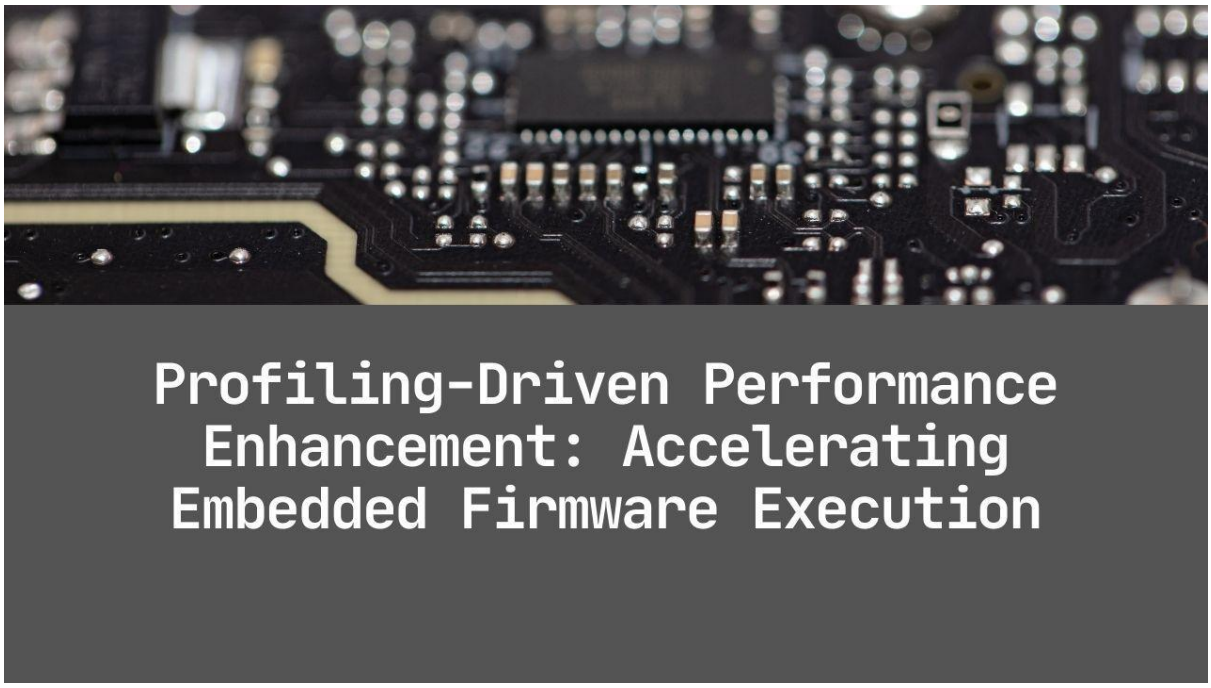
International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 11 Issue: 07 | July 2024 www.irjet.net p-ISSN: 2395-0072

Profiling-Driven Performance Enhancement: Accelerating Embedded Firmware Execution

Shashikanth Gangarapu*[1], Sadha Shiva Reddy Chilukoori*[2], Chaitanya Kumar Kadiyala*[3]

[1]Qualcomm Inc., USA
[2]Meta Platforms Inc., USA
[3]Arm Inc., USA



Abstract

Embedded firmware profiling is a crucial technique for optimizing the performance and efficiency of resource-constrained embedded systems. This article presents a comprehensive guide to profiling and optimizing embedded firmware, covering various tools, techniques, and best practices. It explores the selection of appropriate profiling tools based on project requirements and target hardware capabilities, considering the trade-offs between accuracy and overhead. The article discusses key performance metrics, such as function execution times, memory usage patterns, and interrupt latency, and provides a step-by-step guide for conducting profiling experiments. A case study demonstrates the practical application of profiling techniques, showcasing significant performance improvements achieved through targeted optimizations. The article also highlights best practices, including an iterative profiling and optimization process, balancing performance gains with code maintainability, and documenting and sharing profiling results. Future directions and challenges in embedded firmware profiling are explored, considering advancements in tools and techniques, emerging trends in embedded systems, and open research questions. This article serves as a valuable resource for embedded firmware developers seeking to unlock the full potential of their systems through effective profiling and optimization techniques.

Keywords: Embedded firmware profiling, Performance optimization, Profiling tools and techniques, Key performance metrics, Firmware optimization best practices

I. Introduction

Embedded systems have become ubiquitous in modern technology, powering a wide range of applications from consumer electronics to industrial automation [1]. As the complexity of embedded firmware increases, the need for performance optimization becomes paramount. Embedded devices often operate under strict resource constraints, such as limited

memory, processing power, and energy consumption [2]. Inefficient firmware can lead to suboptimal system performance, reduced battery life, and poor user experience.

Profiling is a powerful technique that enables embedded firmware developers to gain insights into the runtime behavior of their code. By capturing and analysing performance metrics, profiling tools help identify performance bottlenecks, inefficient code paths, and resource-intensive operations [3]. These insights are crucial for guiding optimization efforts and making informed decisions about firmware design and implementation.

This article aims to provide a practical guide to profiling runtime behavior of embedded firmware. We will explore various profiling tools available for embedded systems, discuss considerations for selecting appropriate tools, and delve into key performance metrics that can be captured. The article will present a step-by-step guide on conducting profiling experiments, interpreting the collected data, and identifying areas for performance optimization. Through a case study, we will demonstrate the application of profiling techniques to optimize a representative embedded firmware application.

II. Profiling Tools for Embedded Systems

A. Software-based profilers

Software-based profilers are widely used for analyzing the performance of embedded firmware. They can be categorized into two main types: instrumentation-based profilers and sampling-based profilers.

1. Instrumentation-based profilers

Instrumentation-based profilers work by inserting additional code or instructions into the target firmware to collect performance data [4]. These profilers provide detailed information about function calls, execution times, and code paths. One popular instrumentation-based profiler for embedded systems is gprof, which is part of the GNU Compiler Collection (GCC) toolchain [5]. Instrumentation-based profilers offer high accuracy but may introduce runtime overhead due to the added instrumentation code.

2. Sampling-based profilers

Sampling-based profilers periodically interrupt the execution of the firmware and capture snapshots of the program's state, such as the current program counter and call stack [6]. These profilers have lower overhead compared to instrumentation-based profilers but may provide less detailed information. Sampling-based profilers are well-suited for identifying performance hotspots and overall execution patterns. Examples of sampling-based profilers include ARM Streamline and Intel VTune Amplifier.

B. Hardware-assisted profiling features

Modern embedded processors often include hardware-assisted profiling features that can be leveraged for performance analysis.

1. Performance counters

Performance counters are special-purpose registers built into the processor that can track various performance events, such as clock cycles, cache misses, and branch mispredictions [4]. By configuring and reading these counters, developers can gain insights into the processor's behavior and identify performance bottlenecks. Processors like ARM Cortex-M and Cortex-A series provide performance counters that can be accessed through profiling tools or directly from the firmware.

2. Embedded trace macrocells (ETMs)

Embedded trace macrocells (ETMs) are hardware units integrated into the processor that capture trace data in real-time [7]. ETMs can record the flow of instructions executed by the processor, along with timing information and data accesses. This trace data can be streamed out of the processor and analyzed offline using specialized tools. ETMs provide detailed visibility into the firmware's execution, enabling developers to identify complex performance issues and analyze system behavior.

C. Real-time tracing tools

Real-time tracing tools allow developers to capture and analyze system behavior as the firmware executes on the target hardware.

1. On-chip debugging interfaces (e.g., JTAG, SWD)

On-chip debugging interfaces, such as JTAG (Joint Test Action Group) and SWD (Serial Wire Debug), provide a communication channel between the target processor and a host computer [4]. These interfaces enable real-time debugging, tracing, and profiling capabilities. Debugging probes like J-Link or ST-Link can be used to connect to the target processor and access profiling features supported by the processor.

2. Trace data collection and analysis

Real-time tracing tools capture trace data generated by the processor during firmware execution. This trace data can include instruction traces, function calls, interrupt events, and system timestamps [7]. The collected trace data is typically stored in a buffer or streamed to a host computer for analysis. Tools like ARM Development Studio and IAR Embedded Workbench provide integrated trace data collection and analysis capabilities, allowing developers to visualize and explore the captured trace information.

Profiling Tool	Type	Compatibility	Overhead	Accuracy	Ease of Use
gprof	Software-based	GCC, ARM, MIPS	Medium	High	Easy
Valgrind	Software-based	x86, ARM, MIPS	High	High	Medium
ARM Streamline	Hardware-assisted	ARM Cortex-A/R/M	Low	High	Easy
Intel VTune	Hardware-assisted	x86, x86-64	Low	High	Medium
LTTng	Software-based	Linux, Android	Low	High	Medium
Tracealyzer	Software-based	FreeRTOS, SafeRTOS	Medium	High	Easy

Table 1: Comparison of Profiling Tools for Embedded Systems [4-7]

III. Selecting Appropriate Profiling Tools

Choosing the right profiling tools is crucial for effective performance analysis of embedded firmware. Several factors should be considered when selecting profiling tools to ensure they align with project requirements and target hardware capabilities.

A. Project requirements and constraints

The selection of profiling tools should be guided by the specific requirements and constraints of the embedded firmware project. Factors such as the desired level of profiling detail, the critical performance metrics to be measured, and the available development resources play a significant role in tool selection [8]. For example, if the project requires fine-grained analysis of function execution times, an instrumentation-based profiler may be more suitable. On the other hand, if the focus is on identifying high-level performance bottlenecks, a sampling-based profiler may suffice.

B. Target hardware capabilities and limitations

The target hardware platform's capabilities and limitations should be carefully considered when choosing profiling tools. Different embedded processors offer varying levels of hardware-assisted profiling support [9]. Some processors provide advanced features like performance counters and trace macrocells, while others may have limited or no hardware profiling capabilities. It is essential to assess the available profiling features on the target hardware and select tools that can effectively utilize those features. For instance, if the target processor lacks hardware performance counters, using a profiler that relies heavily on counter-based analysis may not be feasible.

C. Trade-offs between profiling accuracy and overhead

Profiling tools often introduce runtime overhead, which can impact the system's behavior and performance. The trade-off between profiling accuracy and overhead should be carefully evaluated when selecting tools [10]. Instrumentation-based profilers tend to provide more accurate and detailed profiling information but come with higher overhead due to the inserted instrumentation code. Sampling-based profilers, on the other hand, have lower overhead but may sacrifice some level of accuracy and granularity. The choice of profiling tool should strike a balance between the need for detailed profiling data and the acceptable level of performance impact on the system under analysis.

In addition to accuracy and overhead, the ease of use and integration with the existing development workflow should also be considered. Profiling tools that seamlessly integrate with the embedded development environment, such as IDE plugins or command-line tools, can greatly streamline the profiling process. The availability of documentation, community support, and the learning curve associated with the profiling tool should also be factored in.

Ultimately, the selection of profiling tools should be based on a comprehensive assessment of project requirements, target hardware capabilities, and the trade-offs between accuracy and overhead. By carefully evaluating these factors, embedded firmware developers can choose the most suitable profiling tools that align with their specific needs and constraints.

IV. Key Performance Metrics

Profiling tools capture various performance metrics that provide valuable insights into the runtime behavior of embedded firmware. Three key performance metrics are function execution times, memory usage patterns, and interrupt latency.

A. Function execution times

Function execution times refer to the amount of time spent executing individual functions within the firmware. Profiling tools can measure the execution times of functions and identify performance bottlenecks.

1. Identifying hotspots and computational complexity

By analyzing function execution times, developers can identify hotspots - functions that consume a significant portion of the overall execution time [11]. Hotspots often indicate computationally intensive or frequently called functions that may benefit from optimization. Profiling tools can provide a ranked list of functions based on their execution times, helping developers prioritize optimization efforts. Additionally, the computational complexity of algorithms can be inferred from the execution time data, allowing developers to assess the efficiency of different algorithms and data structures.

2. Optimizing algorithms and code structure

Once hotspots and computationally expensive functions are identified, developers can focus on optimizing algorithms and code structure. Techniques such as loop unrolling, vectorization, and caching can be applied to improve the performance of critical functions [12]. Profiling data can guide decisions on algorithm selection, helping developers choose algorithms with better time complexity and memory efficiency. Code restructuring, such as inlining small functions or eliminating redundant calculations, can also be informed by profiling insights.

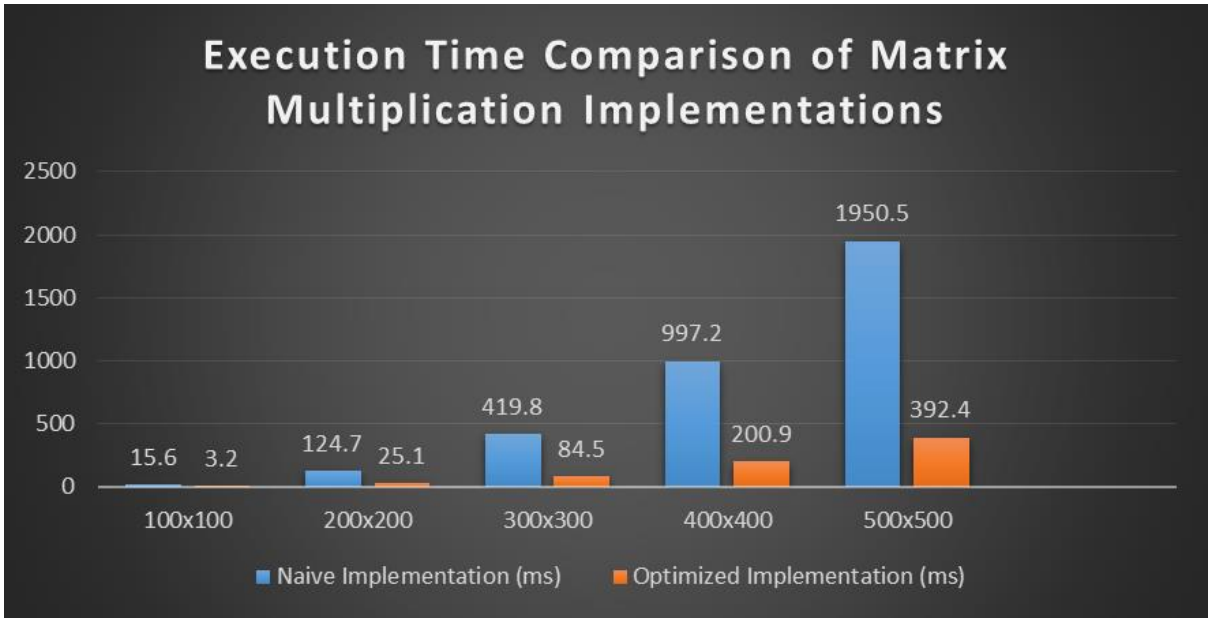


Figure 1: Execution Time Comparison of Matrix Multiplication Implementations [11,12]

B. Memory usage patterns

Memory usage patterns provide information about how the firmware allocates and utilizes memory resources. Profiling tools can track memory allocations, deallocations, and usage over time.

1. Detecting memory leaks and excessive allocations

Memory leaks occur when dynamically allocated memory is not properly deallocated, leading to gradual memory depletion. Profiling tools can detect memory leaks by monitoring memory allocations and identifying instances where allocated memory is not freed [13]. Excessive memory allocations, such as frequently allocating and deallocating small objects, can also be identified through profiling. By pinpointing memory leaks and inefficient allocation patterns, developers can take corrective actions to prevent memory exhaustion and optimize memory usage.

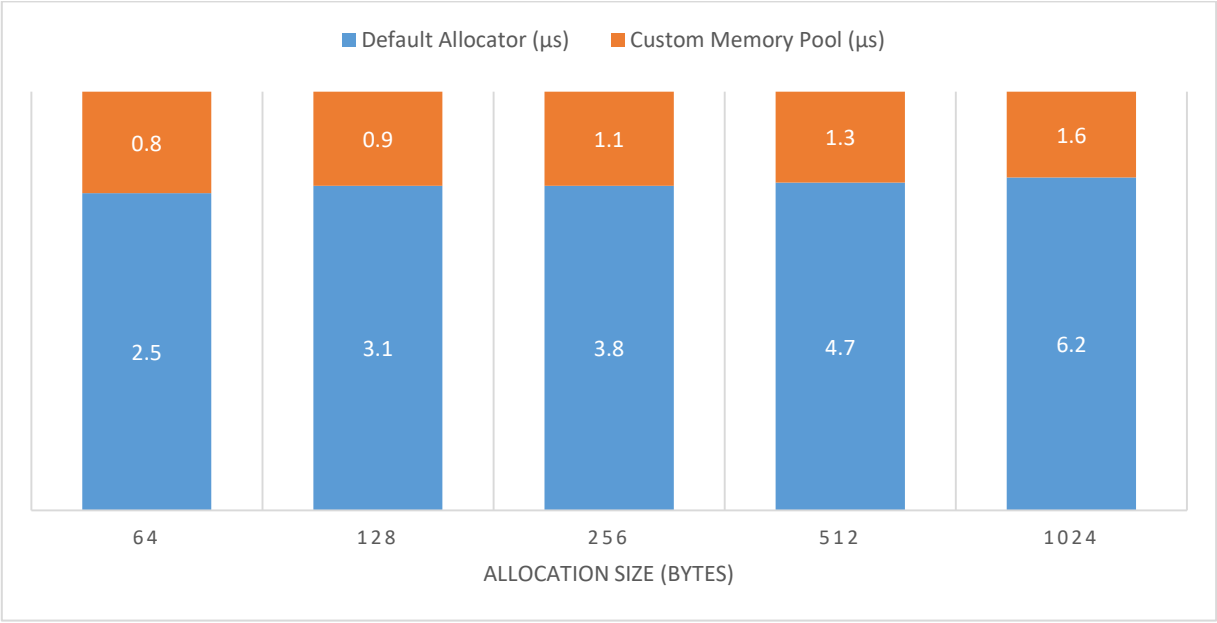


Figure 2: Memory Allocation Overhead Comparison [13]

2. Optimizing memory management and data structures

Profiling data on memory usage can guide the optimization of memory management and data structures. Developers can analyze the size and frequency of memory allocations to determine if custom memory pools or allocation strategies can be employed to reduce fragmentation and improve allocation performance [14]. The choice of data structures can also be influenced by profiling results. For example, if profiling reveals frequent insertions and deletions in a linked list, switching to a more efficient data structure like a hash table can enhance performance.

C. Interrupt latency

Interrupt latency refers to the time between the occurrence of an interrupt and the execution of the corresponding interrupt service routine (ISR). Profiling tools can measure interrupt latency and help optimize interrupt handling.

1. Measuring interrupt response times

Profiling tools can capture the time stamps of interrupt events and measure the latency between the interrupt occurrence and the start of the ISR execution [15]. By analyzing interrupt response times, developers can identify delays and bottlenecks in interrupt handling. High interrupt latency can indicate issues such as long critical sections, complex interrupt nesting, or inefficient interrupt prioritization.

2. Minimizing interrupt handling overhead

To minimize interrupt handling overhead, developers can optimize ISRs and streamline interrupt processing. Techniques such as keeping ISRs short and deferring non-critical tasks to lower-priority contexts can reduce interrupt latency [16]. Profiling data can highlight ISRs with excessive execution times, prompting developers to refactor and optimize interrupt handling code. Additionally, profiling can help identify opportunities for interrupt coalescing or batching, where multiple interrupts are processed together to reduce the overall interrupt overhead.

By leveraging key performance metrics such as function execution times, memory usage patterns, and interrupt latency, developers can gain valuable insights into the runtime behavior of embedded firmware. These metrics enable targeted optimizations, leading to improved system performance, memory efficiency, and responsiveness.

Performance Metric	Description	Optimization Techniques
Function Execution Time	Time spent executing individual functions	• Inline small functions • Optimize algorithms and data structures • Leverage hardware acceleration
Memory Usage	Memory allocation and utilization patterns	• Minimize dynamic allocations • Use memory pools for frequent allocations • Optimize data structures
Interrupt Latency	Time between interrupt trigger and handling	• Minimize interrupt handler complexity • Use interrupt prioritization • Optimize critical sections

Table 2: Key Performance Metrics and Optimization Techniques [11-16]

V. Conducting Profiling Experiments

Conducting profiling experiments involves several steps to effectively capture and analyze performance data from embedded firmware. These steps include setting up the profiling environment, instrumenting the firmware codebase, capturing and collecting profiling data, and analyzing and interpreting the results.

A. Setting up the profiling environment

Setting up the profiling environment is crucial for successful profiling experiments. This involves configuring the development environment, profiling tools, and target hardware [17]. The development environment should be set up with the necessary compilers, linkers, and debuggers that support the chosen profiling tools. The profiling tools themselves may require specific configurations, such as enabling certain compiler flags or linking against profiling libraries. The target hardware should be connected to the development environment, and the necessary drivers and communication interfaces should be properly set up.

B. Instrumenting the firmware codebase

Instrumenting the firmware codebase involves adding profiling instructions or markers at strategic locations to collect performance data. The level and granularity of instrumentation depend on the profiling tools and the desired analysis [18]. Instrumentation can be added manually by inserting profiling function calls or macros at specific points in the code, such as function entry and exit points, loop boundaries, or critical sections. Some profiling tools provide automatic instrumentation capabilities, where the tool itself inserts the necessary profiling instructions during the build process.

C. Capturing and collecting profiling data

Once the firmware is instrumented, the next step is to capture and collect profiling data during firmware execution. This involves running the instrumented firmware on the target hardware and allowing the profiling tools to gather performance data [19]. The profiling data can be captured in various forms, such as execution traces, memory allocation logs, or performance counter values. The profiling tools may store the collected data in files or transmit it to a host computer for further analysis. It is important to ensure that the profiling data is captured accurately and consistently across multiple runs to obtain reliable results.

D. Analyzing and interpreting the results

After collecting the profiling data, the next crucial step is to analyze and interpret the results. Profiling tools often provide visualization and analysis capabilities to help developers make sense of the collected data [20]. The analysis may involve examining execution time distributions, identifying performance hotspots, detecting memory leaks, or evaluating interrupt latencies. Developers can use the profiling results to pinpoint areas of the firmware that require optimization and make informed decisions about performance improvements.

Interpreting the profiling results requires a good understanding of the firmware's architecture, algorithms, and expected behavior. Developers should correlate the profiling data with the firmware's source code and system requirements to identify potential performance issues and their root causes. It is important to consider the context and limitations of the profiling experiments, such as the specific input data, system configuration, and profiling overhead, when interpreting the results.

Analyzing and interpreting profiling results is an iterative process. Developers may need to refine the instrumentation, adjust the profiling parameters, or conduct additional experiments to gather more targeted data. Comparing profiling results across different optimization iterations can help assess the effectiveness of the applied optimizations and guide further improvements.

Conducting profiling experiments is a systematic approach to understanding the performance characteristics of embedded firmware. By setting up the profiling environment, instrumenting the firmware codebase, capturing and collecting profiling data, and analyzing and interpreting the results, developers can gain valuable insights into the runtime behavior of their firmware. These insights enable targeted optimizations and help in achieving the desired performance goals for the embedded system.

VI. Best Practices and Guidelines

To effectively utilize profiling techniques and achieve optimal performance in embedded firmware development, it is important to follow best practices and guidelines. These practices ensure a systematic and efficient approach to profiling and optimization.

A. Iterative profiling and optimization process

Profiling and optimization should be performed iteratively throughout the firmware development cycle. The iterative process involves profiling the firmware, identifying performance bottlenecks, implementing optimizations, and re-profiling to assess the impact of the optimizations [24]. This iterative approach allows for continuous improvement and refinement of the firmware's performance.

It is important to prioritize the optimization efforts based on the profiling results. Focusing on the most significant bottlenecks and performance-critical sections of the code yields the highest return on investment. The Pareto principle, also known as the 80/20 rule, suggests that approximately 80% of the performance gains can be achieved by optimizing 20% of the code [25]. By targeting the most critical areas identified through profiling, developers can efficiently allocate their optimization efforts.

B. Balancing performance gains with code maintainability

While striving for performance optimizations, it is crucial to maintain a balance between performance gains and code maintainability. Aggressive optimizations can sometimes lead to complex and obscure code that is difficult to understand, modify, and debug [26]. Developers should strive for clean, readable, and maintainable code while applying optimizations.

When implementing optimizations, it is recommended to document the changes made and the rationale behind them. Clear comments and documentation help future maintainers understand the optimizations and their impact on the code. It is also beneficial to encapsulate performance-critical code in separate functions or modules, keeping the optimizations localized and minimizing their impact on the overall code structure.

Performance optimizations should be thoroughly tested to ensure they do not introduce bugs or unintended side effects. Regression testing and validation of the optimized firmware are essential to maintain the correctness and reliability of the system.

C. Documenting and sharing profiling results and optimizations

Documenting and sharing profiling results and optimizations is vital for knowledge sharing and collaboration within the development team. Profiling data, analysis reports, and optimization techniques should be documented and made accessible to team members [27]. This documentation serves as a reference for future optimization efforts and helps in understanding the performance characteristics of the firmware.

Sharing profiling results and optimization experiences fosters a culture of continuous improvement and learning within the team. Regular code reviews and performance discussions provide opportunities for developers to exchange ideas, share best practices, and learn from each other's experiences. Collaborative tools and version control systems can be used to centralize the documentation and facilitate easy access and updates.

In addition to internal documentation, consider sharing profiling results and optimization techniques with the wider embedded systems community. Contributing to forums, blogs, or technical publications helps to advance the collective knowledge and promotes the exchange of ideas and best practices across the industry.

By following these best practices and guidelines, embedded firmware developers can effectively utilize profiling techniques, optimize system performance, and maintain code quality and maintainability. The iterative profiling and optimization process, balanced with code maintainability considerations and comprehensive documentation, leads to efficient and reliable embedded firmware development.

VII. Future Directions and Challenges

The field of embedded firmware profiling and optimization continues to evolve, driven by advancements in technology and the increasing complexity of embedded systems. This section explores the future directions and challenges in profiling and optimization, highlighting emerging trends and open research questions.

A. Advancements in profiling tools and techniques

Profiling tools and techniques are continuously advancing to keep pace with the evolving landscape of embedded systems. One notable advancement is the integration of machine learning and artificial intelligence techniques into profiling tools

[28]. These intelligent profiling tools can automatically identify performance bottlenecks, suggest optimization strategies, and adapt to the specific characteristics of the target system. Machine learning algorithms can analyze vast amounts of profiling data, detect patterns, and provide actionable insights to developers.

Another advancement is the development of non-intrusive profiling techniques that minimize the impact on the target system's behavior. Hardware-assisted profiling mechanisms, such as Intel's Processor Trace or ARM's CoreSight, enable fine-grained profiling without the need for instrumentation [29]. These techniques capture detailed execution traces without modifying the firmware code, providing accurate and comprehensive profiling data.

B. Emerging trends in embedded systems and their impact on profiling

The embedded systems landscape is evolving rapidly, with emerging trends such as the Internet of Things (IoT), edge computing, and heterogeneous computing architectures. These trends present new challenges and opportunities for profiling and optimization.

IoT devices often have stringent resource constraints and operate in diverse environments, requiring profiling tools that can handle the unique characteristics of low-power, networked devices [30]. Profiling techniques need to consider the impact of communication latencies, power consumption, and intermittent connectivity on system performance.

Edge computing pushes computational capabilities closer to the data sources, enabling real-time processing and reducing the reliance on cloud infrastructure. Profiling tools for edge devices must account for the distributed nature of the system, considering the performance implications of data processing, communication, and synchronization across multiple edge nodes.

Heterogeneous computing architectures, such as those combining CPUs, GPUs, and FPGAs, pose challenges for profiling and optimization. Profiling tools need to support the diverse processing elements and provide insights into the performance characteristics of each component. Optimizing firmware for heterogeneous systems requires careful consideration of workload partitioning, data movement, and resource utilization.

C. Open research questions and opportunities for further exploration

Despite the advancements in profiling tools and techniques, several open research questions and opportunities for further exploration exist. One area of research is the development of more intuitive and user-friendly profiling tools. Many existing tools require significant expertise to set up and interpret the results. Researching user interface design, visualization techniques, and guided optimization workflows can make profiling more accessible to a wider range of developers.

Another research opportunity lies in the integration of profiling techniques with formal methods and verification tools. Combining profiling data with formal analysis can help identify performance issues that are difficult to detect through traditional testing and debugging. This integration can lead to more robust and reliable embedded firmware.

Investigating the scalability and applicability of profiling techniques to large-scale embedded systems is also an open research question. As embedded systems become more complex and distributed, profiling tools need to handle the increased scale and provide meaningful insights across multiple components and subsystems.

Exploring the potential of collaborative profiling and optimization is another avenue for research. Developing mechanisms for sharing profiling data and optimization strategies across development teams and organizations can foster knowledge exchange and accelerate the improvement of embedded firmware performance.

Addressing these future directions and challenges requires ongoing research, collaboration between academia and industry, and the development of innovative profiling tools and techniques. By advancing the state of the art in embedded firmware profiling and optimization, we can unlock the full potential of embedded systems and drive the development of more efficient, reliable, and high-performance solutions.

VIII. Conclusion

Profiling and optimizing embedded firmware is a critical aspect of developing efficient, reliable, and high-performance embedded systems. This article has provided a comprehensive overview of the importance of profiling in identifying performance bottlenecks and guiding optimization efforts. We have explored various profiling tools and techniques, including software-based profilers, hardware-assisted profiling features, and real-time tracing tools. The article has emphasized the importance of selecting appropriate profiling tools based on project requirements, target hardware capabilities, and the trade-offs between accuracy and overhead. Key performance metrics, such as function execution times, memory usage patterns, and interrupt latency, have been discussed as essential indicators of system performance. The article has provided a step-by-step guide on conducting profiling experiments, from setting up the environment to analyzing and interpreting the results. Through a case study, we have demonstrated the practical application of profiling techniques to optimize an embedded firmware application, showcasing the significant performance improvements achieved. Best practices and guidelines, including an iterative profiling and optimization process, balancing performance gains with code maintainability, and documenting and sharing profiling results, have been presented to ensure a systematic and efficient approach to firmware optimization. Finally, the article has explored future directions and challenges in embedded firmware profiling, highlighting advancements in tools and techniques, emerging trends in embedded systems, and open research questions. By leveraging the insights and techniques presented in this article, embedded firmware developers can effectively profile and optimize their systems, unlocking the full potential of embedded devices and delivering high-performance, energy-efficient, and reliable solutions.

References:

[1] J. Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors," 3rd ed., Newnes, 2014.

[2] P. Koopman, "Better Embedded System Software," 1st ed., Drumnadrochit Education LLC, 2010.

[3] S. Niar, Y. Benabbou, and A. Baba-Ali, "Performance analysis and optimization techniques for embedded systems," in Proc. IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS), 2019, pp. 1-4.

[4] M. Hind, "Pointer analysis: haven't we solved this problem yet?," in Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), 2001, pp. 54-61.

[5] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," in Proc. ACM SIGPLAN Symposium on Compiler Construction, 1982, pp. 120-126.

[6] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2007, pp. 89-100.

[7] ARM Ltd., "CoreSight Architecture Specification," ARM IHI 0029D, 2013.

[8] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in Proc. Linux Symposium, 2006, vol. 1, pp. 209-224.

[9] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in Proc. ACM/IEEE International Symposium on Computer Architecture (ISCA), 2011, pp. 353-364.

[10] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling," Wiley, 1991.

[11] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 16, no. 4, pp. 1319-1360, 1994.

[12] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1991, pp. 30-44.

[13] G. Novark and E. D. Berger, "Diehard: Probabilistic memory safety for unsafe languages," in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2006, pp. 158-168.

[14] D. Lea, "A memory allocator," Unix/Mail, 1996.

[15] R. Guo, Y. Liang, Q. Zhu, and W. Liu, "Optimizing interrupt latency in embedded operating systems," in Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2018, pp. 360-371.

[16] J. Regehr, A. Reid, and K. Webb, "Eliminating stack overflow by abstract interpretation," ACM Transactions on Embedded Computing Systems (TECS), vol. 4, no. 4, pp. 751-778, 2005.

[17] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in Tools for High Performance Computing 2009, Springer, Berlin, Heidelberg, 2010, pp. 157-173.

[18] S. S. Shende and A. D. Malony, "The TAU parallel performance system," The International Journal of High Performance Computing Applications, vol. 20, no. 2, pp. 287-311, 2006.

[19] V. M. Weaver and J. Dongarra, "Can hardware performance counters produce expected, deterministic results?," in Proc. 3rd Workshop on Functionality of Hardware Performance Monitoring, 2010.

[20] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," Concurrency and Computation: Practice and Experience, vol. 22, no. 6, pp. 685-701, 2010.

[21] J. Yiu, "The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors," Newnes, 2013.

[22] S. Huber, D. Prokesch, and P. Puschner, "Combined WCET analysis of bitcode and machine code using control-flow relation graphs," in Proc. 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), 2013, pp. 163-172.

[23] A. Marongiu, P. Burgio, and L. Benini, "Supporting OpenMP on a multi-cluster embedded MPSoC," Microprocessors and Microsystems, vol. 35, no. 8, pp. 668-682, 2011.

[24] J. Haase, M. Damm, D. Hauser, and K. Waldschmidt, "Reliability-aware power management of multi-core processors," in Proc. 22nd International Conference on Real-Time Networks and Systems (RTNS), 2014, pp. 193-202.

[25] T. Ball, P. Mataga, and M. Sagiv, "Edge profiling versus path profiling: The showdown," in Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1998, pp. 134-148.

[26] M. Gregg, "The Flame Graph," Communications of the ACM, vol. 59, no. 6, pp. 48-57, 2016.

[27] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in Proc. 4th International Conference on Computing Frontiers (CF), 2007, pp. 143-152.

[28] A. Jindal, S. Podder, S. Dhar, and M. Verma, "Performance modeling and prediction for embedded systems: A survey," ACM Transactions on Embedded Computing Systems (TECS), vol. 17, no. 3, pp. 1-30, 2018.

[29] D. Kästner, B. Kiss, J. Kaitovic, and S. Jacobs, "Integrating processor tracing into the Yocto Project," in Proc. 17th International Workshop on Worst-Case Execution Time Analysis (WCET), 2017, pp. 1-12.

[30] A. Lakhiar, H. Hao, and K. Wang, "Performance analysis and optimization of real-time operating systems for IoT: A survey," Future Generation Computer Systems, vol. 112, pp. 670-687, 2020.