$E = MC^2$: Efficient Mobility Centric Caching

(A Position Paper)

Chen Ding
Ben Reber
University of Rochester
Rochester, New York, USA

Dorin Patru
Alexander H. Kneipp
Marcus Figorito
Rochester Institute of Technology
Rochester, New York

ABSTRACT

Mobility-centric design revolves around the concept of a *lease*, which is used by hardware to manage the cache memory and by software to program the hardware. This position paper presents a design of collaborative software and hardware programming of a multicore cache hierarchy. This design is the first to allow any mix of software and hardware policies on the same machine. It promises more powerful cache optimization than what is possible with existing cache designs but also introduces the problem of multi-level cache programming.

1 INTRODUCTION

Today's computers such as CPUs, GPUs and accelerators have complex memory systems that are almost always hierarchical, parallel, dynamically allocated and shared. This complexity is rapidly increasing with new technology, e.g. high-bandwidth memory (HBM), new material, e.g. SSDs, and new architectures, e.g. heterogeneous systems.

Traditional caching is under strain. Memory has become ever larger and more diverse, with different materials, configurations, and interconnects having different trade-offs among capacity, speed, cost and other factors. Compartmentalized solutions with carefully tuned heuristics are increasingly brittle for several reasons. They may not scale well under different types of applications and varying degrees of parallelism. Performance under a dynamic workload can be extremely unpredictable and unfair. There can be pathologically bad performance caused by poor interactions between memory components and their management policies. Last but not least, the memory utilization cannot be fully optimized unless software and hardware can operate in concert.

The complexity of today's memory is too great for conventional automatic solutions to be fully effective and robust. It warrants studies of new theories of memory that aim at formal rigorous performance and correctness models and optimization that are based on mathematics, ensure reproducible results, and have provable guarantees.

The new design focuses on minimizing the data movement. We do not consider prefetching. Although prefetching is critical to performance, it does not reduce the data movement. We limit our consideration to just the data movement. The cost and the performance of the computing infrastructure depend on its memory hierarchy, and the data movement frequently poses a limit to scalability and power efficiency.

2 BACKGROUND

2.1 Cache Programming by Leases

Conventional cache design is reactive — deciding what to replace when the cache is full. Cache allocation is prescriptive. The eviction time of a data block is *prescribed* at its latest access by a lease. The lease is measured by the number of accesses rather than the physical time. A lease of 1000 means that the lease cache keeps the data block until 1000 accesses later. The lease is renewed if the data block is accessed before the end of the lease; otherwise, the block is evicted from the cache.[1]

It helps to draw an analogy with human mobility, in that allocating the space in cache is analogous to renting the rooms in a hotel next to a busy factory. A worker stays in the hotel in case they are needed for some work. If they are, a new reservation is made to extend the hotel stay; otherwise, the worker leaves when their reservation expires. A cache lease is analogous to a hotel reservation.

	LRU cache	Lease cache
Primary policy	LRU, automatic	leases, programmed
Info used	history only	history, prog., or both
Secondary policy	N/A	random eviction

The preceding table shows a comparison with the least-recently-used cache (LRU), a common baseline policy where the least recently used cache block is selected for eviction

[1]With leases, the control is not just imposed on data insertion but also updated on each data use. This differs from the previous concepts of cache leases used in distributed caching [9], Web caching [8] and TLB [1].

at every cache miss. Unlike the LRU cache which is reactive and uses only the history information, the lease cache is programmed and can be so based on program information. If a program requires more cache space than what is available, the lease cache has a secondary policy, which randomly evicts a data block at a cache miss. As an allocation, lease programming is similar to heap management, but the goal is not to minimize the size of a heap, but to obtain as many cache hits as possible.

2.2 Leasing Policies and Properties

Three types of leasing policies have been developed, with increasingly stronger performance guarantees. They are shown in the following table and then described in more detail. MRC refers to *miss ratio curve*.

Policy	Performance	Use
Uniform lease	no worse than LRU, mono. MRC	any prog.
Dual lease	convex MRC	
CARL	minimal misses[2], mono. / convex MRC, sub-partition mono.	prog. amenable to compiler analysis

Uniform Lease. We call a new cache design *performance safe* if it is at least as good as the LRU cache. For the lease cache, this amounts to comparing UL and LRU. We will compare them both in theory and through experimentation. We have shown in theory and simulation that UL and LRU perform the same [5]. The safety implies monotonicity in that LRU is a stack algorithm. It cannot happen that a larger cache incurs more misses than a smaller cache.

Dual Lease. The first extension is a *dual lease*, which assigns a long lease for a (randomly selected) portion of accesses and a short lease for the rest. By choosing the right portion size, the dual lease accomplishes the effect of Talus [2] and SLIDE (for LRU caches) [23].

CARL Leases. Given a program, a compiler assigns a lease for each load and store instruction in the program. The lease is called a *reference lease*, assigned to each load or store instruction at the program binary. When the program executes, each access is given the lease of its load or store instruction. For programs whose data access can be analyzed at compile-time, e.g., scientific kernels, reference leases can be assigned optimally by the CARL (compiler-assigned reference lease) algorithm [7, 17]. CARL takes a target cache size and set of reference RI (reuse interval) distributions and assigns leases to maximize the number of hits per unit of a lease.[2]

[2]CARL uses a dual lease for a reference, while the policy of Dual Lease uses it for the whole program.

CARL has strong properties [7]. It is optimal in that it selects the best possible lease for a program.[3] Across all cache sizes, the miss ratio function is not just monotone but also convex. It also has sub-partition monotonicity in that with finer-grained information, CARL will not lose performance.

The optimality claim goes beyond lease caching. Lease programming is based on statistical prediction of program behavior. This differs from hardware replacement policies such as LRU and OPT which are based on precise prediction. For example, consider a simple trace like $aaaxxx\ldots axaxax$ where a, x are accessed alternately consecutive and interleaved. The hardware cache tries to predict the exact next reuse. A lease is assigned based on the knowledge that about half of the reuse intervals are 1 and the other half 2. The hardware makes the prediction at run time at each access. The lease is assigned statically at compile time and is associated with a program instruction. Using the same statistical information, no algorithm, based on lease or not, can do better than CARL.

3 CACHE CO-PROGRAMMING

Mobility is encoded by leases. As a programming interface, they can be used by both software and hardware. The key idea of the paper is software and hardware collabrative caching. This section explains the concepts, the design for new hardware, and programming techniques.

3.1 Concepts

We call it *cache programming* if the cache control is encoded through program code. To distinguish, we call it *cache management* if the control is purely based on program data. By this definition, the LRU cache policy is cache management, while CARL (i.e., reference leases) is cache programming. The uniform lease and dual lease treat a program as one with a single reference, so they are also cache programming.

We define *collaborative cache programming* as the combined use of cache programming in both software and hardware, or *cache co-programming* in short. The following table shows how mobility centric caching accomplishes cache co-programming by combining leasing policies in software and hardware.

Cache programming	Technique	Collaboration
Hardware	dual lease	run-time, universally applicable
Software	CARL	compile-time, prog./user knowledge

[3]The optimality is proved for variable-size cache. For fixed-size caches, CARL is applied per program phase [16, 17] or per loop nest [19].

Lease-based cache co-programming leverages the convex MRC provided in the dual lease policy in hardware and optimal MRC provided by CARL in software. The combination allows optimization of the shared cache. Their programming, however, must support a multi-level cache hierarchy. The rest of the section discuss these issues.

3.2 Co-programmable Hardware

We are building a hardware prototype shown in Figure 1. It has two RISC-V cores, each connected to private instruction caches (IC) and data caches (DC). The last level cache is the lease cache and shared by both cores. On modern processors, the first-level caches are optimized for latency. Following the conventional design, the private caches are set associative and use pseudo-least recently used (PLRU) replacement [20, §III], rather than leases.

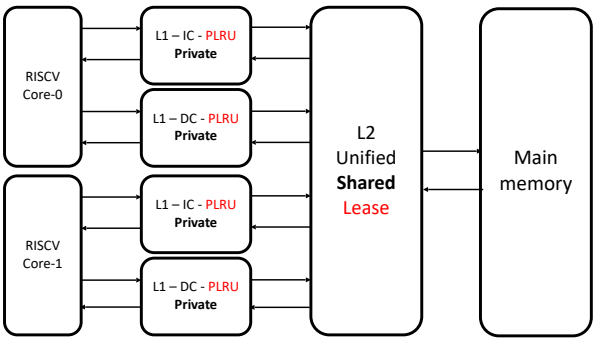

Figure 1: Dual-core and two-level cache hierarchy with a shared lease cache

This is based on a previous FPGA prototype, which has a single RISC-V core and a single-level lease cache [17, 19]. The availability of the programmable logic array offers a unique opportunity to implement a *soft* lease cache and its associated controller. Specifically, at run-time *dynamic reconfiguration* of a part of the programmable logic, a feature present in all high performance SoCs and FPGAs, makes the implementation of a lease cache even more attractive. In effect, the availability of the programmable logic array allows the lease cache to adapt to the needs of the application as prescribed by the compiler through the lease values.

3.3 Optimal Cache Sharing

Shared cache emerged at the paradigm shift of the industry to multicore. On the one hand, cache sharing enables the pooling of cache resources and allows a core to use most of the communal cache when other cores are not using it. On the other hand, it introduces the problem of negative interference and uncertainty. Many techniques have been developed to effectively address these problems. One remaining, however, is a robust method to maximize the utilization of the shared cache for any given workload, i.e., allocating the shared cache to minimize the total number of cache misses.

Lease cache has convex miss ratios. The convexity permits optimal partitioning of cache. It is a well-known result that optimal cache partition is one where the derivative of the miss ratio function of all co-run programs is equal [21, 22]. The following theorem shows the maximization of the cumulative misses $\sum_i f_i(x_i)$ by dividing the total amount of memory X.

THEOREM 3.1. *Let $\{f_i\}_{i=1}^n$ be a set of differentiable decreasing convex functions, and let X be a positive real number. Then the sum $\sum_i f_i(x_i)$ is minimized subject to the constraint $\sum_i x_i \le X$ at a point where $\forall i, j \in \{1..n\}(f_i'(x_i) = f_j'(x_j))$.*

The optimal allocation can be done in time linear to the memory size, by greedily allocating each cache block to the program with the most reduction in the miss ratio.

In practice, it is often desirable to ensure fairness or a baseline quality of service (QoS). Elasticity is an idea for individual programs to "volunteer" a portion of its cache partition if others can make a better use "for the greater good." For monotone miss ratio functions, Ye et al. [25] used dynamic programming for optimization. Convex miss ratios enable the same optimization in linear time.

Cache programming enables operating system (OS) control. The following table compares OS management of physical memory, which is used by almost all today's machines, and the possibility of OS management of the shared cache.

	Virtual memory	Shared cache
Mechanism	page tables	lease tables
Allocation frequency	every page fault	every memory access

Compared to physical memory, the shared cache is smaller, allocated at smaller granularity, and more importantly, managed at the frequency of every cache access, instead of OS page allocation. The cache allocations and evictions are too frequent for any software to manage it directly. Cache programming is necessary to allow fine-grained OS control.

In current systems, OS controls privileged instructions to partition the cache memory among programs and let each program make the best use of it. On Intel systems with CAT, the granularity of partitioning is a cache way. With cache programming, the OS may partition the cache at finer granularity. More importantly, it may efficiently find the optimal cache partitioning.

Unlike physical memory, the cache system is hierarchical. We next consider the programming of multiple cache layers.

3.4 Two-level Cache Programming

In our hardware design (Section 3.2), level-one (L1) caches use PLRU, and they share a single level-two (L2) cache which is programmed by leases. For current multicore processors, the private cache and the shared cache are exclusive on AMD machines and non-inclusive on Intel machines. Here we consider exclusive caches, where the shared cache serves as the victim cache for private caches. Since L1 is a conventional cache, the goal is to program L2 but remain mindful of L1 and minimize L2 misses.

Cache hierarchy poses a significant challenge for collaborative cache – software can make inferences about the *source level* access stream, i.e. the memory accesses issued by a process. Meanwhile, L2 does not see the stream of source level accesses but rather a *target level* access stream comprised of L1 misses and evictions.

A hardware-only approach sees only the target level access stream and may react to it dynamically. However, in order to obtain the benefits of software control, the collaborative approach must account for this discrepancy somehow. There are two primary options. The first is for software to assign leases according to program order, and trust hardware to provide information about the source level access stream to the target cache dynamically. In our design, this would mean updating every L2 lease on every L1 access. The second approach requires software to account for this filtering effect during lease assignment. Thus, leases could be assigned based on predicted L2 reuse statistics. We propose a simple compromise between these two options – software can compute and assign leases based on source level access stream, and trust hardware to dynamically scale them down by the L1 hit ratio when L2 stores a block.

In terms of cache allocation, there are two choices. The first is *L1 oblivious*, where L2 leases are assigned based on L2 cache size, completely ignoring L1. The second is *L1 conscious*, where L2 leases are selected differently based on how large L1 caches are. The first choice is portable but does not fully use the combined space of the two caches.

In both cases, we need two support from hardware. The first requires the hardware to record additional information in each cache block in L1. In particular, each time an L1 block is accessed, we record the reference. When the block is evicted from L1, this information is used to look up the L2 lease in the lease table.

Since the lease starts from the last time the data is accessed, not when it is evicted from L1, the L2 lease needs to be adjusted. Specifically, the time between the last access to the block and the block's eviction from L1 should be deducted from the lease before using it. We call the lease before the deduction the *programmed* lease and the *adjusted* lease after it. For LRU caches, the average eviction time (AET) has been studied and used in modeling both software and hardware caches [6, 11, 12]. We use the hardware to measure AET at run time and use it to adjust L2 leases.

Based on AET, L1 conscious leasing computes the effect of the private cache. There are two effects. First, it identifies data reuses in L1 and does not assign them L2 leases. Second, it calculates the demand to L2 after removing L1 hits and assigns leases for the adjusted demand. Effectively, L1 is now an extension of L2, and L2 leases are assigned for the combined space.

4 PRELIMINARY RESULTS

Figure 2 presents preliminary results on the performance of four CARL-based lease assignment algorithms for benchmarks from the Polybench suite. We use PolyBench/C 4.2.1, which contains 30 numerical kernels [15] and show the results for the small dataset size. We use PolyBench because the benchmark suite is relatively easy to port through the FPGA tool chain to allow testing on a real system.

We show the L2 miss ratios for four different lease assignment algorithms on our machine. Previous work has presented only L1 lease cache performance. CLAM[17] is a direct implementation of the CARL algorithm on our hardware, and PRL, SHEL and C-SHEL each attempt to improve performance via load balancing[17, 19]. For the sake of brevity, the details of the particular differences between each lease assignment algorithm are elided. We see that regardless of which lease assignment technique is used, performance over PLRU improves on average.

5 RELATED WORK

We categorize existing methods in four groups in the following table based on whether and how they combine software and hardware caching.

		Hardware	
		managed	programmed
Software programming	no	LRU, RRIP, Talus, SLIDE	Hawkeye, LEU
	yes	collaborative caching	this paper

Collaborative caching inserts hints, such as evict-me [24] or those on Intel Itanium architecture [3], into a program. It is collaborative and uses cache programming in software but not in hardware. The hardware policy combines most recently used (MRU) replacement policy and LRU, and the combined policy is a stack algorithm [10]. Its MRC is monotone but not convex, so collaborative caching cannot achieve optimal cache sharing described in Section 3.3.

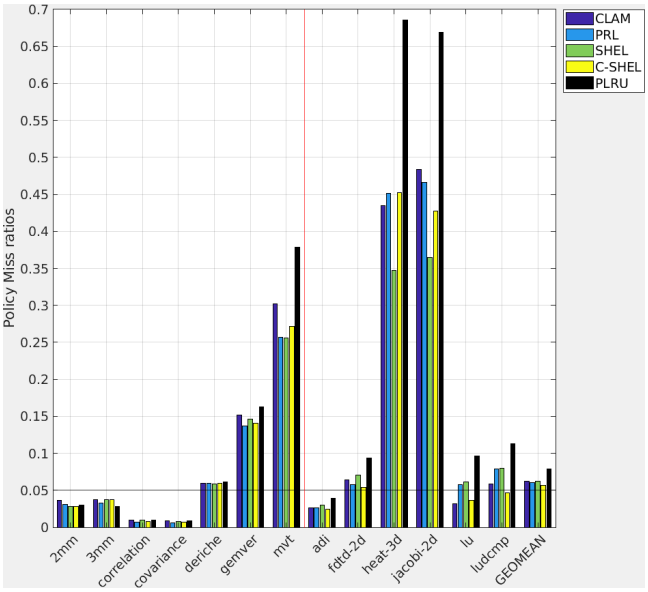

Figure 2: L2 miss ratios for four different lease assignment algorithms - CLAM, PRL, SHEL and C-SHEL, on Polybench programs with small dataset size compared to PLRU. Each algorithm is a modification of the CARL algorithm discussed in section 2.2.

LRU and its numerous improvements such as RRIP [14] and set dueling [18] are purely cache management techniques in hardware, so are Talus and SLIDE mentioned in Section 2. More recent improvements in caching include Hawkeye [13] and Least Expected Use (LEU) replacement [4], which are cache programming in hardware but not in software. Widely used hardware techniques such as branch prediction and prefetching use code-based analysis and control, e.g. the branch prediction table. They would fit in the upper right quadrant with Hawkeye and LEU if they were caching techniques. These techniques are not collaborative between hardware and software. Co-programming allows caching be optimized based on program or programmer knowledge. However, it needs multi-level cache programming discussed in Section 3.4.

6 SUMMARY

In this position paper, we have presented collaborative software and hardware programming of a two-level cache hierarchy for a dual-core RISC-V processor. Dual Lease improves the basic caching for any program, and CARL achieves optimal caching for programs amenable to compiler analysis. The system is the first to allow any mix of these two policies on the same machine, efficiently optimize their cache sharing, and enable cache programming in a multi-level hierarchy.

By focusing on mobility, the new design may minimize the number of cache misses and hence the energy consumption due to data movement. Mobility centric caching is a promising approach to further reduce the energy cost of computing systems.

REFERENCES

[1] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 273–287. https://doi.org/10.1109/PACT.2017.38

[2] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 64–75. https://doi.org/10.1109/HPCA.2015.7056022

[3] Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250.

[4] Sayak Chakraborti, Zhizhou Zhang, Noah Bertram, Chen Ding, and Sandhya Dwarkadas. 2023. Blast from the Past: Least Expected Use (LEU) Cache Replacement with Statistical History. In *Proceedings of the International Symposium on Memory Management*, Stephen M. Blackburn and Erez Petrank (Eds.). 124–136.

[5] Dong Chen, Chen Ding, Fangzhou Liu, Benjamin Reber, Wesley Smith, and Pengcheng Li. 2021. Uniform lease vs. LRU cache: analysis and evaluation. In *ISMM '21: 2021 ACM SIGPLAN International Symposium on Memory Management, Virtual Event, Canada, 22 June 2021*, Zhenlin Wang and Tobias Wrigstad (Eds.). ACM, 15–27.

[6] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality analysis through static parallel sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 557–570. https://doi.org/10.1145/3192366.3192402

[7] Chen Ding, Dong Chen, Fangzhou Liu, Benjamin Reber, and Wesley Smith. 2022. CARL: Compiler Assigned Reference Leasing. *ACM Transactions on Architecture and Code Optimization* 19, 1 (2022), 15:1–15:28.

[8] Brad Fitzpatrick. 2004. Distributed caching with Memcached. *Linux Journal* 2004, 124 (2004), 5.

[9] Cary G. Gray and David R. Cheriton. 1989. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the ACM Symposium on Operating System Principles*. 202–210. https://doi.org/10.1145/74850.74870

[10] Xiaoming Gu and Chen Ding. 2011. On the theory and potential of LRU-MRU collaborative cache management. In *Proceedings of the International Symposium on Memory Management*. 43–54.

[11] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *Proceedings of USENIX Annual Technical Conference*. 351–364. https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu

[12] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. 2018. Fast Miss Ratio Curve Modeling for Storage Cache. *ACM Transactions on Storage* 14, 2 (2018), 12:1–12:34. https://doi.org/10.1145/3185751

[13] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the International Symposium on Computer Architecture*. 78–89. https://doi.org/10.1109/ISCA.2016.17

[14] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 60–71.

[15] Louis-Noël Pouchet. [n. d.]. PolyBench/C 4.0. http://polybench.sourceforge.net.

[16] Ian Prechtl, Chen Ding, and Dorin Patru. 2020. Design and Evaluation of a Fixed-size Programmable Working-set Cache on FPGAs. preprint online at https://dx.doi.org/10.13140/RG.2.2.24423.60320.

[17] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. 2020. CLAM: Compiler Lease of Cache Memory. In *MEMSYS 2020: The International Symposium on Memory Systems, Washington, DC, USA, September, 2020*. ACM, 281–296.

[18] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel S. Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the International Symposium on Computer Architecture*. 381–391.

[19] Benjamin Reber, Matthew Gould, Alexander H. Kneipp, Fangzhou Liu, Ian Prechtl, Chen Ding, Linlin Chen, and Dorin Patru. 2023. Cache Programming for Scientific Loops Using Leases. *ACM Transactions on Architecture and Code Optimization* 20, 3, Article 39 (jul 2023), 25 pages.

[20] Kimming So and Rudolph N. Rechtschaffen. 1988. Cache Operations by MRU Change. *IEEE Trans. Comput.* 37, 6 (1988), 700–709. https://doi.org/10.1109/12.2208

[21] Harold S. Stone, John Turek, and Joel L. Wolf. 1992. Optimal Partitioning of Cache Memory. *IEEE Trans. Comput.* 41, 9 (1992), 1054–1068. https://doi.org/10.1109/12.165388

[22] Dominique Thiébaut, Joel L. Wolf, and Harold S. Stone. 1992. Synthetic Traces for Trace-Driven Simulation of Cache Memories. *IEEE Trans. Comput.* 41, 4 (1992), 388–410.

[23] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *Proceedings of USENIX Annual Technical Conference*. 487–498. https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

[24] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. Charlottesville, Virginia.

[25] Chencheng Ye, Jacob Brock, Chen Ding, and Hai Jin. 2017. Rochester Elastic Cache Utility (RECU): Unequal Cache Sharing is Good Economics. *International Journal of Parallel Programming* 45, 1 (2017), 30–44. https://doi.org/10.1007/s10766-015-0384-3