to execute only the dependent code for getting the argument(s) to memoization function and also executing the function and filling the memoization table. The main thread would be initiating the helper thread as and when its input(s) are ready and can always take the result of memoization function from the pre-filled table.

5.1.4 Invariant Analysis

In our work we have not handled the case where any function argument is invariant across function calls. The invariant argument can be either memoizable type or non-memoizable type like arrays or large structures. In the first case identifying an invariant argument could mean handling it like a constant as discussed in Section 3.2.4 which can increase the potential performance benefit. Second case is more important for memoization as this could open many more functions which could be memoized [CH99].

5.1.5 Approximation and Prediction

For the whole of the thesis, we have considered precise value computation and no approximation. But in many fields like multimedia approximation is allowed and with approximation applicability of memoization is more as more values are getting repeated [ACV05, ESCB12]. Also, it would be an interesting work to try and predict the future arguments from the already seen ones. This prediction has limited scope in functions where arguments are of floating point type but could be useful for integer arguments.

Appendix

Listing 3: CNDF function in Blackscholes

```
fptype CNDF ( fptype InputX )
{
    int sign;

    fptype OutputX;
    fptype xInput;
    fptype xNPrimeofX;
    fptype expValues;
    fptype xK2;
    fptype xK2_2, xK2_3;
    fptype xK2_4, xK2_5;
    fptype xLocal, xLocal_1;
    fptype xLocal_2, xLocal_3;

    // Check for negative value of InputX
    if (InputX < 0.0) {
        InputX = -InputX;
        sign = 1;
    } else
        sign = 0;

    xInput = InputX;

    // Compute NPrimeX term common to both four & six decimal accuracy calcs
    expValues = exp(-0.5f * InputX * InputX);
    xNPrimeofX = expValues;
    xNPrimeofX = xNPrimeofX * inv_sqrt_2xPI;

    xK2 = 0.2316419 * xInput;
    xK2 = 1.0 + xK2;
    xK2 = 1.0 / xK2;
    xK2_2 = xK2 * xK2;
    xK2_3 = xK2_2 * xK2;
    xK2_4 = xK2_3 * xK2;
    xK2_5 = xK2_4 * xK2;

    xLocal_1 = xK2 * 0.319381530;
    xLocal_2 = xK2_2 * (-0.356563782);
    xLocal_3 = xK2_3 * 1.781477937;
    xLocal_2 = xLocal_2 + xLocal_3;
```

```
    xLocal_3 = xK2_4 * (-1.821255978);
    xLocal_2 = xLocal_2 + xLocal_3;
    xLocal_3 = xK2_5 * 1.330274429;
    xLocal_2 = xLocal_2 + xLocal_3;

    xLocal_1 = xLocal_2 + xLocal_1;
    xLocal  = xLocal_1 * xNPrimeofX;
    xLocal  = 1.0 - xLocal;

    OutputX = xLocal;

    if (sign) {
        OutputX = 1.0 - OutputX;
    }

    return OutputX;
}
```

Glossaire

Memoization : Technique of saving input and results for future reuse
IFM : Intercepting Functions for Memoization
GPU : Graphical Processing Unit
STM : Software Transactional Memory
SSA : Static Single Assignment
LLVM : A compiler tool chain for SSA based optimization
CNDF : Cumulative Normal Distribution Function.

Bibliography

[ACV05] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *Computers, IEEE Transactions on*, 54(7):922–927, July 2005.

[ADE+01] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[APX14] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 529–540, June 2014. doi:10.1109/ISCA.2014.6853207.

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

[BGA03] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275, March 2003.

[BKSL08a] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1454115.1454128.

[BKSL08b] 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, January 2008.

[CF00] Daniel Citron and Dror G. Feitelson. Hardware memoization of mathematical and trigonometric functions. Technical report, 2000.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. URL: http://doi.acm.org/10.1145/115372.115320, doi:10.1145/115372.115320.

[CFR98] Daniel Citron, Dror Feitelson, and Larry Rudolph. Accelerating multimedia 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, pages 252–261, New York, NY, USA, 1998. ACM.

[CH99] Daniel A. Connors and Wen-Mei W. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO)*, 1999.

[CPM+13] Mamadou Ciss, Nicolas Parisey, Fabrice Moreau, Charles-Antoine Dedryver, and Jean-Sébastien 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*, pages 1–9, 2013.

[DL04] Yonghua Ding and Zhiyuan Li. A compiler scheme for reusing intermediate computation results. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 277–288, March 2004. doi:10.1109/CGO.2004.1281681.

[dra] Dragonegg. URL: http://dragonegg.llvm.org/.

[ESCB12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 449–460, Dec 2012.

[Fox] Agner Fox. Calling conventions. URL: http://www.agner.org/optimize/calling_conventions.pdf.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960. URL: http://doi.acm.org/10.1145/367390.367400, doi:10.1145/367390.367400.

[G+11] PARSEC Group et al. A memo on exploration of splash-2 input sets. *Princeton Univ., Elizabeth, NJ*, 2011.

[GTM99] Antonio González, Jordi Tubella, and Carlos Molina. Trace-level reuse. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 30–37. IEEE, 1999.

[Hen06] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. URL: http://doi.acm.org/10.1145/1186736.1186737, doi:10.1145/1186736.1186737.

[HL99] Jian Huang and David J Lilja. Exploiting basic block value locality with block reuse. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 106–114. IEEE, 1999. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=744342.

[Int] Intel Corporation. Intel turbo boost technology.

[JM98] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *Micro, IEEE*, 18(4):60–75, Jul 1998. doi:10.1109/40.710872.

[job] Joblib. URL: https://pypi.python.org/pypi/joblib.

[jug] Jug. URL: https://pypi.python.org/pypi/Jug.

[KD15] Amal Khalil and Juergen Dingel. Incremental symbolic execution of evolving state machines. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 14–23. IEEE, 2015.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. 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*, pages 190–200. ACM Press, 2005.

[LFB+10] Guoping Long, Diana Franklin, Susmit Biswas, Pablo Ortiz, Jason Oberg, Dongrui Fan, and Frederic T. Chong. Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 337–348, Washington, DC, USA, 2010. IEEE Computer Society. URL: http://dx.doi.org/10.1109/MICRO.2010.41, doi:10.1109/MICRO.2010.41.

[LGNG14] Y. Li, C.D. Garrett, M.D. Noakes, and A.G. Gounares. Input vector analysis for memoization estimation, June 10 2014. US Patent 8,752,021. URL: https://www.google.com/patents/US8752021.

[llv] Writing an LLVM pass. URL: http://llvm.org/docs/WritingAnLLVMPass.html.

[lru] Pseudo lru. URL: http://people.cs.clemson.edu/~mark/464/p_lru.txt.

[MH98] Paul McNamee and Marty Hall. Developing a tool for memoizing functions in C++. *SIGPLAN Not.*, 33(8):17–22, August 1998.

[Mic68] Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[Mic08] Pierre Michaud. Atmi manual, 2008.

[MSF+07] Pierre Michaud, André Seznec, Damien Fetis, Yiannakis Sazeides, and Theofanis Constantinou. A study of thread migration in temperature-constrained multicores. *ACM Trans. Archit. Code Optim.*, 4(2), June 2007. URL: http://doi.acm.org/10.1145/1250727.1250729, doi:10.1145/1250727.1250729.

[NSKS14] Bharath Narasimha Swamy, Alain Ketterlin, and André Seznec. Hardware/Software Helper Thread Prefetching On Heterogeneous Many Cores. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Paris, France, October 2014. URL: https://hal.inria.fr/hal-01087752, doi:10.1109/SBAC-PAD.2014.39.

[RC10] Hugo Rito and João Cachopo. 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, pages 89–98, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1852761.1852775, doi:10.1145/1852761.1852775.

[Red88] Uday Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 289–297, New York, NY, USA, 1988. ACM. URL: http://doi.acm.org/10.1145/62678.62721, doi:10.1145/62678.62721.

[Ric92] Stephen E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 1992.

[Ric93] S. E. Richardson. Exploiting trivial and redundant computation. In *Proceedings., 1993 IEEE 11th Symposium on Computer Arithmetic*, pages 220–227, Jun 1993. doi:10.1109/ARITH.1993.378089.

[Rod85] David P. Rodgers. Improvements in multiprocessor system design. In Thomas F. Gannon, Tilak Agerwala, and Charles Freiman, editors, *ISCA*, pages 225–231. IEEE Computer Society, 1985. URL: http://dblp.uni-trier.de/db/conf/isca/isca85.html#Rodgers85.

[RRC+14] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. In *DCE 2014 - International workshop on Dynamic Compilation Everywhere*, Vienne, Austria, January 2014. URL: https://hal.inria.fr/hal-00917950.

[SR] Arjun Suresh and Erven Rohou. if-memo. URL: https://team.inria.fr/alf/software/if-memo//.

[SRS+12] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[SSRS15] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. Intercepting functions for memoization: A case study using transcendental functions. *ACM Trans. Archit. Code Optim.*, 12(2):18:18:1–18:18:23, June 2015. URL: http://doi.acm.org/10.1145/2751559, doi:10.1145/2751559.

[SW12] Terrance Swift and David s. Warren. Xsb: Extending Prolog with tabled logic programming. *Theory Pract. Log. Program.*, 12(1-2):157–187, January 2012. URL: http://dx.doi.org/10.1017/S1471068411000500, doi:10.1017/S1471068411000500.

[TACT08] James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. 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, pages 145–156, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1346281.1346300, doi:10.1145/1346281.1346300.

[Wat95] G.N. Watson. *A Treatise on the Theory of Bessel Functions*. Cambridge Mathematical Library. Cambridge University Press, 1995. URL: https://books.google.fr/books?id=Mlk3FrNoEVoC.

[Wil84] Dan E. Willard. New trie data structures which support very fast search operations. *J. Comput. Syst. Sci.*, 28(3):379–394, July 1984. URL: http://dx.doi.org/10.1016/0022-0000(84)90020-5, doi:10.1016/0022-0000(84)90020-5.

[YPK12] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 144–154, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2338965.2336771, doi:10.1145/2338965.2336771.

List of Figures

1 Speed-Up par Memoization pour les benchmarks SPEC 9
2 Speed-Up par Memoization pour les applications sélectionnées 10
3 Speed-Up de la Memoization matérielle pour les Benchmarks SPEC . . 11
4 Speed-Up de la Memoization matérielle pour les applications sélectionnées 11
5 Importance of Amdahl's law 14
6 Reuse distance of arguments to j0 function in ATMI_Goh 16
7 Repetitive behaviour of arguments to j0 function in ATMI_Goh 17

1.1 Division Unit using a MEMO Table 25
1.2 Implementation of Trace Level Memoization 26
1.3 Scope of Region Based Memoization 28

2.1 Intercepting dynamic call to *sin* 38
2.2 Implementation of Hash Function 39
2.3 Memoization of SPEC benchmarks on Intel Ivy Bridge (GNU compiler) for different table sizes 47
2.4 Memoization of selected applications on Intel Ivy Bridge (GNU compiler) for different table sizes 47
2.5 Memoization of SPEC benchmarks on Intel Ivy Bridge for different table sizes (icc) 49
2.6 Profitability curve of memoizing transcendental functions 51
2.7 Memoization of selected applications on Intel Ivy Bridge for different table sizes (icc) 51

2.8 Memoization of SPEC benchmarks and selected applications on ARM (GNU compiler) for different table sizes 52
2.9 Profitability curve of memoizing transcendental functions 52
2.10 Associative table for ATMI (GNU compiler) on Intel Ivy Bridge 54
2.11 Associative table for ATMI (GNU compiler) on ARM Cortex-A9 56

3.1 Profitability Curve - preload vs inline 63
3.2 Flow Diagram of Memoization 66
3.3 Speed-Up by Memoization for SPEC benchmarks 75
3.4 Speed-Up by Memoization for Selected Applications 76
3.5 Code size change due to Memoization 80

4.1 Profitability Curve- Software vs Hardware 86
4.2 Working of MSCALL/MSUPDATE 88
4.3 A set entry in Hardware Memoization Table 89
4.4 Speed-Up by Hardware Memoization for SPEC Benchmarks 93
4.5 Speed-Up by Hardware Memoization for Selected Applications 93

List of Tables

1 Behaviour of j0 args in ATMI 17
2 Profitability analysis 20

2.1 Experimental set-up on Intel Ivy Bridge 44
2.2 Experimental set-up on ARM Cortex 45
2.3 Function call analysis 46
2.4 Profitability Analysis 50
2.5 Profitability Analysis 53
2.6 Function call analysis for ATMI 55
2.7 Function call-site analysis 57
2.8 Hash-function Comparison 58

3.1 Profitability Analysis with Inlining 64
3.2 Experimental set-up on Intel Ivy Bridge 74
3.3 Calls to memoized functions 79

4.1 Profitability Analysis for Hardware Memoization 85
4.2 Comparing Hardware-Software Memoization 92

List of Algorithms

1 Pseudo code for memoization 38
2 Pseudo-code for LLVM Module Pass 65
3 Pseudo-code for identifying memoizable functions 67
4 Pseudo-code for identifying if input function is safe for memoization . . 68
5 Pseudo-code for identifying if input function has safe arguments 69
6 Pseudo-code for checking if a function is Global Variable safe for memoization 72
7 Pseudo-code for Replacing Memoizable Function Calls 73

Résumé

Nous avons proposé des mécanismes pour mettre en œuvre la mémoïsation de fonction au niveau logiciel dans le cadre de nos efforts pour améliorer les performances du code séquentiel. Nous avons analysé le potentiel de la mémoïsation de fonction sur des applications et le gain de performance qu'elle apporte sur des architectures actuelles. Nous avons proposé trois approches - une approche simple qui s'applique au chargement et qui fonctionne pour toute fonction de bibliothèque liée dynamiquement, une approche à la compilation utilisant LLVM qui peut permettre la mémoïsation pour toute fonction du programme, ainsi qu'une proposition d'implémentation de la mémoïsation en matériel et ses avantages potentiels. Nous avons démonstré avec les fonctions transcendantales que l'approche au chargement est applicable et donne un bon avantage, même avec des architectures et des compilateurs (avec la restriction qu'elle ne peut être appliquée que pour les fonctions liées dynamiquement) modernes. Notre approche à la compilation étend la portée de la mémoïsation et en augmente également les bénéfices. Cela fonctionne pour les fonctions définies par l'utilisateur ainsi que pour les fonctions de bibliothèque. Nous pouvons gérer certains types de fonctions non pures comme les fonctions avec des arguments de type pointeur et l'utilisation de variables globales. La mémoïsation en matériel abaisse encore le seuil de profitabilité de la mémoïsation et donne plus de gain de performance en moyenne.

Abstract

We have proposed mechanisms to implement function memoization at a software level as part of our effort to improve sequential code performance. We have analyzed the potential of function memoization on applications and its performance gain on current architectures. We have proposed three schemes – a simple load time approach which works for any dynamically linked function, a compile time approach using LLVM framework which can enable memoization for any program function and also a hardware proposal for doing memoization in hardware and its potential benefits. Demonstration of the link time approach with transcendental functions showed that memoization is applicable and gives good benefit even under modern architectures and compilers (with the restriction that it can be applied only for dynamically linked functions). Our compile time approach extends the scope of memoization and also increases the benefit due to memoization. This works for both user defined functions as well as library functions. It can handle certain kind of non pure functions like those functions with pointer arguments and global variable usage. Hardware memoization lowers the threshold for a function to be memoized and gives more performance gain on average.