DOI 10.15622/ia.21.2.2

P. BUI, M. LE, B. HOANG, N. NGOC, H. PHAM

DATA PARTITIONING AND ASYNCHRONOUS PROCESSING TO IMPROVE THE EMBEDDED SOFTWARE PERFORMANCE ON MULTICORE PROCESSORS

Bui P., Le M, Hoang B., Ngoc N., Pham H. **Data Partitioning and Asynchronous Processing to Improve the Embedded Software Performance on Multicore Processors.**

Abstract. Nowadays, ensuring information security is extremely inevitable and urgent. We are also witnessing the strong development of embedded systems, IoT. As a result, research to ensure information security for embedded software is being focused. However, studies on optimizing embedded software on multi-core processors to ensure information security and increase the performance of embedded software have not received much attention. The paper proposes and develops the embedded software performance improvement method on multi-core processors based on data partitioning and asynchronous processing. Data are used globally to be retrieved by any threads. The data are divided into different partitions, and the program is also installed according to the multi-threaded model. Each thread handles a partition of the divided data. The size of each data portion is proportional to the processing speed and the cache size of the core in the multi-core processor. Threads run in parallel and do not need synchronization, but it is necessary to share a general global variable to check the executing status of the system. Our research on embedded software is based on data security, so we have tested and assessed the method with several block ciphers like AES, DES, etc., on Raspberry PI3. The average performance improvement rate achieved was 59.09%.

Keywords: embedded software performance improvement, multicore processors, multithread, data partitioning, asynchronous processing.

1. Introduction. Data encryption has been studied with many algorithms such as AES, DES, etc., to ensure data security and is focused on research for sequential processing software systems. It hasn't been being studied on embedded systems. Because encrypting data will increase the cost of data processing time compared to unencrypted data, the problem is to both ensure data security and maximize performance by adapting encryption algorithm based on embedded software configuration.

Today, in the fast development trend of the 4th industrial revolution, information technology systems, especially IoT systems, also flourish in both hardware and programming models. In this trend, multicore processors are widely studied and applied. Not only information technology systems in general, but embedded systems also use more and more multicore processors.

The purpose of using multicore processors is to improve the performance of embedded systems under limited resource conditions. There are many studies that have been done to improve the performance of embedded software. Improvements to embedded software performance can be made on different levels: from processor level to operating system, programming and design levels. However, most of the old programming models, which have only been programmed in serial/sequential processing, have not yet promoted the parallel processing.

In recent years, there have also been several teams that have performed researches and developed methods to improve embedded software performance on multicore processors. These studies mainly focus on multithread models – the distribution of processing flows for parallel implementation as the optimal approach based on co-design is also studied and applied to multicore processors, such as [1]. In the study, the author proposed and developed a method of co-designing hardware and software for network systems on a chip (NoC), which improved the average performance by 33.1%. However, due to a data conflict between the local data set of the failed transaction and the global end-level caching (LLC), the transaction cancellation rate is still high in software transactional memory, and it is costly to identify and update the cancelled data.

In study [2], the authors built a user space memory scheduler that allocates the ideal memory node for tasks by monitoring the characteristics of the heterogeneous memory architecture to optimize application performance for the NUMA multicore processors. The average performance improvement rate achieved by this method is 25%. The authors focused on the development of the scheduler but used the Princeton Application Repository for Shared-Memory Computers (PARSEC); therefore, it is not yet possible to evaluate the compatibility of a scheduler with the ability to parallel the data on the NUMA configuration.

Study [3] presented a multi-threaded approach to the travelling salesman problem to improve performance. Under certain hardware limitations, the proposed math can take full advantage of multi-core chips and effectively balance out the contradiction between increasing data size and computing efficiency, thus gaining a satisfactory solution. Nevertheless, a parallel threaded approach that depends on the set of input parameters of the problem is not new; instead, it just gives case studies on thread programming to solve the problem.

Study [4] presented a platform used in multithreaded programming to improve performance as OpenMP. The platform can be used to develop both desktop applications and embedded apps. This study analyses the effects of different schedules and segment sizes on the at-gain speed of multi-core platforms that use different shared memory in regular workloads. The results illustrated that different multi-core technology showed different acceleration values, and different multi-core platforms were better than others in terms of speed as the number of cores was increased.

In study [5], the authors presented the method of sharing the data to be processed to points to improve the thoughts of points in the problem.

This study demonstrates that each specific problem has its own method of implementing performance improvement.

Studies presented on hardware configuration-based performance improvement methods are in [6]. The authors proposed a single-command, multi-data model to improve performance for multi-disciplinary chips.

Most of these studies focus on solving only one layer of specific problems and have not fully solved the problem of synchronization, linking data between threads. At the same time, the studies have not looked at data independence and data partitioning for parallel processing.

With big data processing problems and independent data sections, a more efficient model is needed to promote data independence and simplify synchronized time, linking data between threads. Therefore, in this study, we propose a model of data partitioning and asynchronous processing to process data in parallel to improve embedded software performance.

In study [7], the authors proposed a parallel multithreaded pipeline to filter, clean and classify information in the information collection phase. We developed the pipeline so that it can be easily re-applied to any type of heterogeneous aggregation and run efficiently on medium to low resource infrastructures where I/O speed is the main limitation.

In consideration of undesirable elements in synchronous programming, asynchronous programming has emerged as a programming style to overcome the limitations. Usually, in asynchronous programming models, the methods are included in the queue list for implementation, and the order in which the method is implemented is serial yet unidentified. Nevertheless, a lag still exists with this method due to the need for serial / sequential execution; therefore, there have been researches being carried out to solve the parallel asynchronous problems. The idea of asynchronous programming is to divide the execution of the original program into tasks that are running over a short period of time. Furthermore, each task is performed as recursive sequence software that defines new methods to be enforced later when accessing shared memory.

The simplicity of the asynchronous program’s combination makes asynchronous programming a preferred option for deploying API or systems that require reactive systems. The use of asynchronous programming for main servers, desktop applications, and embedded systems is increasingly being used and varied with problems [8], such as JavaScript tools of modern web browsers, Grand Central Dispatch in MacOS and iOS, Linux job queues, asynchrony in .NET, and deferred procedure calls in Windows core are all based on asynchronous programming. Even in single processing settings (i.e. without any parallelism), asynchronized frameworks such as Node.js are becoming widely used to design extremely scalable (web) servers.

Many studies have been carried out to present asynchronous programming models. Nevertheless, some tasks posted in asynchronous programs may have different levels of execution priority, as addressed in study [8]. A major drawback of existing tasks considering execution priorities is the complexity of models that host those priorities. An asynchronous program divides the behaviour of the accumulated program into short-running tasks. Each task basically acts as a recursive sedated program, in addition to accessing the memory shared by all tasks that can post new tasks to the task buffer for later execution. Tasks from each buffer perform in succession: when one task is completed, another is taken from the buffer and run until completed. Programming a reactive system requires the designer simply to ensure that no single task is performed for too long to prevent others more urgent tasks from being executed. Figure 1 illustrates the general architecture of asynchronous programs.

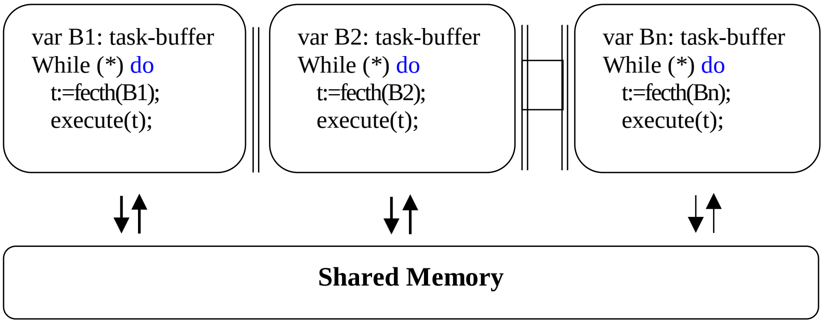

Fig. 1. A general architecture of asynchronous programs with N task buffers. Tasks from the same buffer execute serially, but concurrently with tasks of other buffers [8]

In study [9], the authors and his colleagues proposed an official model of asynchronous event-oriented programs, which narrows the semantic gap between programs and existing models, especially by allowing dynamic creation of tasks, events, buffer tasks and strings simultaneously, and accurately capture the interaction between these quantities. Instead of assigning each calculated task to a blocked dedicated thread for certain conditions, the system maintains simple sets of events that pend tasks, buffers of tasks with enabled events and workflows for performing tasks in the buffer.

Figure 2 presents asynchronous event-driven programs. The pending tasks (drawn as triangles) are moved to their designated task buffers (drawn as boxes) once their designated events (drawn as circles) are triggered. Threads (drawn as diamonds) execute buffered tasks to completion, such that no two tasks from the same buffer (drawn with the same colour) execute in parallel. In addition to the ability to use events to synchronize between tasks, the task cache itself provides another means of coordination: the system can allow tasks from separate buffers to perform in parallel while ensuring that tasks from the same buffer perform sequentially. Authoring the management of events, tasks, buffers, and flows to the system often increases the efficiency of the system.

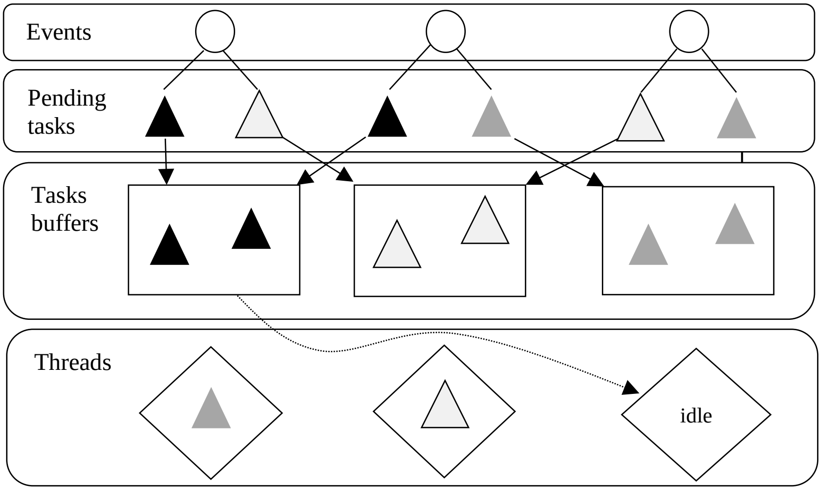

Fig. 2. Asynchronous event-driven programs [9]

Most of the studies only focus on solving a specific class of problems and have not completely solved the problem of synchronization and data linkage between threads. At the same time, studies have not considered the issue of data independence and data division for parallel processing.

With problems related to a big amount of processing data and/or independent pieces of data, a more efficient model is needed to promote the independence of the data and minimize the time for synchronization and linked data between the streams. Therefore, in this study, we propose a model of data division and asynchronous processing for parallel data processing to improve embedded software performance.

The rest of the paper is organized as follows: Section II – Survey, analysis, synthesis of related research; Section III – Presentation on ideas, process and content of method's development; Section IV – Experiment for method's testing and evaluation; Section V – Conclusion and trends of development.

2. Related work. The problem of optimizing performance based on parallel calculations has been researched and developed by many research groups in different approaches such as CPU hardware level, co-design level, operating system level and application level multi-threading. The CPU level parallelization includes typical optimization techniques such as order scheduling, command pipeline, in-of-order, out-of-order, CPU configuration, setting configuration, etc. [10]. These algorithms and methods are the basis for the direction of research and implementation approach of study [10].

Study [11] analysed cipher-related algorithms and then designed and deployed the AES algorithm in CUDA. The data will be divided into 64K blocks by the CPU and transferred to the cores of the GPU and performed in parallel the sub Bytes, Shift row, Mix column and AddroundKey functions for encryption. Encrypted data were collected on the GPU and transferred upwards to the CPU. This technique improved the performance of the AES encryption process with the help of a GPU. However, the cost was involved in transporting data from the CPU to the GPU and vice versa.

Data partitioning as an important sub-factor in database tools has been studied in the previous work. Polychroniou, O. and Ross, K.A. [12] provided extensive analysis of data partitioning across multiple methods, such as partition type (base, hash or range) and shuffle strategy. It has been proved that for more than 16 partitions, partitions that write in combination with direct memory writing and skipping cache memory would work best. Partition throughputs are reported to be 1.1 billion data sets/s for 8,192 partitions with 64 threaded parallel executions on 32-core servers.

Schuhknecht F.M. et al. in [13] presented a set of experiments performing base-based partitioning. Existing optimizations (combinations, time-consuming storage, etc.) are enabled, and new optimizations (pre-fetch for recording, micro-row layout) are added step by step to observe their impact on the total duration of implementation. The fastest FPGA data partition deployment to date was presented by Wang and his associates [14] with 256 Million Data Sets/s for 8,192 partitions. They improved the existing OpenCL implementation of a partitioned and deployed it on the FPGA. The partitioned data are assumed to be in the DRAM that is directly connected to the FPGA. The resulting partitions are recorded to the same DRAM, and the transfer via PCIe to the server memory is necessary if the partitioned data will be used by the CPU for further operations.

Study [15] shows that the probability of parallel calculation of data depends heavily on its data partitions. The solutions implemented by the status quo of the systems have not yet been optimized. Community-recommended techniques for finding optimal data partitions are not applied directly when relevant to complex user-defined data functions and models. A parallel data program compiled into an execution plan chart (Execution Plan Graph -EPG) is a directional AC chart with multiple stages. Therefore, data partitioning affects many aspects of how a job is run in a cluster, including parallelism, workloads for each vertex, and network traffic between vertices. In this study, the authors proposed the system architecture as Figure 3 to partition the data.

Figure 3 shows the architecture of the system. First, the system compiles a certain data parallel program into a work plan chart (EPG) with the original data partitions (e.g., provided by the user). The code analysis module takes this EPG and the code for each vertex in the EPG as input to infer the calculated complexity of the program per-vertex and important data features. This step is important because it not only provides information about the relationship between the input data size versus the cost of calculation and I/O but also guides the process of data analysis, for example, providing suggestions for strategically sampling data and estimating data statistics. This information can then be used to identify image recordings to process and distribute them more evenly. The data Analysis module linearly scans data to create compact data expressions.

The Modelling/Estimation Module uses code and data analysis results to estimate the runtime cost of each vertex, including CPU time, output data size, and network traffic.

The authors in study [15] gave the example of creating a partitioned graph, as shown in Figure 4. The two root nodes represent two partitions of the sampled input data. The cost optimization module inserts an additional partitioning stage into the existing EPG in search of an optimized partitioning scheme. First, the two inputs are divided into 8 partitions (for example, h1(k) mod 8 hash partition) and EPG is updated accordingly. The Cost Estimation module then determines the critical path up to the current period in the updated EPG, including the vertex associated with Partition 5. To reduce costs, it divides Partition 5 into the other two partitions (for example, the h2(k) mod 2 hash partition). Meanwhile, Partitions 0, 2 and 4 all have small costs to reduce I/O, the cost of starting peaks and therefore the underlying overall cost. The process of estimating costs and optimizing by ingesting and separating this recursive data is repeated until it converges. Each iteration is a greedy step towards minimizing overall costs. In summary, study [15] has proposed a system architecture to find the optimal for data partitioning based on greedy algorithms.

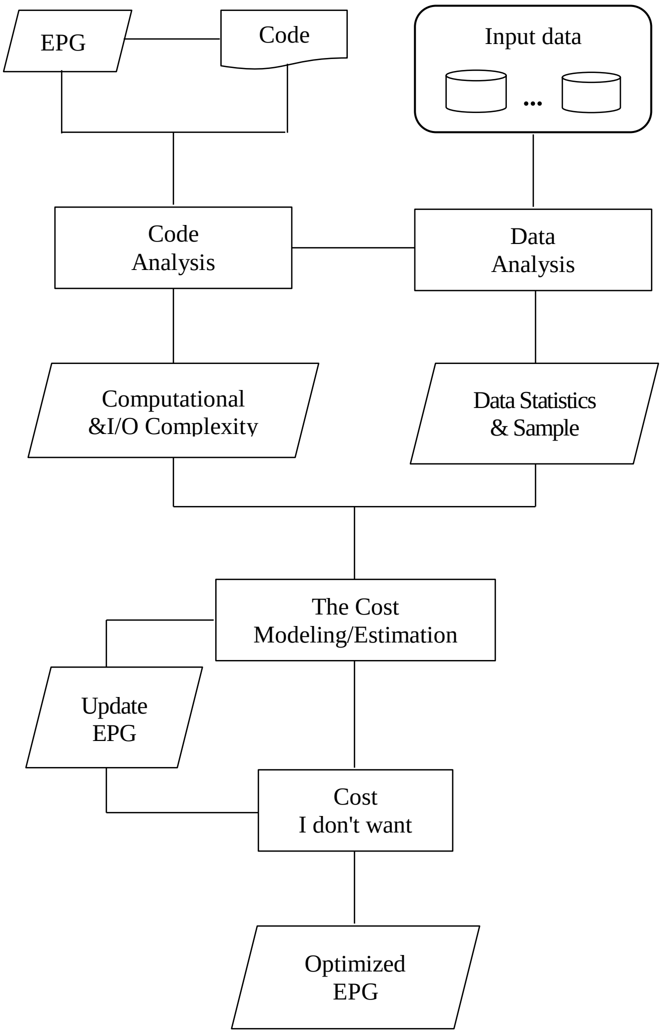

Fig. 3. System architecture, from study [15]

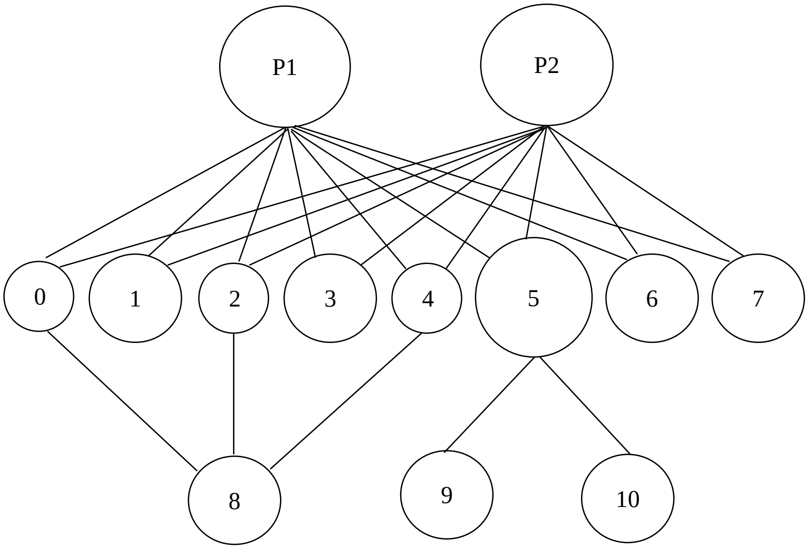

Fig. 4. Example of generating an optimized data partitioning scheme represented by the partition graph

The article [16] proposes a data set manipulation method to divide the training data set into many groups with different characteristics to train each classifier. The author partitioned the training data into groups of the same distance. If each individual SVM is trained using each of these groups, the trained classifiers have increased diversity. The test results showed an average accuracy of 70% by repeating the data set for a single letter four times.

In study [17], the authors et al. proposed an automatic local data partitioning algorithm, which can automatically recognize the local maxima of the data density from experimental observations and use them to serve as the focal point to form data partitions. This method is free of user's parameters and prior assumptions. Numerical results based on the benchmark dataset proved the validity of the proposed algorithm and demonstrated its high performance and computational efficiency compared to modern clustering algorithms.

Kara K. et al. [18] explored the use of FPGA to speed up data partitioning. The paper studies new hybrid architectures in which FPGA is placed as a co-processor located on one socket and has consistent access to the same memory as the CPU located on the other socket. Such an architecture reduces the cost of data transfer between the CPU and the

FPGA, allowing the execution of the combined algorithm in which the partition occurs on the FPGA and the construction and exploration stages of a connection that occurs on the CPU. The FPGA-only partitioning and execution reduce the cost of data transfer to the CPU and vice versa, but storing and performing on big data are matters for further study of the paper.

Study [19] considered the issue of optimal distribution of workloads for the parallel implementation of data between the processing components of non-click calculation systems. Zhong Z. et al. presented a solution that uses functional performance models (FPM) of processing agents and FPM-based data partitioning algorithms. The effectiveness of this method is proven by experiments with parallel matrix multiplication. The FPM-based data partitioning has proven to be fully and efficiently used to balance the workloads of many data parallel applications across modern hem helmless disk forms. However, more effort is still needed to improve this approach in some respects. For example, only the calculated performance of processing units is used to partition data so far, while the cost of communicating between processes is not considered. In some dense matrix applications on highly haemolysed disk forms, the performance of a processing unit may depend on the shape of the matrix block assigned to it. In that case, the multi-dimensional performance model has more than one parameter that may be required to describe the performance accurately. For large complex applications where the computational kernel cannot be easily separated from the application or there is more than one computational kernel, it may take more effort to balance the workload with this approach.

In study [20], the authors compared and evaluated three algorithms AES, DES, and 3DES by nine factors: key length, passcode type, block size, developed, code break resistance, security, ability key, ASCII printable character key, and the time it takes to check all possible keys at 50 billion seconds. The results have proven that AES algorithm~~s~~ are better than DES and 3DES, as shown in Table 1. However, this study did not evaluate the performance of data encryption processing of these three methods.

Study [21] compared the results of the AES encoding techniques to increase encryption efficiency on FPGA integrated circuits. Five techniques were outlined in the article.

The CB-KB-S technique: Both encryption and key creation are in the RAM Block. Processing is carried out in a singly way, first creating an extended key then the encryption process. The speed is slow due to the implementation of the order; however, it gives accurate results, therefore this technique is applied to problems that require high accuracy.

Table 1. Raspberry PI 3 embedded computer configuration

Factors	**AES**	**3DES**	**DES**
Key Length	128, 192, Or 256 bits	(kl, k2 and k3) 168bits (kl and k2 is same) 112 bits	56 bits
Cipher Type	Symmetric block cipher	Symmetric block cipher	Symmetric block cipher
Block Size	128, 192, or 256 bits	64 bits	64 bits
Developed	2000	1978	1977
Cryptanalysis resistance	Strong against differential, truncated differential, linear, interpolation and square attacks	Vulnerable to differential, Brute force attacker could be analyzed plaint text using differential cryptanalysis.	Vulnerable to differential and linear cryptanalysis; weak substitution tables
Security	Considered secure	One only weak which is Exit in DES.	Proven inadequate
Possible Keys	2128, 2192, or 2256	2112 or 2168	256
Possible ASCII printable character key	9516, 9524, or 9532	9514 or 9521	957
Time required to check all possible keys at 50 billion keys per second**	For a 128-bit key: 5x 1021years	For a 112-bit key: 800Days	for a 56-bit key: 400Days

The CB-KB-P technique: Both encryption and key creation are in the RAM Block but are used in a private key writing module. This module will run in parallel with the encryption process instead of performing a sequential key creation according to the CB-KB-S algorithm. With this technique, the application of parallel processing of encryption blocks increases performance due to reduced latency during the sequential process. Still, it reduces security due to the extended key born before the encryption process.

The CB-KC-S technique: With this technique, the encryption process is carried out at Block RAM, and the extension key is done in Logic Block. These two modules are done in a sequence of the extended key that is performed and then encrypted. This technique will reduce the processing in a way that conducts two tasks in two different places. On the other hand, data exchange between the parties also causes certain latency in encryption.

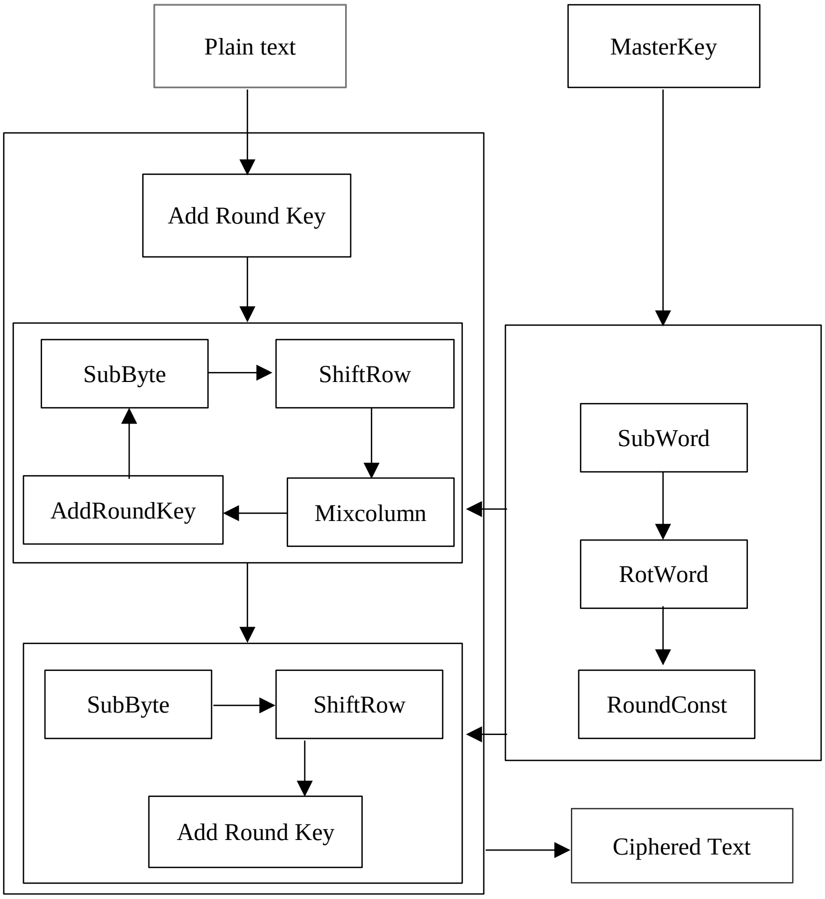

Fig. 5. Standard overview of AES algorithm

The CB-KC-P technique: like the CB-KC-S technique, the encryption process in this technique will be carried out at block RAM, and the extension key is done in Logic Block. The difference with the CB-KC-S technique is the fact that this technique conducts parallel unlocking and encryption. Since this process takes place in parallel, the performance of the technique is considered very good, but the safely of this technique, as well as the CB-KB-P technique, is not high because there is no constraint on the encoding result with extended key birth.

The CC-KC-S technique: this technique brings both encryption and extended key being executed in Logic Block. Because of the same implementation in Logic Block, there is no longer lag due to data exchange between Block RAM and Logic Block, so the efficiency of this technique achieved is the highest.

Most of the applications are written using a synchronous batch programming model. Therefore, they are not optimal for low-latency or asynchronous communication algorithms. The study [22] proposes constructs for asynchronous multi-GPU programming and describes their implementation in a thin runtime environment, performing common math operations and distributed task lists. The authors et al. demonstrated that this approach achieves performance gains and exhibits strong scalability for heterogeneous systems, yielding more than 7x speedup for some algorithms.

In study [23], asynchronous programming was built based on a multithreaded basis due to the fact that APIs have been deployed using multithreaded programs with shared memory. The execution of software flows is not affected by the number of processors in the system, which has been proven by the fact that threads are enforced as recursive sedation software that runs simultaneously with alternating write and read commands. It is likely that being able to walk away, in this case, might cause the complexity of simultaneous programming models. This study gave the authors the idea of using shared variables for this paper's asymable parallel programming solution.

The reduction is then as follows. First, the pre-emptive priority scheduler is made explicit by adding to the program the code shown in Figure 6. The hyper period procedure executes each thread one time, choosing non-deterministically a sleeping thread to execute via the choose operation, which returns an index that satisfies the supplied guard. An infinite cycle of hyper periods is simulated by invoking hyper period in a non-terminating loop. During each hyper period, the scheduler has two tasks: (*i*) it must ensure that each thread *i* is awoken so that *i* can execute its task, and (*ii*) the wake-ups should happen non-deterministically. The first task is handled by defining a Boolean array of size *n*, where each entry in the array denotes whether a thread *t* is sleeping or not. (In Figure 6, the array is named Sleeping.) The scheduler loops until all threads have been awoken and completed their periodic task.

The second task is handled by performing a source-to-source transformation on the code of each thread so that it non-deterministically invokes Schedule before each statement *st*. That is if a thread is comprised of program statements *st1*,. . . , *stk*, then the transformed program will have program statements *st0* 1 ,. . . , *st0 k*, where each *st0* is defined as: *st0,Schedule();st*. In the definition of *Schedule*, the function *nondet* non-deterministically returns *true* or *false*. When *Schedule* is invoked, the code of a higher-priority thread *ti* 0 than the thread *t*i whose code is currently executing may be invoked, which corresponds to *ti* being pre-empted by *ti* 0. Before executing a thread *ti* by invoking *Threads[i].entry()*, the flag *Sleeping[i]* is set to false to ensure that *ti* is executed exactly once per hyper period *H*.

```
// Sleeping flags
   Sleeping[n] = {true,...,true};
// Thread priorities
   Priorities[n] = ...;
// Thread entry points
   Threads[n] = ...;
// 0 => choose any thread
   Prio = 0;

void Hyperperiod() {
   while (Vi Sleeping[i]) {
            j = choose j: Sleeping[j];
            Sleeping[j] = false;
                        Threads[j].entry();
   }
}
```

```
// Wake-up higher-priority thread
void Schedule() {
// Save current priority
   int prevPrio = Prio;
   for i in (1..n) {
         if (Priorities[i] <= Prio)
            continue;
         if (nondet() && Sleeping[i]) {
                Prio = i;
                Sleeping[i]=false;
                Threads[i].entry();
                break;
         }
   }
    Prio = prevPrio;
}
```

Fig. 6. Pseudo-code to execute one hyperperiod [23]

The study [24] proposes a light-weighted asynchronous processing solution that is based on the idea of Pareto principle about coloring algorithms. It is to separate the vertices processing based on the color distribution by partitioning the vertices to achieve the maximum parallelism and to reduce data transmission costs while processing the partitions.

Lian X. et al. have performed research [25] on two common asynchronous parallel implementations for Stochastic Gradient on computer networks and shared memory systems. Two algorithms (ASYSG-CON and ASYSG-INCON) were used to describe the two above implementations. This study was able to explain their converging and accelerating properties, mainly due to the heterogeneity of the majority of deep learning formulas and asym asymable parallel mechanisms.

In study [26], the authors modeled the dependence of the output on tens of thousands of causal events that recursively use a new time-coding scheme for real-time processing of event streams. In tests using real data, the study proved useful in real-world applications.

Based on the results of the study reviewed above, we can see that the problem of optimizing embedded software on multi-core processors also focuses a lot on multi-threaded models to parallelize the processing flows in the program. With independent data problems, it is necessary to process a large amount of data that requires a more efficient parallel model and minimizes synchronized time, linking data between threads. This class of problems can be solved by dividing data into sub-sets for asynchronous processing. This method will be developed, tested and evaluated in the following sections.

3. Ideas of research and experimental processes

3.1. Ideas. The main idea of the proposed method is to build a global data partitioning model based on the configuration of processor builders and asynchronous data processing between threads for performance improvements. Global data are used so that every thread is accessible, and the data division creates independent data to execute concurrent processing on processor cores. An asynchronous data processing model is applied to eliminate intermediate data processing time and link data between threads.

3.2. Research and experimental process. The process of developing and experimenting with the method described as shown in Figure 7 is carried out in 4 steps as follows:

1. Analyse configuration of a processor: From the information of the processor, we analyse the processor's configuration information to give the processor's configuring the filling.
2. Calculate partitioning rates: Based on the configuration of the chromium 4th of the processor, we calculate, determine the data partitioning ratios.
3. Data partitioning: To select the rate type and global data, we divide it into data parts.
4. Multi-thread programming: From the processor configuration information and data sections, we determine the number of data and set the processing flows on each divided data area. Map the threads to the corresponding characters.

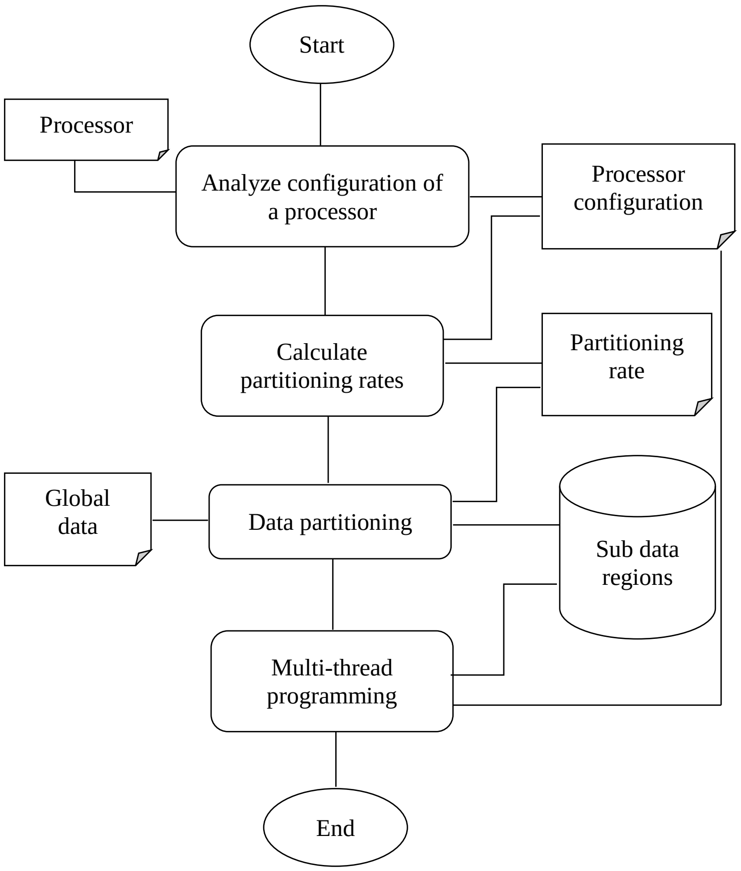

Fig. 7. Research and experimental process

3.3. Data partitioning. Data partitioning is a data delivery technique to improve data processing performance. Processing performance can be improved in one of two ways. Firstly, it is possible in some cases to pre-determine that a partition does not need to be accessed for processing depending on how the data are partitioned. Secondly, when data are partitioned, in some cases, parallelism can be achieved in data processing because different partitions can be accessed and processed in parallel. In this paper, we focus on data partitioning towards parallel processing on multi-core processor cores, since depending on the properties of the processor, the partitioning is based on the processing capacity, the size of the human cache.

Definition 1 – ratio by speed. The speed ratio is the ratio between the processing capacity of the current core in the set of microprocessors. This ratio is used to divide data by the speed of the core. The speed ratio is denoted as r_s, described in Equation (1).

$$r_s = \frac{s_i}{\sum_{i=1}^{N} s_i}, \tag{1}$$

where:

- s_i is the speed of the i^{th} core;
- N is the number of cores in a processor.

Definition 2 – ratio by buffer size. Caching size ratio is the rate at which data are stored on the cache of the current core, the set of microprocessors. This ratio is used to partition data according to the buffer size of the core. The buffer size ratio is denoted as r_c, described in Equation (2):

$$r_c = \frac{c_i}{\sum_{i=1}^{N} c_i}, \tag{2}$$

where:

- c_i is the cache size of the i^{th} core;
- N is the number of cores in a processor.

Definition 3 – aggregate ratio. The aggregate ratio is the ratio of the combination of the speed ratio and the core cache size ratio, currently the set of microprocessors. This scale is used to partition the data according to the aggregate ratio of the core. The aggregate ratio is denoted as r, described in Equation (3):

$$r = \alpha \times r_s + \beta \times r_c, \tag{3}$$

where:

- α and β are positive coefficients;
- $\alpha + \beta = 1$.

From the above definitions, we offer three ways to divide the original data set into sub-set. Call D the size of the data to be processed, and D_i is the size of the component data partition i being calculated by Equations (4), (10) and (11) depending on the division.

Split data proportional to the speed of core/processor. The size of the data divided by the speed ratio of the i^{th} component is determined based on the speed ratio of the i^{th} core and the size of the data to be processed:

$$Ds_i = r_s \times D = \frac{s_i}{\sum_{i=1}^{N} s_i} \times D. \tag{4}$$

Proposition. Suppose we have a processor with N cores, each core has the same speed s_i, has the same cache capacity, and the performance r_s is calculated by definition 1. There is data block D divided by cores that execute concurrently.

According to the normal way of data division, there are N cores, D data block will be divided into N parts then, each core will perform D/N data processing. Since the cores execute concurrently, the time it takes to process the data block D needs to run out t_{max}, where t_{max} is the time of the lowest-speed core s_{min} or, in other words, the time that the least-performing core needs to process the assigned data portion.

$$t_{max} = \frac{D}{N} \times \frac{1}{s_{min}}. \tag{5}$$

We improved the data division to increase the performance of the above D block processing by dividing the volume for each core based on the performance of each core. Then the data is divided according to Equations (4). Since the cores execute concurrently and the data of each core is divided according to the performance of each core, the processing time of each core is equivalent and is called t.

$$t = \frac{D \times s_i}{\sum_{i=1}^{N} s_i} \times \frac{1}{s_i}. \tag{6}$$

Need to prove that $t < t_{max}$.

From (5),

$$t_{max} = \frac{D}{N \times s_{min}} = \frac{D}{\sum_{1}^{N} s_{min}}. \tag{7}$$

From (6),

$$t = \frac{D}{\sum_{i=1}^{N} s_i}. \tag{8}$$

Since s_{min} is always less than or equal to $s_i \; \forall \; i \epsilon 1, .. N$, according to (7) and (8), $t < t_{max}$.

So with the performance of different cores and the data block being large enough, our data division is better than the normal data division.

Since the processing speed of the core is directly proportional to the cache capacity of the core, the above statement is true for the data division Equations (9) and (10).

Split data proportional to buffer size. The size of the data divided by the caching size ratio of the i^{th} component is determined based on the caching size ratio of the i^{th} core and the size of the data to be processed:

$$Dc_i = r_c \times D = \frac{c_i}{\sum_{i=1}^{N} c_i} \times D. \tag{9}$$

Split data using a combination of two buffer sizes and speed parameters. The data size of the i^{th} component is determined based on the aggregate ratio of the i^{th} core and the size of the data to be processed:

$$D_i = r \; \times D = (\; \alpha \times r_s + \beta \times r_c) \times D. \tag{10}$$

3.4. Building a multi-threaded model for asynchronization optimization. When solving the problem in a multi-threaded model, in addition to execution time context transfer time, the system also takes more time to synchronize the flows. Thread synchronized time has a great influence on system performance. When dividing data, if data processed by threads depend on each other, it is imperative to synchronize the flows. However, if the data parts are processed by independent threads, there is no need for synchronizing; just determine when all flows are done. The challenge is how to solve the problem of asynchrony with independent data. To solve this problem, we propose organizations of input, output and counting variables in the global memory area. Because it is a global memory, all threads are retrieved. At that time, the input data will be divided into regions with dimensions determined by formulas (4), (9), (10), depending on the divisions. The output data also are in global memory areas. Each thread stores the input data in the corresponding memory using this global memory. The processing process is shown in Figure 8, including the following steps:

- Init(): Function that implements two alarms of data area *D* as a global byte array.
- Division(*D*): Divides array *D* into D_i sub-arrays and calculates

the number of threads to perform, assigning global variables using the number of threads to perform.

- Multithread(): Execution of threads performed on 1 Array of D_i. When a thread is done, the global variable decreases by 1.
- Check (global): Check the global = 0 variable; all threads are finished, no synchronizing and ending is required.

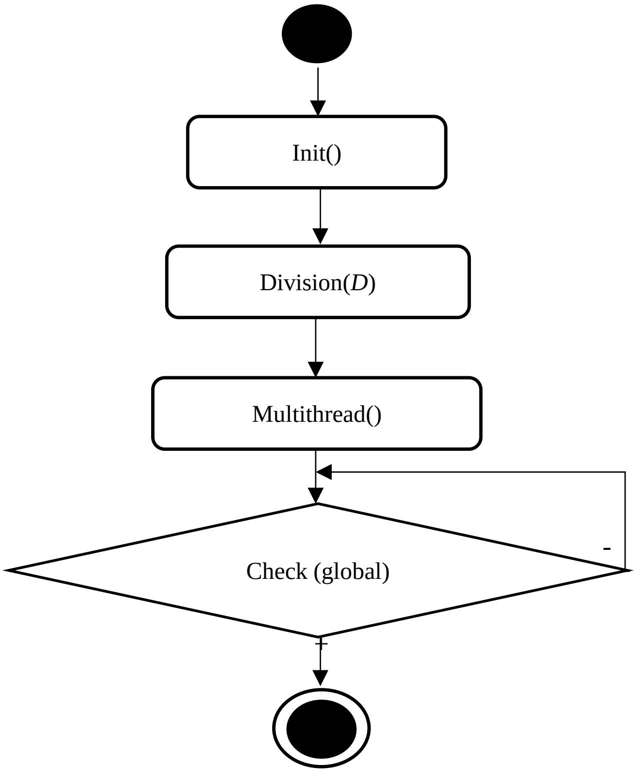

Fig. 8. Processing

4. Experiments

4.1. Experimental description. To evaluate the proposed method, we conduct experiments on the embedded Raspberry PI 3 computer – configured as in Table 2 on the embedded computer. We implement the encryption algorithms AES, DES, 3DES, etc., installed according to the mono-threaded model and multi-threaded model with the data divided according to formulas (4), (9) and (10).

Table 2. Raspberry PI 3 Embedded computer configuration

Ingredients		**Values**
Hardware	CPU	64 bit quad-core CPUARM Cortex A53 processor, 1.2 GHz speed 4 processors
	RAM	1 GB
	Memory card	32 GB
	Peripheral	4 USB ports HDMI port, full HDMI support Ethernet port (or LAN port)
Software	OS	Raspbian
	Application	Python Interpreter

4.2. Experimental system model. The experimental system model shown in Figure 9 includes:

1. **Camera**: Video obtained from the camera will go to the embedded Raspberry computer.
2. **Raspberry embedded computer:** The original video received from the camera is encrypted and transferred the code to storage on the IoT cloud server.
3. **IoT cloud server:** Provides API for execution on the mobile and the computer via a web browser.
4. **Smartphone:** Enforces android app with the server API to decrypt video transmitted from the server.
5. **PC**: Enforce on a web browser with API from the server to decrypt video transmitted from the server.

To evaluate the method, on the Raspberry, we install the AES algorithm, DES, Triple DES (3DES) to encode the video according to the single-threaded model and according to the multi-threaded model based on the proposed method. The Raspberry has a 4-ed processor with a symmetrical architecture, so according to the method developed in previous sections, we divide the data into 4 partitions of the same size; each partition is enforced in one thread. Experimentally, the program is divided into 4 threads corresponding to 4 characters. Each thread handles a corresponding data area.

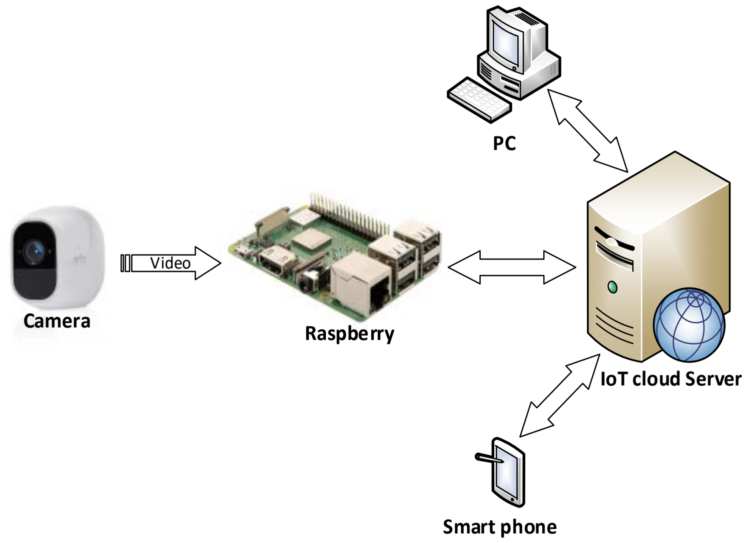

Fig. 9. Experimental model

Table 3. AES encryption algorithm executing time on Raspberry PI 3

Data size (MB)	Execution Time (ms)		Improvement rate (%)
	Single threading	Multithreading	
1	64.23	35.80	44.26
2	118.59	61.44	48.19
4	230.39	112.72	51.07
8	455.22	216.37	52.47
16	912.86	416.25	54.40
32	1851.16	843.74	54.42
64	3798.15	1629.91	57.09
Average			**51.78**

Table 4. DES encryption algorithm executing time on Raspberry PI 3

Data size (MB)	**Execution Time (ms)**		**Improvement rate (%)**
	Single threading	**Multithreading**	
1	90.53	44.19	51.19
2	177.97	76.67	56.92
4	347.27	140.28	59.60
8	699.05	271.78	61.12
16	1398.91	541.06	61.32
32	2769.94	1082.18	60.93
64	5627.24	2136.39	62.03
Average			**57.59**

Table 5. 3 DES encryption algorithm executing time on Raspberry PI 3

Data size (MB)	**Execution Time (ms)**		**Improvement rate (%)**
	Single threading	**Multithreading**	
1	201.91	72.31	64.18
2	401.41	135.38	66.27
4	817.61	264.61	67.63
8	1617.32	520.43	67.82
16	3220.44	1169.88	63.67
32	6431.65	2025.93	68.05
64	12812	4069.99	68.23
Average			**66.55**

4.3. Evaluation of experimental results. The experimental results are aggregated in Table 6 and are described in Figure 10. The experimental results show an average performance improvement rate of 59.09%. In particular, the experimental result with the AES algorithm is 51.78%, DES is 57.59%, and Triple DES is 66.55%. The average performance improvement rate increases as data size grows. When the data size is small, the performance improvement rate is low when using multithreads due to the loss of CPU rotation time between threads and intermediate data processing time. However, when the processing data size is large, the rate of performance improvements is high and gradually progresses to the proportionality to the number of threads. In the proposed model, since it does not take time to process data synchronizing between threads, it only takes time to divide the original data, so it works well when the data size is large. Although it is not only in terms of performance, this model also increases the size of the program's occupied memory.

Table 6. Improvement rate comparison between AES, DES and 3DES

Data size (MB)	**Improvement rate (%)**		
	AES	**DES**	**3DES**
1	44.26	51.19	64.18
2	48.19	56.92	66.27
4	51.07	59.6	67.63
8	52.47	61.12	67.82
16	54.4	61.32	63.67
32	54.42	60.93	68.05
64	57.09	62.03	68.23
Average	**51.78**	**57.59**	**66.55**

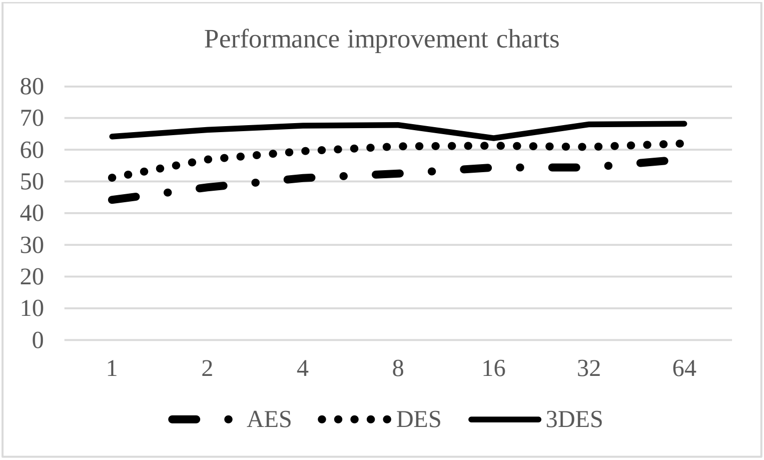

Fig. 10. Performance improvement charts

During the experiment, we also compared and evaluated the performance of data partition and asynchronous processing between the three AES, DES, and 3DES encryption algorithms on the same data set size 1, 2, 4, 8, 16, 64 and as shown in Table 7 and in Figure 11. Out of the three algorithms, the execution time of AES is the least and the greater the data size, the bigger the execution time discrepancy between the three algorithms.

Table 7. Performance comparison between AES, DES and 3DES

Data size (MB)	**Execution Time (ms)**		
	AES	**DES**	**3DES**
1	35.80	44.19	72.31
2	61.44	76.67	135.38
4	112.72	140.28	264.61
8	216.37	271.78	520.43
16	416.25	541.06	1169.88
32	843.74	1082.18	2025.93
64	1629.91	2136.39	4069.99

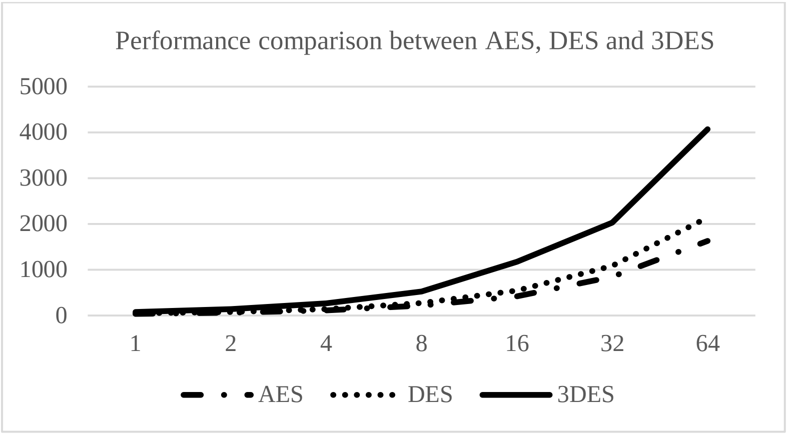

Fig. 11. Performance comparison between AES, DES and 3DES

The graph in Figure 12 compares the rate of performance improvement in the proposed method with other methods [1, 2, 3]

([2] is 25%, [3] – 21%, [1] – 33.1 %. The recommended method's performance improvement rate is also better than the average improvement rate as aggregated in studies [1, 2, 3].

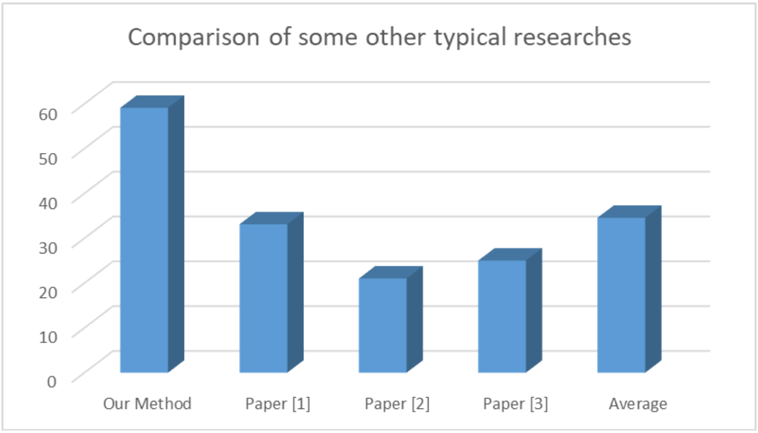

Fig. 12. Comparison of some other typical studies

5. Conclusion and future work. The paper proposes and develops a method to improve embedded software performance on multi-core processors based on data partitioning and asynchronous processing. The issue of performance improvement for embedded software on multi-core processors is of high practical significance. Especially, the problems need to be handled and ensured information security because this type of data takes time to process for data safety. Therefore, the performance improvement based on data partitioning and asynchronous processing of the paper meets the need of improving the speed of embedded software for data with independent divisibility such as block cipher.

Data are divided into sections proportional to the number of cores, the speed of the core and the cache size. Data are declared globally to be shared for all threads, so it does not take time to sync and link data. Each part of the data is mapped to the corresponding execution threads. Threads are performed asynchronously. The proposed method works well, especially when the data size is large. A high-performance improvement rate compared to previous studies has been shown.

Despite the positive results, the proposed method still has some limitations such as large memory appropriation; ineffective when the data size is small; the problem of mapping threads to the corresponding cores; the new paper only calculates the data division based on performance at the beginning of execution, and during execution, the performance of each CPU core changes is not taken into account.

As future work, we will continue to improve the method in several aspects: expanding the problem to process data in sync; map the execution threads to the corresponding cores; and monitor and change the data division according to performance at each time when each CPU core has finished processing data.

References

1. Yao, Y. Power and Performance Optimization for Network-on-Chip based Many-Core Processors. PhD thesis. KTH. School of Electrical Engineering and Computer Science (EECS). 2019.
2. Lim, G. and Suh, S.-B. User-Level Memory Scheduler for Optimizing Application Performance in NUMA-Based Multicore Systems. IEEE 5th International Conference on Software Engineering and Service Science. 2014. 10.1109/ICSESS.2014.6933553.
3. Wei, X., Ma, L., Zhang, H. & Liu, Y. Multi-core, Multi-thread based Optimization Algorithm for Large-scale Traveling Salesman Problem. Alexandria Engineering Journal 60, 2021, pp. 189-197.
4. Khalib, Z.I.A., Ng, H.Q. Elshaikh, M., and Othman, M.N., Optimizing Speedup on Multicore Platform with OpenMP Schedule Clause and Chunk Size, IOP Conference Series. 2020. Materials Science and Engineering 767, 012037.
5. Lingampalli, S., Mirza, F., Raman, T. and Agonafer, D. Performance Optimization of Multi-core Processors using Core Hopping - Thermal and Structural. Proc. of the 28th Annual IEEE Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM). 2012. pp. 112-117.
6. Gunther, N.J., Subramanyam, S., and Parvu, S. A Methodology for Optimizing Multithreaded System Scalability on Multi-cores. Programming Multi-core and Many-core Computing Systems. 2011. CoRR abs/1105.4301.
7. Rengasamy, V., Fu, T.-Y., Lee, W.-C., and Madduri, K. Optimizing Word2Vec Performance on Multicore Systems. Proceedings of the Seventh Workshop on Irregular Applications. 2017. Architectures and Algorithms. Association for Computing Machinery. New York. NY. USA.
8. Wipe, E., Miller, J.E., Choi, I., Yeung, D. Amarasinghe, S.P., and Agarwal, A. Multicore Performance Optimization Using Partner Cores. 2011. in Michael McCool & Mendel Rosenblum. 'HotPar'. USENIX Association.
9. Zhou, Y., He, F., Hou, N., and Qiu, Y. Parallel Ant Colony Optimization on Multi-core SIMD CPUs. Future Generation Computer Systems 79. 2018. pp. 473-487.
10. Emmi, M., Lal, A., and Qadeer, S. Asynchronous Programs with Prioritized Task-buffers. SIGSOFT FSE. 2012. 48.
11. Emmi, M., Ganty, P., Majumdar, R., Rosa-Velardo, F. Analysis of Asynchronous Programs with Event-Based Synchronization ESOP 2015: Programming Languages and Systems. 2015. pp. 535-559.
12. Kornaros, G. Multi-Core Embedded Systems. CRC Press. 2010. Inc., USA.
13. Bodake, V. and Gawande, R.M. A Review on An Encryption Engines For Multi Core Processor Systems. IOSR Journal of Electronics and Communication Engineering (IOSR-JECE). e-ISSN: 2278-2834. p-ISSN: 2278-8735. pp. 38-46.
14. Polychroniou, O. and Ross, K.A. A Comprehensive Study of Main-memory Partitioning and its Application to Large-scale Comparison- and Radix-sort. Print SIGMOD. 2014. pp. 755–766.

15. Schuhknecht, F.M., Khanchandani, P., and Dittrich, J. On the Surprising Difficulty of Simple Things: the case of radix partitioning. VLDB. 8(9): 2015. pp. 934–937.
16. Wu, L., Barker, R.J., Kim, M.A., and Ross, K.A. Navigating Big Data with High-throughput, Energy-efficient Data Partitioning. Print SIGARCH. volume 41. 2013. pp. 249–260.
17. Wang, Z., He, B., and Zhang, W. A Study of Data Partitioning on OpenCL-based FPGAs. In FPL. 2015. pp. 1–8.
18. Ke, Q., Prabhakaran, V., Xie, Y., Yu, Y., Wu, J., Yang, J. Optimizing Data Partitioning for Data-Parallel Computing. Hot Topics in Operating Systems (HotOS XIII) | May 2011. Published by USENIX.
19. Cieslewicz, J. and Ross, K. Data Partitioning on Chip Multiprocessors. DaMoN '08: Proceedings of the 4th international workshop on Data management on new hardware June 2008. pp. 25–34.DOI: 10.1145/1457150.1457156.
20. Zhong, Z. et al. Data Partitioning on Heterogeneous Multicore and Multi-gpu Systems Using Functional Performance Models of Data-parallel Applications in Cluster. 2012. pp. 191–199.
21. Kara, K., Giceva, J., and Alonso, G. FPGA-based Data Partitioning. Proceedings of the 2017 ACM International Conference on Management of Data. May 2017. pp. 433–445. DOI: 10.1145/3035918.3035946.
22. Zhong, Z., Rychkov, V., and Lastovetsky, V. Data Partitioning on Multicore and Multi-GPU Platforms Using Functional Performance Models. IEEE Transactions on Computers. Volume 64. Issue 9. Sept. 1 2015. Doi: 10.1109/TC.2014.2375202.
23. Alanazi, H.O., Zaidan, B.B., Zaidan, A.A., Jalab, H.A., Shabbir, M., and Al-Nabhani, Y. New Comparative Study Between DES, 3DES and AES. J. of computing. volume 2. issue 3. March 2010. ISSN 2151-9617.
24. Farooq, U. and Faisal Aslam, M. Comparative Analysis of Different AES Implementation Techniques for Efficient Resource Usage and better Performance of an FPGA. Journal of King Saud University - Computer and Information Sciences. Volume 29. Issue 3. July 2017. pp. 295-302.
25. Sen, K. and Viswanathan, M. Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In CAV. 2006. pp. 300–314.
26. Kidd, N., Jagannathan, S., and Vitek, J. One Stack to Run Them all: Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling. In SPIN ’10: Proc. of the 17th International Workshop on Model Checking Software. volume 6349 of LNCS, Springer. 2010. pp. 245–261.
27. Lian, X., Huang, Y., Li, Y., and Liu, J. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. NIPS 2015: pp. 2737-2745.
28. Alba, E. and Troya, J.M. Analyzing Synchronous and Asynchronous Parallel Distributed Genetic Algorithms, Future Generation Computer Systems. Vol. 17. Issue 4. January 2001. pp. 451–465.

Bui Phuc — Ph.D., Postgraduate student, Vietnam National University. Research interests: cyber security, embedded software optimization, software technology. The number of publications — 3. phucbh.hvan@gmail.com; 144, Xuan Tu St., 11311, Hanoi, Viet Nam; office phone: +84 24 3858 4615.

Le Minh — Ph.D., Dr.Sci., Head of department, Department of information system security, Information Technology Institute – Vietnam National University. Research interests: cyber security, reliability of information system. The number of publications — 40. quangminh@vnu.edu.vn; 144, Xuan Tu St., 11311, Hanoi, Viet Nam; office phone: +84 24 3858 4615.

Hoang Binh — Software engineer, Technological Application and Production One Member Limited Liability company. Research interests: mobile and embedded software engineering, AIoT, IoT optimization, mobile apps. binhht@teca.vn; 18A, Republic St., 700901, Ho Chi Minh City, Russia; office phone: +84-28 3811 0181.

Ngoc Nguyen — Ph.D., Dr.Sci., Professor, Vice-president, Kyoto College of Graduate Studies for Informatics (KCGI). Research interests: computer science, software engineering, embedded systems & software, machine learning, data mining, information security. The number of publications — 114. nn_binh@kcg.edu; 7, Tanaka Monzencho St., 606-8225, Kyoto, Japan; office phone: +81 75-711-0161.

Pham Huong — Ph.D., Dr.Sci., Vice dean of the faculty, Faculty of information technology, Academy of Cryptography. Research interests: machine learning in information security, cloud computing, AIoT and IoT optimization. The number of publications — 30. huongpv@actvn.edu.vn; 141, Victory St., 100915, Hanoi, Viet Nam; office phone: +84 24 3854 4244.

УДК 004.056 **DOI** 10.15622/ia.21.2.2

Ф.Х. Буй, М.К. Ле, Б.Т. Хоанг, Н.Б. Нгок, Х.В. Фам

РАЗДЕЛЕНИЕ ДАННЫХ И АСИНХРОННАЯ ОБРАБОТКА ДЛЯ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ ВСТРОЕННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА МНОГОКОДНЫХ ПРОЦЕССОРАХ

Буй Ф.Х., Ле М.К., Хоанг Б.Т., Нгок Н.Б., Фам Х.В. **Разделение данных и асинхронная обработка для повышения производительности встроенного программного обеспечения на многокодных процессорах.**

Аннотация. Сегодня обеспечение информационной безопасности крайне неизбежно и актуально. Мы также наблюдаем активное развитие встраиваемых IoT-систем. В результате основное внимание уделяется исследованиям по обеспечению информационной безопасности встроенного программного обеспечения, особенно в задаче повышения скорости процесса шифрования. Однако исследованиям по оптимизации встроенного программного обеспечения на многоядерных процессорах для обеспечения информационной безопасности и повышения производительности встроенного программного обеспечения не уделялось особого внимания. В статье предлагается и развивается метод повышения производительности встроенного программного обеспечения на многоядерных процессорах на основе разделения данных и асинхронной обработки в задаче шифрования данных. Данные используются глобально для извлечения любыми потоками. Данные разбиты на разные разделы, также программа устанавливается по многопоточной модели. Каждый поток обрабатывает раздел разделенных данных. Размер каждой части данных пропорционален скорости обработки и размеру кэша ядра многоядерного процессора. Потоки работают параллельно и не нуждаются в синхронизации, но необходимо совместно использовать глобальную общую переменную для проверки состояния выполнения системы. Наше исследование встроенного программного обеспечения основано на безопасности данных, поэтому мы протестировали и оценили метод с несколькими блочными шифрованиями, такими как AES, DES и т. д. На Raspberry Pi3. В нашем результате средний показатель повышения производительности составил около 59,09%. В частности, наши экспериментальные результаты с алгоритмами шифрования показали: AES - 51,78%, DES - 57,59%, Triple DES - 66,55%.

Ключевые слова: повышение производительности встроенного программного обеспечения, многоядерные процессы, многопоточность, разделение данных, асинхронная обработка.

Буй Фук Хуу — Ph.D., аспирант, Вьетнамский национальный университет. Область научных интересов: кибербезопасность, оптимизация встроенного программного обеспечения, программные технологии. Число научных публикаций — 3. phucbh.hvan@gmail.com; Суан Туи, 144, 11311, Ханой, Вьетнам; р.т.: +84 24 3858 4615.

Ле Минь Куанг — д-р техн. наук, заведующий кафедрой, кафедра безопасности информационных систем, Институт информационных технологий Вьетнамского национального университета. Область научных интересов: кибербезопасность, надежность информационных систем. Число научных публикаций — 40. quangminh@vnu.edu.vn; Суан Туи, 144, 11311, Ханой, Вьетнам; р.т.: +84 24 3858 4615.

Хоанг Бинь Тхань — инженер-программист, Общество с ограниченной прикладной и технической продукцией - TECAPRO. Область научных интересов: мобильное и встроенное программное обеспечение, AIoT, оптимизацию IoT, мобильные приложения. binhht@teca.vn; Республика, 18A, 700901, Хошимин, Россия; р.т.: +84-28 3811 0181.

Нгок Нгуен Бинь — д-р техн. наук, профессор, вице-президент, Киотский колледж аспирантуры по информатике. Область научных интересов: компьютерные науки, разработка программного обеспечения, встроенные системы и программное обеспечение, машинное обучение, интеллектуальный анализ данных, информационная безопасность. Число научных публикаций — 114. nn_binh@kcg.edu; Танака Монзенчо, 7, 606-8225, Киото, Япония; р.т.: +81 75-711-0161.

Фам Хуонг Ван — д-р техн. наук, заместитель декана факультета, факультет информационных технологий, Академия криптографии. Область научных интересов: машинное обучение в области информационной безопасности, облачных вычислений, AIoT и оптимизации IoT. Число научных публикаций — 30. huongpv@actvn.edu.vn; Победы, 141, 100915, Ханой, Вьетнам; р.т.: +84 24 3854 4244.

Литература

1. Yao, Y. Power and Performance Optimization for Network-on-Chip based Many-Core Processors. PhD thesis. KTH. School of Electrical Engineering and Computer Science (EECS). 2019.
2. Lim, G. and Suh, S.-B. User-Level Memory Scheduler for Optimizing Application Performance in NUMA-Based Multicore Systems. IEEE 5th International Conference on Software Engineering and Service Science. 2014. 10.1109/ICSESS.2014.6933553.
3. Wei, X., Ma, L., Zhang, H. & Liu, Y. Multi-core, Multi-thread based Optimization Algorithm for Large-scale Traveling Salesman Problem. Alexandria Engineering Journal 60, 2021, pp. 189-197.
4. Khalib, Z.I.A., Ng, H.Q. Elshaikh, M., and Othman, M.N., Optimizing Speedup on Multicore Platform with OpenMP Schedule Clause and Chunk Size, IOP Conference Series. 2020. Materials Science and Engineering 767, 012037.
5. Lingampalli, S., Mirza, F., Raman, T. and Agonafer, D. Performance Optimization of Multi-core Processors using Core Hopping - Thermal and Structural. Proc. of the 28th Annual IEEE Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM). 2012. pp. 112-117.
6. Gunther, N.J., Subramanyam, S., and Parvu, S. A Methodology for Optimizing Multithreaded System Scalability on Multi-cores. Programming Multi-core and Many-core Computing Systems. 2011. CoRR abs/1105.4301.
7. Rengasamy, V., Fu, T.-Y., Lee, W.-C., and Madduri, K. Optimizing Word2Vec Performance on Multicore Systems. Proceedings of the Seventh Workshop on Irregular Applications. 2017. Architectures and Algorithms. Association for Computing Machinery. New York. NY. USA.
8. Wipe, E., Miller, J.E., Choi, I., Yeung, D. Amarasinghe, S.P., and Agarwal, A. Multicore Performance Optimization Using Partner Cores. 2011. in Michael McCool & Mendel Rosenblum. 'HotPar'. USENIX Association.
9. Zhou, Y., He, F., Hou, N., and Qiu, Y. Parallel Ant Colony Optimization on Multi-core SIMD CPUs. Future Generation Computer Systems 79. 2018. pp. 473-487.
10. Emmi, M., Lal, A., and Qadeer, S. Asynchronous Programs with Prioritized Task-buffers. SIGSOFT FSE. 2012. 48.

11. Emmi, M., Ganty, P., Majumdar, R., Rosa-Velardo, F. Analysis of Asynchronous Programs with Event-Based Synchronization ESOP 2015: Programming Languages and Systems. 2015. pp. 535-559.
12. Kornaros, G. Multi-Core Embedded Systems. CRC Press. 2010. Inc., USA.
13. Bodake, V. and Gawande, R.M. A Review on An Encryption Engines For Multi Core Processor Systems. IOSR Journal of Electronics and Communication Engineering (IOSR-JECE). e-ISSN: 2278-2834. p-ISSN: 2278-8735. pp. 38-46.
14. Polychroniou, O. and Ross, K.A. A Comprehensive Study of Main-memory Partitioning and its Application to Large-scale Comparison- and Radix-sort. Print SIGMOD. 2014. pp. 755–766.
15. Schuhknecht, F.M., Khanchandani, P., and Dittrich, J. On the Surprising Difficulty of Simple Things: the case of radix partitioning. VLDB. 8(9): 2015. pp. 934–937.
16. Wu, L., Barker, R.J., Kim, M.A., and Ross, K.A. Navigating Big Data with High-throughput, Energy-efficient Data Partitioning. Print SIGARCH. volume 41. 2013. pp. 249–260.
17. Wang, Z., He, B., and Zhang, W. A Study of Data Partitioning on OpenCL-based FPGAs. In FPL. 2015. pp. 1–8.
18. Ke, Q., Prabhakaran, V., Xie, Y., Yu, Y., Wu, J., Yang, J. Optimizing Data Partitioning for Data-Parallel Computing. Hot Topics in Operating Systems (HotOS XIII) | May 2011. Published by USENIX.
19. Cieslewicz, J. and Ross, K. Data Partitioning on Chip Multiprocessors. DaMoN '08: Proceedings of the 4th international workshop on Data management on new hardware June 2008. pp. 25–34. DOI: 10.1145/1457150.1457156.
20. Zhong, Z. et al. Data Partitioning on Heterogeneous Multicore and Multi-gpu Systems Using Functional Performance Models of Data-parallel Applications in Cluster. 2012. pp. 191–199.
21. Kara, K., Giceva, J., and Alonso, G. FPGA-based Data Partitioning. Proceedings of the 2017 ACM International Conference on Management of Data. May 2017. pp. 433–445. DOI: 10.1145/3035918.3035946.
22. Zhong, Z., Rychkov, V., and Lastovetsky, V. Data Partitioning on Multicore and Multi-GPU Platforms Using Functional Performance Models. IEEE Transactions on Computers. Volume 64. Issue 9. Sept. 1 2015. Doi: 10.1109/TC.2014.2375202.
23. Alanazi, H.O., Zaidan, B.B., Zaidan, A.A., Jalab, H.A., Shabbir, M., and Al-Nabhani, Y. New Comparative Study Between DES, 3DES and AES. J. of computing. volume 2. issue 3. March 2010. ISSN 2151-9617.
24. Farooq, U. and Faisal Aslam, M. Comparative Analysis of Different AES Implementation Techniques for Efficient Resource Usage and better Performance of an FPGA. Journal of King Saud University - Computer and Information Sciences. Volume 29. Issue 3. July 2017. pp. 295-302.
25. Sen, K. and Viswanathan, M. Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In CAV. 2006. pp. 300–314.
26. Kidd, N., Jagannathan, S., and Vitek, J. One Stack to Run Them all: Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling. In SPIN ’10: Proc. of the 17th International Workshop on Model Checking Software. volume 6349 of LNCS, Springer. 2010. pp. 245–261.
27. Lian, X., Huang, Y., Li, Y., and Liu, J. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. NIPS 2015: pp. 2737-2745.
28. Alba, E. and Troya, J.M. Analyzing Synchronous and Asynchronous Parallel Distributed Genetic Algorithms, Future Generation Computer Systems. Volume 17. Issue 4. January 2001. pp. 451–465.