Leveraging Hardware Caches for Memoization

Guowei Zhang Daniel Sanchez

Abstract—Memoization improves performance and saves energy by caching and reusing the outputs of repetitive computations. Prior work has proposed software and hardware memoization techniques, but both have significant drawbacks. Software memoization suffers from high runtime overheads, and is thus limited to long computations. Conventional hardware memoization techniques achieve low overheads and can memoize short functions, but they rely on large, special-purpose memoization caches that waste significant area and energy.

We propose MCACHE, a hardware technique that leverages data caches for memoization. MCACHE stores memoization tables in memory, and allows them to share cache capacity with normal program data. MCACHE introduces ISA and pipeline extensions to accelerate memoization operations, bridging the gap between software and conventional hardware techniques. Simulation results show that MCACHE improves performance by up to 21×, outperforms software memoization by up to 2.2×, and achieves similar or superior performance over conventional hardware techniques without any dedicated storage.

1 INTRODUCTION

The impending end of Moore's Law is making transistors a scarce resource. Therefore, it is crucial to investigate new abstractions and architectural mechanisms that make better use of existing hardware. The memory system is ripe for this type of optimization: current memory hierarchies employ sophisticated hardware caches, but are hampered by a narrow load/store interface that limits their utility.

In this work we focus on *memoization*, a natural technique to support in the memory system. Memoization caches the results of repetitive computations, allowing the program to skip them. Memoized computations must be pure and depend on few, repetitive inputs. Prior work has proposed software and hardware implementations of memoization, but both have significant drawbacks.

Software memoization is hampered by high runtime overheads [3, 14]. The software caches used to implement memoization take tens to hundreds of instructions per lookup. This limits software memoization to long computations, e.g., with thousands of cycles or longer. But as we later show, many memoizable functions are merely 20 to 150 instructions long. Software techniques not only cannot exploit these short functions, they must perform a careful cost-benefit analysis to avoid memoizing them [6, 7], which would carry heavy penalties.

Prior work has proposed hardware support to accelerate memoization [3, 14, 16]. Accelerating table lookups unlocks the benefit of memoizing short code regions. However, prior techniques incur large hardware overheads because they introduce special-purpose memoization caches. These structures are large, rivaling or exceeding the area of the L1 cache. For example, Da Costa et al.'s proposal [5] consumes 98 KB. While memoization is very effective for applications with repetitive computations, not all applications can benefit from it. In these cases, this dedicated storage not only wastes area that could otherwise be devoted to caches, but also hurts energy consumption [2].

To address the drawbacks of software and conventional hardware memoization, we propose MCACHE, a hardware technique that leverages data caches to accelerate memoization with minimal overheads. Unlike prior hardware techniques, MCACHE stores memoization tables in memory, allowing them to share cache capacity with conventional data. MCACHE then introduces new instructions to perform memoization lookups and updates. These instructions are designed to leverage existing core structures and prediction mechanisms. For example, memoization lookups have branch semantics and thus leverage the core's branch predictors to avoid control-flow stalls.

As a result, MCACHE achieves the low overheads of prior hardware techniques at a fraction of their cost. Simulations show that MCACHE improves performance by up to 21×, outperforms software memoization by up to 2.2×, and achieves comparable performance to conventional hardware techniques without requiring any dedicated storage.

2 BACKGROUND

Memoization was first introduced by Michie in 1968 [10]. Since then, it has been implemented using software and hardware.

Software memoization is the cornerstone of many important algorithms, such as dynamic programming, and is widely used in many languages, especially functional ones. Software typically implements per-function software caches, e.g., using hash tables that store function arguments as keys and results as values. Citron et al. [3], among others, show that software memoization incurs significant overheads on short functions: when memoizing mathematical functions indiscriminately, software memoization incurs a 7% performance loss, while a hardware approach yields 10% improvement. To avoid performance loss, software-based schemes apply memoization selectively, relying on careful cost-benefit analysis of memoizable regions, done by either compilers [7, 14], profiling tools [6], or programmers [15].

Hardware memoization techniques reduce these overheads and thus can unlock more memoization potential. Much prior work on hardware memoization focuses on automating the detection of memoizable regions at various granularities [5, 8, 13, 16], while others rely on ISA and program changes to select memoizable regions [3, 4, 14]. However, all prior hardware techniques require dedicated storage for memoization tables. Such tables require similar or even larger sizes than L1 caches. Therefore, they incur significant area and energy overheads [2], especially for programs that cannot exploit memoization.

Other prior work has proposed architectural [17] or runtime [11] support to track implicit inputs/outputs of memoized functions, enabling memoization of some impure functions. This support is orthogonal to the implementation of the memoization mechanisms, which is the focus of our work. MCACHE could be easily combined with these mechanisms.

3 MCACHE DESIGN

MCACHE leverages two key features to bridge the gap between software and conventional hardware memoization.

First, MCACHE stores the memoization tables in cacheable memory. This avoids the large costs of specialized memoization caches used by conventional hardware techniques, and allows the capacity in the cache hierarchy to be shared between memoization data and normal program data. MCACHE adopts a format for memoization tables that exploits the characteristics of caches to make lookups fast.

• *The authors are with MIT CSAIL, Cambridge, MA, 02139.*
E-mail: {zhanggw,sanchez}@csail.mit.edu

• *This work was supported in part by C-FAR, one of six SRC STARnet centers by MARCO and DARPA, and by NSF grant CAREER-1452994.*
• *Manuscript submitted: 13-Jun-2017. Manuscript accepted: 01-Sep-2017. Final manuscript received: 09-Sep-2017.*

Second, MCACHE introduces two memoization instructions that are amenable to a fast and simple implementation. MCACHE achieves much faster table lookups than software techniques, where a memoization lookup is implemented as a sequence of instructions that include one or more memory accesses to fetch keys and values, comparisons, and hard-to-predict branches. Though these operations are not complex, they cause frequent stalls that hurt performance. Instead, MCACHE memoization lookups are done through a single instruction with branch semantics. The outcome of a memoization lookup (hit or miss) can be predicted accurately by the core's existing branch predictors, avoiding control-flow stalls.

3.1 Memoization table format

MCACHE uses per-function memoization tables, allowing each table to have a homogeneous entry format. Each memoization table is stored in a contiguous, fixed-size region of physical memory, as shown in Fig. 1.

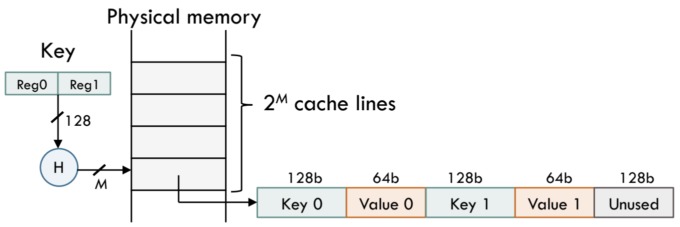


Fig. 1: MCACHE memoization table format.

MCACHE uses a storage format designed to leverage the characteristics of caches. Each cache line stores a fixed number of key-value (i.e., input-output) pairs. For example, Fig. 1 shows the format of a 64-byte cache line for a function with 128-bit keys (i.e., arguments) and 64-bit values. A given entry can map to a single cache line, but can be stored in any position within the line. A lookup thus requires hashing the input data to produce a cache line index, fetching the line at that index (as shown in Fig. 1), and comparing all keys in the line. This design requires accessing a single cache line, but retains associativity within a line to reduce conflict misses. Unlike hash tables, memoization tables do not grow to accommodate extra items. Insertions simply replace one of the line's entries, selected at random.

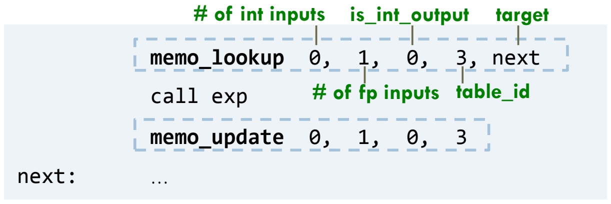
MCACHE uses physical memory for each memoization table, so lookups and insertions, which happen through special instructions, do not check the TLB. The OS reserves and initializes this physical memory, which is not mapped to the program's address space. To avoid the need for valid bits, the OS initializes each line's entries with keys that map to a different line.

3.2 MCACHE ISA extensions

MCACHE provides support for a small number of memoization tables. Cores store the starting address and size of each table in architectural registers. Our implementation supports four table ids, which suffice to have per-function memoization tables on all applications we study. Larger applications could select which functions to memoize or share each table among multiple functions (by including the function pointer as one of the arguments).

MCACHE adds two instructions, `memo_lookup` and `memo_update`. Fig. 2 shows these instructions in action when they are used to memoize the `exp` function.

`memo_lookup` has branch semantics. It performs a lookup in the memoization table specified by **table_id**. If the lookup is a memoization hit, `memo_lookup` acts as a taken branch, setting the PC to the **target** encoded in the instruction (in PC-relative



```
            memo_lookup  0,  1,  0,  3, next
            call exp
            memo_update  0,  1,  0,  3
next:       ...
```

Fig. 2: Example showing MCACHE instructions used to memoize `exp`.

format) and updating the result register with the memoized value. On a memoization miss, `memo_lookup` acts as a non-taken branch. For example, in Fig. 2, a memoization hit jumps to `next`, skipping the call to `exp`.

`memo_lookup` supports functions with up to four integer or floating-point arguments and a single integer or floating-point result, all stored in registers. We leverage the ISA's calling convention to compactly encode the input registers used. As shown in Fig. 2, `memo_lookup` stores only the number of integer and floating-point input registers, and the core decodes them to register numbers. For instance, in x86-64, **num_int_inputs** = 2 means that `rdi` and `rsi` are used as inputs.

`memo_update` is used to update the memoization table upon a memoization miss. Like `memo_lookup`, `memo_update` encodes the input and output registers, and the table id. For example, in Fig. 2, `memo_update` stores the result of the `exp` function in the memoization table.

3.3 MCACHE microarchitecture

MCACHE requires simple changes to cores. We implement MCACHE using a non-pipelined functional unit that executes lookup and insertion instructions. This unit is fed the values of input registers, possibly over multiple cycles, as well as the table id. For a `memo_lookup` instruction, the unit first hashes the input values and table size to find the line index. We restrict the system to use power-of-2 sizes for each memoization table and use XOR-folding to compute the hash value. This is simple and produces good distributions in practice. Then, the MCACHE functional unit loads the appropriate cache line, compares all the keys, and outputs whether there's a match, as well as the memoized result if so. `memo_update` is similar, but the functional unit also takes the result to memoize, and stores the key-value pair in the appropriate cache line.

MCACHE leverages existing core mechanisms to improve performance. We integrate these instructions into an x86-64 core similar to Intel's Nehalem (see Sec. 4.1). The frontend treats `memo_lookup` as a branch, using the branch target buffer and branch predictor to predict whether the lookup will result in a memoization hit. This way, the core overlaps the execution of the lookup with either the execution of the memoized function (if a memoization miss is predicted) or its continuation (if a memoization hit is predicted). We find that this effectively hides the latency of memoization lookups.

In our implementation, the backend executes `memo_lookup` using multiple RISC micro-ops (μops): the decoder produces one or more μops that feed input registers to the MCACHE functional unit, a branch-resolution μop, and, if the lookup is predicted to hit, a μop to move the memoized result into its destination register. `memo_update` uses a similar implementation.

4 EVALUATION

4.1 Methodology

Modeled system: We perform microarchitectural, execution-driven simulation using zsim [12]. We evaluate a single-core OOO processor with parameters shown in Table 2. MCACHE

TABLE 1
Benchmark characteristics.

	Language	Benchmark suite	Input set	Memoizable functions	Memoization table size per func
410.bwaves	Fortran	SPEC CPU 2006	ref	slowpow, pow, halfulp, exp1	256 KB
bscholes	C++	PARSEC	native	CDNF, exp, logf	64 KB
equake	C	SPEC OMP 2001	ref	phi0, phi1, phi2	2 KB
water	C	SPLASH2	1061208	exp	16 KB
103.semphy	C++	BioParallel	220	suffStatGlobalHomPos::get	4 KB
352.nab	C	SPEC OMP 2012	ref	exp, slowexp_avx	4 KB

TABLE 2
Configuration of the simulated system.

Core	x86-64 ISA, 2.0 GHz, Nehalem-like OOO: 16B-wide ifetch, 2-level bpred with 2 K×18-bit BHSRs + 4 K×2-bit PHT, 4+1+1+1 decoders, 6 execution ports, 4-wide commit
L1 cache	64 KB, 4-way set-associative, 3-cycle latency, split D/I
L2 cache	2 MB, 16-way set-associative, 15-cycle latency, inclusive
Main memory	1 controller, 120-cycle latency

supports four table ids, and lookups incur the cost of a cache-line load, plus two cycles to perform key comparisons.

Exploiting memoizable regions: We developed a pintool [9] to identify memoizable (i.e., pure) functions. Then, we manually added `memo_lookup` and `memo_update` instructions to these functions' callsites. We encode these instructions using x86-64 no-ops that are never emitted by the compiler. Due to its low overheads, MCACHE does not need to perform selective memoization based on cost-benefit analysis as in software techniques. Therefore, we memoize every function that our tool identifies as memoizable. We memoize both user-defined and standard-library functions.

Workloads: We analyze programs from six benchmark suites and choose one application with high memoization potential from each suite. Table 1 details these applications and their memoization characteristics. For each application, we use the same memoization table size for all memoized functions. We report the table size that yields the best performance. Sec. 4.3 provides more insight on the effect of table size.

We fast-forward each application for 50 B instructions. We instrument each program with heartbeats that report application-level progress (e.g., when each timestep or transaction finishes), and run the application for as many heartbeats as the baseline system (without memoization) completes in 5 B instructions. This lets us compare the same amount of work across schemes, since memoization changes the instructions executed.

4.2 MCACHE vs baseline

Fig. 3 compares the performance of MCACHE over the baseline, which does not perform memoization. MCACHE improves performance substantially, by 21× on `bwaves`, 8.2× on `bscholes`, 72% on `equake`, 27% on `water`, 22% on `semphy`, and 4% on `nab`.

Table 3 gives more details into these results by reporting per-function statistics. For example, in `bwaves`, memoizing the `pow` function provides most of the benefits. `pow` takes thousands of instructions to calculate x^y if x is close to 1 and y is around 0.75, which is common in `bwaves`. Memoizing `pow` contributes to 99.9% of the instruction reduction in `bwaves`.

Beyond reducing execution time, MCACHE reduces the number of L1 cache accesses significantly, as shown in Fig. 4: L1 access reductions range from 9% on `nab` to 97% on `bwaves`. This happens because the L1 accesses saved through memoization hits exceed the additional L1 accesses incurred by memoization operations. Moreover, MCACHE does not incur much extra capacity contention in L1 caches. Fig. 5 shows that MCACHE increases L1 data cache misses by less than 3% overall. One

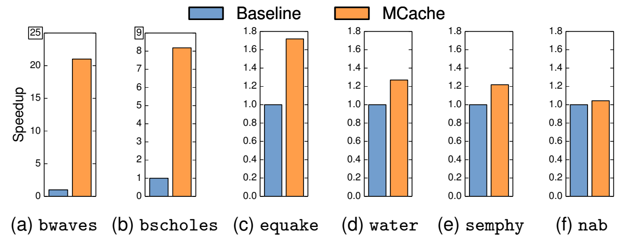


Fig. 3: Per-application speedups on MCACHE. Higher is better.

TABLE 3
Per-function breakdown of `memo_lookups`.

	Function	Instrs/call	# `memo_lookups`	Hit rate
410.bwaves	slowpow	485160	311	4.5%
	pow	12947	370245	97.7%
	halfulp	77	297	0.0%
	exp1	28	8866	5.6%
bscholes	CDNF	193	15547102	99.9%
	exp	115	7789732	100.0%
	logf	56	7773551	100.0%
equake	phi0	119	7953687	100.0%
	phi1	123	7953687	100.0%
	phi2	118	7953687	100.0%
water	exp	116	7806240	100.0%
103.semphy	suffStatGlobal-HomPos::get	19	67123200	94.6%
352.nab	exp	81	29150493	49.8%
	slowexp_avx	14756	0	N/A

exception is `bscholes`, which incurs 36% more L1 misses on MCACHE. However, this is not significant, because the baseline's L1 miss rate is only 0.2%. In fact, such misses bring in valuable memoization data that in the end improve performance by 8.2×. On `equake`, MCACHE even reduces L1 data misses by 3%.

4.3 MCACHE vs conventional hardware memoization

We implement a conventional hardware memoization technique that leverages MCACHE's ISA and pipeline changes, but uses a dedicated storage buffer like prior work [1, 16] instead of using the memory system to store memoization tables. Beyond its large hardware cost, the key problem of conventional hardware memoization is its lack of flexibility: a too-large memoization

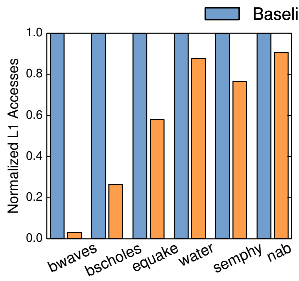


Fig. 4: L1 data cache accesses on baseline and MCACHE, relative to baseline. Lower is better.

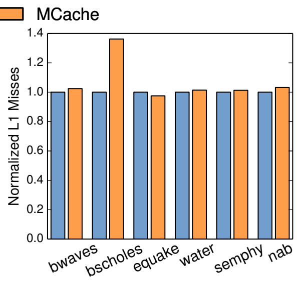


Fig. 5: L1 data cache misses on baseline and MCACHE, relative to baseline. Lower is better.

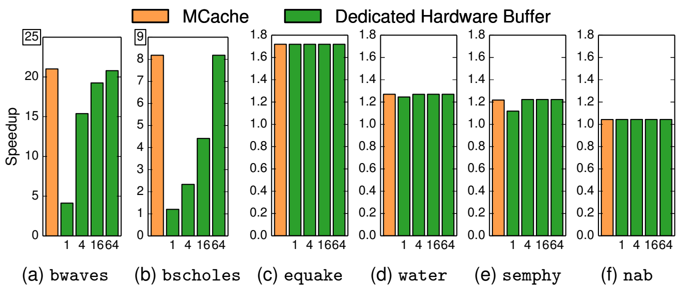


Fig. 6: Per-application speedups of MCache and conventional hardware memoization with different dedicated buffer sizes (in KB). Higher is better.

buffer wastes area and energy, while a too-small memoization buffer sacrifices memoization potential.

Fig. 6 quantifies this problem by showing the performance of hardware memoization across a range of memoization buffer sizes: 1, 4, 16, and 64 KB. The buffer is associative, and entries are dynamically shared among all memoized functions. We optimistically model a 1-cycle buffer access latency.

Fig. 6 shows that applications are quite sensitive to memoization buffer size: 1 KB is sufficient for `equake` and `nab`, while `water` and `semphy` prefer at least 4 KB, and `bwaves` and `bscholes` prefer at least 64 KB. Smaller buffers than needed by the application result in increased memoization misses and sacrifice much of the speedup of memoization.

Finally, Fig. 6 shows that MCache matches the performance of hardware memoization with a dedicated storage size of 64 KB on all applications. This is achieved even though MCache does not require any dedicated storage, saving significant area and energy. The tradeoff is that storing memoization tables in memory causes longer lookup latencies than using a dedicated buffer. However, these lookup latencies are small, as they mostly hit on the L1 or L2, and branch prediction effectively hides this latency most of the time.

4.4 MCache vs software memoization

We implement software memoization using function wrappers similar to Suresh et al. [15]. Per-function memoization tables are implemented as fixed-size, direct-mapped hash tables, accessed before calling the memoizable function and updated after a memoization miss.

Fig. 7 shows the performance of MCache and software memoization. MCache outperforms software memoization by 1.3% on `bwaves`, 2.1× on `bscholes`, 17% on `equake`, 7% on `water`, 2.2× on `semphy`, and 32% on `nab`.

MCache outperforms software memoization due to its low overheads. For example, `semphy`'s memoizable function runs for 19 instructions on average, too short for software memoization. As a result, software memoization is 86% slower than the baseline. This explains why software memoization needs careful cost-benefit analysis to avoid performance degradation. By contrast, MCache improves performance by 22% on `semphy`, outperforming software memoization by 2.2×. Similarly, software memoization makes `nab` 27% slower, while MCache improves performance by 4%.

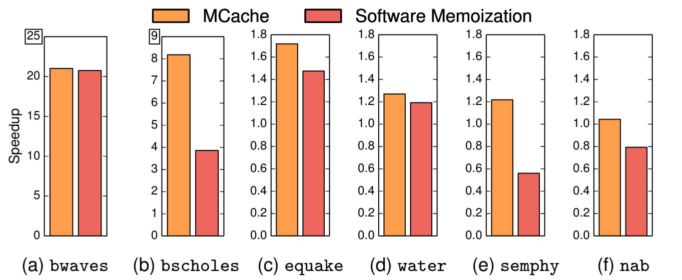


Fig. 7: Per-application speedups of MCache and software memoization with per-function direct-mapped hash tables. Higher is better.

5 Conclusion

We have presented MCache, a technique that leverages data caches for memoization. Unlike prior hardware memoization techniques, MCache requires no dedicated buffers. Instead, MCache stores memoization tables in memory, leveraging the cache hierarchy to achieve low lookup latency and the core's branch prediction machinery to take this latency off the critical path. MCache introduces ISA extensions and pipeline changes to achieve fast lookups and updates to memoization tables.

We have shown that MCache solves the dichotomy of hardware vs software memoization: MCache matches the performance of conventional hardware techniques, but avoids the overheads of large dedicated buffers and supports arbitrarily-sized memoization tables, just like software memoization. As a result, MCache improves performance by up to 21×, outperforming software techniques significantly and saving substantial area and energy compared to conventional hardware techniques. By making memoization practical for short code regions, MCache opens interesting avenues for future work, which could explore compiler techniques to memoize at sub-function granularity. Future work could also explore auto-tuning techniques to dynamically adjust memoization table sizes to maximize performance.

References

[1] P. Chen, K. Kavi, and R. Akl, "Performance enhancement by eliminating redundant function execution," in *ANSS-39*, 2006.
[2] B.-S. Choi and J.-D. Cho, "Partial resolution for redundant operation table," *Microprocessors and Microsystems*, vol. 32, no. 2, 2008.
[3] D. Citron and D. G. Feitelson, "Hardware memoization of mathematical and trigonometric functions," HUJI, Tech. Rep., 2000.
[4] D. Connors and W.-M. Hwu, "Compiler-directed dynamic computation reuse: rationale and initial results," in *MICRO-32*, 1999.
[5] A. T. Da Costa, F. M. G. França, and E. M. C. Filho, "The dynamic trace memoization reuse technique," in *PACT-9*, 2000.
[6] L. Della Toffola, M. Pradel, and T. R. Gross, "Performance problems you can fix: A dynamic analysis of memoization opportunities," in *OOPSLA*, 2015.
[7] Y. Ding and Z. Li, "A compiler scheme for reusing intermediate computation results," in *CGO*, 2004.
[8] J. Huang and D. J. Lilja, "Exploiting basic block value locality with block reuse," in *HPCA-5*, 1999.
[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[10] D. Michie, "Memo functions and machine learning," *Nature*, vol. 218, no. 5136, 1968.
[11] H. Rito and J. Cachopo, "Memoization of methods using software transactional memory to track internal state dependencies," in *PPPJ-8*, 2010.
[12] D. Sanchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA-40*, 2013.
[13] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *ISCA-24*, 1997.
[14] A. Suresh, E. Rohou, and A. Seznec, "Compile-Time Function Memoization," in *CC-26*, 2017.
[15] A. Suresh, B. N. Swamy, E. Rohou, and A. Seznec, "Intercepting functions for memoization: a case study using transcendental functions," *ACM TACO*, vol. 12, no. 2, 2015.
[16] T. Tsumura, I. Suzuki, Y. Ikeuchi, H. Matsuo, H. Nakashima, and Y. Nakashima, "Design and evaluation of an auto-memoization processor," in *PDCN*, 2007.
[17] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas, "SoftSig: software-exposed hardware signatures for code analysis and optimization," in *ASPLOS-XIII*, 2008.