https://doi.org/10.31891/csit-2025-1-14
UDC 004.7

Dmytro MARTINIUK, Oleksii LYHUN, Andriy DROZD
Khmelnytskyi National University
Oleksii BESEDOVSKYI
Simon Kuznets Kharkiv National University of Economics

TASK OPTIMISATION IN MULTIPROCESSOR EMBEDDED SYSTEMS

The relevance of this work lies in the fact that the existing task distribution in multiprocessor embedded systems plays a key role in the development of devices used in various industries. Despite the progress made, there are still many research challenges that require in-depth analysis and implementation of effective solutions. One of the main challenges is to ensure the reliability of embedded systems, especially in environments where safety is critical. Although the functionality of such systems is usually defined at the design stage, ensuring their stable operation in real time remains a challenge. It is necessary not only to guarantee the correctness of calculations, but also to adhere to time constraints, which requires new approaches to managing the resources of multiprocessor systems. Another important problem is the need to meet stringent real-time requirements. This is a characteristic feature of embedded systems, which differ from general-purpose systems that have more flexibility in functionality but do not guarantee such predictability and reliability. Therefore, optimization of task scheduling that takes into account the specifics of embedded systems requires further research. It is also important to take into account the variety of embedded systems, which are divided into control systems and streaming systems that have different data processing requirements. Control systems must respond quickly to environmental events while minimizing delays, while streaming systems process continuous data streams, requiring high throughput and efficiency. The development of universal solutions that can optimize the performance of both types of systems is an urgent task for scientists and engineers. Therefore, task optimization in multiprocessor embedded systems has significant potential for development and is relevant for reliability, real-time guarantees, and efficient resource management, which will contribute to the creation of more secure and productive systems.

In this paper, we develop a method for optimizing task execution using replication in a multiprocessor system, which allows to effectively minimize the total execution time, ensure load balance, and minimize communication delays. The peculiarity of the method is the implementation of task migration according to replication using the optimization objective function. An experiment with the system demonstrated that the chosen optimization method effectively balances the load, but additional objective functions are needed to optimize energy consumption. The simulation results show that an increase in the number of processors leads to a decrease in the maximum load and the number of migrations, an increase in the number of tasks increases the system load and the number of migrations at the initial stages, and the migration mechanism effectively balances the load, especially at the initial stages of execution.

The areas of further research are the detailing of embedded devices and their classification. For each class of embedded devices, it will be necessary to adapt the algorithms and method of task optimization, as well as to develop the target optimization function.

Keywords: Multiprocessor systems, embedded systems, computer systems, optimization

Дмитро МАРТИНЮК, Олексій ЛИГУН, Андрій ДРОЗД
Хмельницький національний університет
Олексій БЕСЄДОВСЬКИЙ
Харківський національний економічний університет імені Семена Кузнеця

ОПТИМІЗАЦІЯ ЗАВДАНЬ У БАГАТОПРОЦЕСОРНИХ ВБУДОВАНИХ СИСТЕМАХ

Актуальність даної роботи полягає в тому, що існуючі розподіл завдань у багатопроцесорних вбудованих системах відіграє ключову роль у процесі розробки пристроїв, що застосовуються в різноманітних галузях. Незважаючи на досягнутий прогрес, залишається чимало дослідницьких викликів, які вимагають глибокого аналізу та впровадження ефективних рішень. Одним із головних викликів є забезпечення надійності вбудованих систем, особливо в умовах, де критично важлива безпека. Хоча функціональні можливості таких систем зазвичай визначені ще на етапі проєктування, забезпечення їхньої стабільної роботи в режимі реального часу залишається складним завданням. Потрібно не тільки гарантувати правильність обчислень, але й дотримуватися часових обмежень, що вимагає нових підходів до управління ресурсами багатопроцесорних систем. Ще однією важливою проблемою є необхідність дотримання жорстких вимог реального часу. Це є характерною рисою вбудованих систем, які відрізняються від систем загального призначення, що мають більшу гнучкість у функціональності, але не гарантують такої передбачуваності та надійності. Тому оптимізація планування завдань, яка враховує специфіку вбудованих систем, потребує подальших досліджень. Важливо також враховувати різноманітність вбудованих систем, які поділяються на системи управління та потокові системи, що мають різні вимоги до обробки даних. Системи управління повинні оперативно реагувати на події зовнішнього середовища, мінімізуючи затримки, тоді як потокові системи обробляють безперервні потоки даних, вимагаючи високої пропускної здатності та ефективності. Розробка універсальних рішень, здатних оптимізувати продуктивність обох типів систем, є актуальним завданням для науковців та інженерів.Тому, оптимізація завдань у багатопроцесорних вбудованих системах має значний потенціал для розвитку і є актуальною щодо надійності, гарантій реального часу та ефективного управління ресурсами, що сприятиме створенню більш безпечних та продуктивних систем.

У даній роботі розроблено метод оптимізації виконання завдань з використанням реплікації у багатопроцесорній системі, який дає змогу ефективно мінімізувати загальний час виконання, забезпечити баланс навантаження і мінімізувати затримки зв'язку. Особливістю методу є реалізація міграції завдань згідно реплікації з використанням цільової функції оптимізації. Проведений експеримент з системою продемонстрував, що обраний метод оптимізації ефективно вирівнює навантаження, але для оптимізації енергоспоживання потрібні додаткові цільові функції. Результати моделювання *показують, що збільшення кількості процесорів призводить до зменшення максимального завантаження і кількості міграцій, збільшення кількості завдань підвищує навантаження на систему та кількість міграцій на початкових етапах, механізм міграції ефективно балансує навантаження, особливо на початкових етапах виконання.*

Напрямами наступних досліджень є деталізація вбудованих пристроїв і їх класифікація. Для кожного класу вбудованих пристроїв необхідно буде адаптувати алгоритми та метод оптимізації завдань, а також розробляти цільову функцію оптимізації.

Ключові слова: багатопроцесорні системи, вбудовані системи, комп'ютерні системи, оптимізація

Introduction

Task optimization in multiprocessor embedded systems is gaining increasing significance in the development of embedded devices across various industries. Despite significant progress, several research challenges still require in-depth analysis and effective solutions.

One of the key challenges is ensuring the reliability of embedded systems, as they often operate in safety-critical environments. Although their functional capabilities are well-defined at the design stage, guaranteeing continuous real-time operation remains a major challenge. It is essential to ensure not only the correctness of results but also their timely execution, necessitating the development of new approaches to resource management in multiprocessor systems.

Another challenge is the need to maintain strict real-time guarantees. Unlike general-purpose systems (e.g., personal computers), which offer more flexible functionality but lack high reliability and predictability, embedded systems must adhere to stringent real-time constraints. Optimizing task scheduling processes to account for these constraints remains an area of active research.

Furthermore, embedded systems can be categorized into control systems and streaming systems, each with distinct data processing requirements. Control systems respond to external events, requiring rapid reaction times and minimal latency. In contrast, streaming systems handle continuous data flows, demanding high throughput and efficient processing of large data volumes. Developing universal approaches that optimize performance for both types of systems remains a critical challenge for researchers and engineers.

In conclusion, research in task optimization for multiprocessor systems holds significant potential for advancement, particularly in the context of embedded devices. Future efforts should focus on addressing reliability issues, real-time guarantees, and efficient resource management, ultimately leading to the development of safer and more efficient systems.

Related works

Optimization of tasks in multiprocessor systems [1, 2], considering the application of computer systems in various fields, has become particularly significant in the context of embedded device development. Most embedded systems have the following characteristics: they are designed to perform a clearly defined set of functional capabilities known at the design stage; they must be reliable, as they often operate in safety-critical environments; and they must provide strict real-time guarantees, meaning their output must not only be correct but also produced within specific time constraints. These characteristics [3, 4] distinguish embedded systems from general-purpose systems, such as personal computers, which offer much greater flexibility in terms of functionality but place significantly less emphasis on guarantees and reliability in real-time operation.

Embedded systems can be categorized into two groups [5, 6] based on the type of functionality they provide: control systems wait for incoming events (or signals) from the environment and respond accordingly, making them suitable for applications such as industrial automation; streaming systems process a continuous, potentially infinite flow of data from the environment, and are commonly used in applications such as audio and video processing.

Given the complexity [7] of task optimization in multiprocessor systems, we will focus on embedded streaming systems as the object of our research. Examples of streaming applications include audio/video encoding and decoding, signal processing, computer vision, medical imaging, navigation systems, security camera systems, and many others.

Complex embedded systems [8, 9], such as those controlling autonomous vehicle driving, actually consist of multiple subsystems that interact with each other. In the case of autonomous driving, some of these subsystems fall into the category of streaming systems, while others belong to control systems.

For example, self-driving cars collect vast amounts of data in the form of continuous streams from onboard cameras and LiDAR sensors. These data streams must be constantly updated and processed to perform motion planning, which involves determining the optimal path and speed for the vehicle, and collision avoidance, which entails detecting and evading unexpected obstacles. These decisions are made by streaming subsystems [10] and then transmitted to control subsystems, which execute actions such as steering, braking, or accelerating the vehicle.

Autonomous vehicle control systems implement motion planning and collision avoidance algorithms [11], which require extremely high levels of interaction. Additionally, these algorithms must produce results within a short and predictable timeframe to ensure the vehicle can quickly respond to external events. For instance, if a pedestrian suddenly crosses the street in front of the vehicle, it must stop as quickly as possible. The high complexity of these algorithms, combined with the need for rapid execution, presents a challenge for system designers to achieve high performance. This requirement is common across many modern embedded systems.

In fact, embedded systems have consistently demonstrated a growing demand for increased performance over the years. Until the mid-2000s, most computing systems were built using single-processor architectures [12], and this increasing performance demand was met by enhancing the computational power of a single processor. However, performance gains between successive generations of single processors began to slow significantly in the early 2000s, primarily due to [13, 14]: diminishing returns from new processor architecture improvements, a very slow increase in clock frequencies due to power leakage issues, and the growing gap between processor and memory speeds.

As a result, to further improve system performance, semiconductor manufacturers shifted their research and development efforts toward multiprocessor architectures starting in the mid-2000s. This technological trend [15, 16], which has influenced both general-purpose and embedded systems, remains promising. Indeed, an increasing number of architectures, proposed by both research institutions and industry, feature a growing number of processing elements. Today, embedded system designers frequently integrate multiple processors, memory units, interconnects, and other hardware peripherals into a single chip, forming what is known as a multiprocessor system-on-chip (MPSoC).

The design of embedded MPSoC [17] is a process that involves multiple stages. It begins with defining the required system functionality using an application model along with the specification of the execution platform on which the application will run. After a series of refinement stages, the design process is completed when a detailed description of the system's hardware and the software running on each processor is obtained.

Design trends, such as the widespread adoption of multiprocessor architectures and scalable interconnections, introduce new design methodologies aimed at achieving high system performance on a multiprocessor architecture. However, ensuring high system performance is not the sole objective of embedded system developers [20].

The term "system adaptability" [18, 19] refers to a system's ability to adjust to changing environmental conditions. These conditions are represented by parameters that can be classified into two categories [21, 22]. The first category includes application-related parameters that affect how the program is executed [23, 24]. For instance, the resolution of a video decoding application is typically defined by two parameters specifying the frame height and width. The second category consists of parameters describing the state of the execution platform. For example, a parameter may indicate the number of active processors in the system. Achieving system adaptability in response to changes in the second set of parameters those describing the execution platform's state is a promising direction for development.

The system is powered by a battery, and as the battery charge depletes, the user may choose to disable a certain number of components to reduce the system's power consumption. This may lead to a decrease in the application's quality of service, such as a reduced video encoding speed. Additionally, this scenario requires the system to redistribute tasks between those that will be disabled and those that will remain active, meaning that task mapping must be adjusted during execution.

System adaptability can be implemented in a computing system in various ways. In the case of embedded multiprocessors, applications are typically scheduled by the operating system, which acts as an intermediate layer. The OS serves as an interface between the application layer and the hardware layer, which resides at the bottom of the stack [25, 26].

Depending on the nature of the tasks and performance requirements, different approaches to migration management can be chosen. Flexible systems with adaptive control are well-suited for scenarios where resilience to changes and unexpected failures is crucial, while deterministic schemes are beneficial for highly regulated environments, such as flight control systems or mission-critical industrial applications. In any case, the choice of method depends on efficiency requirements, predictability, and the overall architecture of the hardware platform.

The objective function for task optimization in multiprocessor systems with migration.

Task optimization in multiprocessor systems with migration requires the development of an objective function that considers all relevant parameters to maximize resource utilization efficiency and ensure timely task execution. The key parameters to take into account include task execution time, migration delays, processor load, energy consumption, task priorities, and adherence to predefined deadlines. It is essential to minimize the overall completion time of all tasks, maintain load balancing across processors, and reduce application overhead associated with task migration. Additionally, optimization should consider energy consumption, as excessive resource utilization leads to increased power costs.

The objective function for such optimization can be defined as follows:

$$F = \min\left[\alpha \sum_{i=1}^{N} C_i + \beta \sum_{m=1}^{M} L_m + \gamma \sum_{k=1}^{K} E_k + \delta \sum_{i=1}^{J} T_j\right], \quad (1)$$

where C_i – task completion time i; L_m – the load factor of processor m, defined as the ratio of processor busy time to the total execution time; E_k – the energy consumption of processor k during task execution; T_j – the total migration time for task j, which accounts for data transfer delays and the setup time of the execution environment on the new processor; $\alpha, \beta, \gamma, \delta$ – weight coefficients that determine the relative importance of each parameter in the objective function.

The goal of optimization is to minimize the objective function F, which balances task completion time, processor load, energy consumption, and migration costs. The weight coefficients can be adjusted according to the system's characteristics and optimization objectives. For example, in real-time systems, priority may be given to minimizing task completion time and meeting deadlines, whereas in systems with limited energy resources, minimizing energy consumption becomes the primary focus. Optimization of such an objective function can be performed using dynamic programming methods, evolutionary algorithms, or machine learning techniques for adaptive parameter tuning in real time.

In multiprocessor systems with migration, the number of tasks affecting the optimization process is a crucial parameter, as it determines the complexity of load balancing, migration costs, and overall execution time. Let N be the total number of tasks to be scheduled, and M be the number of processors in the system. The model assumes that each task i has its own execution time C_i, it can migrate between processors, causing delays T_i, and consume energy E_i during execution.

The objective function should minimize the total completion time of all tasks, balance the load across processors, and minimize energy consumption and migration costs. The refined objective function, considering these parameters, is defined as follows:

$$F = \min[\alpha \sum_{i=1}^{N} C_i + \beta \sum_{m=1}^{M} L_m + \gamma \sum_{i=1}^{N} E_i + \delta \sum_{i=1}^{N} T_i], \tag{2}$$

де $C_i = S_i + W_i + M_i$ – task completion time i, which consists of waiting time S_i, execution time W_i and migration time

M_i; $L_m = \frac{\sum_{i=1}^{N} W_{im}}{\sum_{i=1}^{N} C_i}$ – CPU load m, calculated as the ratio of the execution time of all tasks on the processor m to the total execution time; $E_i = P_i \cdot W_i$ – task energy consumption i, which depends on the processor's power P_i and execution time

W_i; $T_i = \sum_{k=1}^{K} t_{ik}$ – total migration time for the task i, where t_{ik} – task migration time i to the processor k; $\alpha, \beta, \gamma, \delta$ – weight coefficients that determine the relative importance of each parameter in the optimization objective function.

When the number of tasks significantly exceeds the number of processors in an embedded multiprocessor system, optimizing task distribution becomes much more complex due to several key factors. First, it is necessary to consider not only load balancing but also other constraints, such as energy consumption, memory limitations, real-time requirements, and interprocessor delays during data transfer.

As the number of tasks increases, several aspects of optimization become critically important. Migration time costs: With a large number of tasks, frequent migrations can significantly increase overall time costs due to the additional overhead of transferring states between processors. This may lead to situations where the benefits of load balancing are offset by the additional migration costs. Therefore, there is a need to determine the optimal migration points and minimize the migration frequency.

As the number of tasks increases, the optimization algorithm faces exponential growth in the solution space, i.e., the complexity of planning increases. A simple greedy algorithm, which works effectively for a small number of tasks, loses efficiency due to an incomplete exploration of possible configurations. In such cases, a promising approach is to use stochastic optimization methods, such as genetic algorithms, particle swarm algorithms, or machine learning, to predict the optimal distribution.

Energy efficiency becomes crucial as the number of tasks grows, causing the overall energy consumption of the system to increase as well. For embedded systems, this is critical since they often have limited power resources (e.g., batteries). In this context, energy consumption must also be optimized through dynamic voltage and frequency scaling, in combination with task migration.

With the increase in the number of tasks, the likelihood of interdependencies between them grows. This means that migrating one task may introduce delays for others, especially if tasks frequently exchange data. In such cases, the optimal distribution must take into account the topology of the dependencies between tasks, minimizing interprocessor communication costs.

In real systems, tasks can have a dynamic nature, appearing, completing, or changing their priority, meaning the algorithm must exhibit adaptability. This requires adaptive optimization approaches that can respond to such changes in real-time. A promising approach is to use reinforcement learning methods, which allow the algorithm to learn the optimal task distribution based on historical data.

A large number of tasks creates additional load on the cache and local memory of the processors. This can lead to frequent cache misses and increased memory access latency. In such cases, an effective strategy is to consider data locality when migrating tasks, i.e., placing tasks in such a way that they work with local data, minimizing data transfer between processors.

Although the replication approach for task migrations is effective for fast task migration, it shows limited efficiency when the number of tasks significantly exceeds the number of processors. This creates a need for hybrid approaches that combine replication with other strategies, such as process spawning or dynamic scaling. For example, replication can be used for short tasks, while process spawning is more suitable for long, energy-intensive processes.

Another promising direction is the use of multi-agent systems, where each processor is considered an agent that autonomously makes decisions based on local information and communication with other agents. This approach helps avoid a central scheduler, which can become a bottleneck in a system with a large number of tasks.

It is also important to consider the heterogeneity of processors in modern embedded systems. For example, many systems use a combination of high-performance and energy-efficient processors. Optimization in such systems should take into account not only the load but also the energy efficiency of each type of processor.

Thus, there is a need for further improvement and development of adaptive, energy-efficient, and scalable algorithms that can optimally distribute a large number of tasks while considering the constraints of embedded multiprocessor systems. An important direction is the integration of machine learning methods for load prediction and real-time adaptation of optimization algorithms.

Process migration algorithms in multitasking embedded systems

Process migration algorithms in multitasking embedded systems are aimed at optimally distributing computational tasks across multiple processors or cores. The main goal of migration is to balance the load, minimize task execution time, reduce energy consumption, and ensure real-time compliance for critical tasks. Embedded systems have limited resources, so efficient process migration management is particularly important. Depending on the task distribution approach and migration management, algorithms can be divided into several categories: global, cluster-based, hierarchical, and hybrid.

Global algorithms treat all processors as a common set of resources. Any process can run on any processor, and migration occurs dynamically based on the load. Key approaches include load balancing, load distribution, and dynamic load distribution. Load balancing algorithms move processes between processors to achieve uniform load. They use current metrics such as processor utilization or the number of process queues. Examples of such algorithms include the "Least Loaded Processor" algorithm, where tasks are moved to the processor with the least load, and "Round Robin Migration," which distributes tasks in turn among all processors.

Load distribution algorithms assign tasks to free processors at the moment of their creation, rather than during execution, which reduces application overhead for migration but may lead to uneven load distribution. Dynamic load distribution uses real-time system state monitoring to dynamically migrate tasks. An example is the "Work Stealing" algorithm, where processors with low loads steal tasks from overloaded processors, ensuring a more even distribution of tasks.

Cluster migration algorithms divide the system into groups of processors (clusters), and migration occurs within the cluster. This reduces data transfer delays, as clusters often share common cache memory or have faster data exchange. Tasks are scheduled within a single cluster, and if the load is not balanced, task migration between clusters is possible. Additionally, affinity-based migration algorithms are used, which aim to keep tasks on the processors where they were previously executed to minimize cache misses and application migration overhead.

Hierarchical algorithms combine global and cluster-based methods. The system is considered at multiple levels: at the top level, migration occurs between clusters; at the lower level, migration happens within the cluster. This structure allows for a compromise between the flexibility of global algorithms and the efficiency of local ones. An example is the "Multilevel Queue Migration" algorithm, where tasks are grouped by priorities, and migration occurs within each queue.

Hybrid algorithms combine multiple approaches to achieve a balance between efficiency and complexity. They can dynamically switch between different strategies depending on the current system state.

The main challenges for migration algorithms in embedded systems are limited resources, real-time requirements, low energy consumption, and minimizing delays. Effective optimization involves minimizing application overhead for migration, as task movement requires time for data transfer and reinitialization of the execution environment, considering cache affinity since process migration between cores can cause cache misses, which negatively affect performance. It also involves adapting to dynamic changes in load and energy consumption, using machine learning methods to predict load and dynamically adjust scheduling parameters.

The goal is to minimize execution time and energy consumption while maintaining balanced load across processors. We define the objective function as follows:

$$\min\left(\max_{p\in P} T_p\right) + \alpha \textstyle\sum_{p\in P} E_p, \tag{3}$$

where T_p – the total execution time of tasks on a processor p, E_p – the energy consumption of the processor p, α – a coefficient that determines the weight of energy consumption in the objective function.

We will develop optimization algorithms for various parameters in embedded multiprocessor systems.

Algorithm 1 takes into account a ring topology and performs task migration between neighboring processors to achieve load balancing.

Step 1.1. Initialization. Tasks are distributed among processors in a random order.

Step 1.2. Load evaluation. For each processor p, the load is calculated $L_p = \sum C_i$ for all tasks on it.

Step 1.3. Balancing. If the load difference between neighboring processors exceeds a threshold, Δ, migration of a task is performed from the more loaded processor to the less loaded one. The migration occurs in a ring, reducing transfer delays.

Step 1.4. Update the load and repeat Steps 2–3 until the load is balanced.

Step 1.5. Execute tasks within the time quantum Q.

Step 1.6. Check if all tasks are completed. If all tasks are done, finish the process; otherwise, return to Step 2.

Algorithm 2 takes into account task priorities and the possibility of migration between neighboring processors, maintaining processor affinity to reduce cache misses.

Step 2.1. Initialization. Assign tasks to processors based on priority. Tasks with higher priority are assigned first.

Step 2.2. Execute tasks during the time quantum Q. If the task is not completed, it remains in the processor's queue.

Step 2.3. Performance evaluation. If a high-priority task cannot be completed on time on the current processor, migration is performed to a neighboring processor with a lower load.

Step 2.4. Affinity check. Migration is allowed only if the task has not been executed previously on the selected processor to minimize cache misses.

Step 2.5. Repeat Steps 2–4 until all tasks are completed.

Algorithm 3 is aimed at minimizing energy consumption while considering the ring topology and time quantum.

Step 3.1. Initialization. Distribute tasks among processors, taking into account their energy consumption. Tasks with the lowest energy consumption are assigned first.

Step 3.2. Energy consumption evaluation. Calculate the current energy consumption for each processor.

Step 3.3. Optimization. If the energy consumption exceeds the threshold E_{max}, migration is performed to a neighboring processor with lower energy consumption.

Крок 3.4. Execution of tasks during the time quantum Q.

Step 3.5. Update energy consumption and repeat Steps 2–4 until all tasks are completed.

The application of these algorithms enables efficient load balancing, optimal energy consumption, and adherence to real-time constraints in multitasking embedded systems.

Migration algorithms in multiprocessor embedded systems are widely used in various industries where there is a need to efficiently utilize computational resources, minimize task execution time, and optimize energy consumption. Let's consider specific application examples for the three developed algorithms.

Algorithm 1 is particularly effective in systems with a ring topology, such as network routers and multiprocessor computing clusters. In telecommunications networks, routers process data packets, and the traffic volume can dynamically change. In such systems, routers are often organized in a ring topology to ensure fault tolerance and reduce data transmission delays. When one router receives an excessive number of packets, and neighboring routers have available computational resources, task migration for packet processing occurs to less-loaded routers. This ensures load balancing, minimizes data processing delays, and prevents overloading individual routers. As a result, network performance improves, and data transmission latency is reduced.

Algorithm 2 is effective in real-world systems with critical time constraints, such as automotive embedded systems or UAV (unmanned aerial vehicle) control systems. In modern vehicles, numerous electronic control units are responsible for various tasks, such as braking systems, course stabilization, object recognition using cameras, etc. These tasks have different priorities for instance, the braking system has the highest priority. If the processor responsible for processing the braking signal is overloaded with lower-priority tasks (such as object recognition), lower-priority tasks are migrated to neighboring processors with lighter loads. High-priority tasks are executed immediately, ensuring that real-time constraints are met (for example, minimizing the delay in activating the brakes). This improves vehicle safety and ensures the reliable operation of critical systems.

The algorithm is suitable for embedded systems where energy consumption is critical, such as in portable devices or Internet of Things (IoT) systems. Consider a smart watch with multiple processors performing various tasks such as heart rate monitoring, GPS navigation, message processing, and more. Tasks have different energy requirements depending on the complexity of the computations. High-energy tasks, such as GPS navigation processing, can be executed on processors with higher energy efficiency. If one processor becomes overloaded and its energy consumption exceeds the acceptable level, tasks are migrated to a neighboring processor with lower energy consumption. This ensures optimal load distribution with minimal overall energy consumption, extending the device's battery life.

In server clusters for processing large volumes of data, migration algorithms are used for dynamic task distribution between processors to minimize execution time and reduce energy consumption. In Unmanned Aerial Vehicle (UAV) control systems, the algorithm ensures the adherence to real-time constraints for critical tasks (such as flight stabilization and obstacle avoidance), while less critical computations (such as video processing) can migrate to other processors.

Embedded systems in medical devices (e.g., pacemakers) can use Algorithm 3 to ensure continuous operation under limited energy consumption, migrating computational tasks between processors to optimize energy usage.

To achieve system adaptability through process migration, it is also necessary to determine how to transition between the current mapping and the next one. In other words, a mechanism for performing process migration must be anticipated. In the proposed approach, this mechanism is implemented by middleware, specifically through process migration. Let's consider the problem of defining and implementing a process-oriented migration mechanism that meets three requirements: if the migration process is initiated, it must be completed within certain, known deadlines, ensuring predictability; process migration can be triggered at any time in the system, meaning the mechanism must account for scenarios where process migration is required in response to a hardware failure, the occurrence of which is unknown; the code used for process migration must be automatically generated, without manual developer intervention, to relieve developers from the labor-intensive and error-prone task of manually inserting the necessary migration code.

The predictable process migration mechanism allows processes to be redistributed during execution across processors, which is a fundamental requirement for system adaptability. The uniqueness of the solution lies in the fact that by using the operational semantics and structure of the process, migration can actually begin at any point during the execution of the main process without needing to move large states. Furthermore, the upper bound for application costs of process migration can be determined based on the topology and buffer size. Finally, the code used for process migration is minimally invasive to the original code structure and can be fully automated in generation.

Resource management during execution is a well-studied topic in the scheduling of general-purpose distributed systems. Specifically, within this context, mechanisms for process migration must be developed and assessed, enabling dynamic load balancing, fault tolerance, and improving system administration and data access locality. In recent years, execution-time management has become increasingly popular and is also being applied in multiprocessor embedded systems. This domain imposes strict constraints such as cost, power, and predictability, all of which need to be carefully considered by execution-time and process migration management mechanisms.

The proposed migration approach will use fully distributed memory with no direct remote memory access. This means that the task processing element can only directly access its own local memory. All communication and synchronization between processes mapped to different processors can occur only through messaging. The approach for implementing system adaptability involves deploying application processes modeled for their execution time redistribution to adapt the system to changing operating conditions, such as variations in quality of service requirements, resource availability, or power budget constraints. Specifically, system adaptability is supported by using specialized middleware within the software stack. At the top of the software stack, applications are specified as a set of processes implemented as separate threads. An example of a thread representing a process. The basic process structure will be modified to facilitate the implementation of the predictable process migration mechanism. At the bottom of the software stack, the operating system is responsible for all types of process management (creation, deletion, priority setting, suspension, or resumption). These functions are important for managing the system's execution time, particularly for process migration execution. Furthermore, each processor has multitasking capabilities provided by the OS. In the case of a "many-to-one" mapping, where more than one process is mapped to a single processing element, scheduling is governed by data. This means that the process continues its successive iterations until it blocks on a read or write. When the process blocks, it passes control to the next process in the ready queue using a round-robin system. Between the applications and the operating system, there will be middleware consisting of two main components. The first is the communication mechanism, which implements communication and synchronization between processes located on separate processors. A mandatory component is process migration, which is mainly responsible for actions performed during process migration: coordinating the creation and deletion of processes across different processors; ensuring the correct transfer of the process state during migration.

Thus, algorithms for process migration in multitasking embedded systems have been developed, and their application enables efficient load balancing, optimal energy consumption, and adherence to real-time constraints in multitasking embedded systems.

The method of task optimization in embedded systems with multiple processors

The task optimization method will be based on replication during task migration in a multiprocessor system, and the optimization will be carried out using a developed optimization objective function that takes into account the main system parameters, such as task execution time, processor load, memory usage, and communication delays. Optimizing task execution using the replication method involves finding the distribution of processes among processors that minimizes the total task execution time while maintaining load balancing on the processors and minimizing communication costs between them. The optimization objective function is defined as follows:

$$F = \alpha \sum_{i=1}^{N} T_i + \beta \sum_{j=1}^{M} L_j + \gamma \sum_{k=1}^{P} C_k, \tag{4}$$

where T_i — execution time of the i-th task, L_j — load j-th processor, C_k — message passing delay between processors; α, β and γ — weight coefficients that reflect the optimization priorities.

The first component of the objective function aims to minimize the total execution time of all tasks. To achieve this, it is necessary to ensure that tasks are executed as parallel as possible and do not have to wait for resources. The replication method enables quick task switching between processors, as task code copies are already stored in the local memory of all potential processors. Optimizing this component involves dynamically moving tasks to the least loaded processors, as well as minimizing task idle time.

The second component of the objective function is responsible for load balancing between processors. An uneven distribution of tasks can lead to some processors being overloaded while others remain idle. To address this issue, a dynamic load balancing strategy is used, which involves regularly assessing processor load and migrating tasks from overloaded processors to less loaded ones. The replication method significantly simplifies this process since task code is already available on all processors, meaning only the task state needs to be transferred. This reduces the task migration time and ensures quick task resumption.

The third component of the objective function minimizes communication delays between processors. This is especially important for tasks that actively interact with each other. To achieve this, interrelated tasks need to be placed on processors that have minimal communication latency. Optimizing this component involves analyzing the system’s topology and processor placement, as well as calculating communication costs between processors. To minimize delays, dependency graphs between tasks are used to reflect the volume and frequency of message exchanges. Tasks that frequently exchange data are placed on neighboring processors with minimal communication latency.

The optimization method for implementing the objective function consists of several stages. The first stage is the initial distribution of tasks across processors. This can be done using a greedy algorithm, which sequentially assigns tasks to the least loaded processor. The second stage is the dynamic migration of tasks during execution, based on the analysis of the current load on processors and the volume of communication between them. If a processor is found to be overloaded, some tasks are migrated to other processors, taking into account the minimization of communication delays. The third stage involves periodic optimization of task distribution by evaluating possible task placement combinations and selecting the one that minimizes the objective function.

To find the optimal task distribution, combinatorial optimization is used, as the number of possible task distributions between processors increases exponentially with the number of processors and tasks. An effective approach is the use of genetic algorithms or simulated annealing, which provide a quick search for the optimal solution within the space of possible distributions. The genetic algorithm simulates the process of natural selection, using crossover and mutation operations to generate new solutions based on the current ones. Simulated annealing makes random changes to the current task distribution, gradually reducing the probability of accepting worse solutions, which helps avoid local minima of the objective function.

Thus, the replication method combined with the optimization objective function ensures the efficient execution of tasks in a multiprocessor system, minimizing overall execution time, ensuring load balancing between processors, and reducing message exchange delays. This approach is particularly effective for systems with high load dynamics and complex computational node topologies.

The optimization method for task execution in a multiprocessor system using replication and an objective function can be presented in detailed steps. This approach allows dynamic task redistribution between processors to minimize total execution time, balance the load, and minimize message exchange delays. Each step of this method aims to achieve the optimal task distribution within the system, maintaining flexibility and speed in the migration process due to pre-created task code replicas.

Step 1. System initialization and task code replication at the first stage, the following actions are performed: the total number of tasks and the computational resources of the system, including the number of processors and their capacities, are analyzed; for each task, code replicas are created on all processors that are potentially capable of executing the task, ensuring that processors are ready to accept tasks without additional time costs for code transfer; a global state table is created, which stores information about the execution status of each task on all processors, as well as local tables on each processor to track the task execution locations. Additionally, the optimization objective function is initialized, with its value computed according to formula (5).

Step 2. Initial Task Distribution. The initial task distribution is performed between processors based on a greedy algorithm. Tasks are assigned to processors with the least current load. The system's topology is considered to minimize communication delays between processors. Tasks with high interaction levels are placed on neighboring processors. A target function is calculated for the initial task distribution, which includes execution time, processor load, and communication delays. The initial placement is fixed in a global state table.

Step 3. Task Execution and System State Monitoring. Tasks are launched on the corresponding processors selected in the previous step. During execution, the state of each task is regularly saved in the global state repository, including the values of registers, instruction pointer, call stack, and local variables. Processor load and message exchange volumes between processors are monitored. Regular monitoring of the target function values is carried out to assess the efficiency of the current task distribution.

Step 4. Decision on Task Migration. If the target function exceeds predefined threshold values, the migration process is initiated. The current load on each processor and communication delays are analyzed. Bottlenecks such as overloaded processors or high communication delays between tasks that are actively exchanging data are identified. Tasks for migration are selected, particularly those with low data locality or those generating significant volumes of messages to remote processors.

Step 5. Selection of the Target Processor for Migration. An evaluation of all potential processors for executing the migrating task is performed based on the following criteria: current processor load; communication delays with other processors executing related tasks; available free memory and computational core availability. The processor that minimizes the target function, balancing load and communication overhead, is chosen. Resources are reserved on the selected processor to avoid conflicts during migration.

Step 6. Task Migration Execution. The task is paused on the current processor, and its state is fully saved in the global state repository. Since the task's code is already on the new processor, only the task's state is transferred. On the target processor, the task is restored from the saved state. The local state tables on both processors are updated, as well as the global state table to reflect the new task location. The task continues execution from the point where it was paused.

Step 7. Target Function Update and Optimization. After the migration is executed, the target function is recalculated considering the new task distribution. If the target function has not reached its minimum value, the migration process is repeated for other tasks. Optimization is performed using one of the following methods: the genetic algorithm executes crossover and mutations to create new task distribution combinations; the simulated annealing method makes random changes to the task distribution, gradually reducing the probability of accepting worse decisions. The optimization process continues until the target function reaches the specified minimum or a stable task distribution is achieved.

Step 8. Optimization Completion and Load Balancing Maintenance. The optimized distribution is fixed in the global state table. Dynamic load balancing is maintained by periodically updating the target function and reconfiguring the task distribution. Data consistency and state synchronization between processors are ensured.

Thus, the detailed method of task execution optimization using replication in a multiprocessor system effectively minimizes total execution time, ensures load balancing, and minimizes communication delays. A key feature of the method is the implementation of task migration according to replication using the optimization target function.

Research on the Effectiveness of the Task Optimization Method in Multiprocessor Embedded Systems

A program in C++ has been developed to implement the task optimization method using replication and a target optimization function in a multiprocessor embedded system. The program takes into account:

1) the number of processors and tasks;
2) the migration scheme (based on the target function);
3) the state of processes (whether they are running or waiting);
4) the types of tasks (long or short).

The program implements:

1) initialization of the system with a specified number of processors and tasks;
2) initial task distribution among processors;
3) system state monitoring and dynamic task migration to optimize the target function.

The program implements the task optimization method using replication and a target optimization function in a multiprocessor embedded system. The main features of the program include:

1) initialization of the system with random tasks (short or long);
2) initial task distribution among processors based on their load;
3) dynamic optimization with task migration for load balancing;
4) output of the migration process in the console.

Using the developed program, the first series of experiments was conducted. Figure 4.1 shows the graphs of the optimization parameters for a simple case with processes and processors, meaning there is no increased load level in the multiprocessor embedded system.

Program results:

1) initial processor load state - [12, 14, 14, 11];
2) final state after optimization - [12, 14, 14, 11].

The graph shows the dynamics of processor load during optimization. There was no significant redistribution since the initial distribution was relatively balanced.

Figure 4.2 illustrates the graphs of the optimization parameters for a simple case with processes and processors, meaning there is no increased load level in the multiprocessor embedded system. Unlike the first series of experiments, the dynamics of processor load and energy consumption of processors are highlighted.

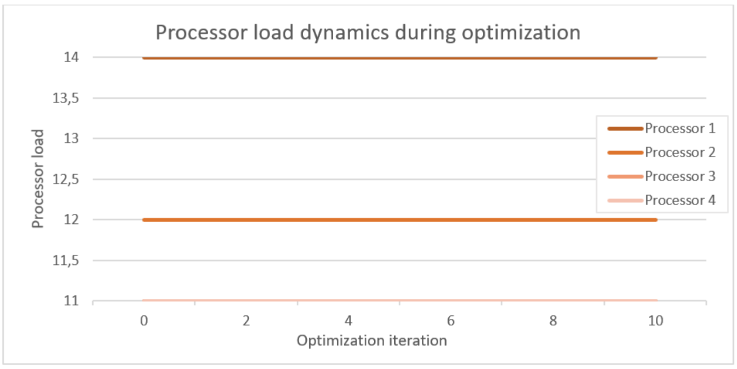

Fig. 1. Results of the first series of experiments

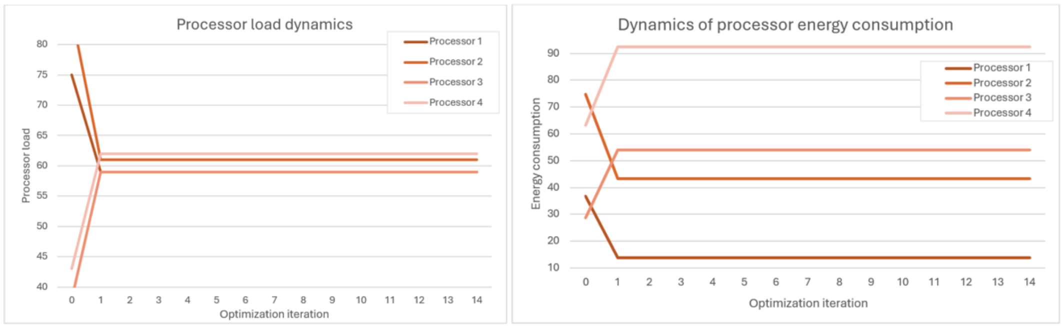

Fig. 2. Results of the second series of experiments

Experiment results:

1) initial processor load - [75, 85, 38, 43];
2) final load after optimization - [59, 61, 59, 62];
3) initial energy consumption: [36.85, 74.71, 28.71, 63.20];
4) final energy consumption: [13.78, 43.34, 54.03, 92.31].

Experiment analysis: Initially, the load was intentionally unbalanced to test the effectiveness of the optimization method. The algorithm performed task migration, balancing the load across processors, as evidenced by the final state where all processors have approximately the same load. However, energy consumption became less uniform, as tasks with higher energy consumption remained on certain processors, while other processors received shorter but more energy-intensive tasks.

In the graphs, the left part shows how the processor load levels out over time, while the right part illustrates the dynamics of energy consumption, which was not a criterion for optimization, so uniformity was not achieved here.

This experiment demonstrates that the chosen optimization method effectively balances the load, but additional target functions are needed for optimizing energy consumption.

Conclusions

The developed task execution optimization method using replication in a multiprocessor system effectively minimizes the overall execution time, ensures load balancing, and reduces communication delays. A key feature of the method is the implementation of task migration according to replication using an optimization target function.

The conducted experiment with the system demonstrated that the chosen optimization method effectively balances the load, but additional target functions are needed for optimizing energy consumption.

Simulation results show that increasing the number of processors leads to a reduction in maximum load and the number of migrations. Increasing the number of tasks increases the system load and the number of migrations during the initial stages, but the migration mechanism effectively balances the load, especially in the early stages of execution. Future research directions include further detailing embedded devices and their classification. For each class of embedded devices, algorithms and task optimization methods will need to be adapted, and an optimization target function will need to be developed.

References

1. Zhou Y., Zhang E., Guo H., Fang Y., Li H. Lifting path planning of mobile cranes based on an improved RRT algorithm. *Adv. Eng. Inform*. 2021, 50, 9. https://doi.org/10.1016/j.aei.2021.101376

2. Zhu A.M., Zhang Z.Q., Pan W. Crane-lift path planning for high-rise modular integrated construction through metaheuristic optimization and virtual prototyping. *Autom. Constr*. 2022, 141, 21. https://doi.org/10.1016/j.autcon.2022.104434

3. Guo H., Zhou Y., Pan Z., Zhang Z., Yu Y., Li Y. Automated Selection and Localization of Mobile Cranes in Construction Planning. *Buildings* 2022, 12, 580. https://doi.org/10.3390/buildings12050580

4. Wang J., Zhang Q., Yang B., Zhang B. Vision-Based Automated Recognition and 3D Localization Framework for Tower Cranes Using Far-Field Cameras. *Sensors* 2023, 23, 851. https://doi.org/10.3390/s23104851

5. Huang L., Pradhan R., Dutta S., Cai Y. BIM4D-based scheduling for assembling and lifting in precast-enabled construction. *Autom. Constr*. 2022, 133, 14. https://doi.org/10.1016/j.autcon.2021.103999

6. Song Y., Xin R., Chen P., Zhang R., Chen J., Zhao Z. Autonomous selection of the fault classification models for diagnosing microservice applications. *Future Generation Computer Systems*, 2024. 153, pp.326-339. https://doi.org/10.1016/j.future.2023.12.005

7. Tao L., Lu X., Zhang S., Luan J., Li Y., Li M., Li Z., Yu Q., Xie H., Xu,R., Hu C. Diagnosing Performance Issues for Large-Scale Microserv ice Systems With Heterogeneous Graph. *IEEE Transactions on Services Computing.* 2024. https://ieeexplore.ieee.org/document/10533869

8. Chen Y., Xu D., Chen N., Wu X. FRL-MFPG: Propagation-aware fault root cause location for microservice intelligent operation and maintenance. *Information and Software Technology*. 2023. 153, p.107083. https://doi.org/10.1016/j.infsof.2022.107083

9. Li X., Wen P., Chen P., Chen J., Wen X., Xia Y. An effective parallel convolutional anomaly multi-classification model for fault diagnosis in microservice system. *Software Quality Journal*. 2024. Pp.1-18. https://doi.org/10.21203/rs.3.rs-5267111/v1

10. Mazraemolla Z.P., Rasoolzadegan A. An effective failure detection method for microservice-based systems using distributed tracing data. *Engineering Applications of Artificial Intelligence.* 2024. 133, p.108558. https://doi.org/10.1016/j.engappai.2024.108558

11. Zhang S., Jin P., Lin Z., Sun Y., Zhang B., Xia S., Li Z., Zhong Z., Ma M., Jin W., Zhang, D. Robust failure diagnosis of microservice system through multimodal data. *IEEE Transactions on Services Computing*. 2023.16(6), pp.3851-3864. https://doi.org/10.48550/arXiv.2302.10512

12. Bedratyuk L. and Savenko O., The star sequence and the general first Zagreb index, *MATCH Communications in Mathematical and in Computer Chemistry*. (2018) 79, 407–414. https://doi.org/10.48550/arXiv.1706.00829

13. Zhang B., Wang X., Wang H. Virtual machine placement strategy using cluster-based genetic algorithm. *Neurocomputing.* 2021. 428, Pp. 310–316. https://doi.org/10.1016/j.neucom.2020.06.120

14. Wei P., Zeng Y., Yan B., Zhou J., Nikougoftar E. Vmp-a3c: virtual machines placement in cloud computing based on asynchronous advantage actor-critic algorithm. *J. King Saud Univ. Comput. Inf. Sci.* 2023. 35, 101549. https://doi.org/10.1016/j.jksuci.2023.04.002

15. Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S. Automated functional and robustness testing of microservice architectures. *Journal of Systems and Software.* 2024. 207, p.111857. https://doi.org/10.1016/j.jss.2023.111857

16. Yin J., Li J., Fang Y., Yang A. Service scheduling optimization for multiple tower cranes considering the interval time of the cross-tasks. *Math. Biosci. Eng.* 2023, 20. Pp. 5993–6015. https://www.aimspress.com/article/doi/10.3934/mbe.2023259

17. Zhang Z.Q., Ma S.L., Jiang X.Y. Research on Multi-Objective Multi-Robot Task Allocation by Lin-Kernighan-Helsgaun Guided Evolutionary Algorithms. *Mathematics* 2022, 10, 4714. https://doi.org/10.3390/math10244714

18. Li K., Duan T., Li Z., Xiahou X., Zeng N., Li Q. Development Path of Construction Industry Internet Platform: An AHP—TOPSIS Integrated Approach. *Buildings* 2022, 12, 441. https://doi.org/10.3390/buildings12040441

19. Lysenko S., Bobrovnikova K., Savenko O., Kryshchuk A. BotGRABBER: SVM-Based Self-Adaptive System for the Network Resilience Against the Botnets' Cyberattacks. *Communications in Computer and Information Science*. 2019. Vol. 1039. Pp.127-143, ISSN: 1865-0929. https://doi.org/10.31891/2307-5732-2024-331-2

20. Lysenko S., Savenko O., Bobrovnikova K., Kryshchuk A. Self-adaptive system for the corporate area network resilience in the presence of botnet cyberattacks. *Communications in Computer and Information Science*, 2018.- 860, - Pp. 385-401. https://link.springer.com/chapter/10.1007/978-3-319-92459-5_31

21. Savenko O., Sachenko A., Lysenko S., Markowsky G., Vasylkiv N. (2020). BOTNET DETECTION APPROACH BASED ON THE DISTRIBUTED SYSTEMS. *International Journal of Computing*, *19*(2), 190-198. https://doi.org/10.47839/ijc.19.2.1761

22. Kashtalian A., Lysenko S., Savenko O., Nicheporuk A., Sochor T., & Avsiyevych V. Multi-computer malware detection systems with metamorphic functionality. *Radioelectronic and Computer Systems*. 2024. Vol. 1. Pp. 152-175. doi: https://doi.org/10.32620/reks.2024.1.13 \

23. Savenko B., Kashtalian A., Lysenko S., Savenko O. Malware Detection By Distributed Systems with Partial Centralization. *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Dortmund, Germany, 2023, pp. 265-270, doi: 10.1109/IDAACS58523.2023.10348773. https://ieeexplore.ieee.org/document/10348773

24. Chong G., Ramiah H., Yin J., Rajendran J., Wong W.R., Mak P.-I., Martin R.P. CMOS cross-coupled differential-drive rectifier in subthreshold operation for ambient RF Energy Harvesting—Model and analysis. IEEE Trans. Circuits Syst. II Express Briefs 2019, 66, 1942–1946. https://ieeexplore.ieee.org/document/8630669

25. Kumar S., Gupta U., Singh A.K., Singh A.K. Artificial Intelligence: Revolutionizing Cyber Security in the Digital Era. J. Comput. Mech. Manag. 2023, 2, 31–42. https://doi.org/10.57159/gadl.jcmm.2.3.23064

26. Chen J., Zhang R., Chen P., Ren J., Wu Z., Wang Y., Li X., Xiong L. MTG_CD: Multi-scale learnable transformation graph for fault classification and diagnosis in microservices. *Journal of Cloud Computing.* 2024. 13(1), p.103. https://doi.org/10.1186/s13677-024-00666-0

Dmytro Martiniuk **Дмитро Мартинюк**	master's degree student, Khmelnytskyi National University, Khmelnytskyi, Ukraine, e-mail: martiniyuk.dim14@gmail.com https://orcid.org/0009-0002-3524-872X	Магістрант, Хмельницький національний університет, м. Хмельницький, Україна
Oleksii Lyhun **Олексій Лигун**	PhD student, Khmelnytskyi National University, Khmelnytskyi, Ukraine e-mail: oleksii.lyhun@gmail.com https://orcid.org/0009-0004-5727-5096	Аспірант, Хмельницький національний університет, м. Хмельницький, Україна
Andriy Drozd **Андрій Дрозд**	PhD student, Khmelnytskyi National University, Khmelnytskyi, Ukraine, e-mail: andriydrozdit@gmail.com https://orcid.org/0009-0008-1049-1911	Аспірант, Хмельницький національний університет, м. Хмельницький, Україна
Oleksii Besedovskyi **Олексій Бесєдовський**	Candidate of Sciences in Economics, Associate Professor, Associate Professor of the Information Systems Department, Simon Kuznets Kharkiv National University of Economics, Kharkiv, e-mail: oleksii.besedovskyi@hneu.net https://orcid.org/0000-0002-9161-4061	Кандидат економічних наук, доцент, доцент кафедри інформаційних систем Харківського національного економічного університету імені Семена Кузнеця, м. Харків