Intercepting Functions for Memoization

Arjun Suresh
Advisor: Erven Rohou
Co-Advisor: André Seznec

Inria, Rennes *May 10, 2016*

Introduction

- We live in a performance hungry world
- Hardware clock frequency has reached its limit
- So, we should try to reduce the no. of cycles required

Memoization

Memoization

- Save the result of execution so that future executions may be avoided
- Hardware as well as Software approaches
  - Instruction level - DIV (x,y)
  - Basic block level - { x = y*y / 6.75;}
  - Function level - sin(x)

Function Memoization

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

```
int fun(int * p, float arg){
   ...
  b = sin(arg);//arg = 6.7
   ...
}
```

- Save the result of a function call
- When arguments repeat, function execution can be avoided
- But function needs to be pure

sin(arg)

Arg. Value	Result
- -	- - -
6.7	0.40484992061
- -	- - -
38	0.2963685787
- -	- - -

Not all pure functions need to be memoized

- Long latency functions
  - to make memoization productive
  - can be statically determined in most cases
- Repeatability in arguments across calls
  - difficult to know before program run
- Critical ones
  - memoized function must be called a lot of times
  - even otherwise we just lose an opportunity for benefit

Examples of memoizable Functions

Transcendental functions in libm

- Trigonometric Functions (`sin, cos, tan`)
- Bessel Functions (`j0, j1`)
- Exponential Functions (`exp, pow, log`)

Required Repetition Rate

Hit in Table

Miss in Table

Original Function

$$H \times t_h \quad + \quad (1 - H)(T_f + t_{mo}) \quad < T_f$$

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

Required Repetition Rate

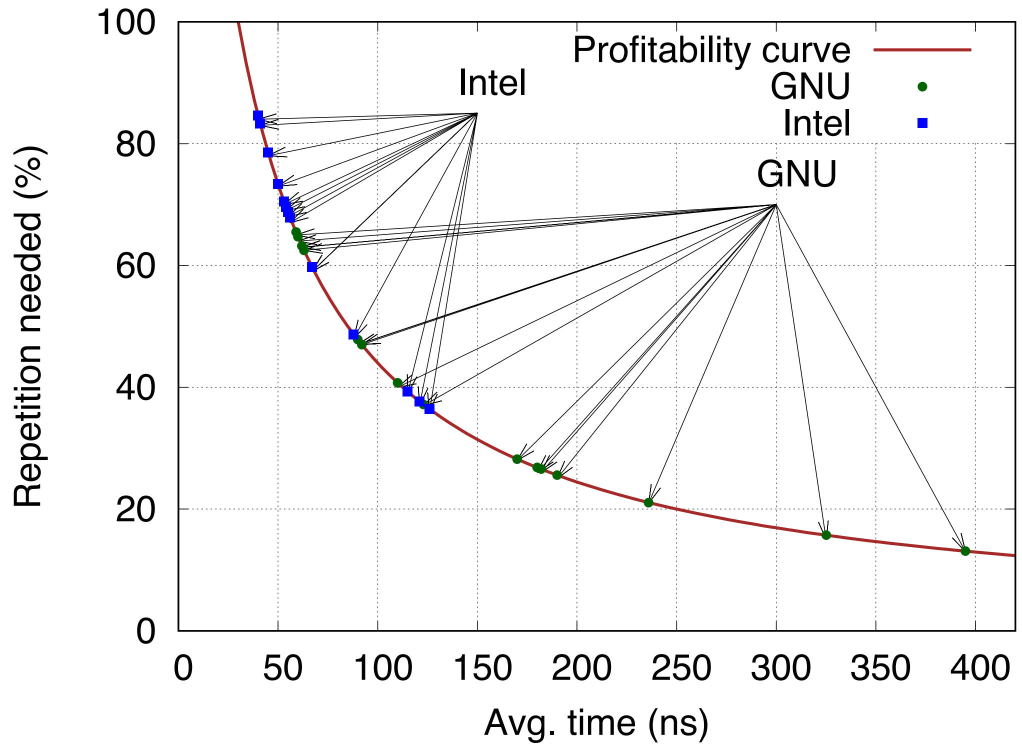
2 GHz Intel Ivybridge

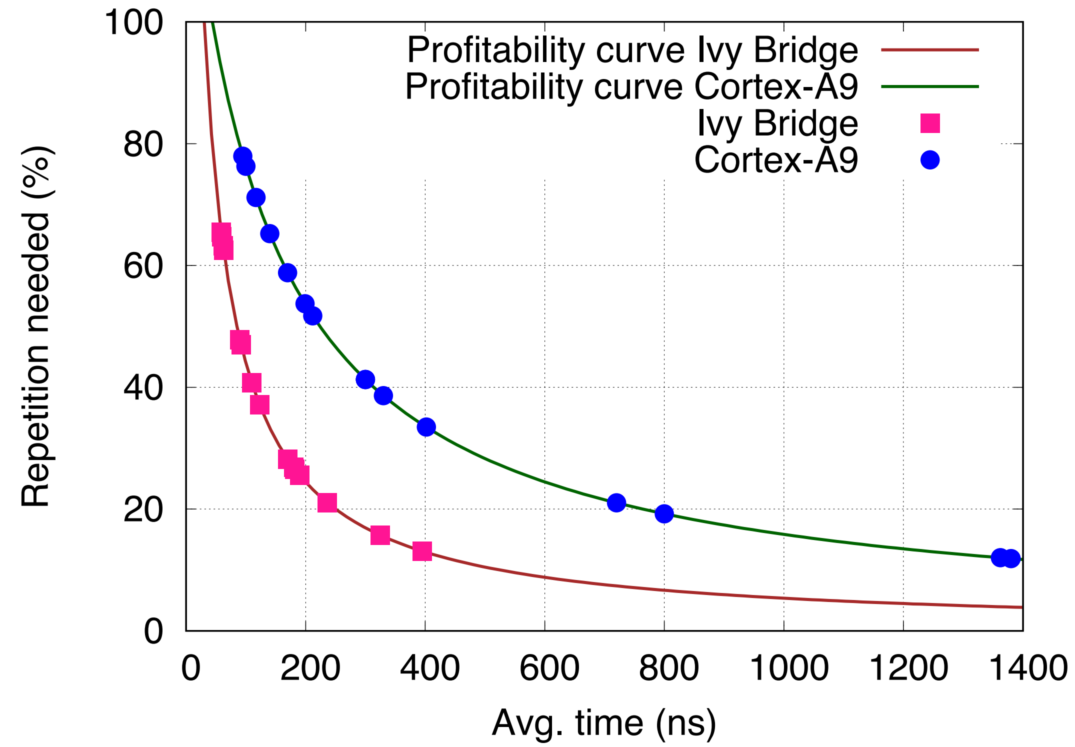
Function	Hit Time- t_h (ns)	Overhead- t_{mo} (ns)	Avg. Time- T_f (ns)	Repetition Needed- H (%)
exp	30	55	90	48
log			92	47
sin			110	41
cos			123	37
j0			395	13
j1			325	16
pow			190	27

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

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

Profitability Curves





gcc vs icc

Intel Ivybridge vs ARM Cortex A9

More repetition needed for icc

More repetition needed for ARM

Are arguments repeating?

Argument behaviour can be classified into three:

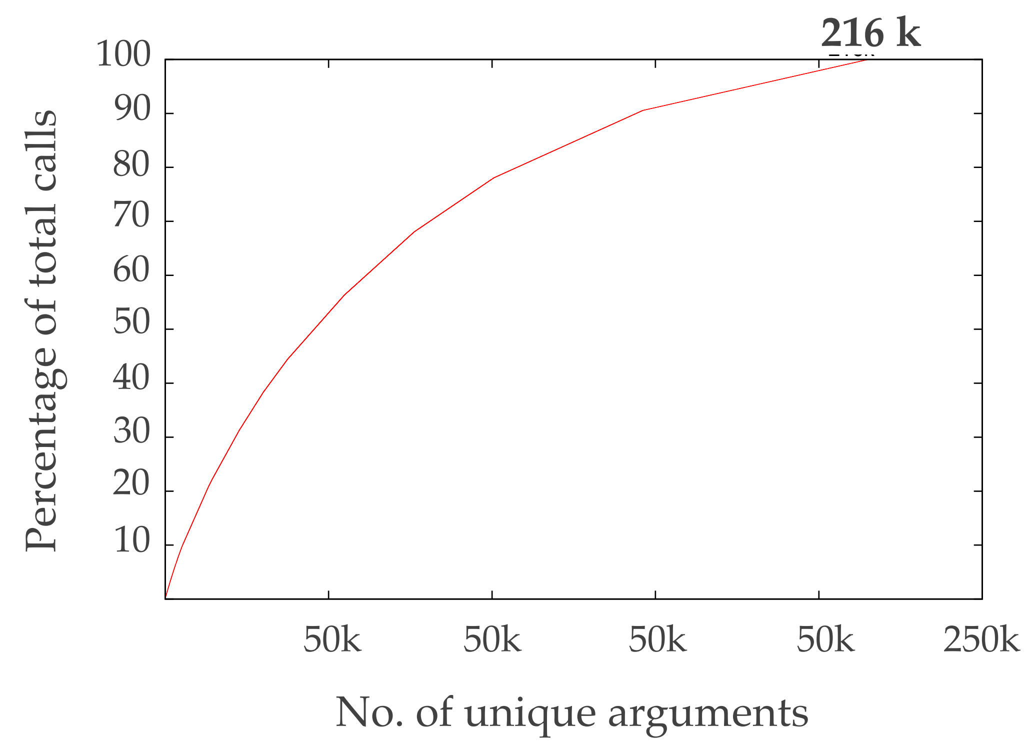
- There are only a few unique arguments
  - A small memoization table is enough
- There are a large number of unique calls but arguments do have repetitions
  - Memoization benefit increases with size of memoization table
- Arguments are largely unique
  - Memoization gives no benefit

An Example of Argument Repetition

Case of `j0()` function in ATMI application

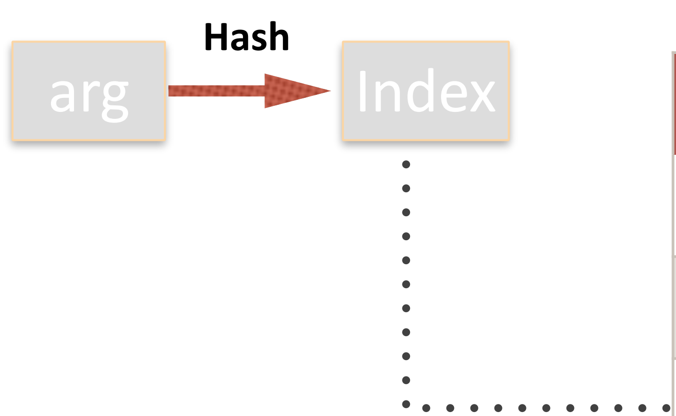
Fully Associative Table Size	%Capture
4k	8
16k	13
64k	28
256k	100

Cumulative call count with respect to no. of unique arguments



Memoization Table

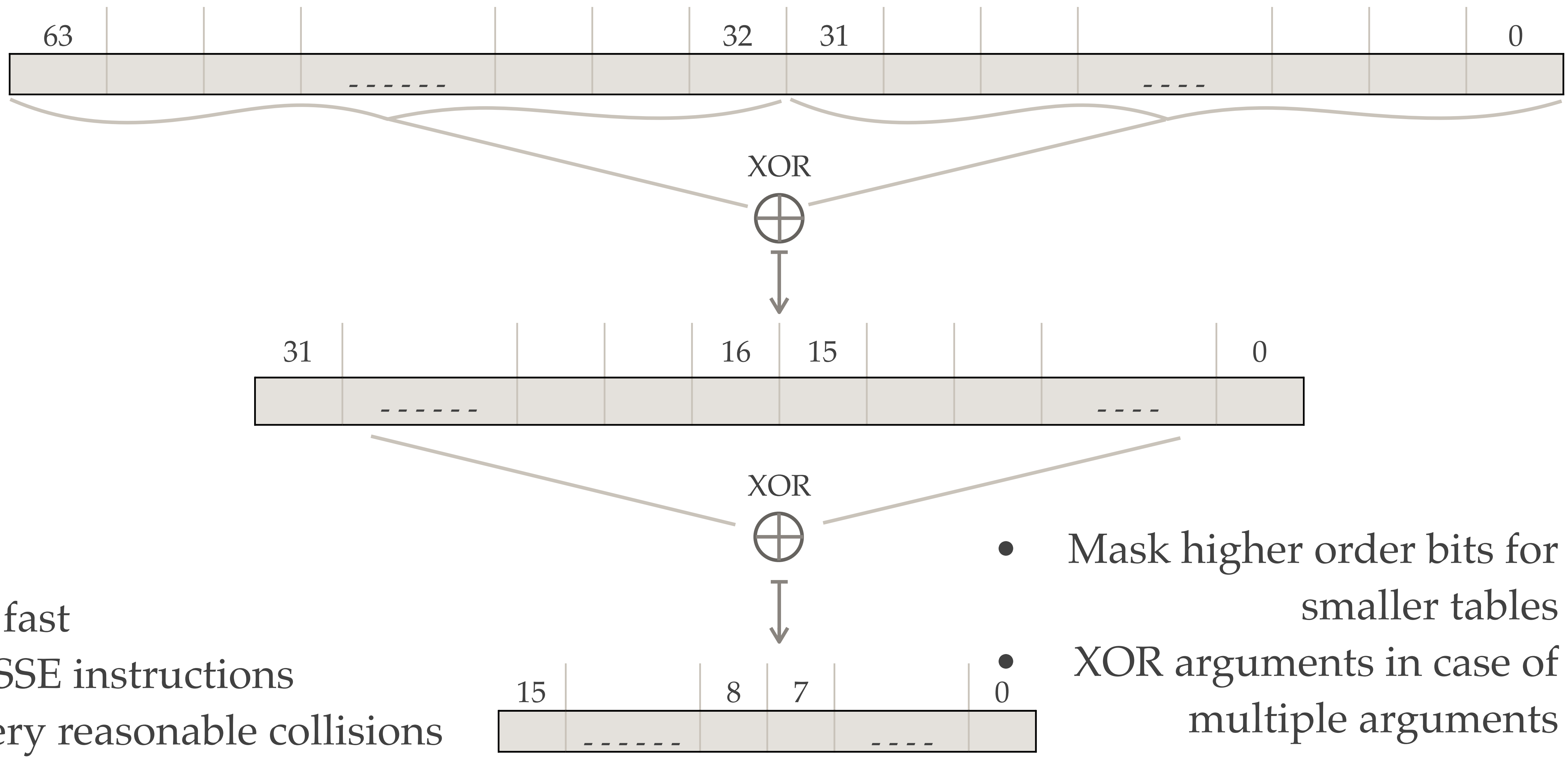
- A direct-mapped hash-table for each memoized function
- Up to 64k entries tagged with arguments
  - Requires up to 1MB of physical memory per memoized function — 2 * 8 bytes * 64 k



Memoization Table

Tag	Value
arg	sin(arg)

XOR Hashing



- Is fast
- 6 SSE instructions
- Very reasonable collisions

- Mask higher order bits for smaller tables
- XOR arguments in case of multiple arguments

Our Memoization Approaches

We propose 3 different approaches for function memoization each with its own merits and demerits

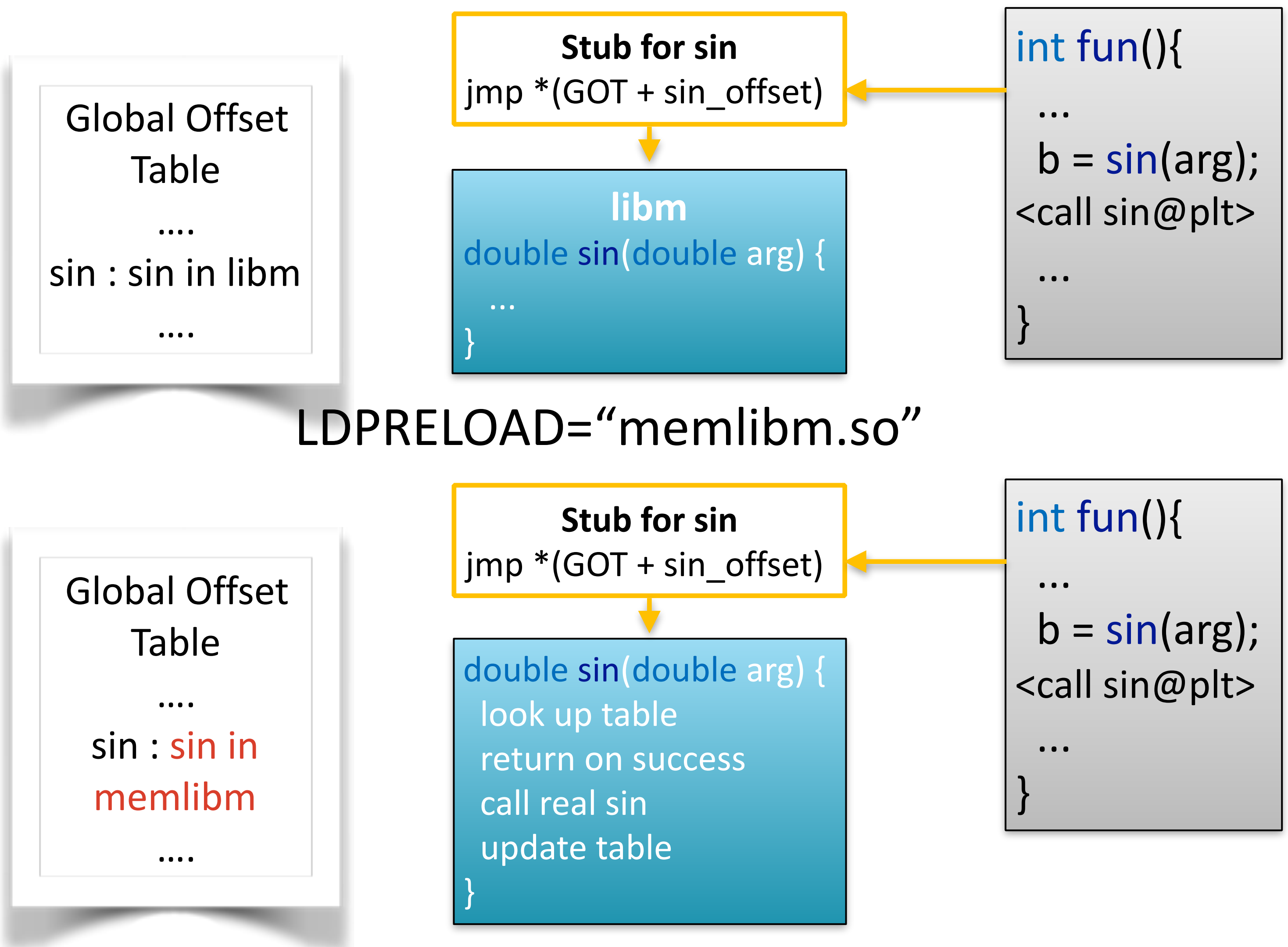
1. Load time Approach
2. Compile time Approach
3. Compile time Approach Assuming Hardware Support

Load-time Function Interception

Load-time Approach

- LD_PRELOAD to intercept dynamic calls to shared library functions
- Memoization wrapper of function from preloaded library gets precedence to original ones and takes care of memoization
- No need of re-compilation or availability of source code
- Can be used even a naive user
- We illustrate the working for libm but same mechanism works for any dynamically linked library

Intercepting Calls to libm



In case of slowdown?

When there are a large number of calls to a memoized function but arguments to them are not repeating enough, memoization can cause slowdown

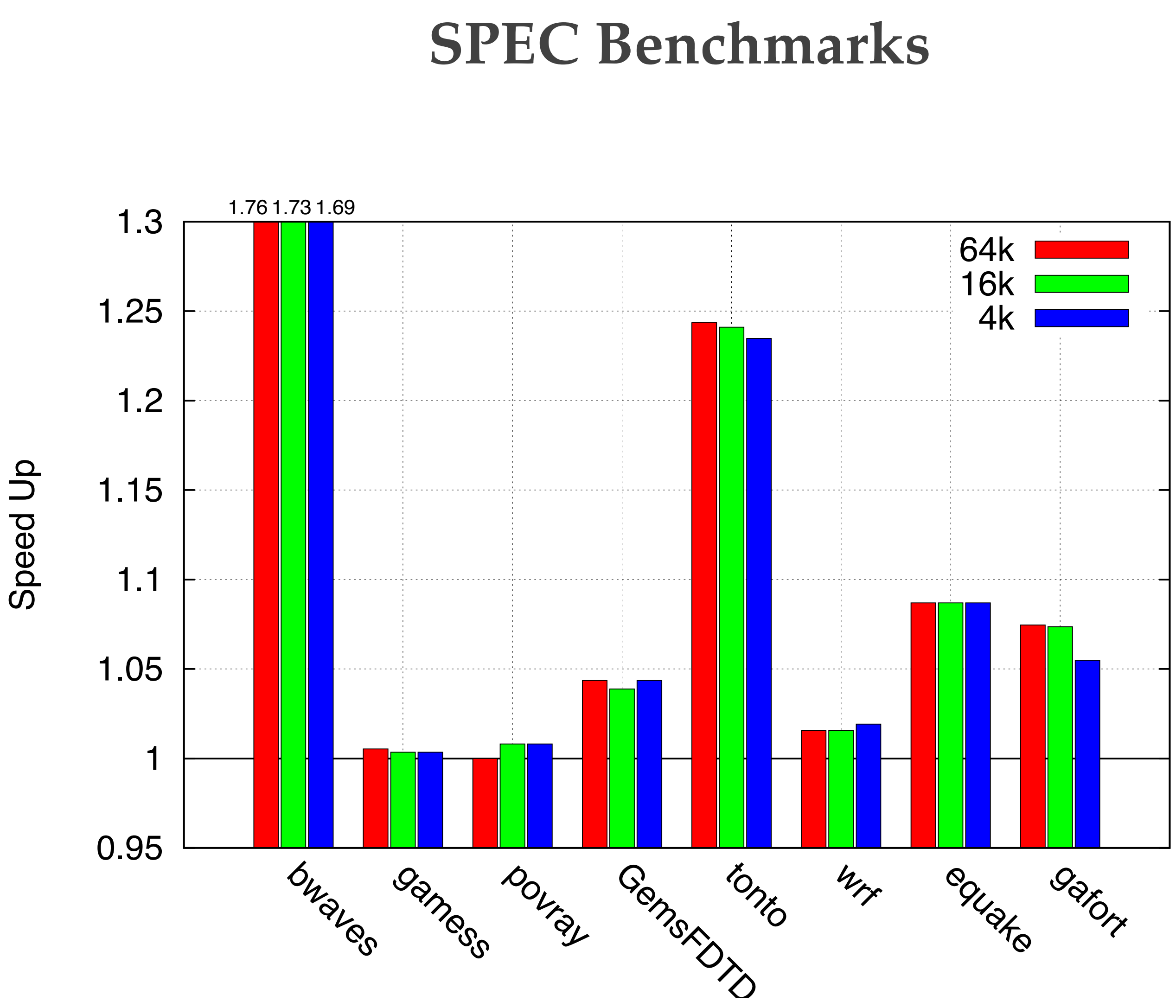
- Turn-off memoization
- Same mechanism as turning-on
  - Modify GOT
    - A helper thread monitors the argument behaviour
    - Replaces the memoizedlibm entry with the libm entry in case of low repetition
    - Application runs exactly as without memoization

Experimental Set-up

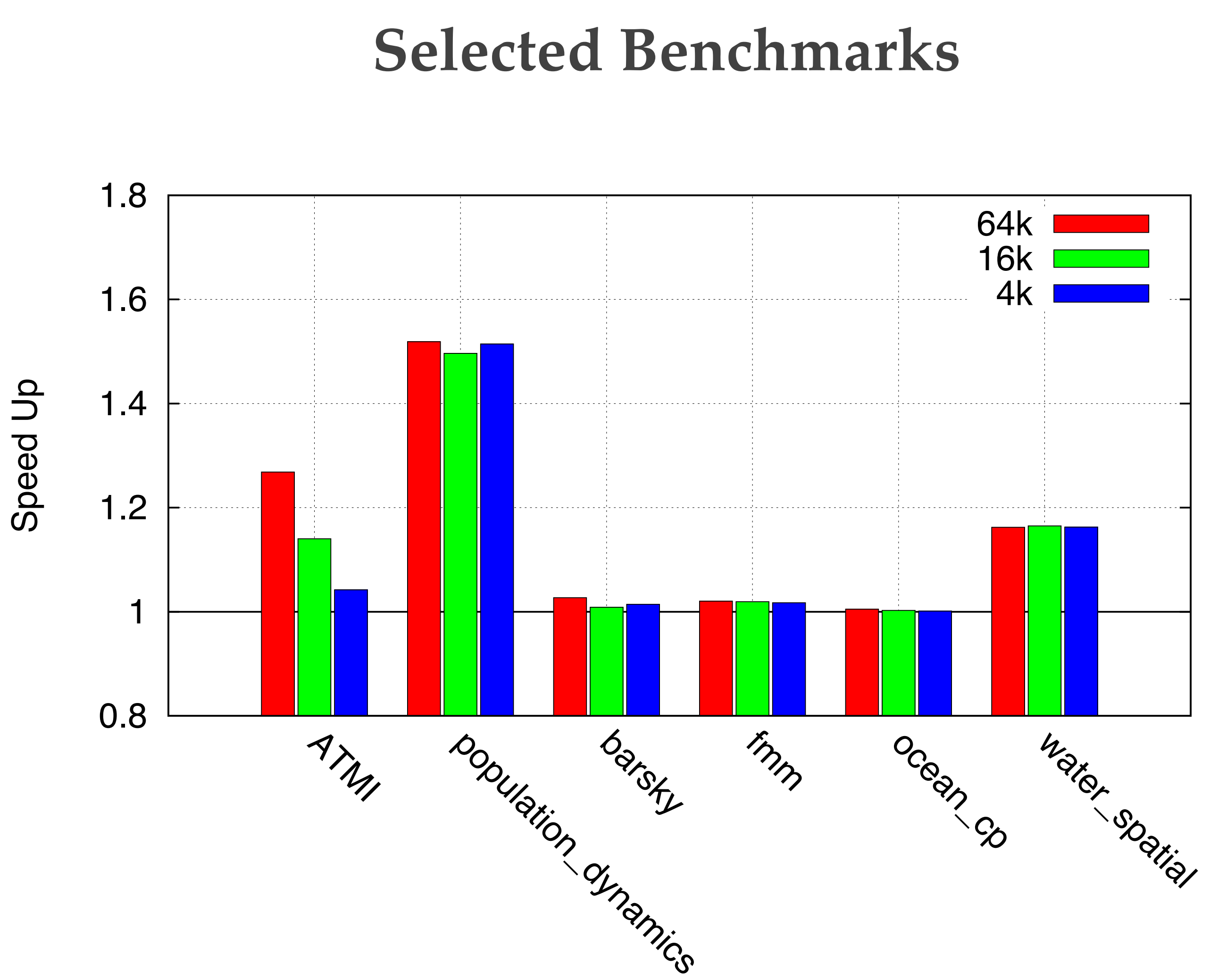
- 2 Application Sets
  - SPEC benchmarks
  - Selected Applications
- Intel Ivybridge and ARM Cortex A9
- gcc as well as icc using Intel Math

	Intel Ivybridge	ARM Cortex A9
Processor	Intel Core i7	ARM Cortex A9
L3 Cache	8 MB	1 MB
Clock Speed	2 GHz	1.2 GHz
RAM	8 GB	1 GB
Linux kernel	3.11	3.11
gcc version	4.8	4.7
icc version	9	—
optimisation flag	-O3	-O2
libm version	2.2	2.2

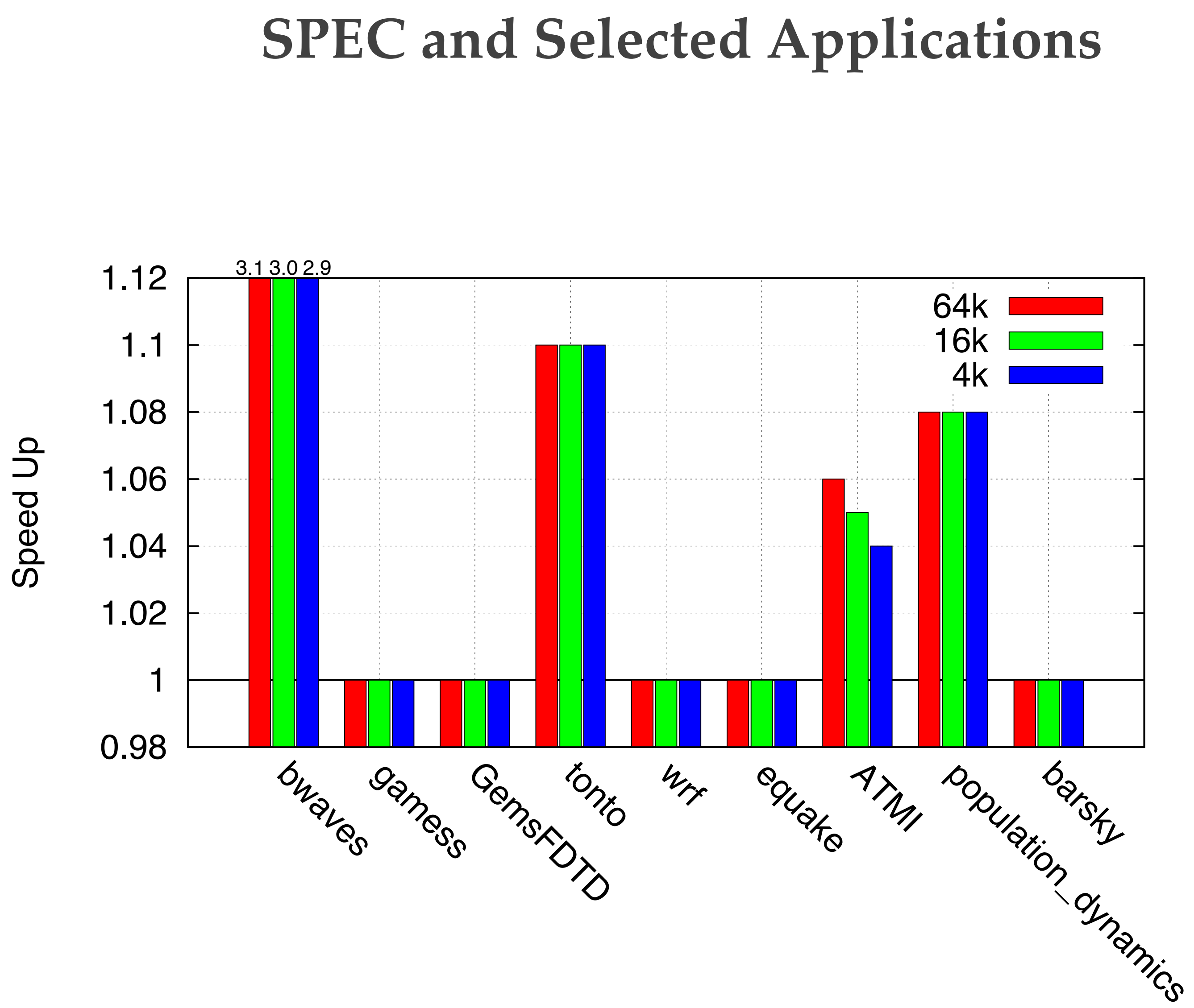
Results - Intel Ivybridge



Results - Intel Ivybridge

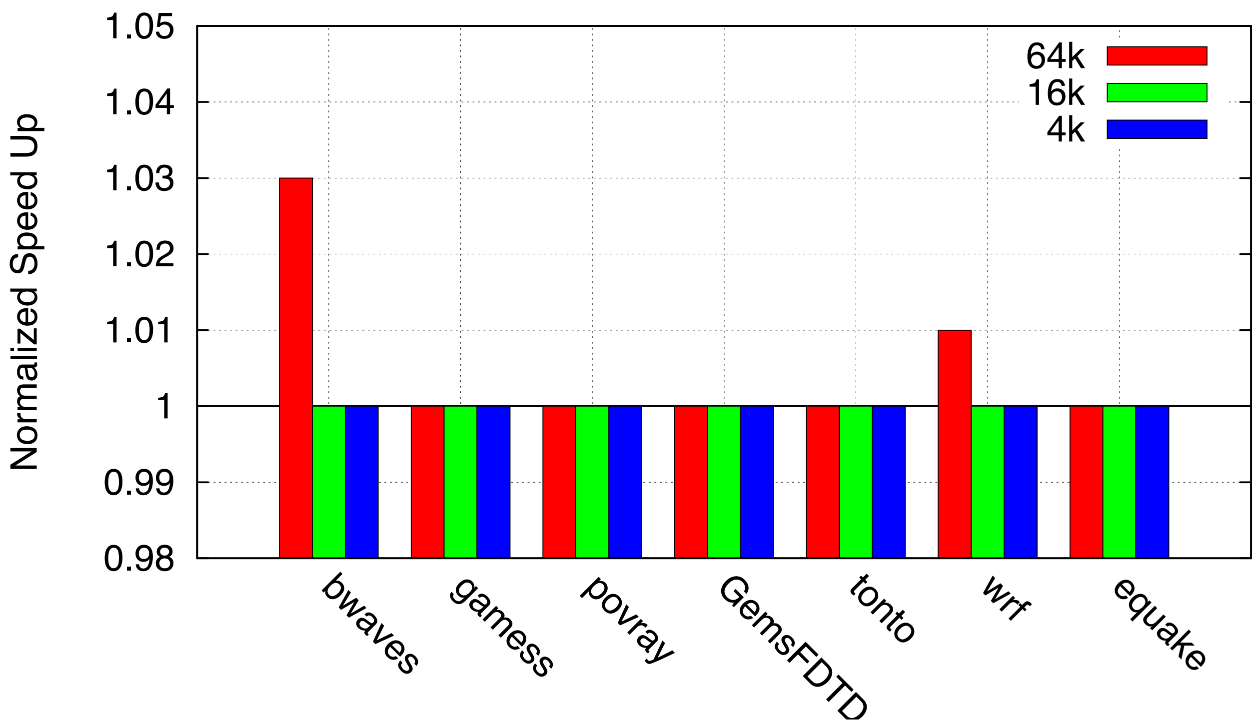


Results - ARM Cortex A9



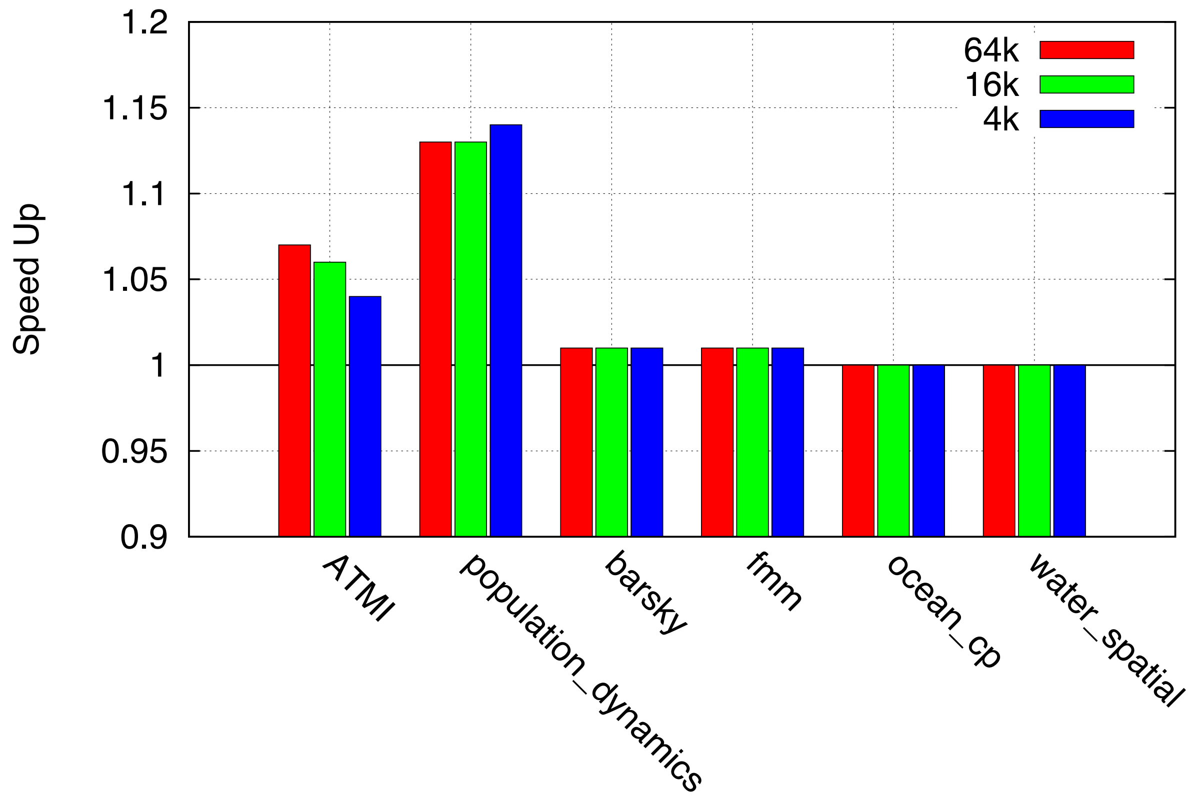
Results - icc

Spec Benchmarks



Results - icc

Selected Applications



Free Result

- bwaves of SPEC 2006 gave 1.76 times improvement on runtime on Intel Ivybridge
- 3.1 times on ARM
- *pow* implementation of **libgcc** has a performance bug
- Happens for *pow(m, n)*, where m is very close to 1 and n is very close to 0.75
- With icc/IntelMath the benefit drops to 3%

Result-Summary

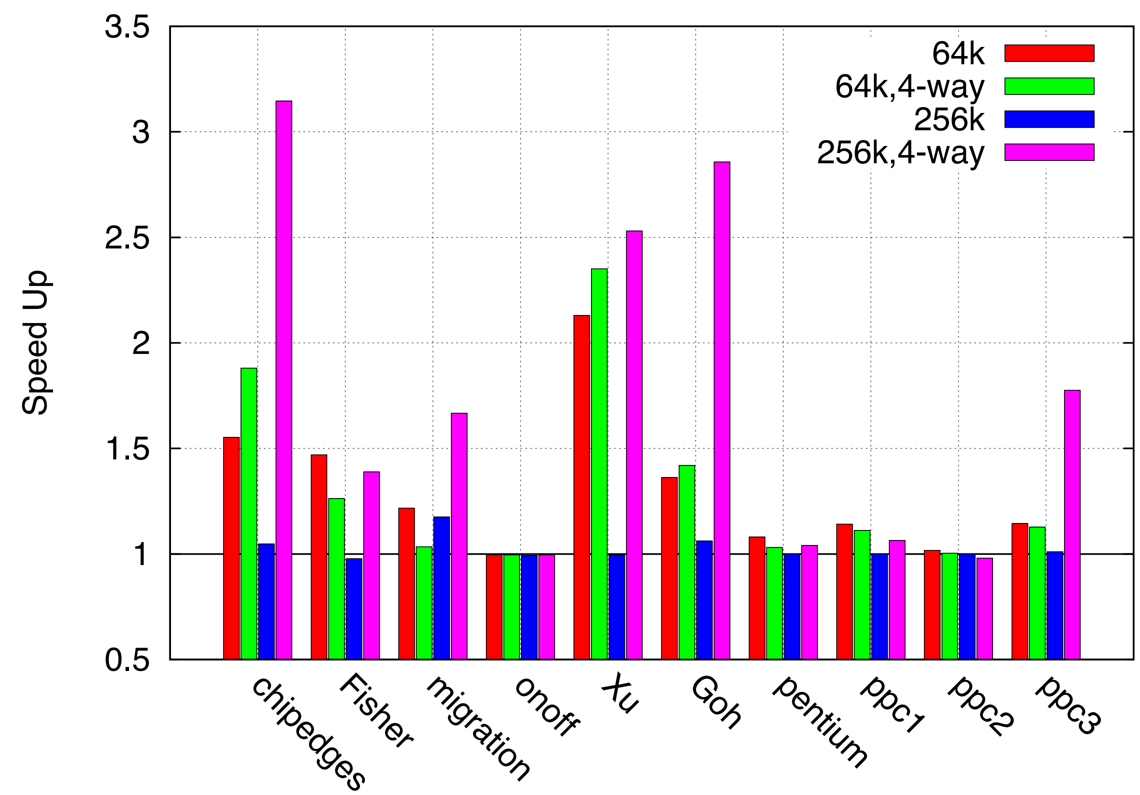
- SPEC benchmarks give 1-24% speedup on Intel Ivybridge
- Drops to up to 10% on ARM Cortex A9
- Other selected applications give 1-50% speed up on Intel Ivybridge
- Drops to up to 14% on ARM Cortex A9
- With icc, speed-ups drop to 1-15%
- Again Selected Applications give more benefit compared to SPEC ones

Associativity

- A 4 way associative hash-table
- 4 sets of 2 X 8 bytes (8 bytes each for a double-precision argument and result) for each table entry
- Each table access requires exactly one cache-line fetch (64 bytes on x86-64)
- Allows for even larger size for memoization table without much penalty of last level cache miss
- 2 way associative implementation for ARM due to 32 byte cache line

Applying Associativity

Associativity on Intel Ivybridge

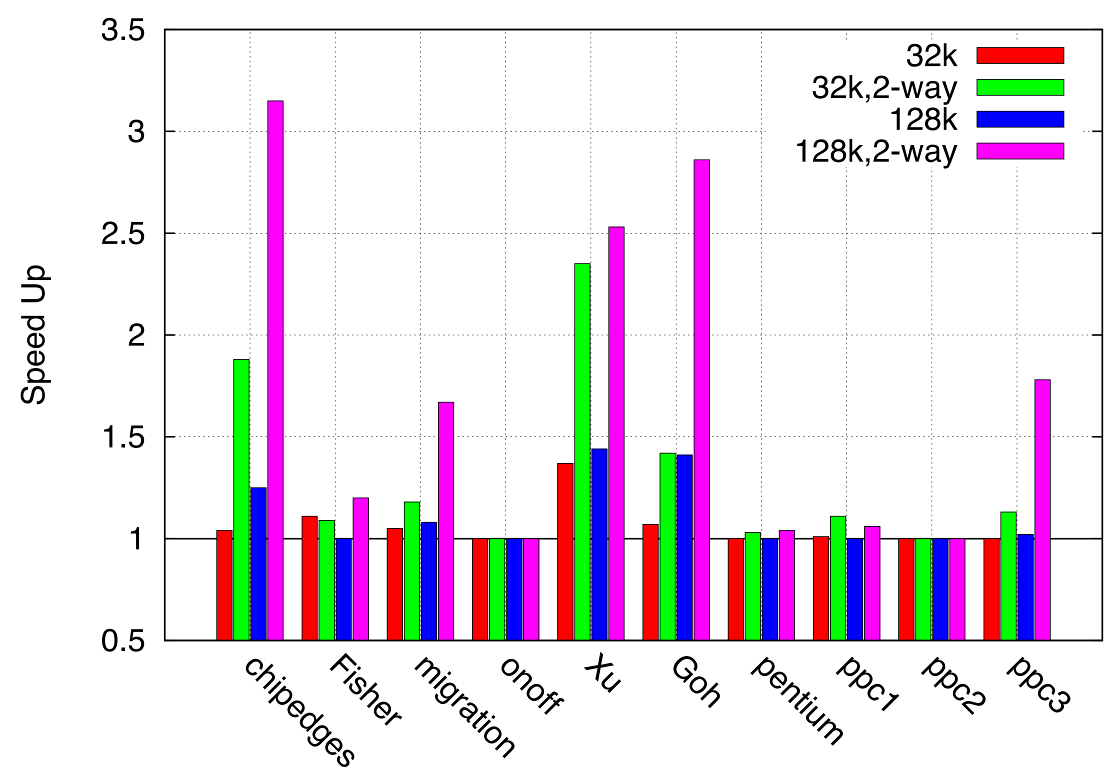


ATMI

Fully Associative Table Size	%Capture
4k	8
16k	13
64k	28
256k	100

Applying Associativity

Associativity on ARM Cortex A9



ATMI

Fully Associative Table Size	%Capture
4k	8
16k	13
64k	28
256k	100

- Associativity gave very good result
- >64k without associativity gave slowdown on Intel Ivybridge

if-memo

- Our memoization tool
- Takes care of function interception
- Maintains the memoization table, hashing and helper thread monitoring
- Works for our selected transcendental functions and can be easily extended for newer functions

Summary

Advantages:

- Simple to use – just set an environment variable
- No need of source code

Disadvantage:

- Works for only dynamically linked functions

Compile Time Function Interception

Compile Time Approach

Advantages:

- More generic- can be applied to any memoizable function
- Facilitates inlining of memoization wrapper
- Allows to handle functions having pointers and global variable usage
- Constants can be handled efficiently

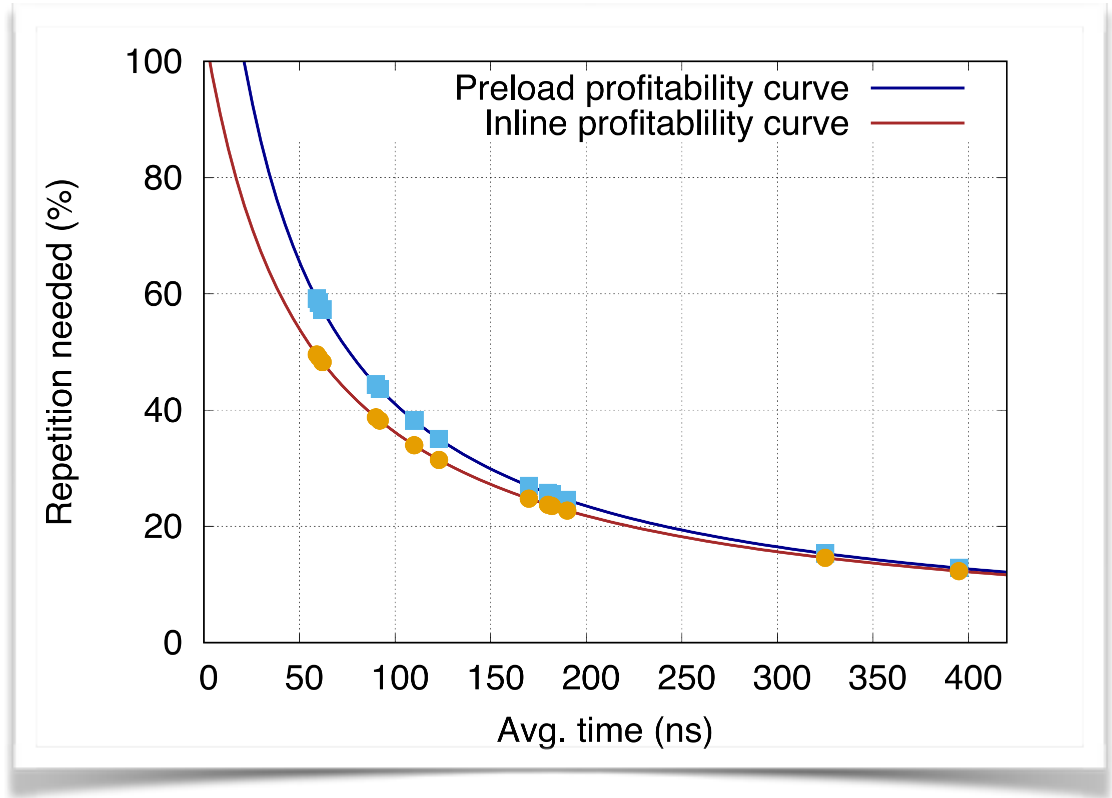
Disadvantage:

- Requires recompilation and hence the availability of user code

Profitability Curve

2 GHz Intel Ivybridge

Function	Hit Time (ns)	Overhead (ns)	Avg. Time (ns)	Repetition Needed (%)
exp	21	55	90	44
log			92	43
11			110	38
cos			123	35
j0			395	13
j1			325	15
pow			190	25



- Inlining lowers the memoization threshold
- Smaller functions require lower repetition rates

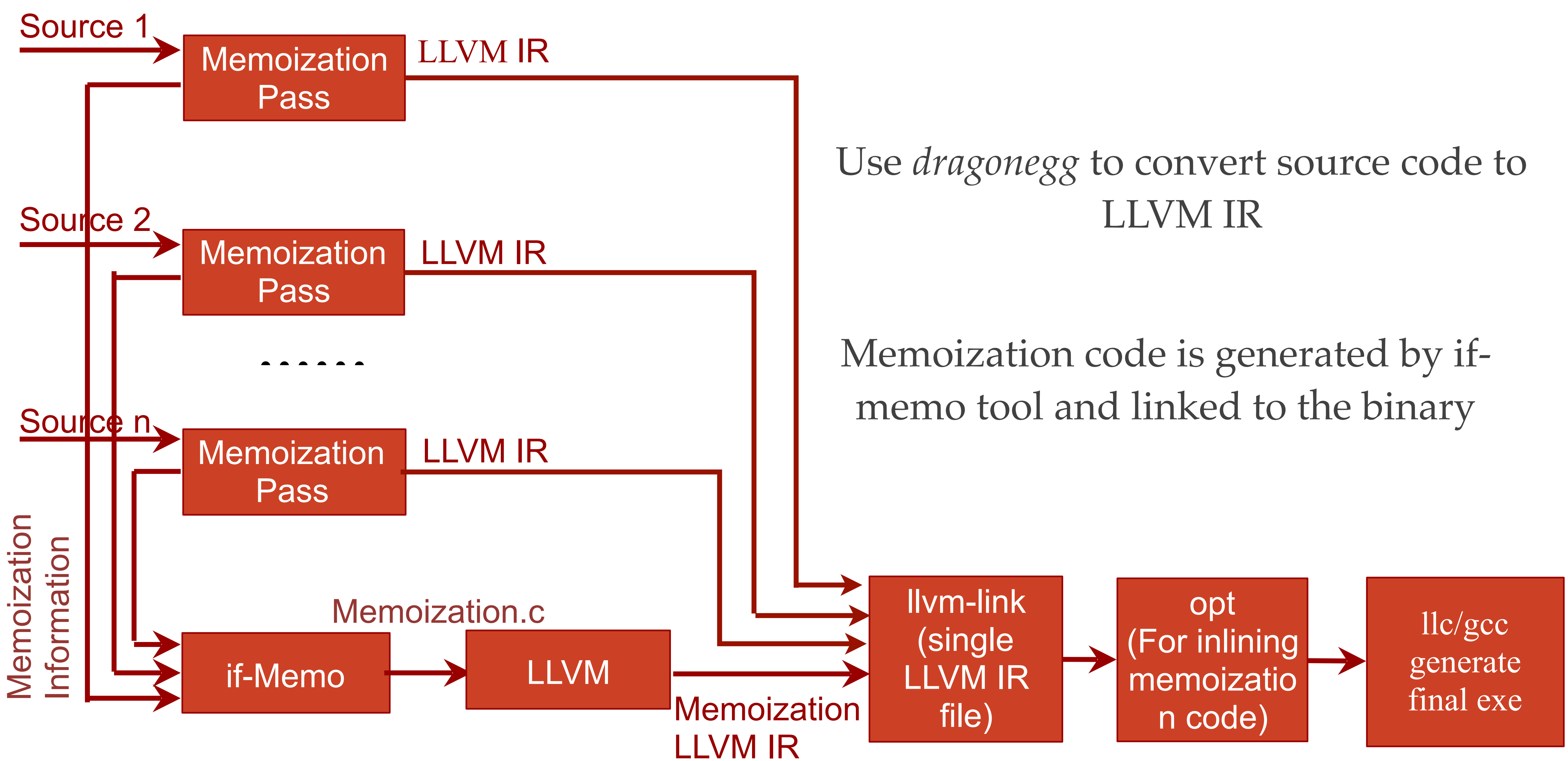
Implementation

- Use LLVM Compiler Infrastructure
- Memoization Pass to identify potential functions
- Re-use the memoization framework — extending "if-memo" tool

Memoization Pass

- Is a transform pass at module level
- Checks each function to be memoizable — pure and calling only pure functions — and generates the corresponding information in a file
- This information — function name and prototype — is used at link time for generating and linking the memoization code

LLVM workflow for Memoization



Use *dragonegg* to convert source code to LLVM IR

Memoization code is generated by if-memo tool and linked to the binary

Function Prototypes

- We handle up to 2 parameters for memoization excluding constants
- Requires a large number of function prototypes to be handled
- Sort the parameters based on their type to reduce the number of memoization function types

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*
```

Pointers

Pointed value is the concern

- RD Only – Pointed value handled like a normal argument. Takes part in table indexing
- WR Only – Pointed value is not used for table indexing, but just to store the result in the memoization table
- RD/WR – Pointed value is used for table indexing and final value is stored in table

Constants

Example:

- foo (2,x) -> memoized_foo(x)
- Memoized_foo(x) calling foo(2,x)

- Facilitated by passing a call string to the memoization wrapper
- Memoization becomes call site specific

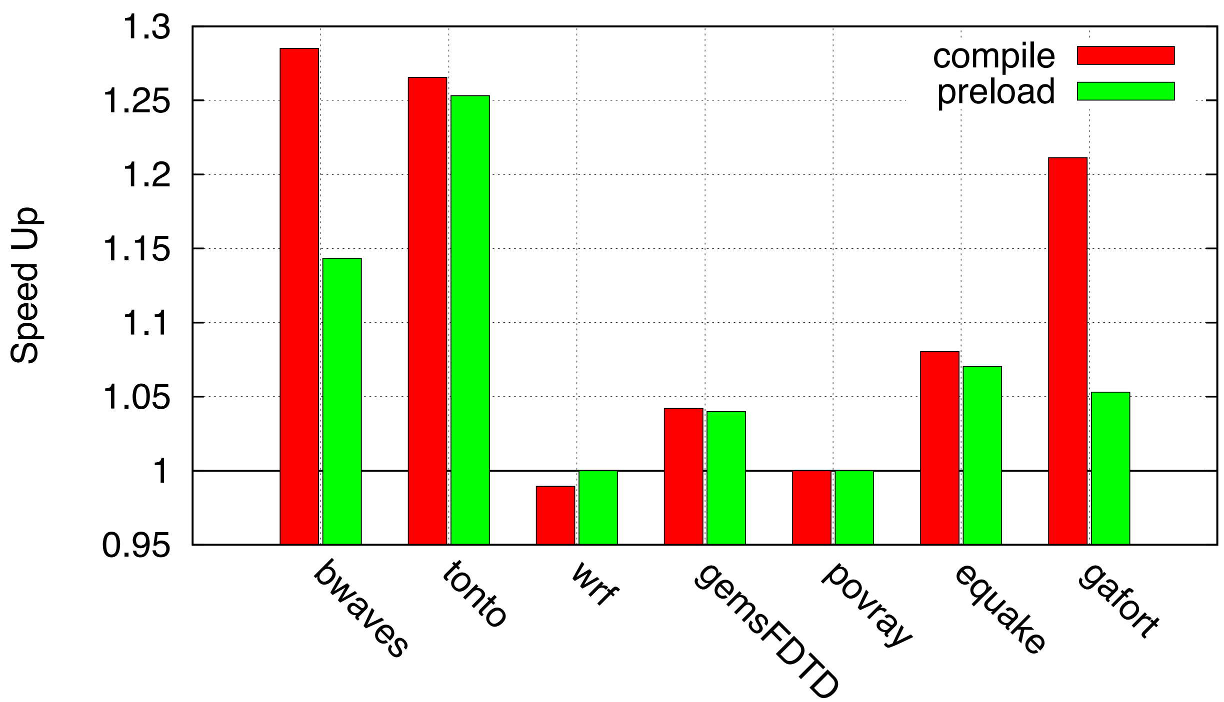
Experimental Set-up

- *blackscholes* from PARSEC and *histo* from Parboil suite are additional benchmarks
- Experiments run on Intel Ivybridge

Processor	Intel Core i7
L3 Cache	8 MB
Clock Speed	2.3 GHz
RAM	8 GB
Linux Version	3.19
llvm version	3.7
optimisation flag	-O3

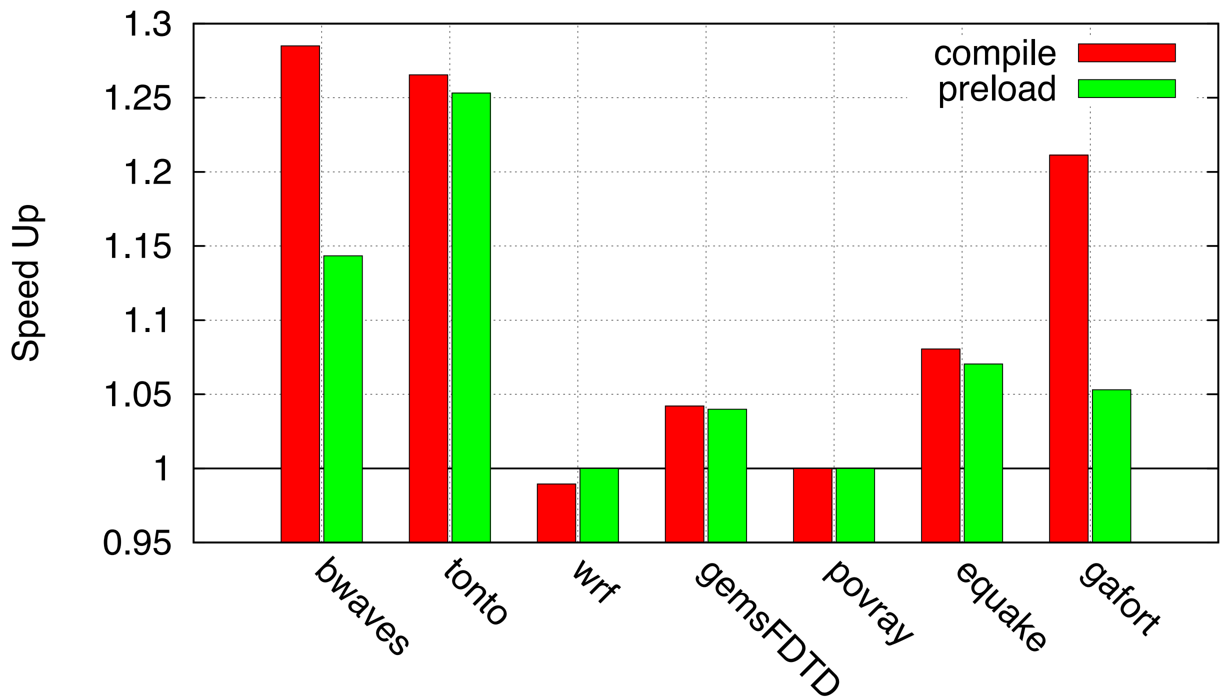
Results: Compile time vs Preload

SPEC benchmarks



Results: Compile time vs Preload

Selected Applications

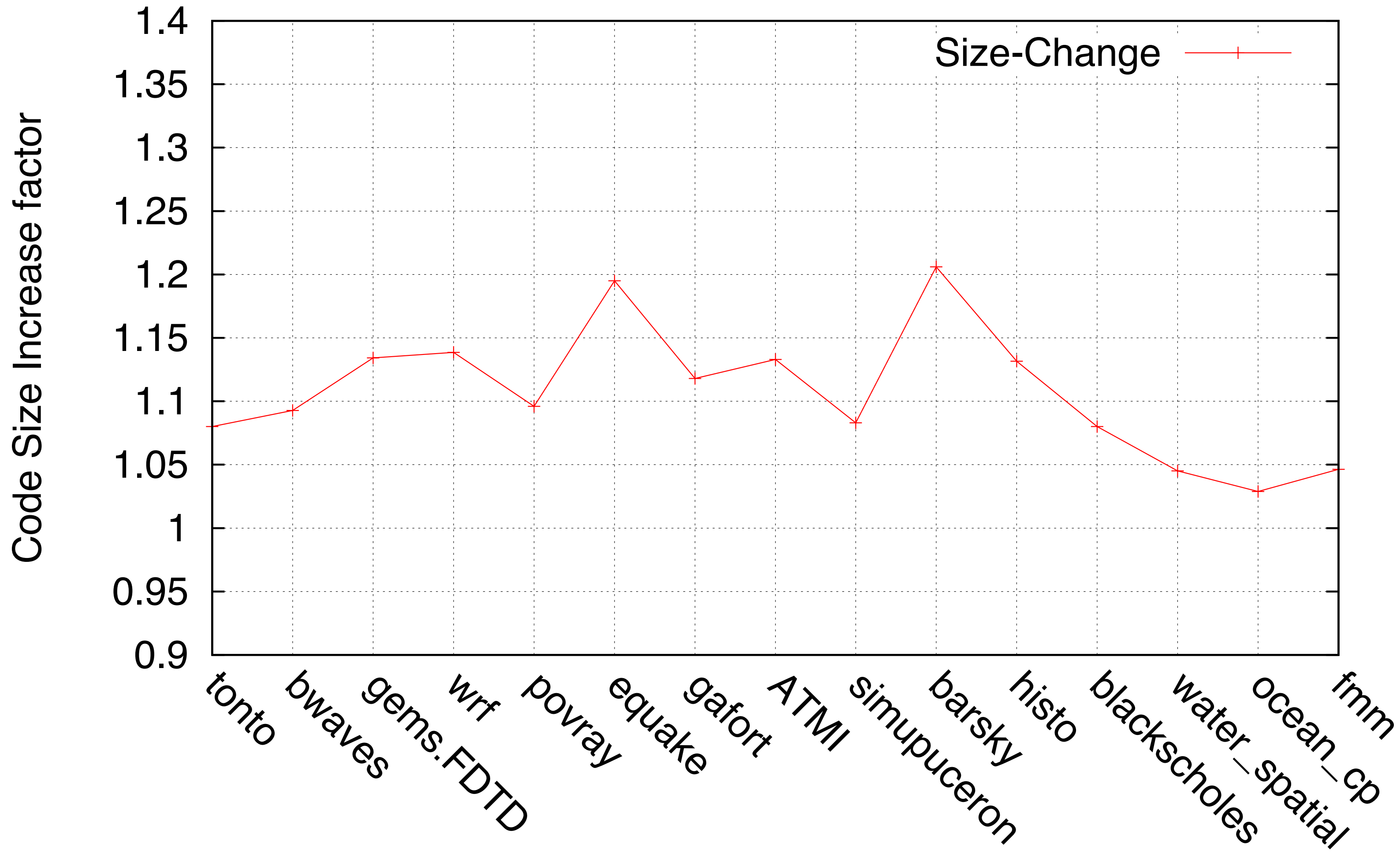


Results

- Most benchmarks gave better runtime compared to LD_PRELOAD approach
- Performance improvement for *histo* and *blackscholes* where memoized functions were user defined (statically linked)

Increase in Code Size

- Inlining memorisation wrapper increases the code size
- Code size increase can affect run time



Summary

- Compile Time approach enables more functions to be memoized
- Increases the efficiency of memoization
- Code size is increased but not much performance degradation

Can we further increase performance benefit?

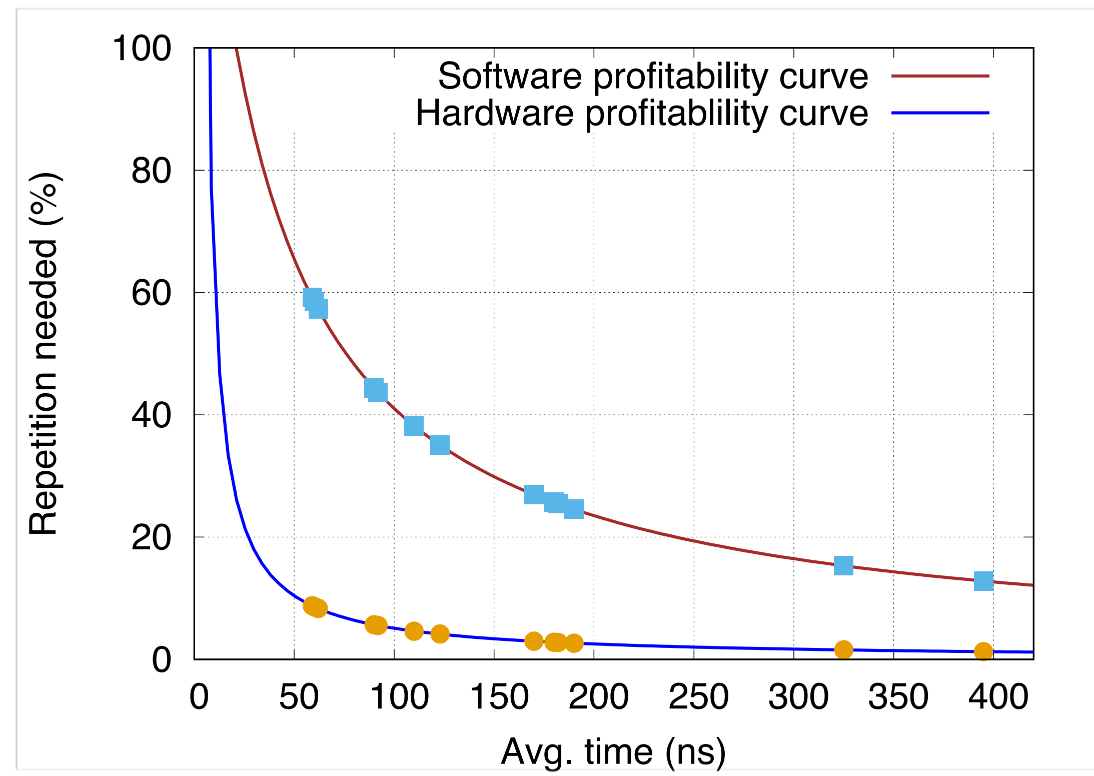
Hardware Memoization

Hardware Support

- Makes memoization more efficient
  - Reduces overhead by faster indexing
  - Faster table look-up
- Enables parallel look-up/execution and hence almost no overhead of look-up failure
- Associativity becomes more practical
- Smaller memoization table size
- Centralised table
- Introduces branch mis-prediction penalty in case of a miss prediction in table look-up

Profitability Curve

Function	Hit Time - t_h (ns)	Miss-prediction Penalty - t_{pen}(ns)	Avg. Time- T_f (ns)	Repetition Needed - H (%)
exp	2	20	90	2
log			92	2
sin			110	2
cos			123	2
j0			395	1
j1			325	1
pow			190	1



$$H \times t_h + (1 - H)(T_f) + t_{pen} < T_f$$

$$\implies H > \frac{t_{pen}}{T_f - t_h}$$

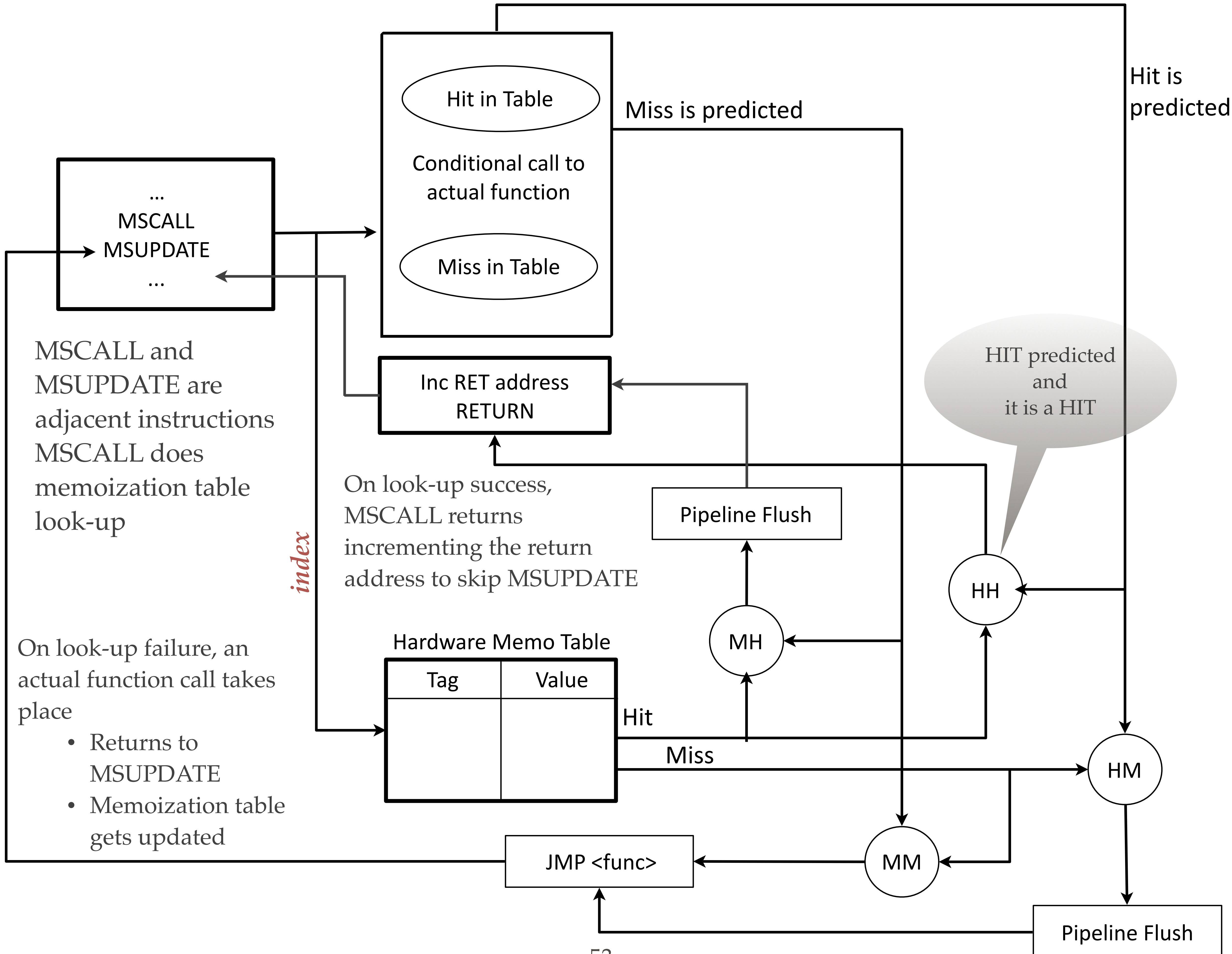
Average mis-prediction penalty

New Instructions

- MSCALL – does indexing and table look-up
- MSUPDATE – updates the memoization table
- Both have different variants for each function prototype being handled
- These two instructions are doing the same function as done by "if-memo" tool but in hardware

Working

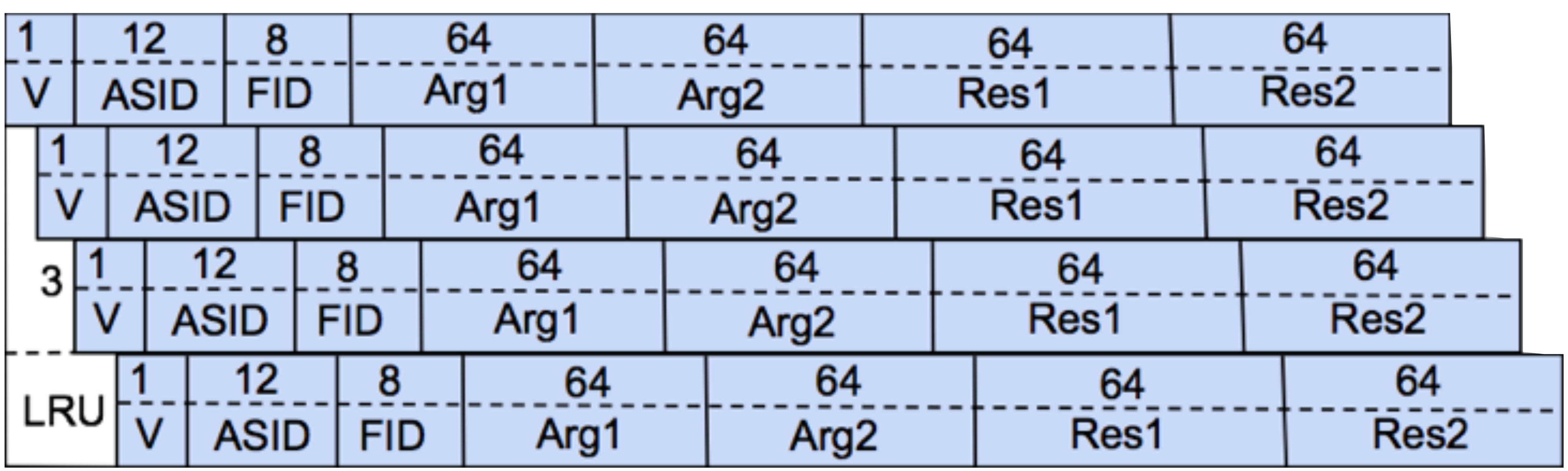
- Same scheme as for compile time approach in identifying memoizable functions
- Instead of calling memoization wrapper, MSCALL and MSUPDATE instructions are inserted at each call site of a memoizable function by the compiler
- Low latency functions also become memoizable now - example *sqrt*



...
MSCALL
MSUPDATE
...
Hit in Table
Conditional call to actual function
Miss in Table
Miss is predicted
Hit is predicted
MSCALL and MSUPDATE are adjacent instructions MSCALL does memoization table look-up
Inc RET address
RETURN
HIT predicted and it is a HIT
index
On look-up success, MSCALL returns incrementing the return address to skip MSUPDATE
Pipeline Flush
HH
MH
Hardware Memo Table
Tag
Value
Hit
Miss
HM
On look-up failure, an actual function call takes place
• Returns to MSUPDATE
• Memoization table gets updated
JMP <func>
MM
Pipeline Flush

Hardware Memoization Table

- Centralised
- Associative
- Pseudo-LRU policy using 3 LRU bits for replacement



Evaluation

- Applications are run with hardware memoization table being implemented in software
- Hit rates and function run times are measured
- Hardware runtime estimated using the found hit rates and the run times assuming 4 clock cycle table look-up time
- No extra overhead in case of table look-up failure
- Table look-up prediction failure causes a pipeline flush

Hardware Time Estimation

Application runtime with hardware memoization is calculated as

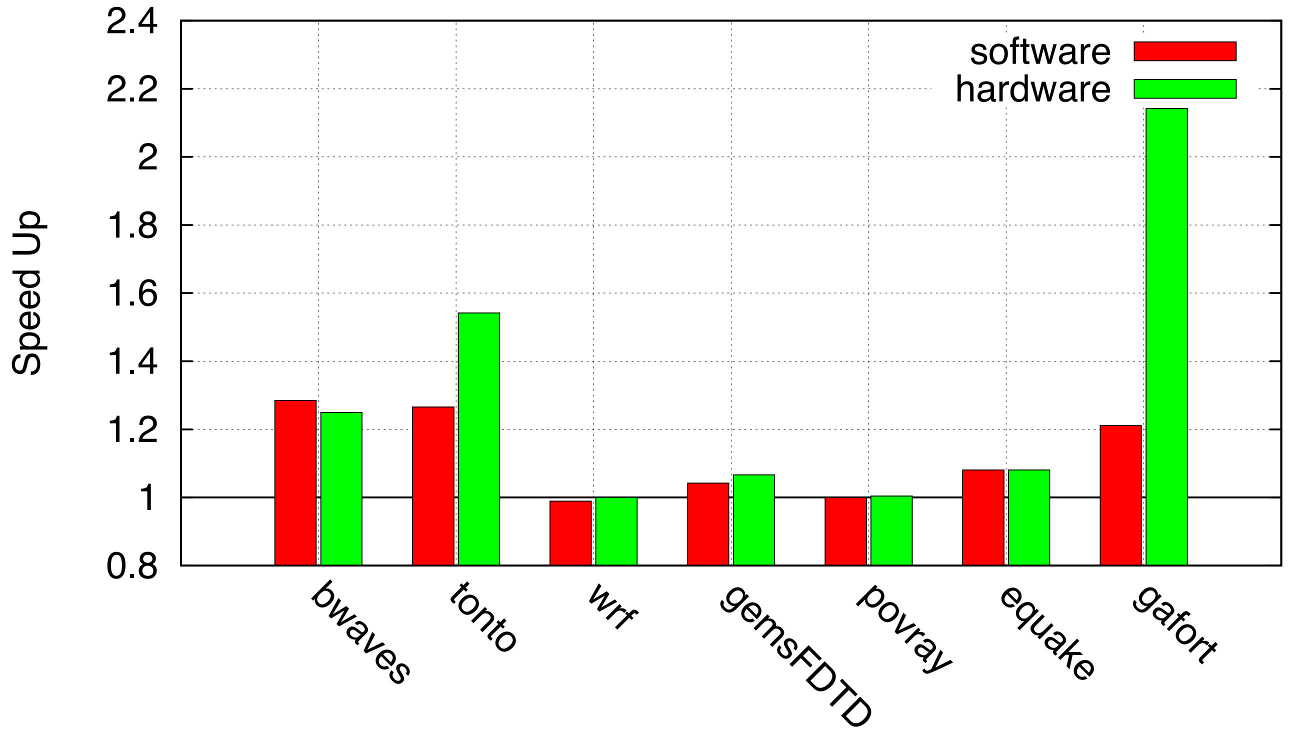
$$H_{time} = E_{time} + \sum_{i=1}^{n} hw_{time_i} + P_{b_i},$$

$\mathbf{E_{time}}$ - application execution time excluding the memoizable functions
$\mathbf{hw_{time_i}}$ - estimated hardware runtime for function *i*
$\mathbf{P_{b_i}}$ - average branch mis-prediction penalty for function *i*

Branch mis-prediction is measured using a single counter software predictor (4 bits and 16 states) and given a 20 cycle penalty on mis-prediction

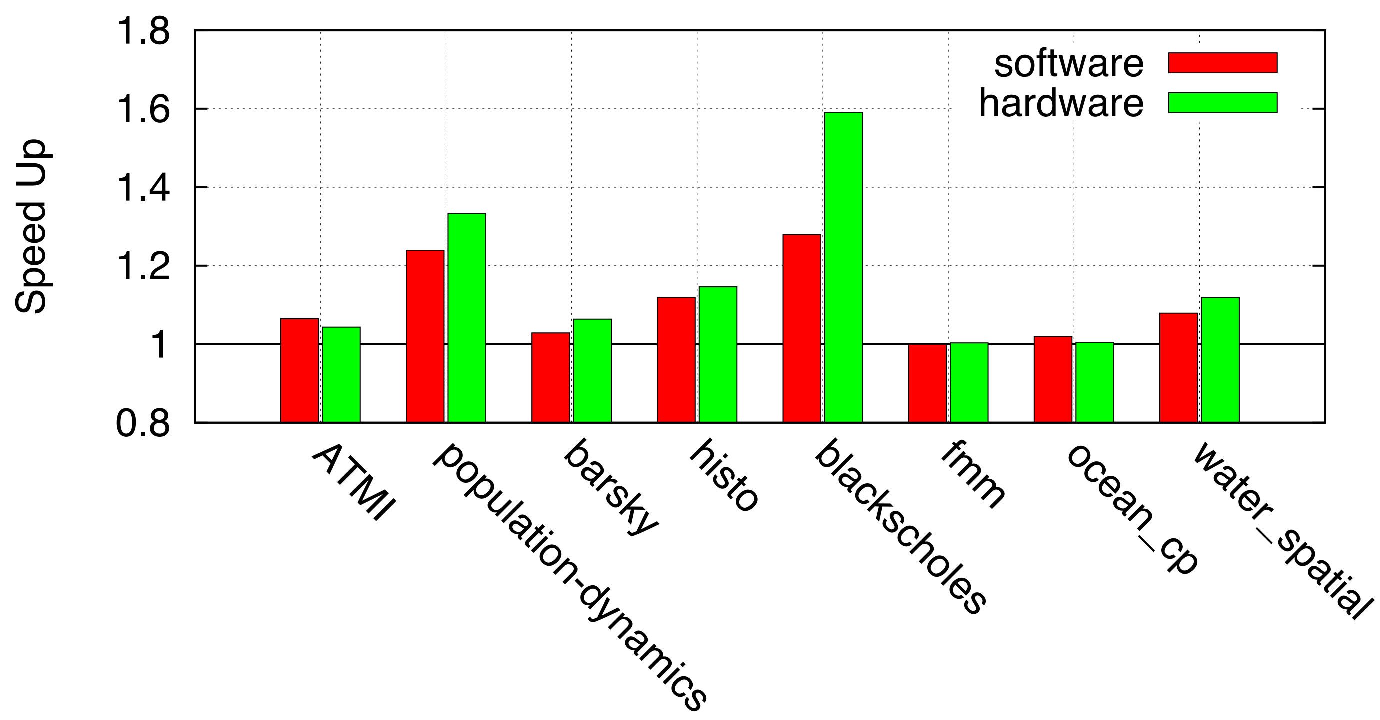
Result: Memoization with Hardware Support

SPEC benchmarks



Result: Memoization with Hardware Support

Selected Applications



Results

- Most benchmarks have a much better speed-up compared to compile time approach
- Some benchmarks — *ocean_cp, bwaves, ATMI* — have a lower speed-up mainly due to increased collision in the memoization table (table being smaller in hardware)
- Very high speed-up for *equake* as the memoized functions were small and highly critical

Conclusion

- At present Memoization gives speed up even in software
- Arguments do repeat a lot in real applications
- A simple LD_PRELOAD technique is good for memoizing dynamically linked functions
- Compile time approach allows memoization of user defined functions and lowers the threshold of memoization by inlining
- Hardware support can further increase the memoization efficiency

Further Extensions

- Hardware Memoization with a back-up software memoization
- Pre-filling memoization table
  - Similar to pre-fetching
  - Requires a helper thread to be made using just enough code to generate the argument to function
  - Requires proper synchronisation with main thread
- Applicability to approximation domains

Thank You