Programming Practices for Cache Memory Optimization with Relevance to Embedded Systems: A Scoping Review

Ramiro V. dos Santos Júnior[2], Francisco Rafael B. de Sousa[1], José Inácio M. Ferreira[1], Raul B. Paradeda[1]

[1]Department of Computer Science
State University of Rio Grande do Norte (UERN) – Natal – RN – Brazil

[2]Federal Rural University of Semi-Arid Region (UFERSA) – Mossoró – RN – Brazil

ramiro.junior@ufersa.edu.br

{franciscobarbosa, inacioferreira}@alu.uern.br

raulparadeda@uern.br

Abstract. *Cache memory efficiency is a cornerstone of performance in computing systems, particularly in embedded applications where hardware resources are constrained. While research on cache replacement and reconfiguration algorithms is well established, software-level programming practices designed to optimize memory hierarchy utilization remain underexplored. This paper presents a scoping review conducted in accordance with PRISMA-ScR guidelines to map evidence on programming techniques that directly enhance cache locality and performance in embedded systems. Empirical studies published between 2015 and 2025 were retrieved from IEEE Xplore, ACM Digital Library, ScienceDirect, and the CAPES Portal. From an initial pool of 2,609 studies, 11 were selected after applying eligibility criteria based on open-access availability, relevance to software-level cache practices, and the presence of empirical results. The selected studies were organized into four thematic categories: data layout and reordering, loop transformations, memory alignment and range propagation, and cache-aware scheduling. Across heterogeneous experimental settings, reported gains demonstrate significant improvements in speedup and energy reduction, depending on workload, platform, and measurement method. Finally, this review identifies existing research gaps and outlines directions for future investigation, including integrating identified techniques into compiler infrastructure and evaluating them on embedded Artificial Intelligence workloads.*

1. Introduction

Cache memory plays a pivotal role in bridging the performance gap between processors and main memory by leveraging locality principles to mitigate high DRAM access latency [Hennessy and Patterson 2017]. In embedded systems, this role is even more critical, as these devices are subject to stringent power, area, and cost constraints, which often preclude the use of large caches or sophisticated hardware-level techniques [Wolf 2016].

A significant portion of the literature focuses on the development of cache replacement or reconfiguration algorithms. Notable examples include LRU (Least Recently Used), widely adopted in commercial systems; ARC (Adaptive Replacement

Cache), which balances recency and frequency of access [Megiddo and Modha 2003]; and various adaptive policies that exploit the dynamic characteristics of applications [Jaleel et al. 2010].

However, programming practices that optimize cache utilization represent an equally essential factor, as they directly influence the spatial and temporal locality of memory accesses. The organization of data structures, the structuring of loops, and the promotion of sequential access patterns can significantly enhance cache efficiency without requiring hardware modifications [McFarling 1995].

In this context, this scoping review aims to map and synthesize existing knowledge on programming practices for cache memory optimization, with an emphasis on embedded systems. Unlike reviews centered on hardware algorithms, this work seeks to understand how software-level implementation choices impact the performance of embedded applications.

This review adheres to the PRISMA-ScR (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) guidelines, a standardized framework for conducting and presenting systematic and scoping reviews. The use of PRISMA ensures transparency and methodological rigor in formulating the research question, study selection, and results synthesis [Page et al. 2021].

The relevance of these optimizations extends beyond isolated computational performance; they are a determining factor for the viability of ubiquitous and pervasive computing. In scenarios such as health monitoring via wearables, for instance, the 30% reduction in energy consumption provided by techniques like memory alignment and range propagation directly impacts the operational autonomy of battery-dependent devices [Mu et al. 2024]. Additionally, in smart city infrastructures where edge computing demands low latency for real-time processing, speedups of up to $2.1\times$ ensure that sensor data analysis occurs efficiently, mitigating communication bottlenecks and enhancing system responsiveness for the user [Inoue and Taura 2015].

Our main contribution is an evidence map of software-level cache-locality techniques, linking each category to the metrics and experimental setups used to assess performance and energy.

2. Theoretical Framework

This theoretical framework presents the fundamental concepts necessary to understand the scope of this review, addressing programming practices for cache memory optimization in embedded systems. To this end, we first discuss the general principles of memory hierarchy and cache organization, followed by the principle of locality, structural aspects of embedded systems, and software optimization techniques. This section is based on three cornerstone works in computer architecture and embedded systems: Hennessy and Patterson [Hennessy and Patterson 2017], Wolf [Wolf 2016], and Stallings [Stallings 2018].

2.1. Cache Memory and Memory Hierarchy

Cache memory is a vital component of the memory hierarchy, designed to bridge the performance gap between the Central Processing Unit (CPU) and main memory. The hierarchy is structured such that proximity to the processor correlates with lower latency

and smaller capacity, while increased distance results in higher latency and larger capacity. Thus, the cache acts as a high-speed intermediary, storing frequently accessed data and mitigating the impact of the "memory wall" [Hennessy and Patterson 2017].

Modern architectures feature multi-level caches (L1, L2, and L3), each with specific size and speed characteristics. System efficiency depends heavily on how well access patterns align with the cache organization [Hennessy and Patterson 2017].

2.2. Internal Cache Organization

Cache behavior is determined by its structural organization. Three primary schemes are employed: direct-mapped, set-associative, and fully associative. Each offers different trade-offs regarding performance, complexity, and cost. Furthermore, replacement policies such as LRU, FIFO, and pseudo-random algorithms are used to determine which blocks should be evicted during storage conflicts [Stallings 2018].

Techniques such as write-back, write-through, prefetching strategies, and bypass mechanisms also influence cache behavior. In embedded systems, these decisions are critical due to stringent power and silicon area constraints [Wolf 2016].

2.3. Principle of Locality

Cache performance relies heavily on the principle of locality, which manifests in two forms:

i. **Temporal locality**: recently accessed data tends to be reused within a short period.
ii. **Spatial locality**: memory accesses usually occur near each other in terms of memory addresses.

These principles are fundamental to designing memory hierarchies and developing optimization techniques aimed at reducing cache miss rates. Modern architectures are designed to exploit the predictability of these patterns [Hennessy and Patterson 2017, Stallings 2018].

2.4. Embedded Systems and Cache

Embedded systems are characterized by rigid constraints on power, performance, real-time response, and physical area. These limitations directly influence the size and policies of the cache memory in such devices. Many embedded platforms utilize smaller or simpler caches to reduce energy consumption and complexity [Wolf 2016]. In these environments, efficient cache utilization can be decisive in achieving performance goals and energy savings. Due to the limited size of these caches, "cache-aware" programming practices are essential to maximize performance without requiring hardware modifications [Wolf 2016]. The interaction between software and hardware is particularly relevant in embedded systems: application performance depends substantially on how effectively it leverages the memory hierarchy [Wolf 2016].

2.5. Software Optimizations for Cache Efficiency

Various software optimization techniques can improve cache behavior:

i. Data structure reorganization and alignment;
ii. Contiguous access patterns and reduction of memory jumps;

iii. Loop transformations (tiling, unrolling, fusion);
iv. Strategic use of prefetching;
v. Reduction of unnecessary main memory accesses.

These techniques enhance both spatial and temporal locality, thereby reducing cache misses and improving overall performance. Classic computer architecture literature provides the theoretical and empirical foundations justifying these practices [Hennessy and Patterson 2017, Stallings 2018].

Understanding these foundational concepts of cache memory, its organization, the principle of locality, and the specific constraints of embedded systems is crucial for appreciating the significance of software-level optimizations. This theoretical background provides the necessary lens through which the programming practices identified and mapped in this review are analyzed, highlighting their mechanisms for improving cache efficiency and their relevance in resource-constrained environments.

3. Related Reviews and Research Context

Existing literature on memory hierarchy optimization in embedded systems has predominantly focused on hardware-level mechanisms, architectural management, or alternative memory structures. For instance, the comprehensive survey by [Gracioli et al. 2015] explores cache management mechanisms for real-time embedded systems, analyzing strategies such as partitioning, locking, and controlled replacement. Other reviews in the memory domain, such as the study by [Tabbassum et al. 2019], investigate software management but restrict their scope to Scratchpad Memories (SPM), which require explicit data movement rather than exploiting the transparent nature of caches. Consequently, these works leave software-level programming practices targeted specifically at cache efficiency largely unexplored.

In the broader domain of software optimization for resource-constrained platforms, recent systematic reviews have investigated compiler techniques and general programming practices aimed at performance enhancement and energy reduction. For example, [Kautish and Gurung 2025] provide a comprehensive synthesis of energy-aware software design, highlighting compiler-level optimizations and dynamic task scheduling for sustainable computing. Similarly, broader surveys on embedded systems, such as the works by [Lahari et al. 2025] and [Salman 2025], discuss system-level power management, efficient coding practices, and the challenges of designing green real-time embedded systems. However, these studies approach optimization from a holistic system perspective or focus on generic low-power code improvements; none of them isolate and systematize programming interventions specifically designed to maximize spatial and temporal cache locality.

This scoping review differentiates itself by bridging these two distinct domains. Unlike existing surveys that detail hardware-centric memory management [Gracioli et al. 2015, Tabbassum et al. 2019] or cover generic energy-efficient embedded programming practices [Kautish and Gurung 2025, Lahari et al. 2025, Salman 2025], this work systematically maps and categorizes software-level interventions—such as data layout reordering and loop transformations—explicitly targeted at cache efficiency. By consolidating these hardware-independent strategies, this review addresses a critical gap in

the literature, providing developers with a focused, actionable framework for cache-aware programming in resource-constrained environments.

4. Methodology

This scoping review was conducted in accordance with the PRISMA-ScR guidelines (Preferred Reporting Items for Systematic Reviews and Meta-Analyses extension for Scoping Reviews) [Tricco et al. 2018], which were adopted to ensure methodological transparency and reproducibility in mapping the existing literature. A scoping review was chosen over a formal systematic review because the primary objective of this study is to broadly map the available evidence, identify categories of techniques, and recognize gaps in the field, without the intention of assessing methodological quality or estimating aggregate effects. This design is particularly appropriate for heterogeneous and emerging domains where multiple experimental approaches, platforms, and metrics coexist [Arksey and O'Malley 2005, Peters et al. 2015].

This study was guided by the following research questions, formulated prior to database searches:

i. RQ1: *Which programming practices have been proposed or empirically evaluated to optimize cache memory utilization in embedded systems*?
ii. RQ2: *Which metrics and experimental methods have been used to assess the effectiveness of these practices*?
iii. RQ3: *Which challenges, limitations, and research opportunities do the selected studies identify in the domain of software-level cache optimization*?

These questions were formulated in response to the observation that software-level interventions targeting cache efficiency remain fragmentarily documented in the literature, as detailed in Subsection 3.

Searches were executed between October 16 and November 30, 2025, across four databases selected for their coverage of computer architecture, systems software, and embedded engineering: IEEE Xplore, ACM Digital Library, ScienceDirect, and the CAPES Portal. These were the only databases that allowed full application of the required filters, open access (to ensure reproducibility of data extraction), document type, and publication period, which was essential for methodological standardization. The search string was validated through pilot searches prior to formal execution. The final string, ("cache memory optimization" OR "cache-aware programming" OR "cache-friendly algorithms" OR "cache locality" OR "memory optimization programming") AND ("embedded systems"), was applied to the primary search fields of each database.

Eligibility criteria are presented in Table 1. Studies published within the 2015–2025 window were targeted, as this period corresponds to intensified experimental activity in cache optimization research [Gracioli et al. 2015].

We also included software–hardware co-design studies when software annotations, compilation, or runtime policies are required to materialize cache-locality benefits.

From an initial corpus of 2,609 retrieved articles, successive filters for access type, publication period, and document category reduced the pool to 162 candidates. Abstract screening then yielded the 11 primary studies ultimately included. The complete selection process is illustrated in Figure 1.

Table 1. Inclusion and exclusion criteria.

Inclusion Criteria	Exclusion Criteria
Published between 2015 and 2025	Duplicates
Full text available (open access)	Purely theoretical studies
Investigates software-level programming practices for cache utilization	Exclusive focus on hardware solutions with no relation to programming practices
Presents empirical results	Absence of experimental results

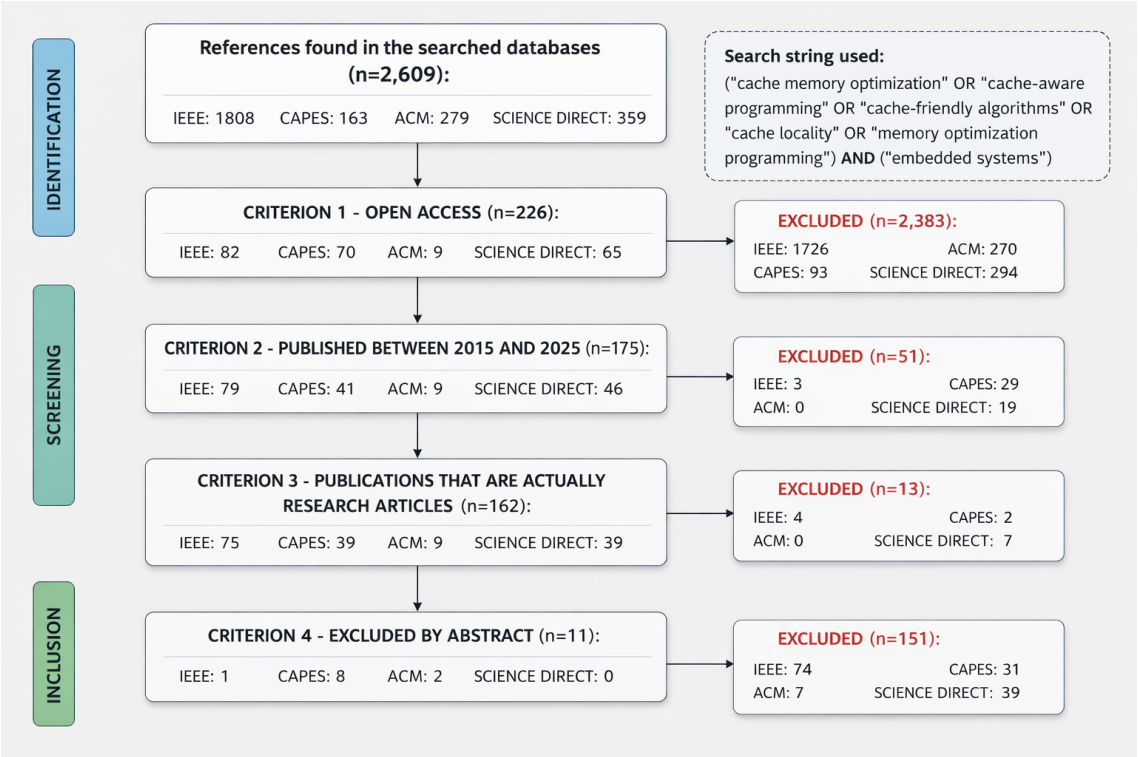

Figure 1. PRISMA flowchart of the study selection process.

For each selected article, we systematically extracted bibliographic data, evaluated techniques, experimental environment, performance metrics, and implications for embedded systems.

5. Results

This review included 11 articles presenting programming practices for cache optimization with full or partial applicability to embedded systems. The studies were grouped thematically into four categories: data layout and reordering, loop transformations, memory alignment and range propagation, and cache-aware scheduling. Table 2 provides a comprehensive synthesis of all included studies, detailing authorship, core techniques, thematic categories, target platforms, evaluation metrics, and reported gains.

This detailed overview in Table 2 serves as an evidence map, illustrating the diversity of approaches and their reported effectiveness across various experimental contexts.

Table 2. Synthesis of the 11 studies included in the review, detailing techniques, platforms, metrics, and reported gains.

ID	Author (Year)	Category	Core Technique	Platform Type	Metrics Evaluated	Reported Gains
1	[Lifflander and Krishnamoorthy 2017]	Loop Transformations	Dynamic splicing with annotations	Intel multi-core	L2 misses	3.7× reduction
2	[Altman et al. 2019]	Data Layout	Cache-aware splitting	Intel i7	Execution time	Up to 44% reduction
3	[Mu et al. 2024]	Memory Alignment	CoSense (sensor ranges)	Embedded (ARM/x86 MCUs)	Energy, Execution Time	Up to 30% energy reduction, 1.5-1.7× speedup
4	[Jung and Yang 2019]	Cache-Aware Scheduling	Context-aware dataflow adaptation	ARM Exynos (big.LITTLE)	Power	29% reduction
5	[Oh et al. 2021]	Data Layout	PMU-guided heap allocation (MaPHeA)	Intel Xeon (HMS)	Execution time	56% speedup
6	[Cattaneo et al. 2015]	Loop Transformations	Streaming SST queuing	Xilinx Virtex-7 (FPGA)	Throughput	1.2 GCells/s
7	[Tasos et al. 2020]	Data Layout	SHAPES (pools and layouts)	Modern architectures	Locality	Qualitative improvement
8	[Inoue and Taura 2015]	Data Layout	Cache-friendly SIMD mergesort	Intel Xeon	Execution time	2.1× speedup
9	[Stratis and Rajan 2018]	Cache-Aware Scheduling	Test suite reordering	General (EEMBC)	Execution time	Up to 17% speedup
10	[Jung et al. 2023]	Data Layout	Dual cache + WST	Simulated	Energy-delay product	17–70% improvement
11	[Herlihy and Liu 2017]	Loop Transformations	Structured futures (single-touch)	Multicore theory	Additional misses	$O(CPT_\infty^2)$

It highlights the prevalence of certain categories, such as data layout and reordering, and underscores the range of platforms and metrics employed in this research domain. Building upon this consolidated overview, the following subsections delve deeper into how these thematic categories translate into practical software-level optimization techniques to enhance cache efficiency.

5.1. Software-Level Optimization Techniques

Five studies address data layout and structural reordering to enhance spatial locality. [Altman et al. 2019] propose cache-aware splitting for multidimensional correlations, achieving up to a 44% reduction in execution time, while [Oh et al. 2021] present MaPHeA, a PMU-guided heap allocation strategy that dynamically reallocates objects based on runtime access profiles, yielding a 56% average performance improvement for graph workloads on heterogeneous memory systems. These two approaches share an orientation toward runtime awareness, demonstrating that static data structure design alone is insufficient when workload access patterns vary dynamically.

The remaining data layout studies operate through static structural reorganization. [Tasos et al. 2020] introduce the SHAPES extension, which provides memory pools and cache-friendly layouts for managed languages without requiring modifications to application logic. [Inoue and Taura 2015] optimize structure ordering through a cache-friendly SIMD mergesort, obtaining a 2.1× speedup over key-index methods, while [Jung et al. 2023] propose a dual cache architecture using a Way Selection Table that improves the energy-delay product by 17–70% relative to baselines.

In the domain of loop transformations, [Lifflander and Krishnamoorthy 2017] apply dynamic splicing with compiler annotations to recursive programs, achieving a 3.7× reduction in L2 misses. [Cattaneo et al. 2015] demonstrate that streaming SST queuing applied to iterative stencil loops on FPGAs reaches a throughput of 1.2 GCells/s, and

[Herlihy and Liu 2017] establish that structured futures with single-touch access bound additional misses to $O(CPT_{\infty}^{2})$, providing a formal theoretical foundation for cache-conscious parallel programming.

Beyond structural and loop-level interventions, [Mu et al. 2024] address memory alignment and range propagation through CoSense, a framework that propagates sensor technical specifications through the LLVM compiler infrastructure to eliminate redundant memory accesses at compile time. Evaluated on embedded ARM and x86 MCUs, CoSense achieved up to a 30% reduction in energy consumption and a 1.5–1.7$\times$ speedup, representing the most directly applicable result to low-power embedded platforms among all included studies.

5.2. Cache-Aware execution and Scheduling

We use 'execution and scheduling' to cover runtime/task mapping, reordering, and scheduling policies that affect cache locality without necessarily changing program semantics. Six studies address cache-aware scheduling, making it the most represented category in the review. A context-aware dataflow adaptation technique for ARM Exynos big.LITTLE architectures achieves a 29% reduction in power consumption by dynamically switching task execution between cores in a cache-informed manner [Jung and Yang 2019]. In a complementary direction, reordering test suites based on instruction distance metrics has been shown to yield a speedup of up to 17% on EEMBC benchmarks, without modifying the underlying application code [Stratis and Rajan 2018]. These two works illustrate that scheduling-level interventions can produce meaningful gains even when the application itself remains untouched.

The remaining scheduling-related studies contribute prioritization mechanisms within work-stealing, streaming, and selective access frameworks [Herlihy and Liu 2017, Lifflander and Krishnamoorthy 2017, Cattaneo et al. 2015, Jung et al. 2023], with their overlap across multiple thematic categories reflecting the interconnected nature of cache optimization strategies in practice. A subset of the included studies evaluates embedded or embedded-class platforms (e.g., ARM SoCs, big.LITTLE, FPGA-based systems, and EEMBC), while others provide transferable evidence from general-purpose platforms. The reported gains range from 1.5–2.1$\times$ in execution time and 20–30% in energy consumption, confirming that software-level cache optimization delivers measurable improvements in resource-constrained environments without hardware modifications.

6. Discussion

The analysis of the 11 included studies reveals that programming practices targeting cache memory optimization have a significant impact on performance and energy efficiency, even on platforms with severe hardware constraints. However, the studies are unevenly distributed across technical categories, and their results must be interpreted within their specific experimental contexts. The predominance of techniques based on data layout and structural reorganization is a primary finding.

Works such as [Altman et al. 2019] and [Inoue and Taura 2015] demonstrate that physical data reorganization alone can yield substantial improvements in spatial locality without modifying access strategies. Conversely, MaPHeA shows that combining PMU-derived information with dynamic heap reallocation captures access patterns that elude

static design [Oh et al. 2021], evidencing a growing shift toward runtime-aware techniques that is particularly promising for embedded systems subject to dynamic workload variations.

In the domain of loop transformations, the literature indicates that such techniques are notably effective in scenarios with high access regularity. [Lifflander and Krishnamoorthy 2017] reports significant gains in L2 miss reduction, while [Cattaneo et al. 2015] demonstrates that transformations coupled with reconfigurable architectures can operate at extremely high throughput rates. However, these methods have less impact on irregular applications or those that depend on dynamic graphs, suggesting that their effectiveness is strongly conditioned by the predictability of the access pattern.

Cache-aware scheduling, in turn, extends this logic to the execution layer: [Jung and Yang 2019] shows that integrating the scheduler with memory hierarchy information can significantly reduce energy consumption in big.LITTLE architectures, while techniques based on work-stealing and task prioritization demonstrate that cache information can be incorporated into high-level scheduling algorithms without modifying application code directly.

A critical observation from our review, as highlighted in Table 2, is the significant divergence in the experimental platforms used across the studies. While the focus of this review is on embedded systems, a substantial portion of the evaluated techniques relies on simulations or high-performance computing (HPC) environments. This disparity implies that the reported gains, while promising, may not directly translate to the unique constraints of low-power embedded devices, which often feature smaller caches, different memory hierarchies, and stricter power budgets. The architectural differences between HPC and embedded platforms mean that cache behaviors and optimization effectiveness can vary considerably, necessitating careful consideration when applying these findings to real-world embedded contexts.

Despite these advancements, the review identified significant gaps in the evidence base. Techniques that simultaneously optimize temporal and spatial locality tend to produce the greatest gains, yet holistic approaches that consider multiple dimensions of the memory hierarchy remain underexplored. Similarly, the growing use of runtime-obtained information, such as PMU data, sensor statistics, and compiler-level metadata, points toward a trend of adaptive and context-aware optimizations that has not yet been systematically studied in the embedded domain.

Furthermore, while the identified techniques offer substantial performance and energy benefits, their practical adoption is often hindered by the effort required for manual implementation and integration. Few research efforts integrate these techniques into modern toolchains such as LLVM passes, which substantially hinders practical adoption. This lack of automated integration implies a higher development cost and potential for introducing bugs, making it challenging for developers to leverage these optimizations effectively in real-world embedded software projects. Addressing this gap by developing compiler-level support for automated cache-aware transformations represents a significant opportunity for future research.

There is also a notable absence of studies addressing complex data structures,

irregular applications, and embedded AI workloads, domains expected to grow considerably in the coming years.

7. Limitations

Several methodological constraints shape the interpretation of these findings. Restricting the search to four databases and requiring open access likely introduced selection bias, potentially excluding relevant paywalled or otherwise indexed studies. This decision, while potentially limiting the breadth of included literature, was made to ensure the full reproducibility of our data extraction process, a core tenet of rigorous scoping reviews. Future work could explore strategies to include paywalled content, provided a robust and reproducible extraction methodology can be maintained. The selected literature also exhibits considerable methodological heterogeneity. Most evaluations rely on simulators or High-Performance Computing HPC environments rather than actual low-power microcontrollers (MCUs) or System-on-Chip (SoC).

This platform divergence, coupled with varying metrics, ranging from execution time to cache miss rates, precludes direct quantitative comparisons and limits the generalizability of reported gains to real-world embedded constraints. Furthermore, the scarcity of detailed energy measurements extracted directly from physically embedded hardware weakens the empirical basis for energy-related conclusions.

Consistent with scoping review methodologies, this study did not formally assess the methodological quality of the included articles, meaning some reported evidence may carry unevaluated experimental limitations. This approach is inherent to scoping reviews, which prioritize mapping the breadth of evidence over in-depth critical appraisal of individual study quality. Therefore, readers should interpret these results as a structured mapping of the state of the art rather than a conclusive synthesis of technique effectiveness, acknowledging that the methodological rigor of individual studies was not a primary focus of this review.

8. Conclusions and Future Work

This scoping review systematically mapped 11 empirical studies. It categorized the programming practices they report into four thematic groups: data layout and reordering, loop transformations, memory alignment and range propagation, and cache-aware scheduling, all designed to optimize cache memory utilization in embedded systems. The comprehensive synthesis presented in Table 2 serves as a valuable evidence map, clearly delineating the diverse techniques, their target platforms, and the performance and energy gains achieved across various experimental settings.

This work's contributions include clear guidelines for developers, summarized theoretical bounds, and the identification of critical research gaps. The practical implications are significant: achieving substantial reductions in energy consumption and latency without requiring hardware modifications.

Building upon the identified gaps, particularly the observed platform divergence and the limited integration into modern development workflows, future research should prioritize several key areas. Equally important is the automatic integration of the identified techniques into compiler infrastructure, such as LLVM/Clang passes, enabling data

layout and scheduling optimizations to be applied without manual developer intervention, including the exploration of software prefetching as a complementary strategy. Finally, future studies should address the gap in irregular application domains, particularly dynamic graphs and embedded AI workloads involving complex pointer structures, which are expected to represent a growing share of embedded computing in the coming years.

References

Altman, E. A., Vaseeva, T. V., and Aleksandrov, A. V. (2019). Cache-aware algorithm for multidimensional correlations. *Journal of Physics: Conference Series*, 1260(4):042001.

Arksey, H. and O'Malley, L. (2005). Scoping studies: towards a methodological framework. *International Journal of Social Research Methodology*, 8(1):19–32.

Cattaneo, R., Natale, G., Sicignano, C., Sciuto, D., and Santambrogio, M. D. (2015). On how to accelerate iterative stencil loops: A scalable streaming-based approach. *ACM Transactions on Architecture and Code Optimization*, 12(4):1–26.

Gracioli, G., Alhammad, A., Mancuso, R., Fröhlich, A., and Pellizzoni, R. (2015). A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys*, 48(2).

Hennessy, J. L. and Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6 edition.

Herlihy, M. and Liu, Z. (2017). Well-structured futures and cache locality. arXiv:1309.5301 [cs.DC]. Disponível em: `https://arxiv.org/abs/1309.5301`.

Inoue, H. and Taura, K. (2015). Simd- and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment*, 8(11):1274–1285.

Jaleel, A. et al. (2010). Adaptive insertion policies for managing shared caches. *IEEE Micro*, 30(1):19–31.

Jung, B.-S., Kim, H.-R., and Lee, J.-H. (2023). Using cache locality for cache memory system design. *Asia-pacific Journal of Convergent Research Interchange*, 9(10):41–50.

Jung, H. and Yang, H. (2019). Efficiently switchable context-aware dataflow adaptation technique for low-power multi-core embedded systems. *IEEE Access*, 7:177974–177986.

Kautish, S. and Gurung, D. (2025). Advancing sustainable computing: A systematic literature review of software, hardware, and algorithmic innovations. *ICCK Transactions on Sustainable Computing*, 1(1):1–19.

Lahari, R., Lohith, T., Manjunath, M., and Raju, R. G. (2025). A review on power optimization energy efficient techniques for embedded systems. *International Journal of Innovative Research in Technology (IJIRT)*, 11(11):390–397.

Lifflander, J. and Krishnamoorthy, S. (2017). Cache locality optimization for recursive programs. Technical Report SAND2017-5070C, Sandia National Laboratories, Albuquerque, NM, USA.

McFarling, S. (1995). Program optimization for instruction caches. Technical report, Digital Western Research Laboratory.

Megiddo, N. and Modha, D. S. (2003). Arc: A self-tuning, low-overhead replacement cache. In *Proceedings of USENIX FAST*.

Mu, P., Mavrogeorgis, N., Vasiladiotis, C., Tsoutsouras, V., Kaparounakis, O., Stanley-Marbell, P., and Barbalace, A. (2024). Cosense: Compiler optimizations using sensor technical specifications. *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24)*.

Oh, D.-J., Moon, Y., Lee, E., Ham, T. J., Park, Y., Lee, J. W., and Ahn, J. H. (2021). Maphea: A lightweight memory hierarchy-aware profile-guided heap allocation framework. *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '21)*.

Page, M. J. et al. (2021). The prisma 2020 statement: an updated guideline for reporting systematic reviews. *BMJ*, 372:n71.

Peters, M. D. J., Godfrey, C. M., Khalil, H., McInerney, P., Parker, D., and Soares, C. B. (2015). Guidance for conducting systematic scoping reviews. *International Journal of Evidence-Based Healthcare*, 13(3):141–146.

Salman, M. (2025). Challenges and practices identification via systematic literature review in the design of green/energy-efficient embedded real-time systems. *International Journal of Innovations in Science & Technology*, 7(1):190–209.

Stallings, W. (2018). *Computer Organization and Architecture*. Pearson, 11 edition.

Stratis, P. and Rajan, A. (2018). Speeding up test execution with increased cache locality. *Software Testing, Verification and Reliability*, 28(4).

Tabbassum, K., Talpur, S., Narejo, S., and Laghari, N.-u.-Z. (2019). Management of scratchpad memory using programming techniques. *Mehran University Research Journal of Engineering & Technology*, 38(2):305–312.

Tasos, A., Franco, J., Drossopoulou, S., Wrigstad, T., and Eisenbach, S. (2020). Reshape your layouts, not your programs: A safe language extension for better cache locality. *34th European Conference on Object-Oriented Programming (ECOOP 2020)*.

Tricco, A. C., Lillie, E., Zarin, W., O'Brien, K. K., Colquhoun, H., Levac, D., Moher, D., Peters, M. D. J., Horsley, T., Weeks, L., et al. (2018). Prisma extension for scoping reviews (prisma-scr): Checklist and explanation. *Annals of Internal Medicine*, 169(7):467–473.

Wolf, M. (2016). *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann, 4 edition.