

# Central processing unit load reduction through application code optimization and memory management

Sowmya Kandiga Bhadrappa, Vishwas Bangalore Ravishankar

Department of Electronics and Communications, R V College of Engineering, Bengaluru, India

## Article Info

### Article history:

Received Jun 10, 2024

Revised Sep 22, 2024

Accepted Sep 28, 2024

### Keywords:

CPU profiling

Data tightly coupled memory

Embedded trace microcell

Instruction tightly coupled memory

Tightly coupled memory

## ABSTRACT

Central processing unit (CPU) loading refers to the amount of processing power a CPU uses to execute a given set of commands or perform an exact task. Higher CPU load can lead to slower, sluggish performance, reduced lifespan, and reduced system stability. Using the CPU Load trace results, the performance bottlenecks can be identified and suitable methods can be adopted to reduce the load on the CPU. For an ideal embedded system, the CPU should be in idle state for around 70% of CPU usage time. In this paper, three types of optimization techniques are implemented, which include application code optimization, memory management, and implementing interrupt-driven data transfer. Application code can be optimized by getting rid of redundant code, duplicate functions and function inlining, function cloning which reduces the size of the code with increase in reusability. By moving the data, variables to data tightly coupled memory (DTCM) and instructions, functions to instruction tightly coupled memory (ITCM), the speed of the CPU increases which reduces the load on CPU. The conventional polling method which increases the CPU load can be reduced by implementing the same in interrupt-driven data transfer. The load on the CPU has reduced from 89.53% to 29.58%.

This is an open access article under the [CC BY-SA](#) license.



## Corresponding Author:

Sowmya Kandiga Bhadrappa

Department of Electronics and Communications, R V College of Engineering

Bengaluru, India

Email: sowmyakb@rvce.edu.in

## 1. INTRODUCTION

The central processing unit (CPU) is the major component of a computer system conscientious for executing instructions and doing calculations. The fetched instructions of CPU from memory, decodes it, and executes it to perform the tasks required by the software running on the system. The CPU is typically made up of two main components: the control unit (CU) and the arithmetic logic unit (ALU). The CU associates the flow of data between the CPU and other components, while the ALU performs mathematical and logical operations. Modern CPUs can have several cores, permitting them to invoke multiple instructions concurrently, and may include characteristics such as cache memory, virtualization support, and hardware acceleration for specific tasks. In embedded systems, the CPU is the primary component responsible for executing instructions and controlling the behavior of the system. Embedded CPUs are typically designed to meet specific requirements such as low power consumption, small size, and high performance. Embedded CPUs come in a variety of architectures and instruction sets. They are often integrated with other system components such as memory, input/output interfaces, and sensors to form a complete system. Embedded systems often have strict performance and power requirements, so optimizing the CPU's usage is critical to ensure the system runs efficiently.

Application code in embedded systems refers to the software program that runs on an embedded system, which is a computer system that is designed for a specific task or application. The application code is written for a feeder protective relay which specifies the functionality of the relay. A feeder protective relay is a type of protective relay used to protect electrical power systems against faults and other abnormal conditions in electrical feeders. Feeder protective relays monitor the electrical parameters of the feeder, such as voltage, current, frequency, and power, and detect any abnormal changes in these parameters. The primary function of a feeder protective relay is to detect faults in the feeder and isolate the faulty section to prevent further damage to the system. When a fault occurs, the protective relay quickly sends a signal to the breaker to trip, thereby disconnecting the faulty section from the rest of the system.

CPU profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Profiling is a technique that identifies sections of code that consume large proportions of the total execution time [1]-[3]. It is usually more productive to focus optimization efforts on code segments that are executed very frequently, or that take a significant proportion of total execution time than to optimize rarely used functions or code that takes only a small proportion of total execution time. CPU Profiler shows what functions consume what percent of CPU time [4]-[6]. This data provides better information on how the utilization is executed, and how exactly assets are allocated. Once the analysis is finished, the profiler visualizes the output data in the reports [7], [8]. The primary objective of CPU profiling is to identify performance congestion in an application's code that are causing the application to use more CPU resources than necessary [9]. CPU profiling can be used to achieve a variety of specific objectives, such as: i) identifying the functions or methods that are consuming the most CPU time, ii) detecting CPU-intensive loops or algorithms, iii) finding functions or methods that are called too frequently, iv) analysing CPU usage across different threads or processes, and v) comparing the performance of different versions of an application or different hardware platforms.

In embedded systems, CPU loading refers to the amount of processing being performed by the CPU at a given time, just like in any other computing system [10]. However, in embedded systems, CPU loading is critical because of the limited processing power and resources available. High CPU loading in embedded systems can cause performance issues, such as slower response times, system instability, and increased power consumption. In some cases, it can even lead to system failure or crashes. Managing CPU loading in embedded systems is essential to ensure optimal system performance and stability [11]-[13]. This can be done by designing the system to have a sufficient processing power to handle the required tasks [14], optimizing the software to minimize CPU usage [15], [16], and implementing real time operating system (RTOS) that can efficiently manage CPU resources. RTOS provides scheduling and prioritization mechanisms [17] that ensure that the most critical tasks are executed first and that the CPU resources are efficiently used. This helps to avoid overloading the CPU and ensures that the system can handle its intended workload. In addition to RTOS, other strategies for managing CPU loading in embedded systems include using hardware accelerators and offloading processing to other devices or systems, such as cloud-based servers or edge devices. These approaches can help to reduce the processing workload on the CPU and improve system performance and efficiency.

## 2. METHOD

The methodology is as shown in Figure 1. CPU loading procedure is performed to obtain the load on the CPU. The CPU loading procedure consists of many activities that need to be carried out simultaneously, which ensures maximum load is applied. The load results are analysed using a tool called Tracealyzer. The performance bottlenecks or tasks and functions which are utilising the processing power of the CPU is noted and suitable measures to reduce the load are taken.

The load on the CPU will be maximum when different tasks are being executed simultaneously. Before releasing the relay to the market, the load on the CPU should be monitored in order to ensure longer lifespan, reliability and better performance of the device. To ensure the device can perform as per expectations even in extreme working conditions, CPU loading procedure is carried out where all the protection functions, measurement functions and other tasks are being executed properly. In order to reduce the CPU load, memory should be utilized efficiently. Instead of storing all the instructions, data and variables in one single memory component, they can be allocated different memory segments such as instruction tightly coupled memory (ITCM), data tightly coupled memory (DTCM) and on chip random access memory (OCRAM). As tightly coupled memory (TCM) is placed near the core, the execution time is reduced as well as the latency, which reduces the load on the CPU. Application code can be optimized by removing the redundant code, unused code, duplicate functions and one-line functions. Some lines of code which are used

only for testing purpose are also removed. Some lines of code are used for simulation purpose, which are also retracted. This has no effect on the actual application functionality.

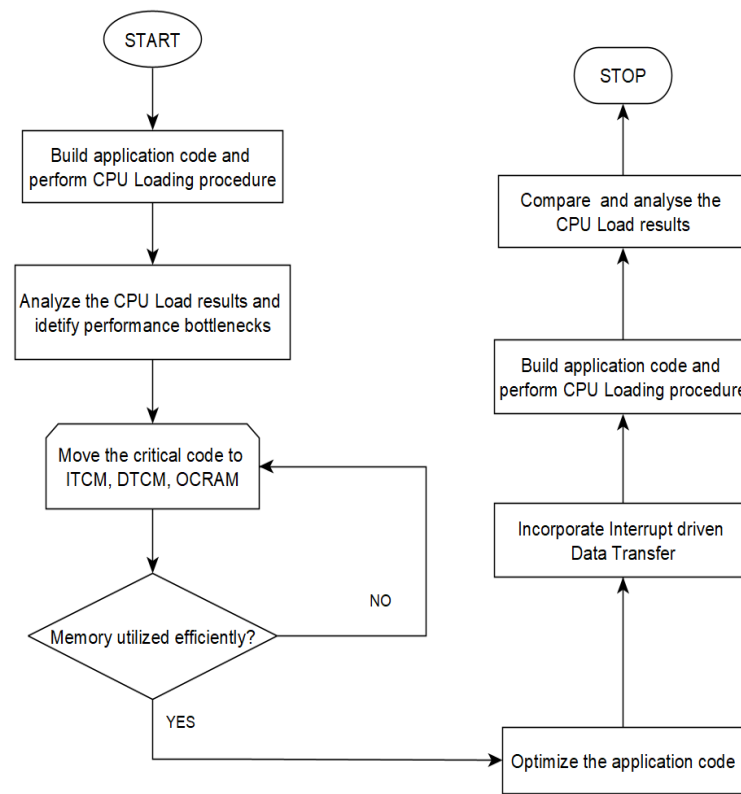


Figure 1. CPU loading methodology

This increases the readability, reusability and efficiency of the code. The size of the application code reduces which also reduces the load on the CPU. The final method which is used is implementation of Interrupt based data transfer, which plays a major role in CPU load reduction. The conventional polling method is replaced by the interrupt-based data transfer. A significant amount of CPU load can be reduced by incorporating this method. After the optimization measures are taken, the CPU loading procedure is performed once again and the results are compared with the results taken in the beginning. The CPU loading procedure can be performed after each optimization method to compare the results and also check if the method implemented has had any effect on the CPU load.

### 3. DESIGN AND IMPLEMENTATION

The application code is written for a feeder protective relay. The relay has a number of protection functions, measurement functions, and supervision function. It also has keypad, liquid crystal display (LCD) display, light emitting diode (LEDs) in the local human machine interface. In order to apply maximum load, all the tasks and functions of the relay should be active and during this period, the CPU loading procedure should be conducted.

#### 3.1. Central processing unit loading procedure

Application code in embedded systems refers to the software program that runs on an embedded system, which is a computer system that is designed for a specific task or application. The application code is written for a feeder protective relay which specifies the functionality of the relay. A feeder protective relay is a type of protective relay used to protect electrical power systems against faults and other abnormal conditions in electrical feeders. Feeder protective relays monitor the electrical parameters of the feeder, such as voltage, current, frequency, and power, and detect any abnormal changes in these parameters. The primary function of a feeder protective relay is to detect faults in the feeder and isolate the faulty section to prevent further damage to the system. When a fault occurs, the protective relay quickly sends a signal to the breaker

to trip, thereby disconnecting the faulty section from the rest of the system. This helps to prevent damage to the equipment and reduces the risk of injury or death to personnel. The CPU load is observed for a period of 5 seconds.

The relay comes with a number of tasks and functions. To ensure maximum load on the CPU, all these tasks and functions should be enabled at the same time. The various tasks and functions are discussed below:

- Local human machine interface (LHMI): the LHMI is used for setting, monitoring and controlling. The LHMI of the relay contains following elements: i) LED indicators/signals, ii) LCD display/demonstrate, and iii) navigation buttons. These navigation keys are used to navigate the LHMI menu, selecting characters and for configuration purposes. During the testing phase the navigation buttons/keys should be used randomly to ensure maximum load is exerted on the CPU, the LEDs should operate properly, the key press speed may be less than one second, ensure LCD is working properly and there is no delay.
- Supervision: the intelligent electronic device (IED) is provided with an extensive self-supervision system which continuously supervises the software and the hardware. It handles the runtime fault simulations and informs the user about a fault through the LHMI. At the time of testing, supervision should be running at all times.
- Fault record: the relay keeps track of analog points for the last 20 trip events in non-volatile memory. The trip signal triggers the fault recording of a protection function. Every fault record provides the root mean square (RMS) current values of basic components for all three phases and the neutral current at 20 different times along the trip event. During testing, for every fault, a trip event should occur which can be validated by reading the fault records using the front port.
- Events: these events include trip circuit supervision, protection start, protection trip, reset, breaker open, breaker close, remote trip, internal relay fault (IRF), blocking, and memory read fail. To store 100 such events, the relay incorporates a non-volatile memory. The event log includes the event along with time and date of occurrence. These event logs are stored sequentially, the most recent being the first and so on.
- Modbus RS485: during the testing phase, both the front port and the rear port should be used simultaneously, the polling rate should be set to 100 ms, i.e., the polling should occur at every 100 ms. The data, which is communicated in the form of packets, should not be lost in the process. Modbus RS485 can be used to read measurement data, read and write configuration, read, and write settings.
- Application function logic (AFL): the relay has around 27 AFLs including measurement functions. Some of these include protection functions such as undercurrent protection, overcurrent protection, thermal overload protection, phase discontinuity, inrush current detection, reclosing and measurement functions. During the testing phase, maximum number of these AFLs should be active. Current injection and communication to and from the relay should be conducted during the testing phase.
- Power failure: preconfigured functionality facilitates easy and fast commissioning of the relay. The relay has a universal power supply 24-265 V AC/DC. The relay has configurable binary inputs/outputs which can be configured using local HMI or communication interface. During the testing phase, power failure should be detected, during the event of power failure, no data should be lost.

### 3.2. Optimization techniques

Optimization of the load on the CPU is the main goal. There are many optimization techniques available. Techniques such as task scheduling, code optimization, and power management can be used to minimize CPU usage and extend battery life. Additionally, hardware acceleration and specialized coprocessors can be used to offload specific tasks from the CPU and improve overall system performance. The optimization techniques used are discussed below.

#### 3.2.1. Memory optimization

Memory optimization plays a critical role in reducing CPU load by reducing the amount of time the CPU spends accessing memory. When memory access is slow, the CPU must wait for the memory to provide data, which can result in wasted processing cycles and increased CPU load. One way to optimize memory access is by reducing the number of memory read and write operations. This can be achieved by optimizing algorithms and data structures to use memory more efficiently, reducing the number of times data is copied between different parts of the system. Another way to optimize memory access is by using cache memory. By using cache memory, the CPU can access frequently used data more quickly, reducing the number of memory read and write operations and improving system performance. Memory fragmentation can also increase CPU load by forcing the CPU to spend more time searching for available memory. To reduce memory fragmentation, developers can use memory allocation algorithms that reduce the number of small gaps in memory and improve memory utilization. The memory optimization implemented in this work

utilizes TCM, ITCM, and DTCM. By using ITCM, DTCM, and OCRAM, system performance can be improved [18], and CPU load can be reduced by reducing the number of memory accesses required. By partitioning the data and code by storing different components in different memory components, the load on the CPU can be reduced thereby increasing the speed.

TCM facilitates low-latency memory approach that the core can utilize with the predictability of access time which is a feature of caches [19]. While making use of external cacheable memory, an appealed instruction or piece of data might be in the cache, giving a fast access, or might not be in the cache, requiring a delayed access to external memory. When using TCM, the access time is accordant [20]. The TCM is used to take action on time-critical routines, such as interrupt handling routines or real-time tasks where the uncertainty of a cache is objectionable. Usually, TCM accesses are set up to collect or send data in a single cycle [21]. The processor can access time-critical procedures, such as exception handlers, immediately by storing them in the TCM instead of waiting for an initial code retrieve from external memory.

DTCM is a type of memory architecture used in microcontrollers and processors. DTCM is a fast, low-latency memory that is tightly integrated with the processor or microcontroller, allowing it to execute instructions and access data quickly and efficiently [22]–[25]. DTCM is typically used for storing frequently accessed data, such as variables and stack data, that are critical for the performance of the processor. DTCM is usually implemented as a small amount of on-chip memory that is physically located close to the processor or microcontroller.

ITCM is a type of memory that is closely integrated with a processor or microcontroller, allowing the processor to execute instructions at a higher speed [26] and with lower latency than if the instructions were stored in external memory. ITCM is typically used in embedded systems, where speed and performance are critical factors. By storing frequently used instructions in ITCM, the processor can quickly access them, reducing the overall execution time of the program.

### 3.2.2. Application code optimization

Application code can be optimized using a number of techniques which include elimination of redundant code [27], one-line functions, duplicate functions [28]. Function cloning and reduction of function call chain can also be implemented to reduce the load on the CPU. Very often, different functions may be declared which perform the same task. Identifying and eliminating such functions can reduce the CPU load [29], make the code reusable and reliable. Inline functions are those function whose definitions are small and be substituted at the place where its function call is happened [30]. Function substitution is totally compiler choice. Another technique used is dynamic memory management which is a process that allocates memory for variables and data structures at runtime when the program requests it [31]. This provides chance for adaptability and capacity, as the magnitude and position of memory blocks can be varied as per the logic of program and size of a data. It also enables the creation and manipulation of complex and dynamic data structures, such as linked lists, trees, graphs, and hash tables. Furthermore, dynamic memory allocation allows the program to adapt to different environments and user inputs, as the memory usage can be adjusted at runtime. This can save memory space and the readability and reusability of code increases.

### 3.2.3. Interrupt driven data transfer

An interrupt is something that alerts the CPU to take immediate action. To put it a different way, we can say that this device alerts the CPU to an issue that exists. The CPU typically suspends its current job and begins running the relevant interrupt handler when an interrupt occurs. When this task is finished, the previously halted task is resumed. The device notifies the CPU that it requires attention when there is an interrupt. It is not a protocol, but a hardware mechanism. The Interrupt handler makes the system get functioned. Functionality works any time. In case of an interrupt, if the device is in need of assistance, then that is indicated by the interrupt-request line.

The steps involved in the interrupt driven data transfer scheme are as follows:

- Transferring data efficiently utilizes the processor time.
- In this scheme, the processor starts off the I/O device for transfer of data.
- After the device is initiated, the processor will continue to execute the instructions in the program.
- At the nth step of an instruction, the processor checks for a right interrupt signal.
- If there is no interrupt signal, then the processor continues to execute the instructions.
- If the I/O device is ready, it interrupts the processor.
- The processor completes the execution of the current instruction and saves the processor status in the stack.
- The processor calls function named interrupt service routine.
- The final procedure of ISR is the processor status gets retrieved from the stack and the main program gets executed.

#### 4. RESULTS AND DISCUSSION

The optimization techniques, including memory management, application code optimization, and incorporating interrupt-driven data transfer, are used to conduct the CPU loading procedure. The results are shown in Figure 2, where the y-axis represents the number of kernel service calls made by the actor and service referencing any object. The kernel call intensity graph, post optimization as shown in Figure 2, displays the number of kernel service calls (y-axis) over time (x-axis). It allows for the identification of hot spots with a high volume of kernel service calls made.

Figure 3 shows the Scheduling intensity view which displays the amount of context switches (y-axis) over time (x-axis). By default, it shows the entire trace divided into 100 intervals. For each time interval, a bar is drawn for each actor beginning or resuming execution at least once in that interval. The height of the bars corresponds to the number of times that actor has begun or resumed execution in the given interval, i.e., the number of fragments of the actor. Time stamp of 5 seconds is provided on the x-axis of Figures 2 and 3.



Figure 2. Kernel call intensity graph post optimization

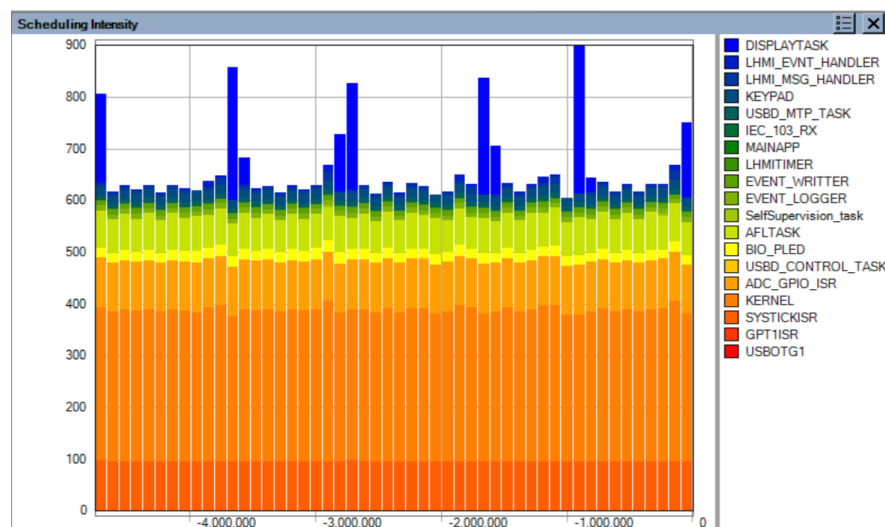


Figure 3. Scheduling intensity post optimization

Figure 4 shows the CPU load trace view after implementation of all the optimization techniques. The main component of this view is scheduling trace, showing fragments of actors displayed as color-coded rectangles. This is the trace view after implementing all the optimization techniques. It can be observed that display task is being executed at different time intervals. Self-supervision task is also consistent as it is one of the most important features of the relay.

The CPU load graph taken after implementing all the discussed optimization techniques is depicted in Figure 5. All the functions or tasks can be seen which can be differentiated using the legend provided on the right side of the window. Time stamp of 5 seconds is provided on the x-axis and the CPU load (y-axis) at particular instants. It can be observed for each instant of time, what tasks are executed and what percentage of load is exerted on the CPU. Display task is implemented by continuously pressing random keys on the relay's LHMI and the LCD should work accordingly. AFL task is executed at all instances as it is the main application.

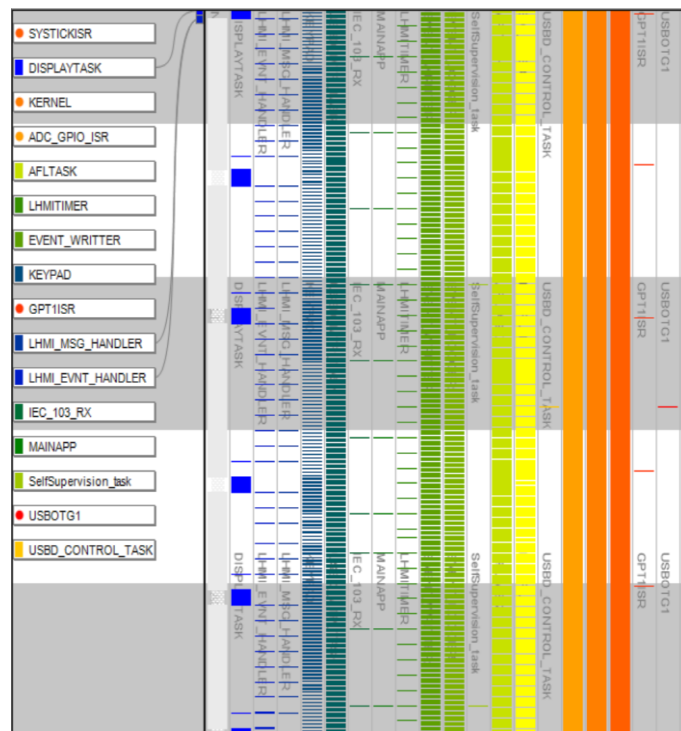


Figure 4. CPU load trace view post optimization

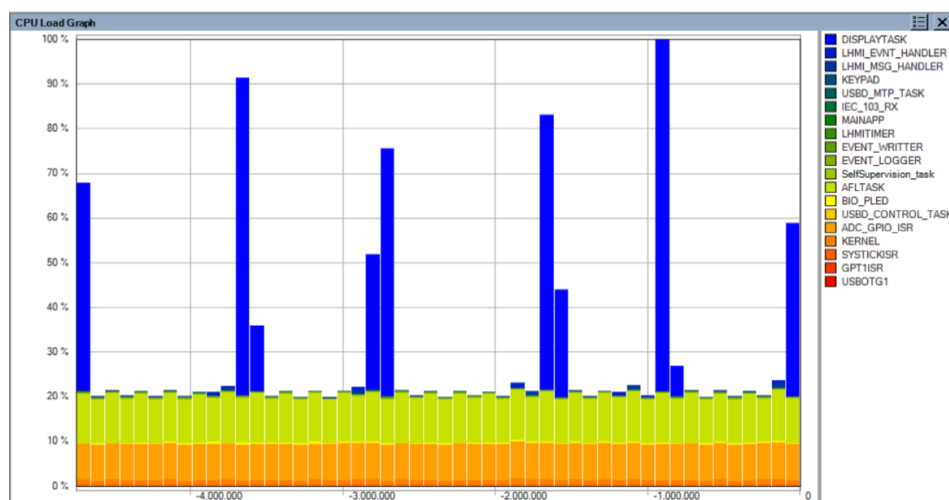


Figure 5. CPU load graph post optimization

It can be observed from the Figure 6 that the CPU usage % for NULLTASK is 70.418%. This shows that the total load on the CPU when all the tasks and functions were active is around 29%. This is an acceptable load on the CPU for an ideal embedded system. The CPU load after implementing each and every optimization technique is presented in the Table 1.

Actor	Priority		Count	CPU Usage	
	Min	Max		%	
Interrupt USBOTG1	1	1	1	0.001	
Interrupt GPT1ISR	2	2	5	0.001	
Interrupt SYSTICKISR	5	5	4752	0.274	
Interrupt KERNEL	15	15	14675	1.158	
Interrupt ADC_GPIO_ISR	15	15	4753	8.097	
Task USBD_CONTROL_TASK	1	1	1	0.001	
Task BIO_PLED	4	4	950	0.344	
Task AFLTASK	5	5	475	10.447	
Task SelfSupervision_task	7	7	2	0.003	
Task EVENT_LOGGER	10	10	475	0.191	
Task EVENT_WRITTER	11	11	475	0.089	
Task LHMIMER	12	12	48	0.004	
Task MAINAPP	30	30	10	0.001	
Task IEC_103_RX	30	30	10	0.001	
Task USBD_MTP_TASK	32	32	475	0.101	
Task KEYPAD	35	35	821	0.131	
Task LHMI_MSG_HANDLER	36	36	42	0.098	
Task LHMI_EVT_HANDLER	36	36	44	0.091	
Task DISPLAYTASK	36	36	11	8.551	
Task NULLTASK	127	127	1	70.418	

Figure 6. CPU lod statistic report after optimization

Table 1. Analysis of CPU load results

Method of optimization	CPU load (%)
Memory optimization	59.37
Application code optimization	42.37
Interrupt driven data transfer	29.58

The application code is written for a feeder protection relay which is commonly used in substations and in the manufacturing and process industry. The relay comprises of both hardware and software. The software consists of the application code which describes the functionality of the relay and the various components of the relay. The load on the CPU before optimization was found to be around 78.9%. Ideal load on the CPU has to be around 30%. Optimization techniques discussed previously are implemented and load is brought down to around 29.58%.

## 5. CONCLUSION

CPU load refers to the amount of processing power being used by CPU at a particular time. The application programmed to the digital board is made to carry out all the tasks and functions, thereby applying full load on the CPU. For an ideal embedded system, the idle time or null task should occupy about 70% of CPU usage. The load on the CPU before any optimization was found to be around 90%. Firstly, memory optimization which translates to efficient usage of memory is carried out. After the implementation of this step, the load on the CPU reduced to around 60%. The next step is the application code optimization which comprises of removal of duplicate functions, nil functions and any other redundant code. After optimizing the application code, the load on the CPU reduced to 43%. The load on the CPU is found to be around 29% after the implementation of interrupt driven data transfer. The CPU load has decreased from 90% to almost 30%. This ensures higher lifespan, reliability, and better system performance and lower power consumption.







## REFERENCES

- [1] R. Elnaggar, K. Basu, K. Chakrabarty, and R. Karri, "Runtime malware detection using embedded trace buffers," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 35–48, Jan. 2022, doi: 10.1109/TCAD.2021.3052856.
- [2] A. K. N. Ramachandra and A. K. Kannur, "Analysis of CPU utilisation and stack consumption of a multimedia embedded system," in *Proceedings - 4th IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, IEEE, Jan. 2008, pp. 89–94, doi: 10.1109/DELTA.2008.38.
- [3] K. Olukotun, T. Mudge, and R. Brown, "Performance optimization of pipelined primary caches," in *Proceedings of the Ninth Annual International Symposium on Computer Architecture*, IEEE, 1993, pp. 181–190, doi: 10.1145/146628.139726.
- [4] K. B. Bey, F. Benhammadi, A. Mokhtari, and Z. Guessoum, "CPU load prediction model for distributed computing," in *8th International Symposium on Parallel and Distributed Computing, ISPD 2009*, IEEE, Jun. 2009, pp. 39–45, doi: 10.1109/ISPD.2009.8.
- [5] S. Lee, S. Lee, and D. Shin, "Design and implementation of a configurable embedded trace macrocell for ARM Cortex-M3 processor," *IEEE Transactions on Consumer Electronics*, 2011.
- [6] C. Kim, H. Lee, and J. Kim, "Design and implementation of a high-performance embedded trace macrocell for a multi-core processor," *IEEE Transactions on Very Large-Scale Integration (VLSI) Systems*, 2017.
- [7] J. Lin, Y. Chen, and S. Wang, "A power-efficient embedded trace macrocell for ARM processors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2013.
- [8] Y. Li, Y. Liu, and H. Li, "Design of an efficient embedded trace macrocell for ARM processors," *IEEE Transactions on Very Large - Scale Integration (VLSI) Systems*, 2016.
- [9] K. B. Sowmya, S. Gomes, and V. R. Tadiparthi, "Design of UART module using ASMD technique," in *Proceedings of the 5th International Conference on Communication and Electronics Systems, ICCES 2020*, 2020, pp. 176–181, doi: 10.1109/ICCES48766.2020.09138098.
- [10] F. Tao, C. Sun, and J. Wu, "Energy-efficient CPU load balancing for virtualized cloud environments," *IEEE Transactions on Cloud Computing*, 2015.
- [11] L. Yang, I. Foster, and J. M. Schopf, "Homeostatic and tendency-based CPU load predictions," in *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2003*, 2003, p. 9, doi: 10.1109/IPDPS.2003.1213129.
- [12] J. Liang, K. Nahrstedt, and Y. Zhou, "Adaptive multi-resource prediction in distributed resource sharing environment," *2004 IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2004*, pp. 293–300, 2004, doi: 10.1109/ccgrid.2004.1336580.
- [13] Y. Zhang, W. Sun, and Y. Inoguchi, "CPU load predictions on the computational grid," *IEICE Transactions on Information and Systems*, vol. E90-D, no. 1, pp. 40–47, 2007, doi: 10.1093/ietisy/e90-1.1.40.
- [14] R. Kaur, N. Kumar, and V. Kumar, "Design of a high-speed on-chip embedded trace macrocell for ARM processors," *IEEE Transactions on Very Large - Scale Integration (VLSI) Systems*, 2015.
- [15] J. Liang, J. Cao, J. Wang, and Y. Xu, "Long-term CPU load prediction," in *Proceedings - IEEE 9th International Conference on Dependable, Autonomic and Secure Computing, DASC 2011*, IEEE, Dec. 2011, pp. 23–26, doi: 10.1109/DASC.2011.28.
- [16] J. Kim and T. Kim, "Memory access optimization through combined code scheduling, memory allocation, and array binding in embedded system design," in *Proceedings - Design Automation Conference*, pp. 105–110, 2005, doi: 10.1145/1065579.1065611.
- [17] H. Zhou, J. Song, and X. Pu, "The design of a novel modbus TCP/RTU gateway for high reliable communication," in *Proceedings - 24th IEEE International Conference on High Performance Computing and Communications, 8th IEEE International Conference on Data Science and Systems, 20th IEEE International Conference on Smart City and 8th IEEE International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application, HPCC/DSS/SmartCity/DependSys 2022*, IEEE, Dec. 2022, pp. 2039–2042, doi: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00303.
- [18] K. H. Salem, Y. Kieffer, and S. Mancini, "Efficient algorithms for memory management in embedded vision systems," in *2016 11th IEEE Symposium on Industrial Embedded Systems, SIES 2016 - Proceedings*, IEEE, May 2016, pp. 1–6, doi: 10.1109/SIES.2016.7509426.
- [19] T. Kim and J. Kim, "Integration of code scheduling, memory allocation, and array binding for memory-access optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 1, pp. 142–151, Jan. 2007, doi: 10.1109/TCAD.2006.882639.
- [20] R. S. Lin, Y. C. Wei, and R. X. Zhang, "Reduction of CPU computation load based on OpenCL for speech codec," *2018 IEEE International Conference on Consumer Electronics-Taiwan, ICCE-TW 2018*, pp. 1–2, 2018, doi: 10.1109/ICCE-China.2018.8448515.
- [21] I. Enesi, E. Zanaj, S. Kokonozi, and B. Zanaj, "Performance evaluation of statefull load balancing in predicted time intervals and CPU load," in *17th IEEE International Conference on Smart Technologies, EUROCON 2017 - Conference Proceedings*, IEEE, Jul. 2017, pp. 89–94, doi: 10.1109/EUROCON.2017.8011083.
- [22] M. Yang, H. Wang, and J. Zhao, "Research on load balancing algorithm based on the unused rate of the CPU and memory," in *Proceedings - 5th International Conference on Instrumentation and Measurement, Computer, Communication, and Control, IMCCC 2015*, IEEE, Sep. 2016, pp. 542–545, doi: 10.1109/IMCCC.2015.120.
- [23] N. H. Yatahiri and K. B. Sowmya, "Low power self-controlled pre-charge free content addressable memory," in *Proceedings of the 3rd International Conference on Electronics and Communication and Aerospace Technology, ICECA 2019*, IEEE, Jun. 2019, pp. 1225–1229, doi: 10.1109/ICECA.2019.8821852.
- [24] Z. Ning and F. Zhang, "Hardware-assisted transparent tracing and debugging on ARM," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1595–1609, Jun. 2019, doi: 10.1109/TIFS.2018.2883027.
- [25] G. Paroux, B. Toursel, R. Olejnik, and V. Felea, "A Java CPU calibration tool for load balancing in distributed applications," in *Proceedings - ISPD 2004: Third International Symposium on Parallel and Distributed Computing/HeteroPar '04: Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Hete*, IEEE, 2004, pp. 155–159, doi: 10.1109/ISPD.2004.2.
- [26] M. W. Bhat, R. Kaartik, and K. B. Sowmya, "Design and implementation of power efficient clock gated dual-port SRAM," *Journal of Physics: Conference Series*, vol. 2325, no. 1, Aug. 2022, doi: 10.1088/1742-6596/2325/1/012034.
- [27] A. Bouchi, R. Olejnik, and B. Toursel, "A new estimation method for distributed Java object activity," in *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2002*, 2002, p. 116, doi: 10.1109/IPDPS.2002.1016500.
- [28] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating android applications' CPU energy usage via bytecode profiling," in *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*, 2012, pp. 1–7, doi: 10.1109/GREENS.2012.6224263.





- [29] K. B. Sowmya and A. Thejaswini, "Systematising troubleshooting of disputes in network," *International Journal of Reconfigurable and Embedded Systems*, vol. 10, no. 1, pp. 32–36, Mar. 2021, doi: 10.11591/ijres.v10.i1.pp32-36.
- [30] J. G. Tong, "Software profiling for an FPGA-based CPU core," University of Windsor, 2007.
- [31] I. Baldini, S. J. Fink, and E. Altman, "Predicting GPU performance from CPU runs using machine learning," in *Proceedings - Symposium on Computer Architecture and High Performance Computing*, 2014, pp. 254–261, doi: 10.1109/SBAC-PAD.2014.30.

## BIOGRAPHIES OF AUTHORS



**Dr. Sowmya Kandiga Bhadrayya**     received the B.E. and M. Tech. degree in electronics and communication engineering from Visvesvaraya Technological University, Belagavi, India in 2006 and 2012, respectively, and the Ph.D. degree in VLSI design and signal processing from Visvesvaraya Technological University, Belagavi, India in 2021. She is currently a full time Assistant Professor of electronics and communication engineering with the R V College of Engineering, Bengaluru, India. In the past, she was associated with PA College of Engineering, Mangalore, India. She has been working on VLSI, HDL, system Verilog, physical design, and system on chip, since 2006. She has contributed many national and international journals, book chapters to various reputed journals and conference. She can be contacted at email: sowmyakb@rvce.edu.in.



**Vishwas Bangalore Ravishankar**     received his bachelor's degree in electrical and electronics engineering from University Visvesvaraya College of Engineering. He is pursuing his masters in VLSI design and embedded systems from R V College of Engineering. He is currently working as an R&D Intern in ABB Innovation Center, Bengaluru. He can be contacted at email: vishwasbr17@gmail.com.