Coding for Efficient Caching in Multicore Embedded Systems

Tosiron Adegbija and Ravi Tandon
Department of Electrical and Computer Engineering
University of Arizona, USA
Email: {tosiron,tandonr}@email.arizona.edu

***Abstract*—We present an information theoretic approach to caching in multicore embedded systems. In contrast to conventional approaches where caches are treated independently, we present novel *cache placement* and *coded data delivery* algorithms that treat the caches holistically, and provably reduce the communication overhead resulting from main memory accesses. Our approach intelligently places data across the processors' caches such that in the event of cache misses, the main memory opportunistically sends coded data blocks that are simultaneously useful to multiple processors. Using architectural simulations, we demonstrate that the coded caching approach significantly reduces the communication overhead, thus reducing the overall memory access energy and latency, while imposing minimal overheads. In a quad-core embedded system, compared to conventional caching schemes, the coded caching approach reduced the access energy and latency by an average of 36% and 16%, respectively.**

***Index Terms*—Cache optimization, coded caching, energy savings, low-power embedded systems.**

I. Introduction and Motivation

Caches are commonly used in embedded systems to bridge the processor-memory performance gap by exploiting executing applications' spatial and temporal locality. Caches also account for a significant portion of an embedded system's power/energy consumption, which has necessitated much research focus on cache optimization techniques [17]. Due to high memory latency and memory bandwidth limitations, cache optimization is critical for improving the end-to-end performance of embedded systems. Cache optimization, however, is challenging, especially in embedded systems, since these systems typically have stringent design constraints with respect to size, battery capacity, real-time deadlines, cost, etc. Despite these design constraints, embedded systems are expected to execute algorithmically complex and memory-intensive applications due to consumer demands for complex applications and functionalities. To satisfy this growing demand, embedded systems are being equipped with multicore processors that feature complex memory hierarchies, as opposed to single-core processors. These technological advances make cache optimization even more challenging.

To meet the often conflicting goals of achieving best possible cache performance and energy efficiency in modern embedded microprocessors, several researchers have proposed circuit-level [18], [22] and architectural [4], [10] cache optimization techniques. Due to the increasing compute and memory complexity of modern embedded systems applications, novel radical and innovative techniques are required to achieve optimal caching that maximizes performance and energy efficiency, without introducing the attendant overheads of traditional optimization techniques, such as area, design time, and computational complexity.

In this work, we approach cache optimization by focusing on minimizing the required cache to main memory communication during application executions, as this communication is a major source of overhead in embedded systems [19]. In the event of a cache miss, data must be transferred from a lower memory level (e.g., level two (L2) cache or main memory) to the first level cache (L1) for subsequent use by the processor. Modern embedded systems microprocessors' memory subsystem consume significant amounts of energy and time by continuously transferring large amounts of data from main memory to processor caches [19]. In several cases, the data transferred to the cache is only briefly utilized by the processor and subsequently replaced by other data that are not reused often. Sometimes, the data stored in the cache feature high levels of redundancy. These caching behaviors exacerbate cache optimization challenges.

Drawing from information theory concepts, we explore the fundamental limits of optimal caching in order to improve the efficiency of caching in multicore embedded systems. Efficient cache utilization is especially critical for multicore systems that feature random scheduling [23] and highly persistent applications—applications that reoccur several times throughout a system's lifetime (e.g., smartphone apps). Our goal is to reduce the overheads resulting from main memory accesses in these systems, while introducing minimal optimization overheads. Recent work in information theory [16] has shown that conventional approaches of treating distributed caches independently can be far from optimal, and coding across caches can be leveraged to reach the information theoretic limits. Therefore, we leverage this information theoretic approach to analyze the optimal utilization of multicore caches in state-of-the-art and emerging embedded systems microprocessors.

In this work, we investigate our main idea of coding data across caches and jointly operating the caches in order to significantly reduce the communication overheads and provably improve optimization goals, such as memory access energy and latency. To this end, we propose a runtime *Systematic Cache Placement Algorithm (SCPA)* that intelligently deter-

mines the cache fill data, and a *Systematic Cache Delivery Algorithm (SCDA)* that delivers the appropriate data block(s) to the processor in the event of a data request. Our approach places data across the processors' caches such that in the event of cache misses, the main memory opportunistically sends *coded data blocks* that are simultaneously useful to multiple processors, thus significantly reducing the overhead from main memory accesses.

We model state-of-the-art embedded systems' memory hierarchy using GEM5 [6] architectural simulations, and benchmarks from the SPEC2006 benchmark suite [3] to represent the increasing compute and memory- complexity of emerging embedded systems applications. Using our experiments, we show that coded caching can reduce the average energy consumption and improve performance by 36% and 16%, respectively, as compared to conventional caching.

II. Background and Related Work

The work presented herein is complementary to previous cache optimization techniques. Thus, in this section, we present a brief background and overview of related work on cache optimization in computer architecture, and recent results on information theoretic aspects of caching.

A. Cache Optimization

The memory hierarchy can consume more than 50% of the total system power, especially in embedded systems. As a result, much research has focused on optimization techniques to improve the efficiency of the memory hierarchy, especially the cache [17]. Significant emphasis has been placed on dynamic optimizations, as an alternative to static cache optimizations [11]. Unlike static optimizations, dynamic optimizations can accommodate changing application behaviors during runtime and appreciably improve performance and energy savings potential.

Several cache optimizations target low cache access latency through data migration, replication, or banked shared level two (L2) caches and techniques that place data in cache banks close to the referencing core [7]. However, these methods are usually not fully dynamic and cannot react to changing runtime application dynamics [5]. Cache partitioning [9] is another well researched optimization that dedicates portions of a shared last level cache (typically the L2 cache) to executing tasks in order to increase performance.

Our approach for cache optimization complements previous techniques; we exploit information theory concepts to optimize cache data placement in order to minimize main memory accesses. The proposed approach incurs minimal overheads in terms of hardware, computational complexity/execution time, and energy, and is therefore practical for resource constrained embedded systems. Furthermore, the proposed approach requires minimal design time effort and reduces overall memory access energy and latency, as a direct consequence of reducing the main memory accesses.

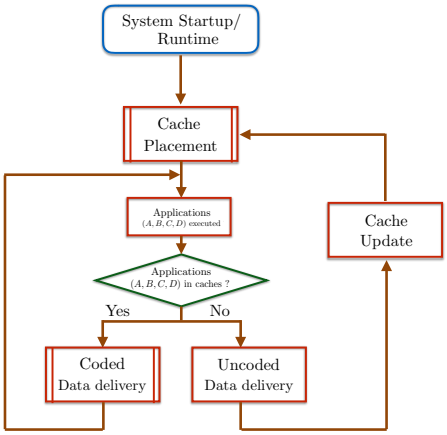

Fig. 1: A high-level overview of our coded caching approach.

B. Information Theory of Distributed Caching

Much recent research has focused on understanding the fundamental information theoretic limits of distributed caching systems. A novel information theoretic model for distributed caching was recently introduced in [16]; the authors showed that coded content delivery could lead to significant reduction in communication overhead in contrast to conventional approaches that treat caches independently. This work has been extended to a variety of problems and scenarios, such as wireless device to device (D2D) networks [13], minimizing content delivery latency over wireless networks [21], etc.

In addition to these wide variety of applications, significant recent progress has also been made on understanding the fundamental tradeoffs between the memories of distributed caches and the communication overhead in the event of cache miss. In particular, the optimal tradeoff between cache memories and the communication was characterized to within a constant factor of 12 in [16]. Recent work [20] improved the approximation ratio to 8 by developing new information theoretic converse techniques.

To the best of our knowledge, there is no prior work that explores the benefits of coded caching in multi-core embedded systems. The information theoretic and coding aspects of our proposed approach and algorithms are inspired by recent advances in our understanding of distributed caching.

III. Coding for Efficient Caching

Figure 1 illustrates the high-level flow of our coded caching approach. At system startup—or during runtime, when new applications are executed—our cache placement algorithm preemptively fills all the caches with data from the most persistent executing applications. Note that our approach also applies to application threads, however, we describe our work using applications for brevity. When applications are executed across the different cores, if the caches contain some or all of the applications' data, the coded data delivery algorithm systematically fetches data from the main memory to be useful to all the cores, while complementing the data already in the caches.

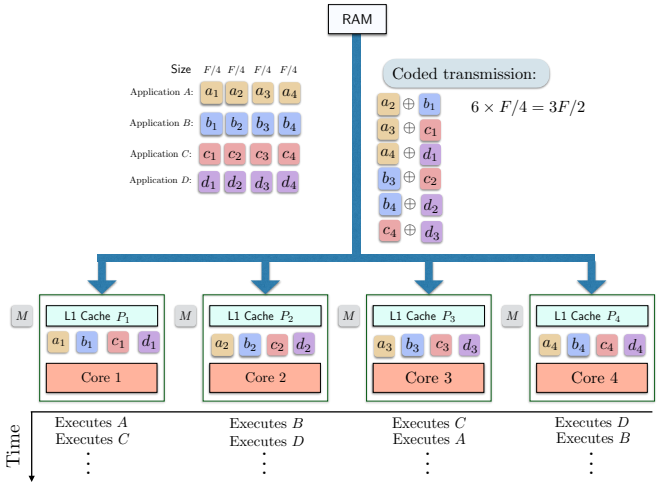

Fig. 2: Coded caching scheme for a quad-core microprocessor.

If the applications' data are not in the caches, a cache update is performed, wherein the caches are filled using the cache placement algorithm. The proposed approach will be most effective for a system with persistent applications, such as smartphones, where multiple applications may be executed several times throughout the system's lifetime, during different time intervals, and on different cores. In this section, we describe our approach for optimally utilizing multicore systems' caches using a specific sample system, and generalize our approach to a more arbitrary system.

A. Description of Coded Caching for Multicore Embedded Systems

Figure 2 illustrates the coded caching approach in a quad-core microprocessor. For this description, we make the following simplifying assumptions: each core has a private level one (L1) cache, P_1, P_2, P_3, and P_4, respectively; there are four executing applications, A, B, C, D; all the applications have equal persistence on all cores (i.e., they are randomly scheduled and have equal probability of being executed); and the data blocks have equal persistence. We generalize our technique to an arbitrary system in Section III-B. Our proposed caching technique comprises of two key stages:

1. Cache placement stage: At system startup or at execution intervals, the cores' L1 caches are filled with coded data based on our proposed caching techniques, while satisfying the memory constraints. This stage is agnostic of the applications that will be requested by the processors.

In the cache placement stage, each application's data is broken into four parts, and preemptively stored in each cache. For example, application A is broken into four 8 KB parts, a_1, a_2, a_3, and a_4, which are stored in caches P_1, P_2, P_3, and P_4, respectively. Similarly, B, C, and D are broken into four parts and stored in each cache. Thus, cache P_1 is filled with a_1, b_1, c_1, and d_1, P_2 is filled with a_2, b_2, c_2, and d_2, and so on.

2. Data delivery stage: During runtime, when each core requests data blocks, a portion of the data blocks will be guaranteed cache hits in the core's private L1 cache; the remaining data, if any, is then fetched from the main memory.

In the data delivery stage, our approach intelligently fetches coded data from the main memory in order to be simultaneously useful for multiple cores. Assuming the cores with caches P_1, P_2, P_3, and P_4 are executing applications (or threads) A, B, C, and D, respectively, six coded (XOR'd) data 'chunks' are sent from main memory on a common data bus [15] as follows: $a_2 \oplus b_1$ for P_1 and P_2; $a_3 \oplus c_1$ for P_1 and P_3; $a_4 \oplus d_1$ for P_1 and P_4; $b_3 \oplus c_2$ for P_2 and P_3; $b_4 \oplus d_2$ for P_2 and P_4; and $c_4 \oplus d_3$ for P_3 and P_4. If during the next time period, the four cores execute the applications in a different order (C, D, A, and B, for example), the same data chunks are fetched from main memory to satisfy all four cores.

B. Generalizing Coded Caching to an Arbitrary Multicore System

The description in Section III-A is a first step towards using coded caching for optimal cache utilization in multicore embedded systems. However, some of the assumptions in the description introduce some important caveats that present opportunities for future work. To address some of these caveats, we present a *systematic cache placement algorithm (SCPA)* and *systematic coded delivery algorithm* that create coding opportunities across all cores in order to significantly reduce the communication overhead. These algorithms [16] generalize coded caching to an arbitrary number of cores, K, any number of applications, N, and any L1 cache size M. We will generalize the proposed techniques to multilevel caches in future work.

Systematic Cache Placement Algorithm (SCPA). Consider cache size $M \in \{0, \frac{N}{K}, \frac{2N}{K}, \frac{3N}{K}, \ldots, N\}$, and set $t = KM/N$, such that t is an integer between $\{0, 1, \ldots, K\}$. The cache placement algorithm works as follows: each application's data, D_n is split into $\binom{K}{t}$ non-overlapping sub-blocks, where the size of each sub-block is $F/\binom{K}{t}$. Formally, D_n is split into sub-blocks as follows:

$$D_n = \{D_{n,\mathcal{S}}, \quad \mathcal{S} \in \{1, \ldots, K\}, \quad |\mathcal{S}| = t\}$$

The cache placement scheme works as follows: cache of core k stores the sub-block $D_{n,\mathcal{S}}$ if $k \in \mathcal{S}$. Hence, each cache stores $N\binom{K-1}{t-1}$ sub-blocks. The total amount of memory used at each cache is thus, $N\binom{K-1}{t-1} \times \frac{F}{\binom{K}{t}} = FNt/K = FM$, thereby satisfying the cache memory constraint. We can observe that the cache placement described in Figure 2 follows this algorithm, for $K = 4$ processors, $N = 4$ applications, and $M = N/K = 1$.

Systematic Coded Delivery Algorithm (SCDA). Now, consider that each core executes a particular application at a given time. We denote the requested applications at any given time as $r_1, r_2, \ldots, r_K$, where $r_k \in \{1, \ldots, N\}$. In other words, core k references data D_{r_k} corresponding to application r_k. The coded delivery of data blocks works as follows: consider a subset $\mathcal{S}$ of size $|\mathcal{S}| = t+1$ cores. Every t cores in this subset have sub-blocks stored in the cores' local

TABLE I: Cache and main memory configuration parameters.

Configuration	Parameters
L1 data cache	32 KB, 4-way, 64 byte blocks, 0.67 ns access time, 1 R port, 1 W port, 4 banks, 32nm technology
DRAM main memory	2 GB, 4 banks, 1 R/W ports, 128 bits, 9.12 ns access time, 32nm technology

caches, and these sub-blocks are needed at the other cores in the set $\mathcal{S}$. Given a core s, the sub-block $D_{r_s,\mathcal{S}\backslash\{s\}}$ is needed at core s to process the application r_s. Hence, for each subset $\mathcal{S} \subset \{1,\ldots,K\}$ of cardinality $\mathcal{S} = t+1$, the main memory can transmit $\oplus_{s\in\mathcal{S}} D_{r_s,\mathcal{S}\backslash\{s\}}$ over the common bus to all the cores, where $\oplus$ denotes bitwise XOR operation. Each of these XOR'd sums contribute to a communication of $F/\binom{K}{t}$ bits, and the total number of such subsets is $\binom{K}{t+1}$. Hence the total communication by the main memory over the common data bus is:

$$FR^{\text{coded}} = \binom{K}{t+1}\frac{F}{\binom{K}{t}} = F\frac{K-t}{t+1} = F\left(\frac{K(1-M/N)}{1+KM/N}\right)$$

In summary, by exploiting coding opportunities, the total amount of communication [16] is:

$$R^{\text{coded}} = \underbrace{K(1-M/N)}_{\text{Local caching gain}} \times \underbrace{\frac{1}{1+KM/N}}_{\text{Global Caching gain}}$$

The conventional scheme (treating each core independently) leads to a total communication of $R^{\text{conventional}} = K(1-M/N)$. We thus observe that the conventional schemes can only extract a local caching gain of $1-M/N$, since each core/cache is treated independently. On the other hand, the coded scheme extracts not only a local caching gain, but a *global caching gain* of $\frac{1}{1+KM/N}$ by virtually coupling the caches together and exploiting this strategy to create coding opportunities.

IV. Experimental Results

A. Experimental Setup

To quantify the effectiveness of coded caching approach, we evaluated the energy and latency of the executing applications' cache and main memory accesses; every other metric remained constant. To represent real-world applications, we used fifteen benchmarks from the SPEC CPU2006 benchmark suite, cross-compiled for the ARM instruction set architecture, and executed using the reference input sets. The benchmarks were selected to represent a variety of application characteristics (memory/compute intensity and integer/floating point applications). We used SPEC benchmarks, as opposed to traditional embedded systems benchmarks (e.g., EEMBC [2]) because SPEC benchmarks exhibit greater execution memory and compute complexity, and more accurately represent the increasing complexities of emerging embedded systems applications.

To reduce simulation time, while maintaining the overall application behaviors, we profiled the SPEC benchmarks using a combination of Simpoint [12], and techniques described in previous work [4] to extract the benchmarks' execution phases and the phases' persistence. A phase is a length of execution during which an application executes stable execution characteristics (e.g., cache miss rates, branch mispredicts, instructions per cycle, etc.), while the persistence is the rate of that phase's recurrence throughout the application's execution. Thus, the higher a phase's persistence, the more accurately it represents the application's full execution. We used 100000000 instructions of each benchmark's most persistent phases for our simulations.

TABLE II: Workload groups used in our experiments. Each workload comprises of four SPEC2006 benchmarks running on each core.

Workload	Benchmarks
workload_1	calculix/h264ref/bzip/bwaves
workload_2	omnetpp/milc/gromacs/xalancbmk
workload_3	libquantum/hmmer/mcf/namd
workload_4	astar/gobmk/gobmk/soplex
workload_5	soplex/bwaves/hmmer/omnetpp
workload_6	astar/mcf/calculix/xalancbmk
workload_7	gobmk/h264ref/milc/gromacs
workload_8	bzip2/libquantum/namd/bwaves
workload_9	bwaves/omnetpp/h264ref/xalancbmk
workload_10	namd/hmmer/libquantum/gobmk

To simulate our approach, we used GEM5 [6] to model a quad-core embedded system microprocessor with cache configurations similar to the ARM Cortex A15 microprocessor [1] featuring 32 KB, 4-way set associative private L1 caches with 64 byte line sizes. We used CACTI to determine the access energy and latencies for the L1 caches and main memory as shown in Table I. For this paper, we focused on the L1 data cache. We have begun to evaluate the scalability to more complex systems with more than four cores and with multi-level caches, but limit the results presented herein to a quad-core single-level cache system.

In order to reduce the sensitivity of the results to a particular set of simulated workloads, we created ten multiprogrammed workloads by randomly selecting four from the fifteen SPEC2006 benchmarks; each workload executes one application to completion on each core. This kind of execution is analogous to a multithreaded application with no inter-core data dependencies [8]. Table II lists the workload groups and benchmarks used in our experiments. To account for the potential difference in application data sizes, we bounded the XOR'd data by the smallest data sizes in the workload groups.

B. Main memory accesses

The key goal of the proposed approach is to reduce the overall access energy and latency by reducing the number of main memory accesses, which constitute a significant portion of access overheads. The approach targets a system with no application-core affinity, i.e., equal application persistence and equal probability of execution on a given core; these kinds of systems are more prone to overheads from memory accesses. Figure 3 depicts the percentage reduction in main memory accesses achieved by coded caching as compared to uncoded caching. On average, over all the workloads, coded caching reduced the number of memory accesses by 31%. The main memory accesses reduced by 27% to 34%, illustrating

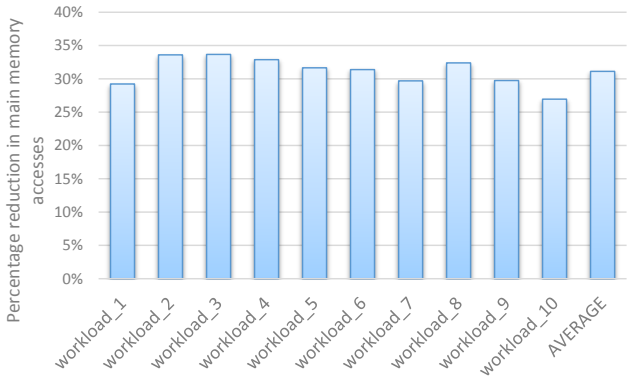

Fig. 3: Percentage reduction in memory accesses achieved by coded caching as compared to uncoded caching.

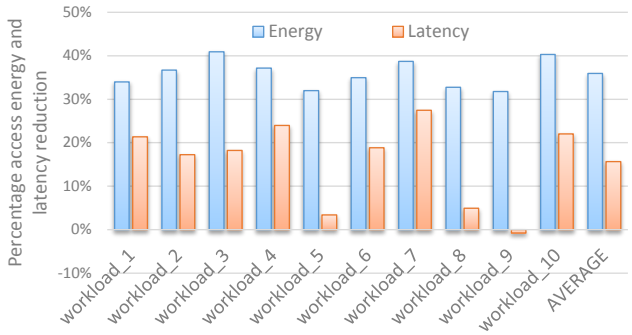

Fig. 4: Percentage access energy and latency reduction of coded caching compared to uncoded caching.

the consistency of the coded approach in a system with no application-core affinity.

We also investigated a system with high application-core affinity—applications were executed multiple times on the same core before executing on a different core. In this system, coded caching only reduced the average number of main memory accesses by 1%; the maximum reduction was 6%, while the number of main memory accesses *increased* for one workload (*workload_10*) by 5% (graphs omitted for brevity). We plan to explore, in future work, the tradeoff point with respect to application-core affinity that allows benefits from coded caching.

C. Overall Access Energy and Latency

In this subsection, we compare the overall access (cache + main memory) energy and latency achieved by the coded caching technique to that of conventional uncoded caching. Note that the coded approach can be used in synergy with other cache optimizations; however, we ignore the impact of other cache optimizations, as they are orthogonal to our approach. We assume a system with random scheduling of applications across cores and no application-core affinity, similar to the illustration in Figure 2. The access energy and latency evaluations include cache fill, cache accesses, and main memory accesses, and consider a dynamic execution scenario, where the executing applications are not known a priori. We limited the access latency of each workload by the application with the longest latency.

In general, the proposed approach reduced the overall access energy and latency as a direct result of reducing the number of main memory accesses (Section IV-B). Figure 4 depicts the percentage access energy and latency reduction achieved by the coded caching approach compared to the traditional uncoded caching approach for different workloads (Table II). On average over all the workloads, coded caching reduced the access energy and latency by 36% and 16%, respectively. Even though coded caching reduced the latency by up to 27% for *workload_7*, there was a wide range of behaviors across the different workloads. Thus, we analyzed the individual applications' memory characteristics (memory references, working set size, data reuse, etc.), and observed that the efficiency of coded caching is strongly tied to the memory characteristics of the applications being executed.

For example, coded caching achieved a much lower latency reduction for *workload_5* and *workload_8*, and *increased* the latency for *workload_9*. The common factor among these workloads is that the workloads include *bwaves*. We initially attributed this behavior to *bwaves*' high memory footprint (as a function of the working set size) compared to the other applications. However, we observed that the benefits of coded caching is more impacted by applications' data reuse than memory footprint. *Bwaves* exhibits lower data reuse than most of the other applications, resulting in a higher potential for conflict cache misses and more main memory accesses. Thus, *bwaves'* low data reuse resulted in lower coded caching benefits for its workload group.

We observed a similar behavior with workloads involving *omnetpp*. Even though *omnetpp*'s working set size was similar to other applications in the workload groups, it exhibited low data reuse and high instruction dependency (with very low IPC), resulting in higher communication overhead. Thus, *workload_9*'s overall access latency increased because it contains both *bwaves* and *omnetpp*. *Workload_9*'s access energy reduction was also the lowest among all the workloads at 32%.

D. Key Observations

Our experimental results demonstrate three key observations. First, coded caching is most effective in systems with high application persistence and random execution, i.e., systems with no determinism to the scheduling of applications or application threads to cores. Second, coded caching is significantly impacted by the application execution characteristics, especially data reuse. Applications with high data reuse benefit more from coded caching; however, some benefit can be derived from coded caching even with applications featuring low data reuse by co-scheduling them with other applications that have high data reuse. Finally, our results show that coded caching has much potential, which warrants further studies on the intricacies of coded caching, especially in synergy with other cache optimization techniques (e.g., configurable caches).

E. Implementation Overhead

The implementation overheads comprise of the computational complexity of the proposed algorithms, and the hardware overhead of implementing the XOR operations. The hardware overhead comprises of a low-overhead coprocessor, with bitwise XOR gates, that codes the data blocks before transmission from the main memory. Since XOR operations are very cheap operations, we propose that the system cores can decode the data prior to execution without imposing significant computational overhead.

The proposed algorithms' computational complexity is proportional to the number of sub-blocks created. In particular, for a K core processor (Section III-B), we create $\binom{K}{t}$ sub-blocks, where t is a parameter which depends on the amount of memory per cache. Since $K^t/t^t \leq \binom{K}{t} \leq K^t/t!$, the worst case complexity of the coded caching process is exponential in K, the number of cores. While this may be reasonable for small values of K, i.e., $K = 2, 4$, an interesting future research direction is to design coded caching mechanisms with linear complexity.

To estimate the coprocessor's overhead, we assumed a system similar to the ARM Cortex A15 [15] with a 128-bit AXI data bus. For maximum throughput, the coprocessor can perform bitwise XOR operations for 128-bit inputs. Using standard XOR cells from a 180nm fabrication process [14], the coprocessor's propagation delay, power, and area are $91.23ps$, $0.019mW$, and $5.85\text{x}10^3 mm^2$, respectively. Relative to most state of the art microprocessors (e.g., ARM Cortex A15), these overheads are negligible. We plan to validate these estimates in future work.

V. Conclusions and Future Work

In this paper, we explored the potentials of coded caching to minimize the communication overhead between L1 caches and the main memory. Drawing from information theoretical concepts, we explore the use of cache data placement and coded data delivery algorithms, such that in the event of cache misses, the main memory opportunistically sends coded data blocks that are simultaneously useful to multiple processors. Our experiments show that coded caching can reduce the energy consumption and improve the performance by up to 36% and 16%, respectively, as compared to uncoded caching.

We have begun to extend the approach presented herein to more complex systems comprising of more than four cores and multilevel caches, and we intend to present this extended approach in future work. In addition, we intend to validate the hardware overheads estimated herein, and implement our techniques in other embedded systems microprocessor data storage and transfer components that impact system performance and energy consumption. For example, instruction window resources, such as the reorder buffer (ROB), instruction queue (IQ), and load-store queue (LSQ), can benefit from the proposed techniques, since they serve as a form of caching for application instructions and data. Finally, we are currently working on generalizing this approach for a runtime scenario wherein the embedded systems' executing applications are not known a priori and change over time.

References

[1] Arm. http://www.arm.com. Accessed: January 2016.

[2] The embedded microprocessor benchmark consortium. http://www.eembc.org/. Accessed: January 2016.

[3] Spec cpu2006. http://www.spec.org/cpu2006. Accessed: January 2016.

[4] T. Adegbija and A. Gordon-Ross. Phase-based cache locking for embedded systems. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 115–120. ACM, 2015.

[5] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 319–330. IEEE, 2004.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *Computer Architecture News*, 40(2):1, 2012.

[7] J. A. Brown, R. Kumar, and D. Tullsen. Proximity-aware directory-based coherence for multi-core processor architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 126–134. ACM, 2007.

[8] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. Barrierpoint: Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 2–12. IEEE, 2014.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE, 2005.

[10] H. Hajimiri and P. Mishra. Intra-task dynamic cache reconfiguration. In *VLSI Design (VLSID), 2012 25th International Conference on*, pages 430–435. IEEE, 2012.

[11] H. Hajimiri, P. Mishra, and S. Bhunia. Dynamic cache tuning for efficient memory based computing in multicore architectures. In *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*, pages 49–54. IEEE, 2013.

[12] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[13] M. Ji, G. Caire, and A. F. Molisch. Wireless device-to-device caching networks: Basic principles and system performance. *arXiv : 1305.5216*, May 2013.

[14] K. Juretus and I. Savidis. Reduced overhead gate level logic encryption. In *Proceedings of the 26th edition on Great Lakes Symposium on VLSI*, pages 15–20. ACM, 2016.

[15] T. Lanier. Exploring the design of the cortex-a15 processor. *URL: http://www. arm. com/files/pdf/atexploring the design of the cortex-a15. pdf (visited on 12/11/2013)*, 2011.

[16] M. A. Maddah-Ali and U. Niesen. Fundamental limits of caching. *IEEE Transactions on Information Theory*, 60(5):2856–2867, May 2014.

[17] S. Mittal. A survey of architectural techniques for improving cache power efficiency. *Sustainable Computing: Informatics and Systems*, 4:33–43, 2014.

[18] S. Motaman, A. Iyengar, and S. Ghosh. Synergistic circuit and system design for energy-efficient and robust domain wall caches. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 195–200. ACM, 2014.

[19] O. Mutlu and L. Subramanian. Research problems and opportunities in memory systems. *Supercomputing Frontiers and Innovations*, 1(3), 2014.

[20] A. Sengupta, R. Tandon, and T. Clancy. Improved approximation of storage-rate tradeoff for caching via new outer bounds. In *IEEE International Symposium on Information Theory*, pages 1691–1695, June 2015.

[21] A. Sengupta, R. Tandon, and O. Simeone. Cache aided wireless networks: Tradeoffs between storage and latency. In *To appear 50th Annual Conference on Information Sciences and Systems (CISS)*, March 2016.

[22] L. Yavits, A. Morad, and R. Ginosar. Cache hierarchy optimization. *Computer Architecture Letters*, 13(2):69–72, 2014.

[23] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, volume 45, pages 129–142. ACM, 2010.