**Journal of Electrical and Electronic Engineering**
2020; 8(2): 42-46
http://www.sciencepublishinggroup.com/j/jeee
doi: 10.11648/j.jeee.20200802.11
ISSN: 2329-1613 (Print); ISSN: 2329-1605 (Online)

# Embedded Software Optimization for Computation - Intensive Applications

**Amitkumar Mistry[1], Rahul Kher[2, *]**

[1]Object Video Labs, McLean, USA

[2]Department of Electronics & Communication Engg, G H Patel College of Engg & Tech, Vallabh Vidyanagar, India

## Email address:

mistry_amit@yahoo.com (A. Mistry), rahul2777@gmail.com (R. Kher)

[*]Corresponding author

## To cite this article:

Amitkumar Mistry, Rahul Kher. Embedded Software Optimization for Computation - Intensive Applications. *Journal of Electrical and Electronic Engineering*. Special Issue: *Soft Computing Methods for Electrical and Electronics Engineering Applications*. Vol. 8, No. 2, 2020, pp. 42-46. doi: 10.11648/j.jeee.20200802.11

**Received**: April 20, 2020; **Accepted**: May 9, 2020; **Published**: May 27, 2020

**Abstract:** Optimization metrics for compiled code are not always measured in execution clock cycles on the target architecture. Modern cellular telephone or wireless devices, which may download executables over a wireless network connection or backhaul infrastructure, it is often advantageous for the compiler to reduce the size of the compiled code that must be downloaded to the wireless device. By reducing the size of the code, savings are achieved in terms of bandwidth required for each wireless point of download. These are metrics correlated to the dynamic run-time behaviour of not only the compiled code on the target processor, but also the underlying memory system, caches, DRAM, and buses, etc. Despite new generation of embedded systems are getting innovative and computationally powerful with upcoming embedded processors, the market demands more computational-intensive embedded software to be developed on embedded systems. It is very essential to implement efficient embedded software to meet the market demand of embedded systems. These embedded systems are special-purpose computing systems and built to perform very specific embedded applications. And, these embedded applications mainly use three key resources of embedded systems: (1) CPU (2) Run-time memory (3) Persistent memory i.e. NAND/NOR flash memory. This paper summarizes several effective embedded software optimization techniques to optimize CPU usage, Run-time memory, and Persistent memory.

**Keywords:** System-on-Chip (SoC), CPU, Run-Time Memory, Persistent Memory, Optimization Techniques

## 1. Introduction

Mostly, the embedded systems do not have much freedom to choose most powerful processors to develop applications flexibly and they often have very limited resources are available. Therefore, the effective embedded software optimization techniques are so important to use the embedded systems' resources efficiently. These embedded software optimization techniques bring multiple advantages: (1) Use the embedded systems' resources efficiently (2) Allow to improvise existing functionalities (3) Add functionalities to upgrade the systems (4) Lower the power consumption of embedded systems. Including these techniques in design and implementation of embedded software from very beginning is important to meet the requirements and allow flexibilities to improvise and add more functionalities later. In this paper, we propose variety of optimization techniques to utilize CPU, Run-time memory, and Persistent memory.

With the advance of system level integration and system-on-chip, the high-tech industry is now moving toward multiple-core parallel embedded systems using hardware/software co-design approach. To design and optimize an embedded system and its software is technically hard because of the strict requirements of an embedded system in timing, code size, memory, low power, security, etc. while optimizing a parallel embedded system makes research even more challenging. The loops should be focused more because they are usually the most critical parts to be optimized in digital signal processing (DSP) or any computation-intensive applications. The basic idea of fully

parallelizing nested loops not only minimizes the code size overhead but the technique based on multidimensional retiming, any uniform nested loops can be transformed with minimal overhead such that all the computations in the new loop body can be executed simultaneously [1].

Image processing on a Digital Signal Processor (DSP) often requires image data to be stored in external memory, because the amount of fast on-chip memory is usually very limited. Processing images in external memory causes significant performance drawbacks. A double buffering method using Direct Memory Access (DMA), called Resource Optimized Slicing (ROS-DMA), may be used instead of a Level 2 (L2) data cache. The idea of ROS-DMA is to transfer image slices into small intermediate buffers of fast internal memory, where the processing can be completed utilizing the full processing power. Use of DMA enables the data transfers and the processing to be accomplished in parallel. The method has the advantage of a modular implementation, making it easy to re-use components for various image processing operations. The sequence of transfers is organized in such a way that use of processor resources is optimized to achieve the shortest possible execution time. ROS-DMA can yield substantially better performance compared to using L2 cache. With ROS-DMA it will be easier to obtain reliable and tight Worst-Case Execution Times (WCETs). Test runs achieved up to six times faster execution with ROS-DMA compared to using the L2 cache on a C6416 DSP from Texas Instruments [2].

In [3]-[16], authors have explained various aspects pf CPU performance optimization like energy reduction/ power optimization in high performance computing; loop transformation and loop alignment for memory access optimization; data partitioning issues in on-chip and off-chip memory; low power, multi-module, multi-port memory design; structure packing and code vectorization. This paper particularly discusses how to optimize the CPU, run-time memory and persistent memory in some detail.

# 2. Embedded Software Optimization

Applications running on Embedded systems mainly use three key resources (1) CPU (2) Run-time memory (3) Persistent memory i.e. NAND/NOR flash memory. Based on our recent work, we propose variety of optimization techniques to optimize each individual resource to obtain enough level of performance. In later sections, we categorically go through each resource optimization techniques.

### *2.1. CPU Optimization*

In embedded systems or any general computing systems, efficient code optimization can greatly contribute to CPU's performance. This paper outlines effective optimization techniques to implement very efficient embedded code and design approaches to run embedded applications faster on CPU and consume less CPU. These approaches are considered at different level during embedded application development i.e. design, development, and upgrade level.

#### *2.1.1. Compiler Options*

The use of compiler optimization for embedded software can itself take entire paper. To justify other optimization techniques, we recommend basic optimization levels, specific optimization options and architecture specific optimization options. We highly recommend reviewing your processor's compiler documentation for specific optimizations.

Most compilers of embedded processors provide built-in optimization levels to optimize the code size and execution time as shown in Table 1. With each -O level, different set of optimization options are automatically enabled by the compiler making it easy for the developer to avoid picking individual compiler options.

***Table 1.*** *Common Compiler optimization options.*

| Option | Optimization Level |
|---|---|
| -O0 | Most optimization disabled. Fast compile time. |
| -O1 | Reduces the code size and execution time keeping compile time balanced. |
| -O2 | Generates highly optimized code. Most commonly used optimization option. |
| -O3 | All supported optimizations of -O2 plus more aggressively. |
| -Os | Optimize the program space usage. |

Along with -O options, following compiler options or similar supported options of your compiler can be used to fine-tune very specific desired optimizations as shown in Table 2.

***Table 2.*** *Specific Compiler optimization options.*

| Option | Optimization behavior |
|---|---|
| -funroll-loops | Unroll loops whose number of iterations can be determined at compile time. |
| -funsafe-math-optimizations | Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards. |
| -ffast-math | Sets -fno-math-errno, -funsafe-math-optimizations, -ffinite-math-only, -fno-rounding-math, -fno-signaling-nans and -fcx-limited-range. It may yield faster code however it can result in incorrect output of program if it depends on exact implementation of IEEE or ISO rules. |

The architecture specific optimization options are aimed to generate optimized code for embedded processor of your embedded system. The architecture specific optimization options are summarized in the following Table 3.

***Table 3.*** *Architecture Compiler options.*

| Option | Optimization behavior |
|---|---|
| -mcpu | Enables code generation for a specific processor. Some compiler allows architecture with optional extensions. |
| -mtune | Performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike -march). |
| -march | Generates machine code for specific machine type. Allows option architecture extensions like simd, vfp etc.. |
| -mfpu | Specifies the target FPU architecture, that is the floating-point hardware available on the embedded processor. |

### *2.1.2. Execution Timing Optimization*

Optimizing for embedded system timing requires typically its own techniques and methods. This section describes several tips that embedded developers can follow when optimizing their embedded applications.

Firstly, we recommend using performance profiler tools that can provide key information of execution time of functions and their hit count i.e. number of calls to each function during the embedded application run. Both execution time and hit count provides great information to target what needs to be optimized to run the embedded application faster. We recommend checking for following things once most time-consuming and hit count functions are identified:

a) Restructuring loops: Function or multiple functions using large for loops using same data should be restructured in single function without changing program's output. It not only reduces the loop iterations, but the data localization helps using cached data effectively.

b) Loop tiling: Break loop into smaller inner loops to fit the inner loops data in cache, which helps reducing the cache miss drastically

c) Early exit: Transform functions or loop in a way that it can exit early instead running unnecessary code or iterations.

d) Aligned pointer array: Aligning pointer array ensures the data is going to be available in the ideal location in the memory for the processor to fetch and perform necessary operations on them.

e) Restricted pointer array: If there will be no other pointer pointing to the same memory address of an array pointer, you can declare that pointer as a restrict pointer. This will let the compiler know that it can change order of certain operations involving pointer array to make your code faster.

f) Data locality: Data locality is a very key method to speed up the function execution. It's similar to concepts mentioned earlier: 1) Restructuring loops 2) Loop tiling. If your function data stored and far apart from each other in the memory, the processor will have to reach out to different part of memory to fetch all data before performing any mathematical computations.

g) Laying out the data in memory intelligently ensures all data required to perform each math computation stored close to each other and in cache at the same time in the function, which makes data tiling very important to fit the data in the cache once to avoid cache misses.

h) Double buffering: Embedded systems with on-chip peripheral DMA comes very handy to get the data in and out when the data transfer takes longer than processing the data. This technique helps the data to be available when CPU is ready to process next batch of the data. In more complicated embedded systems, DMA channel should be set up with multiple buffers so that the processor can access the current buffer while the next is being filled. It may involve multiple simultaneous block transfers. For example, in addition to accessing the current block and collecting the next block, it may be necessary to send out the last processed block for future use.

i) Vectorization: Embedded processors supporting Single Instruction-Multiple Data (SIMD) could take use of this optimization technique. In Vectorization, special set of intrinsic functions take advantage of data parallelism. Data parallelism occurs in when the same operation needs to be applied to large amounts of data in a sequentially independent manner. Vectorization enables to pull most possible performance from Embedded systems.

j) Hand-written assembly: It's the most low-level optimization technique applied to most time-consuming functions. Embedded developer identifies the most time-consuming functions and implements these functions in processor specific machine code i.e. instructions set.

k) Faster algorithms: We recommend embedded developers to evaluate the algorithms used by their application and replace them with faster and more efficient algorithms i.e. search algorithm.

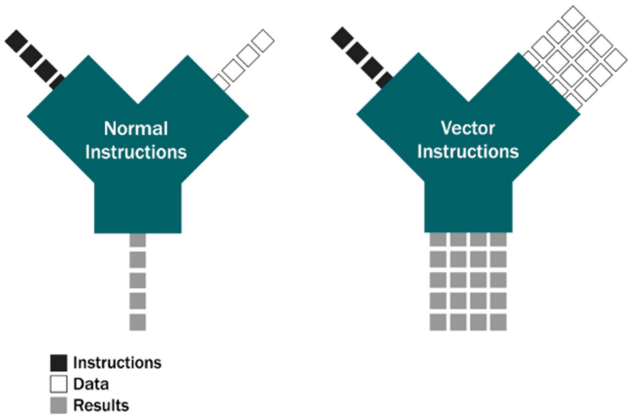


***Figure 1.*** *Vectorization.*

### *2.2. Optimizing Run-time Memory*

Most embedded systems constrained by limited memory resource forcing embedded developers to use the memory

thoughtfully. Run-time memory is a key bottleneck in embedded systems. Embedded application can fail to perform necessary functions if required memory is not available during its run-time. We recommend following tips to optimize the run-time memory usage:

a) Release memory: Releasing memory is very important if allocated memory is no longer needed. It creates more chances to succeed next allocation requests.
b) Static array allocation: Malloc calls with constant size arguments are replaced with a static array declaration to optimize the allocation and free process.
c) Advance memory estimation: In advance memory estimation technique, it assesses the memory requirements of a module or an application in advance by embedded developers. This tremendously optimizes with memory allocation and free process by pre-allocating memory for all modules. This technique helps reduce the heap fragmentation by less frequent allocations during run-time.
d) Scratch memory: Scratch memory concept is greatly useful to optimize the run-time memory. We recommend allocating one scratch memory pool for all intermediate temporary buffers so it reduces the memory allocations, avoids the memory fragmentation and the scratch pool can be reused throughout entire application thus achieving huge run-time memory optimization and reduce malloc/free calls.
e) Data type determination: Embedded developers must assess the data type required for their structures & buffers. In this technique, it assesses the range of values stored in buffers or structures. Once the range is determined, we can make informed decision to use the correct data type for buffers/structures. For example, the values stored in an integer type buffer stays in range of -10 to 120 can be converted to char type buffer. This would result in significant memory saving of 75%.

Structure reordering and packing: Embedded applications using thousands and hundreds of thousands C struct instances can be reduced run-time memory usage significantly. Most Embedded processors like ARM don't normally start at arbitrary byte addresses in memory. Rather, each data type except char has an *alignment requirement*. Thus, compilers will have to add padding if the data type is not aligned. One of the important criteria to pack structure effectively by lay out primitive data type in proper order so compiler's padding can be reduced. As shown in Table 4, *myStruct's float c* field will be aligned to 4-byte address so the compiler will 3 bytes padding after *char b* field. By reordering structure fields could save 8-12 bytes padded bytes for 32/64-bit processors.

***Table 4.*** *Reordering Structures.*

| Structure | Reordering Structure |
|---|---|
| struct myStruct { | struct myStruct { |
| int a; | long h; |
| char b; | int a; |
| float c; | float c; |
| char d; | int f; |
| char e; | char b; |
| int f; | char d; |
| char g; | char e; |
| long h; | char g; |
| } | } |

### *2.3. Optimizing Persistent Memory*

The persistent memory is used to store an embedded application program and persistent state of a program i.e. configuration. Optimizing embedded application's program size would reduce the persistent memory usage. We recommend following tips to optimize the persistent memory usage:

a) Log messages: Log messages used in printf/ count are embedded statically in the program footprint. Unknowingly, it could take up quite a-bit space of program memory. Shortening the messages or removing unnecessary messages could reduce the program footprint.
b) Shared library: On embedded systems, multiple embedded applications using same static library would compile and embed all machine code of static library in all applications' program memory. Converting static library to shared library would perform the linking process on the fly when embedded applications are executed. Thus, it's one copy of library occupies the persistent memory space even though multiple applications uses the same library. The -fPIC flag enables "position independent code" generation, which is a requirement for shared libraries. This allows the code to be located at any virtual address at runtime.
c) Compression: Compressing configuration data of an embedded application would reduce the persistent memory usage. In some cases, the application can be broken into a small program and shared object library. This shared object library can be compressed to reduce the program size. It would add an overhead of one-time decompression of shared object library if the free room in persistent memory is running very low.
d) Look-up tables: Sometimes embedded applications use large constants lookup tables resulting program footprint increase. Increase of program footprint would take up more persistent memory. Deriving lookup tables at run-time would eliminate such large lookup tables from program footprint thus reducing the persistent memory usage.

# 3. Conclusion

There are various aspects of real-time embedded systems' performance optimization like energy reduction/ power optimization in high performance computing; loop transformation and loop alignment for memory access optimization; data partitioning issues in on-chip and off-chip memory; low power, multi-module, multi-port memory design; structure packing and code vectorization. We have discussed about the optimization of three key resources- CPU, run-time memory and persistent memory in some detail in this paper.

# References


[1] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," ACM Transactions on Design Automation of Electronic Systems (TODAES) April 2001.

[2] Christian Zinner, Wilfried Kubinger, "A DMA Double Buffering Method for Embedded Image Processing with Resource Optimized Slicing", RTAS 2006.

[3] Bellas, N., Hajj, I. N., Polychronopoulos, C. D., and Stamoulis, G. 2000. Architectural and compiler techniques for energy reduction in high-performance microprocessors. IEEE Trans. Very Large Scale Integr. Syst. 8, 3 (June), 317–326.

[4] Danckaert, K., Catthoor, F., and Man, H. D. 2000. A preprocessing step for global loop transformations for data transfer and storage optimization. Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, San Jose, USA.

[5] Fraboulet, A., Huard, G., and Mignotte, A. 1999. Loop alignment for memory access optimisation. In Proceedings of the 12th ACM/IEEE International Symposium on System-Level Synthesis (San Jose CA, Dec.). ACM Press, New York, NY, 70–71.

[6] Kirovski, D., Lee, C., Potkonjak, M., and Mangione-Smith, W. 1999. Application-driven synthesis of memory-intensive systems-on-chip. IEEE Trans. Computer-Aided Des. 18, 9 (Sept.), 1316–1326.

[7] Masselos, K., Catthoor, F., Goutis, C. E., and Man, H. D. 1999. A performance-oriented use methodology of power optimizing code transformations for multimedia applications realized on programmable multimedia processors. In Proceedings of the IEEE Workshop on Signal Processing Systems (Taipeh, Taiwan). IEEE Computer Society Press, Los Alamitos, CA, 261–270.

[8] Panda, P. R., Dutt, N. D., and Nicolau, A. 2000. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. ACM Trans. Des. Autom. Electron. Syst. 5, 3 (July), 682–704.

[9] Shiue, W. and Chakrabarti, C. 1999. Memory exploration for low power, embedded systems. In Proceedings of the 36th ACM/IEEE Conference on Design Automation (New Orleans LA, June). ACM Press, New York, NY, 140–145.

[10] Shiue, W.-T., Tadas, S., and Chakrabarti, C. 2000. Low power multi-module, multi-port memory design for embedded systems. In Proceedings of the IEEE Workshop on Signal Processing Systems (Lafayette, LA, Oct.). IEEE Press, Piscataway, NJ, 529–538.

[11] Eric S. Raymon, "The Lost Art of Structure Packing," http://www.catb.org/esr/structure-packing/.

[12] Kevin Kredit, "Write Vectorized Code and Optimize Your CPU Performance", https:// dornerworks.com/ blog/ write-vectorized-code-and-optimize- your-cpu- performance/.

[13] Yemliha, Taylan, "Performance and Memory Space Optimizations for Embedded Systems" (2011). Electrical Engineering and Computer Science - Dissertations. 300. https://surface.syr.edu/eecs_etd/300

[14] Editor's note. https://www.embedded.com/ achieving- better-software-performance-through-memory-oriented-code-optimization-part-1/

[15] Ph. D Thesis. Design and Optimization of Architectures for Data Intensive Computing, available at http://users.ece.northwestern.edu/~jay/PhD_Dissertation.pdf

[16] E. H. M. Sha, "Parallel embedded systems: optimizations and challenges," IEEE Int. Conf. on Emerging Information Technology, Taipei, 2005, pp. 4-9.