International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Mathematical Modelling of Engineering Problems
Vol. 12, No. 3, March, 2025, pp. 971-981
Journal homepage: http://iieta.org/journals/mmep

Fuzzy Dynamic Load Balancing for Real-Time Systems on Heterogeneous Multiprocessors

Ridha Mehalaine*, Djamel Nessah, Meriem Djezzar, Mounir Hemam

ICOSI Laboratory, Computer Science Department, Abbes Laghrour University, Khenchela 4000, Algeria

Corresponding Author Email: r_mahalaine@univ-khenchela.dz

https://doi.org/10.18280/mmep.120323

Received: 26 December 2024
Revised: 14 February 2025
Accepted: 20 February 2025
Available online: 31 March 2025

Keywords:
load balancing, real-time systems, scheduling, multiprocessor, earliest deadline first algorithm

ABSTRACT

The large market of real-time systems and the rapid evolution of processing capabilities to ensure compliance with time limits or constraints have motivated researchers to maximize the computing power, which makes the distribution and balancing of load between the different resources a major problem in these systems. The objective of any load balancing technique is to optimize the utilization of processors by specifying the locality of tasks. In this paper, we propose a new dynamic load balancing approach based on a real-time scheduling algorithm where tasks are distributed across three heterogeneous processor classes based on the average load of each class and the task deadline using a proposed fuzzy approach, which significantly reduces the energy consumption by assigning tasks with far too low deadlines to processor classes. The objective of the presented work is to distribute the load among processors fairly while respecting the time and energy constraints. In terms of load balancing our proposed approach gives very good results for strict periodic tasks in a multiprocessor real-time system with a significant reduction in energy consumption, while respecting task deadlines.

1. INTRODUCTION

With the increase in machine power, real-time systems were almost exclusively developed using dedicated systems. This operating system, attracting real-time users by its many advantages such as ease of development and freedom of extension of the system [1]. Real-time systems are crucial in applications where timing is critical and where failure to meet timing requirements can lead to serious consequences [2]. A real-time operating system (RTOS) must respect temporal constraints. RTOS is a multitasking operating system intended for real-time applications. An RTOS generally uses a scheduling algorithm specific to real-time systems, in order to provide developers with the ability to produce applications with deterministic and predictable behavior in the final system. On a multiprocessor machine there are two types of processors: those specialized for real-time that only execute real-time tasks and those that execute all non-real-time tasks. However, resource management in this type of infrastructure obviously poses much more complex problems than those posed by traditional systems. The problem of obtaining an optimal distribution of tasks to processors in a system is very complex and is well known to be NP-complete. The problem of load balancing aims to ensure that no processor is underloaded or overloaded and to establish a uniform load on all processors. It consists in taking advantage, in the best way, of the possibilities of using resources; in other words, load balancing must maintain an equivalent load on all processors. Our objective is to propose a real-time scheduling algorithm on multiprocessor machines that meets the needs of load balancing. We are trying to provide a new solution that is not penalizing for time and energy.

Load balancing of tasks in real-time systems is a major problem for industrial and academic research. Users are increasingly demanding solutions that offer their real-time applications features that allow them to run faster and work to reduce their energy consumption. These needs generate challenges to meet important constraints (deadline, energy, load balancing).

Load balancing ensures a uniform distribution of needs between different processors in order to generate more computing power. Load balancing techniques allow both to optimize the response time for each task, while avoiding unevenly overloading the processors. Load balancing was broadly classified into two categories, namely static load balancing and dynamic load balancing.

The load balancer always seeks to address a specific problem. Among others, the hardware architecture on which the algorithms will operate, the nature of the tasks that will be executed, the algorithmic complexity that we allow ourselves, the energy consumption or the tolerance of errors that we agree must be taken into account. A compromise must therefore be found to meet the need for load balancing, but existing standard load balancing solutions are very penalizing in terms of performance, do not have a complete analysis and evaluation of the algorithms designed to manage a mixture of real-time tasks, including periodic, aperiodic and sporadic tasks. In addition, they can easily fail in a dynamic and adaptive context, which makes their adoption a question that is not always obvious.

2. RELATED WORK

The following section presents the existing body of related work, providing a background for the current research effort.

Alam and Varshney [3] proposed a dynamic load balancing strategy for a homogeneous multiprocessor system and apply it on a cube-based interconnect network called Folded Crossed Cube network. The experimental results show that a lower load imbalance factor has been achieved as well as the execution time. Parallel tasks are solved with the largest number of tasks. As the number of tasks increases, the execution time decreases with a lower load imbalance factor.

Efficient architecture for running thread (EARTH) is a multithreaded programming and execution model that supports fine-grained and non-preemptive threads in a distributed memory environment. In this paper, the authors present the load balancing strategies for the runtime of a multithreaded system. They describe the design and implementation of a set of dynamic load balancing algorithms and study their performance in divide-and-conquer, regular, and irregular applications. The experimental study on the distributed memory multiprocessor IBP SP-2 indicates that a random load balancer performs as well as, and often better than, history-based load balancers [4].

Tan et al. [5] proposed a high-performance, real-time, dynamic multicore load balancing method for microkernel operating system. It has been implemented and tested on a microkernel operating system named Mginkgo. The results show that in case of load imbalance in the system, load balancing can be performed automatically so that all processors in the system can try to achieve the maximum throughput and resource utilization.

A real-coded genetic algorithm has been proposed by Panwar et al. [6] to balance the load on each processor. To achieve the specified objective, a dual function is used here; first, the fitness function is used to reduce the execution time, while the second is used to maximize the load on the individual processor. The proposed algorithm is tested on a total of 12 problems from the literature as well as three additional benchmark problems. The analysis shows that the proposed algorithm is efficient compared to what was previously known.

Nirmala and Girijamma [7] proposed a hybrid genetic algorithm (HGA) combined with a stochastic development process to designate and order real-time tasks with priority requirements, the proposed algorithm works on the CPU utilization, the CPU queue length and the distance to its current load as linguistic inputs while framing the fuzzy set. The proposed algorithm has been evaluated with similar existing methods to prove its efficiency. The results prove that FLLBHGATS outperforms other techniques with respect to the quality of the solution.

Ali and Suleman [8] presented various dynamic load balancing algorithms including round-robin, minimal connections, and weighted load balancing in real-time scenarios. This paper explores the implementation of dynamic load balancing strategies that adapt to different workloads in real-time, thereby improving user experience while minimizing latency and resource waste. By using adaptive techniques that consider both current workloads and future demand forecasts, distributed systems can achieve more balanced load distribution, resulting in improved performance and user satisfaction.

The works that use static load balancing are often insufficient in real-time systems, leading to inefficient resource utilization. Dynamic load balancing addresses these challenges by continuously monitoring system performance metrics such as CPU utilization, energy consumption and redistributing tasks across processors based on real-time data, ensuring that no processor is overloaded while others remain underutilized. Most of the existing works do not consider very important constraints such as energy consumption and timing constraints while solving the load balancing problem. And in real-time systems, both of these constraints play a very important role for the system efficiency. In order to guarantee the processing of uncertain information (task arrival, expected completion date, etc.) of tasks across a set of tasks that run on multiple processors and satisfy certain constraints, the use of fuzzy logic is more than necessary.

3. LOAD BALANCING

3.1 Components of a load balancing system

Since the balancing problem is a relatively old problem, many approaches have been proposed to solve it in different platforms. In computer systems and networks, load balancing algorithms [9-11] are methods used to distribute tasks or workloads across multiple resources (such as processors, servers, or network links) of a system to manage a fair amount of work. Load balancing allows distributing a set of tasks of a parallel program among the different processors of a multiprocessor system.

To improve the system performance, load balancing optimizes the utilization of processors by specifying the locality of tasks through optimal decomposition, which allows distributing work fairly among the different processors to reduce the average response time and idle time.

A load balancing system as shown in Figure 1 is composed of two essential elements: policies and mechanisms. Mechanisms physically realize the distribution of the load and provide the information required by policies, while policies consider the set of choices to be made to distribute a workload [12].

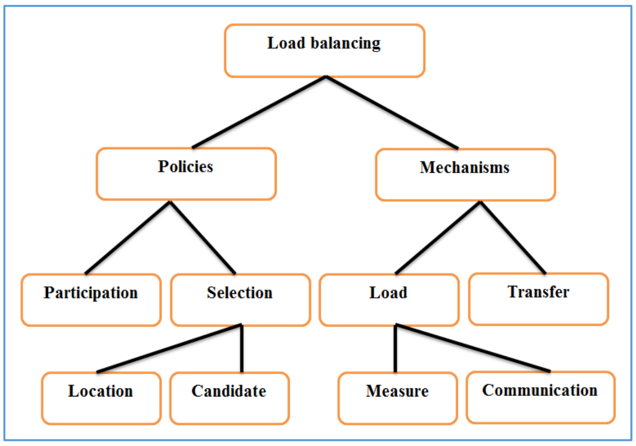


Figure 1. Components of a load balancing system [12]

A load balancing mechanism can be described by two essential elements: a system load state manager, a control and decision-making element, and a load transfer element. The load state of the different processors is the main source of information for balancing techniques. Considering the exchange of statistical data and the processors must provide additional resources for communication, collecting information on the load state of each processor causes additional costs. Load balancing algorithms can be classified in different ways; two main classifications are:

• **Static vs. dynamic approach:** In static load balancing the assignment of tasks to processors is determined before the execution of the program based on some predefined criteria, such as the size, complexity or priority of the tasks, or the deadline, availability or location of resources, and remains fixed throughout its execution. The information about the execution time of tasks and the dynamic characteristics of the processors are assumed to be known a priori. Static load balancing solves many problems (e.g. those caused by the heterogeneity of processors) for most applications that have regular, predictable and homogeneous workloads and resources, because it can reduce the overhead and complexity of load balancing. The transient load due to multiple users on a network of workstations requires a dynamic approach to balance the load [13]. During execution, the static approach cannot adapt to changes in workload or resource conditions, such as failures that also require prior knowledge of the workload and resource characteristics, which may not be available or accurate. The dynamic approach, is a technique in which the assignment of tasks to processors is performed during execution; the assignment of tasks to processors is decided based on information that is collected about the load state of the system. The decision-making process is based on real-time measurements of the current state of the system and the characteristics of the tasks. This helps to improve the execution performance of tasks. Dynamic load balancing is more responsive, more flexible, and more robust.

• **Homogeneous and heterogeneous:** resources are homogeneous in terms of capacity, which simplifies the estimation of the execution time of tasks. Homogeneous load balancing algorithms are designed to evenly distribute computational or network loads across multiple resources that are the same or similar in terms of processing capacity. Unlike homogeneous load balancing, heterogeneous load balancing algorithms take into account variations in processing power, memory, or other attributes among the available resources.

The goal of the participation policy is to determine whether a processor is in an appropriate state to participate in the migration of a task as a source (overloaded processor) or as a receiver (underloaded processor). The selection policy determines the different unbalanced processors. There are three classes of selection policy: systematic policies based on theoretical results, cause work exchanges in a given order and designate processors alternately. A threshold-based policy where a processor compares its load with one or more thresholds. Depending on the result of this comparison, the processor will become a load transmitter or receiver. A comparison policy can also be used, where the processor adopts a behavior depending on the state of the set or a subset of processors [14].

3.2 Load assessment

More precisely, load balancing algorithms can be distinguished by the Information Update Policy which attempts to obtain a state on the system by collecting information on the load state of the different processors (the main source of information for balancing techniques). The first goal of load measurement is to estimate the amount of processing allocated for each processor [15]. We distinguish 3 classes of policies:

• Periodic policies: in this case the information is collected regularly and it is stored either centrally or distributed on a set of processors.

• On-demand policies: the information is assembled and sent each time a processor needs the information.

• State change policies: transmit an update of the information because they pass from one remarkable state to another [14].

A load index is associated with each processor. The chosen load index must be easily representable, by a number if it represents a measured state or a logical level of load if it is compared with a threshold. Several indices are used to evaluate the load of a processor such as: the length of the queue of a processor, number of tasks processed, the execution time of the tasks currently running and the average response time of the tasks.

3.3 Architecture of a balancing system

Schopf [16] proposed phases to follow to design and develop a load balancing strategy in a system. As shown in the Figure 2, the proposed scheme is composed of three main phases: resource discovery, which generates a list of potential resources; gathering information about these resources and selecting candidate resources to participate in a balancing; and task execution, which includes storing and cleaning files.

3.4 Properties of a load balancer

The way a load balancer improves task execution is by moving a task to another processor or not moving it at all. To distinguish the quality of different load balancers, there are three main properties, which we seek to improve when modifying a load balancer [14].

1). Efficiency: When there is work available, processors should spend the minimum amount of idle time. The main quality of a load balancer should be able to use the machine's power to its maximum.

2). Fairness: At a given time, tasks with the same priorities should be assigned the same execution time. This property can consist of comparing the execution times of two tasks with the same priority, if fairness is good, they are always very close to each other [14].

3). Locality: Tasks that have a particular affinity for one or more processors should be assigned to them first.

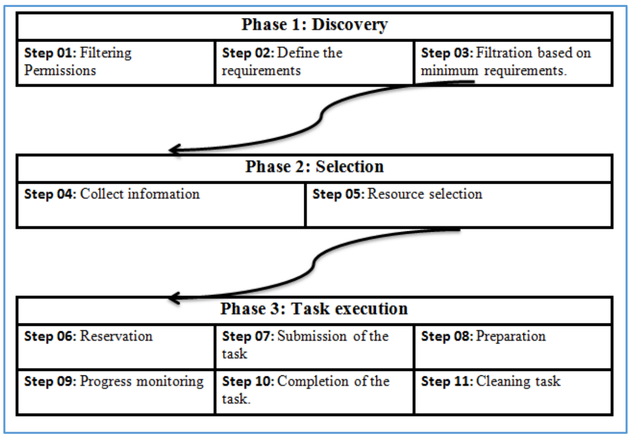

Phase 1: Discovery
Step 01: Filtering Permissions
Step 02: Define the requirements
Step 03: Filtration based on minimum requirements.
Phase 2: Selection
Step 04: Collect information
Step 05: Resource selection
Phase 3: Task execution
Step 06: Reservation
Step 07: Submission of the task
Step 08: Preparation
Step 09: Progress monitoring
Step 10: Completion of the task.
Step 11: Cleaning task

Figure 2. Architecture of a balancing system [16]

4. REAL-TIME SYSTEMS

Generally, current systems must respect temporal constraints. A system is called real-time when the data it collects and processes remains relevant for a well-defined duration. These systems are mainly used in the field of process control, where the execution of programs must be completed before a specific deadline. In other words, a system is real-time if it is able to respect temporal deadlines [17].

Real-time scheduling algorithm is an algorithm that can provide a sequence of the work performed by the processor(s); if the task deadlines are respected, the sequence is considered valid. Single-processor and multiprocessor scheduling algorithms are presented in this subsection [18].

• "Rate-monotonic" (RM): Scheduling algorithm: is a static method that gives the highest priority to the task with the highest frequency (i.e., the shortest period). The disadvantage of this algorithm is that it is only used for periodic tasks with deadlines on request.

• "Inverse deadline" (ID): In this static scheduling algorithm, the highest priority is given to the task with the shortest deadline. Inverse Deadline is a method where the priorities of tasks are determined based on their respective deadlines.

• "Least laxity first" (LLF): The "least laxity first" algorithm grants, at time t, the highest priority to the task with the smallest laxity. The calculation of the laxities of the tasks can be done in two ways: The laxity of a task represents its maximum deviation from its deadline to (re)start its execution, when the task is executed alone.

• "Earliest deadline first" (EDF): Grants, at time t, the highest priority to the task with the earliest deadline. The EDF* notation is used to denote the EDF algorithm, where among the tasks with the same deadline, the one that comes first is selected [19].

If the tasks of our system can be executed on several available processors then the scheduling is of the multiprocessor type. Real-time systems must respect their constraints, particularly in strict real-time systems. Thus, the scheduling algorithm must be shared the tasks between the processors in an equitable manner to meet the load balancing needs. Our objective is to provide a new solution for load balancing, non-penalizing in terms of time constraint with minimized energy consumption. To achieve this objective in this work we must propose a real-time scheduling algorithm on a multiprocessor machine that estimates the workload of a system and distributes the load equally between the processors that are either overloaded or underloaded using a load balancing scheme.

5. MULTIPROCESSOR CONCEPTS AND ARCHITECTURE

The computational need of some applications, such as mechanical computing, image processing, is increasing even faster, there is a technological limit on the speed of processors; If the workload cannot be satisfactorily handled by one processor, the solution may be to apply multiple processors (parallelism is an attempt at an answer that is still relevant today). The use of multiple processors generates design considerations that must be taken into account for satisfactory operations and performance.

A multiprocessor is a type of computer system that contains several processing units (processors) in a single machine. Running a program on a multiprocessor architecture is a difficult problem, because it involves deciding which task will be executed by which processor. The distribution of tasks on the different processors is a bigger problem. This can lead to a problem called: load imbalance. The different combinations of design solutions and trade-offs give rise to a wide variety of architectures (hardware and software) of multiprocessor systems (Figure 3).

There are two main types of multiprocessor architectures:

• Symmetric multiprocessor (SMP): is a simplest type of multiprocessor architecture in which all processors have equal access to memory and other resources. In this type of systems, all processors can perform any task and the operating system is responsible for distributing tasks equally among them. But it can be difficult to scale to a large number of processors.

• Asymmetric multiprocessors (AMP): is a more difficult type of multiprocessor architecture in which each processor has its own local memory and access to remote memory is slower. These systems are used in specific tasks where a more powerful processor is required for a specific task.

To ensure that processors work efficiently and in a coordinated manner multiprocessor systems use synchronization and coordination techniques to avoid conflicts. Multiprocessors are used in a wide variety of applications, they are common in web servers and databases, where multiple processors can handle a large number of requests simultaneously. They are also used from industrial process control systems to high-end supercomputers.

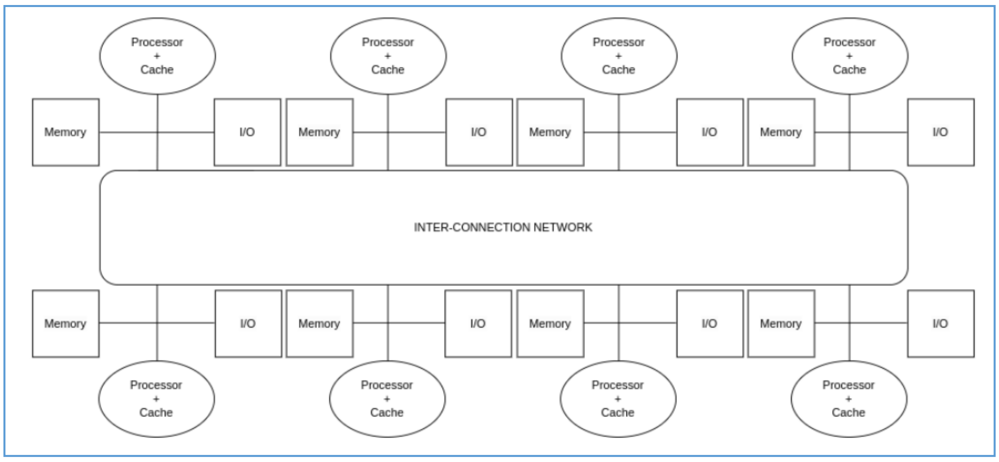


Figure 3. Multi-processor architecture

6. RESULTS AND DISCUSSIONS

6.1 The proposed approach

Our approach explores the implementation of dynamic load balancing strategies that adapt to real-time systems. Figure 4 presents the proposed approach to solve the load balancing problem for real-time systems. The proposed dynamic load balancing continuously monitors system performance metrics, such as CPU utilization, energy consumption, and allows the system to redistribute workloads across available processors based on real-time data, ensuring that no processor is overloaded while others remain underutilized. The paper presents a dynamic load balancing algorithm that uses the EDF* real-time scheduling algorithm to schedule tasks in 03 different queues; namely, the high-speed processor queue HSPQ, the medium-speed processor queue MSPQ, and the low-speed processor queue LSPQ. The assignment of tasks to queues is done by the proposed fuzzy approach based on the task deadline and the utilization factor of each processor category. We examine the impact of our approach on system performance metrics such as response time, throughput and energy consumption. By using adaptive techniques that take into account workloads to achieve a more balanced load distribution.

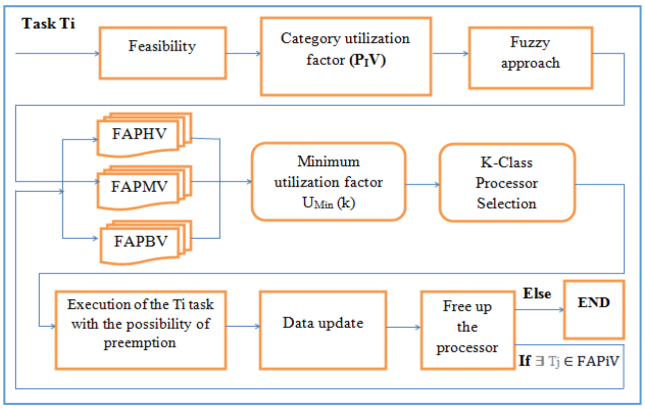

Task Ti
Feasibility
Category utilization factor (P_IV)
Fuzzy approach
FAPHV
FAPMV
FAPBV
Minimum utilization factor U_{Min} (k)
K-Class Processor Selection
Execution of the Ti task with the possibility of preemption
Data update
Free up the processor
Else
END
If ∃ Tj ∈ FAPiV

Figure 4. The proposed approach

In this section, fuzzy logic is used to balance the load in order to share the load equally among the processors and to minimize the energy consumption in the real-time task scheduling algorithm for the multiprocessor environment. The proposed model expresses the different steps to be followed to solve the load balancing problem for multiprocessor real-time systems. The model we propose is based on the load balancing mechanism to make it capable of meeting all the constraints. Once the scheduling is triggered, it complies with the scheduling policy, which defines which resources to assign in priority to a given task.

This paper proposes a dynamic load balancing algorithm that addresses the common problems that can reduce the efficiency of a multiprocessor system. These problems are response time, failure rate and processor utilization.

(a) Response time: This is the time required for a task to be processed by the system. A good load balancer should reduce this time, ensuring that critical tasks are executed in the required time.

(b) CPU utilization: This metric measures how much CPU resources are being used. An effective balancer should maximize this utilization while avoiding overloading certain processors.

(c) Task failure rate: This indicates the percentage of tasks that fail to be processed in the expected time. A high failure rate may signal poor load management and requires adjustments in the balancing mechanism.

6.2 Assumptions and considerations

Some considerations and assumptions are made regarding the multiprocessor system for which the algorithm is designed. The main assumptions characterizing this multiprocessor system:

the use of a system with homogeneous processors is not always the case in reality, but most of the current systems use heterogeneous processors to meet the needs of our system or the needs of users, which allows us to propose an approach on a system with heterogeneous processors (at the characteristic point the maximum execution speed).

in reality most of the systems are multiprocessor systems (heterogeneous) which process dependent or independent tasks, periodic or aperiodic and in order not to complicate our approach by several parameters, we assumed that our system processes only independent tasks, periodic or aperiodic. In the context of real-time systems running on multiprocessor architectures, load balancing is crucial to ensure optimal performance, minimize response times, and maximize the utilization of processor resources. In this paper, the multiprocessor system is assumed to be configured using heterogeneous processors.

Migrating a task to another processor makes the cache contents invalid for the first processor and the cache of the second processor must be repopulated. When executing a task on a specific processor, the most recently accessed data of the task is stored in the processor cache and successive memory accesses of the task are often satisfied in the cache. Due to the high cost of cache invalidation and repopulation, most SMP systems try to avoid process migration from one processor to another and try to keep a process running on the same processor.

Unlike many existing works that use static methods, in our proposed approach, we used dynamic techniques that adjust the distribution of running tasks, often in response to load variations, to balance the load across different processors. This can increase the efficiency of the system, but requires careful management to avoid frequent interruptions and that can dynamically react to load changes. Dynamic load balancing algorithms have shown particular promise, allowing optimized redistribution of tasks in response to load variations.

A single load manager monitors and distributes tasks across processors. This provides a holistic view of the system, which facilitates optimization, and to avoid inconsistencies and inefficiencies if communications between processors are not well managed.

In our multiprocessor system, multiple tasks must be executed on multiple processors, each with specific timing constraints. The scheduling of these tasks is crucial to ensure that all tasks meet their deadlines (hard real-time system), especially in critical applications such as embedded systems or real-time control systems.

6.3 Task modeling

Each task will have a set of properties, including its period, execution time, and deadline.

- *ri*: The wake-up date *ri* of the kth instance: *ri*=*r*0+*i*P*
- *Pi*: Period
- *Di*: Critical delay
- *Di*: The deadline of the ith instance:

$$di = ri + D = r0 + i^{*}P + D \tag{1}$$

- *Ci*: induced load (execution time)

It is clear that Ci depends on the processor used, so the duration of each instruction must be configurable. In the following we denote a real-time task system composed of n tasks Ti (i=1…n) by {T1(r1,C1,D1,P1), T2(r2,C2,D2,P2), ..., Tn(rn,Cn,Dn,Pn)} or more briefly by {Ti(ri,Ci,Di,Pi)}i = 1..n.

Study period: If H is the length of such an interval, then the program in $[0, H]$ is the same as that in $[kH, (k+1)]$ for any integer k>0. This is the minimum time interval after which the program repeats itself. For a set of periodic tasks activated synchronously at time t=0, the study period is given by the least common multiple of the periods: $H = P(T1, \ldots, Tn)$.

6.4 Feasibility

The CPU utilization factor *U* is the fraction of CPU time spent on the task set execution. Since *Ci/Pi* is the fraction of CPU time spent on task *i* execution, the utilization factor for *n* tasks is given by:

$$U = \sum_{i=0}^{n} Ci/Pi \tag{2}$$

Processor heterogeneity: allows to integrate multiple processor types and computing units within our system to achieve optimized performance and efficiency. In such an environment, various processors collaborate to execute various computational tasks. The essence of processor heterogeneity lies in its ability to distribute workloads based on the strengths of each processor type. In our proposed approach, we group processors according to their execution speed into 03 different categories. Each processor class excels at handling specific types of tasks: high-speed (PHV) processors are well suited for tasks with close deadlines, medium-speed (PMV) processors for processing tasks with medium-term deadlines, and low-speed (PBV) processors for tasks with long-term deadlines. This distribution improves performance, as tasks are processed faster and more efficiently by the most suitable processors. In addition, it improves energy efficiency by reducing the computational load on less suitable processors, thereby reducing energy consumption. If the number of processors in a category I is defined by the term nbrPr (PIV), then the utilization factor of a category of processors $U(PIV)$ is given by:

$$U(PIV) = (\sum_{i=0}^{n} Ci/Pi)\,/nbrPr(PIV) \tag{3}$$

To assign a task Ti to a free processor Pr(j) of category k that belongs to the set {PHV, PMV, PBV} we must calculate UMin (k) which represents the processor of category k with a minimal utilization factor.

UMin (k)=Prj knowing that

$$U(\Pr j) = Min\{U(\Pr i), \forall i \in K\} \tag{4}$$

The EDF algorithm is a dynamic scheduling rule that selects tasks according to their absolute deadlines. More precisely, tasks with closer deadlines will be executed at higher priorities.

The arrival of a task Ti in the system is an event that requires a feasibility test to ensure that the insertion of this task does not exceed the capacity of all the processors available in our system.

According to the reference [20], the feasibility test is defined by a sufficient condition and a second necessary condition. if these two conditions are verified, we ensure the existence of a real-time scheduling of the tasks.

$$\begin{gathered}\sum_{i=0}^{n} Ci/Pi \leq 1 \\ \sum_{i=0}^{n} Ci/Di \leq 1\end{gathered} \tag{5}$$

When our system consists of R processors and N concurrent periodic tasks with individual timing constraints, the operating system must guarantee that each periodic instance is regularly activated at its own pace and completed within deadlines. The feasibility test becomes as follows:

$$\sum_{i=0}^{n} Ci/Pi \leq R \tag{6}$$

6.5 The fuzzy approach

In our proposed approach, each task is inserted into one of the existing queues (queue of high-speed processors (HSPQ), queue of medium-speed processors (MSPQ), queue of low-speed processors (LSPQ)) according to the following fuzzy inference rules (Figure 5):

1)- If Di=Soon and U(PHV)≠Hight then insert Ti into HSPQ

2)- If Di=Soon and U(PHV)=Hight and U(MSPQ)≠Hight then insert Ti into MSPQ

3)- If Di=Soon and U(PHV)=Hight and U(MSPQ)=Hight and U(LSPQ)≠Hight then insert Ti into LSPQ

4)- If Di=Soon and U(PHV)=Hight and U(MSPQ)=Hight and U(LSPQ)=Hight then insert Ti in HSPQ

5)- If Di=medium and U(PMV)≠Hight then insert Ti in MSPQ

6)- If Di=medium and U(PMV)=Hight and U(PHV)≠Hight then insert Ti in HSPQ

7)- If Di=medium and U(PMV)=Hight and U(PHV)=Hight and U(PBV)≠Hight then insert Ti in LSPQ

8)- If Di=medium and U(PMV)=Hight and U(PHV)=Hight and U(PBV)=Hight then insert Ti in MSPQ

9)- If Di=far and U(PBV)≠Hight then insert Ti in LSPQ

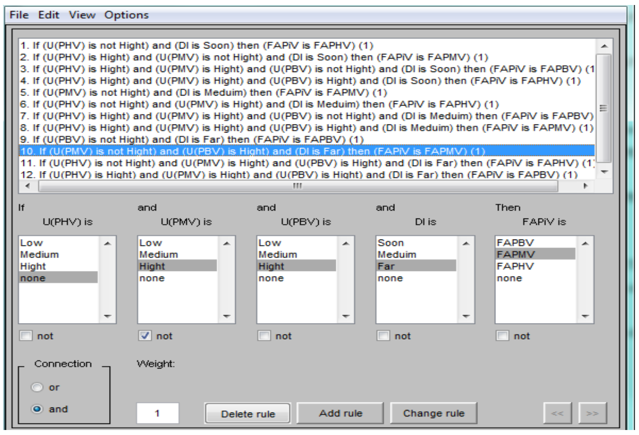
10)- If Di=far and U(PBV)=Hight and U(PMV)≠Hight then insert Ti in MSPQ

11)- If Di=far and U(PBV)=Hight and U(PMV)=Hight and U(PHV)≠raise then insert Ti into HSPQ

12)- If Di=far and U(PBV)=Hight and U(PMV)=Hight and U(PHV) = raise then insert Ti into LSPQ

Fuzzy control plays a major role in solving problems that involve inaccurate and uncertain information. As shown in Figure 6, the proposed fuzzy approach uses 04 input variables and one output variable: U(PHV), U(PMV), U(PBV), task deadline Di and the output variable FAPiV which represents the 03 categories of queues (HSPQ, MSPQ and LSPQ). Fuzzification is the transformation of numerical inputs Xi into a set of membership values in the interval [0, 1] in corresponding fuzzy sets. Fuzzification can be seen as a conversion of real variables into fuzzy variables (also called linguistic variables) defined on a representation space related to the inputs. The number of membership functions to be defined for each language variable is defined using human expertise [21]. The rules are formulated on the expert's knowledge of the system. These rules express the relationship between the fuzzy input sets and the corresponding fuzzy control sets. This representation space is normally a fuzzy subset.

In our approach, each task is described with a 4tuple: Ti=Ti(ri,Ci,Di,Pi) where each linguistic variable corresponds to the triplet (X, T(X), U); X is a variable (U(PHV), U(PMV), U(PBV), Di). T(X) is the range of values of the variable and U is a finite or infinite set of fuzzy subsets. Figure 7 shows the fuzzification of the input variable U(PHV) that models the utilization factor of the high-speed processor category (PHV). It is presented as follows: X=U(PHV), T(X)=[0, 1], U={Hight, medium, low}.

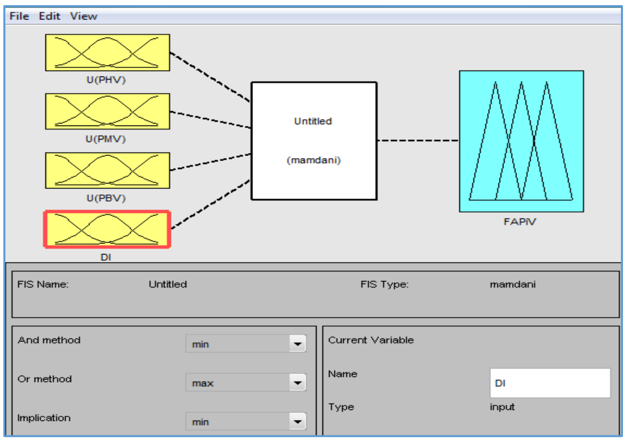

1. If (U(PHV) is not Hight) and (DI is Soon) then (FAPiV is FAPHV) (1)
2. If (U(PHV) is Hight) and (U(PMV) is not Hight) and (DI is Soon) then (FAPiV is FAPMV) (1)
3. If (U(PHV) is Hight) and (U(PMV) is Hight) and (U(PBV) is not Hight) and (DI is Soon) then (FAPiV is FAPBV) (1
4. If (U(PHV) is Hight) and (U(PMV) is Hight) and (U(PBV) is Hight) and (DI is Soon) then (FAPiV is FAPHV) (1)
5. If (U(PMV) is not Hight) and (DI is Meduim) then (FAPiV is FAPMV) (1)
6. If (U(PHV) is not Hight) and (U(PMV) is Hight) and (DI is Meduim) then (FAPiV is FAPHV) (1)
7. If (U(PHV) is Hight) and (U(PMV) is Hight) and (U(PBV) is not Hight) and (DI is Meduim) then (FAPiV is FAPBV)
8. If (U(PHV) is Hight) and (U(PMV) is Hight) and (U(PBV) is Hight) and (DI is Meduim) then (FAPiV is FAPMV) (1)
9. If (U(PBV) is not Hight) and (DI is Far) then (FAPiV is FAPBV) (1)
10. If (U(PMV) is not Hight) and (U(PBV) is Hight) and (DI is Far) then (FAPiV is FAPMV) (1)
11. If (U(PHV) is not Hight) and (U(PMV) is Hight) and (U(PBV) is Hight) and (DI is Far) then (FAPiV is FAPHV) (1
12. If (U(PHV) is Hight) and (U(PMV) is Hight) and (U(PBV) is Hight) and (DI is Far) then (FAPiV is FAPBV) (1)

If U(PHV) is	and U(PMV) is	and U(PBV) is	and DI is	Then FAPiV is
Low	Low	Low	Soon	FAPBV
Medium	Medium	Medium	Meduim	FAPMV
Hight	Hight	Hight	Far	FAPHV
none	none	none	none	none

Connection: or [✓ and]

Weight: 1

Figure 5. Fuzzy inference rules

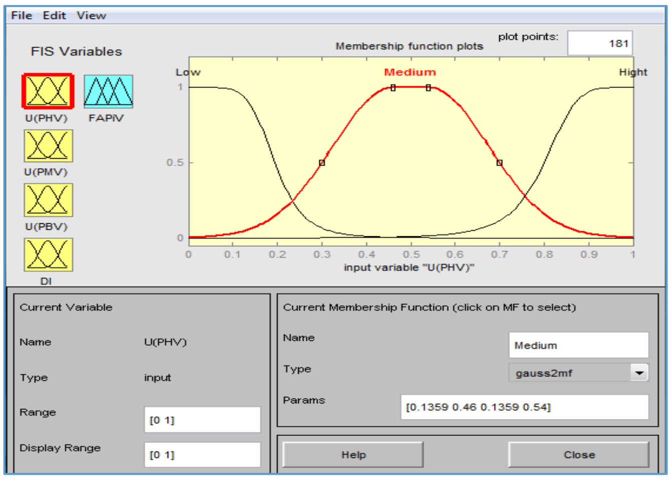


FIS Name: Untitled
FIS Type: mamdani
And method: min
Or method: max
Implication: min
Current Variable
Name: DI
Type: input

Figure 6. The approach fuzzy

We have chosen 03 linguistic values {High, medium, low} of the input variable U(PHV) to express the utilization rate of the high-speed processor category that allow us to decide to which queue our task should be assigned according to the linguistic values, knowing that if the utilization factor U(PHV) is high then we avoid assigning the task Ti to their queue. Our approach uses fuzzy inference rules that use 04 fuzzy input variables and one output variable, based on the idea of assigning the task to the queue reserved for processors with the lowest utilization factor considering the task deadline. i.e. avoid assigning a task with a very close deadline, to the category of processors with medium or low speed to ensure the execution of this task before their deadline.

Figure 8 shows the structure of the decision space of our fuzzy approach. The decision space shows how these decisions influence each other. To evaluate the proposed algorithms, we performed simulations under MATLAB 7.9.0.529.

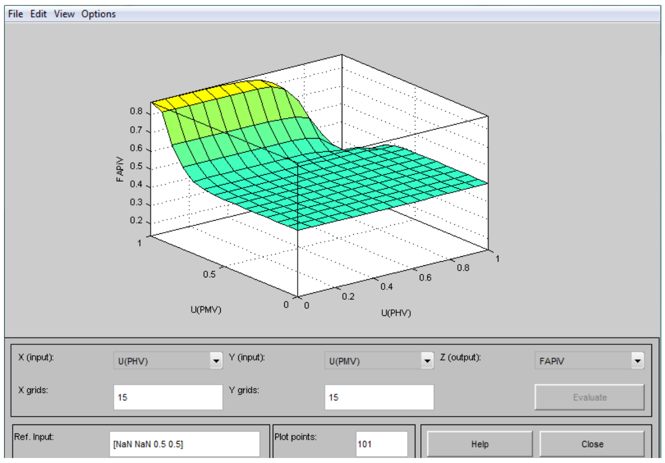


plot points: 181

Current Variable
Name: U(PHV)
Type: input
Range: [0 1]
Display Range: [0 1]

Current Membership Function (click on MF to select)
Name: Medium
Type: gauss2mf
Params: [0.1359 0.46 0.1359 0.54]

Figure 7. The input variable U (PHV)



X (input): U(PHV)
Y (input): U(PMV)
Z (output): FAPiV
X grids: 15
Y grids: 15
Ref. Input: [NaN NaN 0.5 0.5]
Plot points: 101

Figure 8. The set of decisions

6.6 Simulation

The simulation of our proposed approach is based on the implementation of our proposed scheduling algorithm under MATLAB 7.9.0.529 and by using a HP ProBook 4530s laptop with an Intel(R) Core TM i7 processor and a 09 GB RAM. Our goal in this simulation is to describe the execution scenario of tasks to meet the load balancing needs, and not the actual execution of tasks.

In our approach, each task is described with a 4-tuple: Ti(ri,Ci,Di,Pi); ri : The wake-up date ri of the k-th instance, Ci: induced load (execution time), Pi: period, Di: critical delay. The choice of values for these criteria in our case study is based on scenarios that can apply in real machines.

To demonstrate the effectiveness of our proposed approach, we propose a case study presented in Table 1.

In our case study, our system consists of 09 processors grouped in 03 different categories:

PHV: category of high speed processors which contains 03 processors (Pr 02, Pr 05 and Pr 07) with an execution speed Vi=1.

PMV: category of medium speed processors which contains 04 processors (Pr 01, Pr 04, Pr 06 and Pr 08) with an execution speed Vi=0.9.

PBV: category of low speed processors which contains 02 processors (Pr 03 and Pr 09) with an execution speed Vi=0.8.

And 20 periodic tasks; each task described with a 4-tuple: Ti = (DA, Ci, Di, Pi).

Table 1. Tasks description

No.	Arrival Date	Execution Time	Deadline	Period
T1	00	01	05	06
T2	00	04	09	10
T3	00	03	13	15
T4	01	08	27	30
T5	01	02	05	05
T6	01	05	09	10
T7	01	06	12	15
T8	02	02	06	06
T9	03	03	09	10
T10	03	07	14	15
T11	03	06	15	15
T12	04	13	26	30
T13	04	04	15	15
T14	04	03	14	15
T15	05	02	10	10
T16	06	01	14	15
T17	07	02	14	15
T18	07	08	26	30
T19	08	03	17	30
T20	24	01	27	30

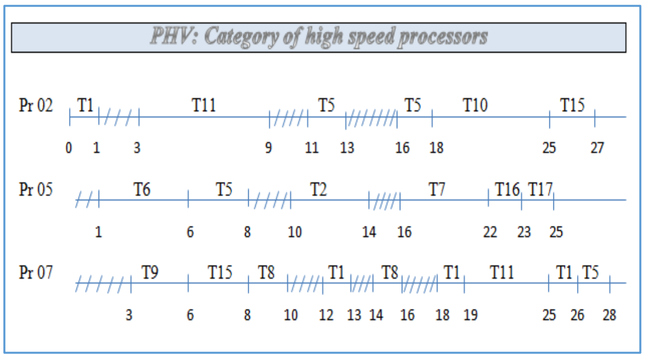


Figure 9. Execution of tasks by PHV category processors

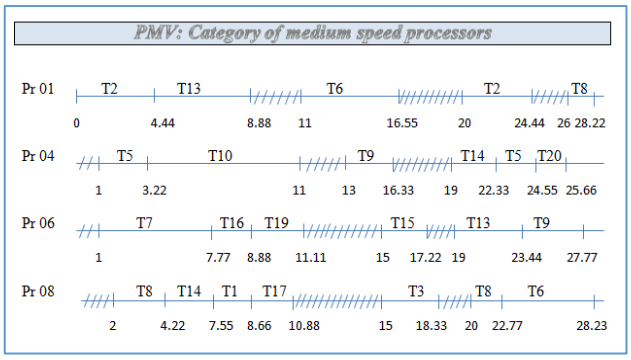


Figure 10. Execution of tasks by PMV category processors

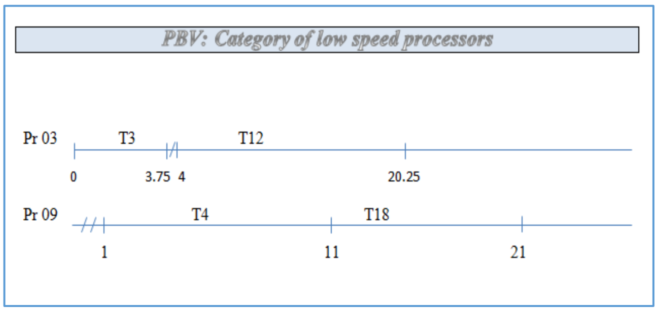


Figure 11. Execution of tasks by PBV category processors

Figures 8 and 9 represent the execution of tasks in the two categories of PHV and PMV processors respectively. Figure 10 presents the execution of tasks in the category of PBV processors.

6.7 Discussion of case study results

The feasibility test is a necessity in our approach for each arrival of a task Ti; this requires calculating Ptot first (the global period of all tasks), Ptot=PPCM (6,10.15, 30, 5, 10, 15, 6, 10, 15, 15, 30, 15, 15, 10, 15, 15, 30, 30, 30)=30. Calculating feasibility after each task arrives is an important step. The 03 tasks T1, T2, T3 arrived in the system at time t=0. After the positive feasibility test, the proposed fuzzy approach associates for each task their appropriate queue (HSPQ, MSPQ, LSPQ) according to the utilization factor of each category of processors and the deadline of the task. then the 03 tasks are inserted into the 03 queues in the following manner:

- T1 insert into the HSPQ queue.
- T2 insert into the MSPQ queue.
- T3 insert into the LSPQ queue.

After inserting the tasks into the queues in the order calculated by the EDF* real-time scheduling algorithm, the first task of each queue must be assigned to the available processor with a low utilization factor.

With an execution speed V2=1, the task T1 uses the processor Pr 02 from the date t=0 for an execution duration C1=1, at the same time the task T2 uses the processor Pr 01 with C2=4 and the execution speed V1=0.9, the task T3 used the processor Pr 03 from the date t=0 with C3=3 and the execution speed V3=0.8. In date t=task T1 completes execution for its first period and the arrival of tasks T4, T5, T6, T7 in the system. then the 04 tasks are inserted into the 03 queues in the following manner:

- T4 insert into the LSPQ queue.
- T5 insert into the MSPQ queue.
- T6 insert into the HSPQ queue.
- T7 insert into the MSPQ queue.

Task T4 used processor Pr 09 from date t=1 with C4=8 and execution speed V9=0.8, Task T5 used processor Pr 04 from date t=1 with C5=2 and execution speed V1=0.9, Task T6 used processor Pr 05 from date t=1 with C6=5 and execution speed V3=1 and Task T7 used processor Pr 06 from date t=1 with C7=6 and execution speed V3=0.9. In date t=2 it is the arrival of task T8 in the system and inserted into the MSPQ queue. Processor Pr 08 is available, then task T8 is executed by Pr 08 with C8=2 and execution speed V8=0.9. In date t= 3, arrival of tasks T9, T10, T11 in the system. then the 03 tasks are inserted into the 03 queues in the following manner:

- T9 insert into the HSPQ queue.
- T10 insert into the MSPQ queue.
- T11 insert into the HSPQ queue.

Task T9 used processor Pr 07 from date t=3 with C9=3 and execution speed V7=1, Task T11 used processor Pr 02 from date t=3 with C11=6 and execution speed V2=1, Task T10 used processor Pr 04 from date t=3.22 (end of execution of the T5 task for its first period) with C10=7 and execution speed V4=0.9. In date t=4, arrival of tasks T12, T13, T14 in the system. then the 03 tasks are inserted into the 03 queues in the following manner:

- T12 insert into the LSPQ queue.
- T13 insert into the MSPQ queue.
- T14 insert into the MSPQ queue.

Task T12 used processor Pr 03 from date t=4 with C12=13 and execution speed V3=0.8, Task T13 used processor Pr 01 from date t=4.44(end of execution of the T2 task for its first period) with C13=4 and execution speed V1=0.9, Task T14 used processor Pr 08 from date t=4.22 (end of execution of the T8 task for its first period) with C14=3 and execution speed V8=0.9.

Similarly, all tasks complete their execution and meet their deadlines using our proposed approach. The scheduling of tasks by our proposed approach on different processor categories are shown in Figures 9-11.

Table 2. Case study statistics

No.	Response Time for Each Period	Respect Deadline	Processor	Utilization Factor
T1	0, 1.55, 0, 0, 1	Yes	Pr 01	70.67%
T2	0, 0, 0	Yes	Pr 02	67.67%
T3	0, 0	Yes	Pr 03	67.67%
T4	0	Yes	Pr 04	67.67%
T5	0, 0, 0, 0, 1.33, 0	Yes	Pr 05	67.67%
T6	0, 0, 1.77	Yes	Pr 06	70.33%
T7	0, 0	Yes	Pr 07	67.67%
T8	0, 0, 0, 0, 0	Yes	Pr 08	68.13%
T9	0, 0, 2.44	Yes	Pr 09	67.67%
T10	0.22, 0	Yes	PHV	67.67%
T11	0, 1	Yes	PMV	69.20%
T12	0	Yes	PBV	67.67%
T13	0.44, 0	Yes	Task failure rate	
T14	0.22, 0	Yes	00%	
T15	1, 0, 0	Yes	Average utilization factor	
T16	1.77, 1.77	Yes	68.35%	
T17	1.66, 1	Yes	Average response time	
T18	4	Yes	0.43%	
T19	0.88	Yes	Number of migrations	
T20	0.55	Yes	00%	

The results presented in Table 2 express that the proposed approach for dynamic load balancing is efficient by the fair distribution of load between the different processors and the respect of deadline for all the tasks. The utilization factor of each processor is very close for the average utilization factor and that the average response time is very low equal to 0.43. Without using the migration our proposed approach allows the optimization of the performances and the use of the resources in an efficient way.

The most studied energy calculation model in the literature is defined as follows: if the speed of a machine is equal to a value 'V' for a given duration Δ, then the energy consumption power is given by Vα for a constant α>1, and the total energy consumed is Vα×Δ. If we now assume that the speed of the machine is given by a function of time, V(t), then the total energy consumed during the duration Δ is calculated by integrating the power over the duration Δ: ∫Δ S(t)αdt (15) [FAD 12]:

Energy (Pr 02)=1*(1)2+6*(1)2+2*(1)2+2*(1)2+7*(1)2+2*(1)2=20 joules.

Energy (Pr 05)=5*(1)2+2*(1)2+4*(1)2+6*(1)2+1*(1)2+2*(1)2=20 joules.

Energy (Pr 07)=3*(1)2+2*(1)2+2*(1)2+1*(1)2+2*(1)2+1*(1)2+6*(1)2+1*(1)2+2*(1)2=20 joules.

Energy (Pr 01)=4.44*(0.9)2+4.44*(0.9)2+5.55*(0.9)2+4.44*(0.9)2+2.22*(0.9)2=17.08 joules.

Energy (Pr 04)=1.11*(0.9)2+7.77*(0.9)2+3.33*(0.9)2+3.33*(0.9)2+2.22*(0.9)2+ 1.11*(0.9)2=15.28 joules

Energy (Pr 06)=6.66*(0.9)2+1.11*(0.9)2+2.22*(0.9)2+2.22*(0.9)2+4.44*(0.9)2+ 4.44*(0.9)2=17.08 joules

Energy (Pr 08)=2.22*(0.9)2+3.33*(0.9)2+2.22*(0.9)2+3.33*(0.9)2+2.22*(0.9)2+6.66*(0.9)2=16.18 joules.

Energy (Pr 03)=3.75*(0.8)2+16.25*(0.8)2=12.80 joules

Energy (Pr 09)=10*(0.8)2+10*(0.8)2=12.80 joules

Total energy our approach=20+20+20+17.08+15.28+17.08+16.18+12.80+12.80= 151.22 joules

Total energy by EDF*.=5+12+6+8+12+15+12+ 10+9+14+12+13+8+6+6+2+4+8+3+1=166 joules

Table 3 compares between our work and some pertinent related works. We defined nine comparison criteria that are:

• Level: at what level, load balancing is considered (managed) (RTOS for Real Time Operating System)
• TP: Tasks Periodicity (P: periodic/ A: aperiodic/ S: sporadic/ M: Mix)
• TD: Tasks Dependency (I: independent, D: dependent)
• TC: Temporal Constraints (H: hard, S: soft, F: firm, M: Mix)
• SP: HM: homogeneous, HT: heterogeneous
• EC: Energy consumption reduction
• LB: Load balancing (D: dynamic, S: static)
• UD: Uncertain Data support

Table 3. Comparison between our work and some pertinent related works

Work	Level	TP	TD	TC	SP	EC	LB	UD
MAH16	Application	A	I	S	HM	No	D	No
PRA 01	RTOS	A	I	S	HT	No	S	No
KUN21	RTOS	M	D	M	HM	No	S	No
OOP22	RTOS	A	I	S	HM	No	D	No
NIR22	RTOS	A	I	H	HM	No	D	Yes
ASA24	Application	A	I	S	HM	Yes	D	No
Our work	RTOS	P	I	H	HT	Yes	D	Yes

7. CONCLUSION

Dynamic load balancing is an essential strategy to ensure the equitable distribution of tasks among different processors in real-time systems, which allows for the optimization of performance and efficient resource utilization.

The work of this paper will allow designers and developers of real-time systems to implement more efficient and reliable systems in terms of load balancing, also meeting the requirements of energy consumption and respecting the deadline of all tasks. By implementing efficient techniques for assigning tasks to processors and by continuously monitoring performance indicators to ensure that the system remains efficient under varying loads. Due to the uncertainty of task information such as the exact execution time (before the task execution) and the arrival dates of future tasks arrive, which justify the use of fuzzy logic for the assignment of tasks in different queues.

A significant improvement in the efficiency of real-time systems by the proposed dynamic load balancing presented by the results of our approach despite the arrival of a large number of tasks at different dates and with different execution times, in a system that uses heterogeneous processors. The 20 tasks of our case study their utilization factors do not exceed 1.98% as a difference compared to the average utilization factor equal to 68.35% in a system that uses heterogeneous processors for the execution of different tasks.

The work in this paper opens the way for interesting future research in real-time systems and load balancing. We plan to make a comparative study on the impact of different scheduling strategies on load balancing. Moreover, the effectiveness of the EDF algorithm in more complex scenarios such as sporadic and not only periodic task usage could be promising research avenues.

REFERENCES

[1] Daraghmi, E., Hamoudi, A., Abu Helou, M. (2024). Decentralizing democracy: Secure and transparent E-Voting systems with blockchain technology in the context of Palestine. Future Internet, 16(11): 388. https://doi.org/10.3390/fi16110388

[2] Nelaturi, N.P., Rajesh, V., Syed, I. (2024). Real-time liver tumor detection with a multi-class ensemble deep learning framework. Engineering, Technology & Applied Science Research, 14(5): 16103-16108. https://doi.org/10.48084/etasr.8106

[3] Alam, M., Varshney, A.K. (2016). A new approach of dynamic load balancing scheduling algorithm for homogeneous multiprocessor system. International Journal of Applied Evolutionary Computation (IJAEC), 7(2): 61-75. https://doi.org/10.4018/IJAEC.2016040104

[4] Kakulavarapu, P., Maquelin, O.C., Amaral, J.N., Gao, G.R. (2001). Dynamic load balancers for a multithreaded multiprocessor system. Parallel Processing Letters, 11(1): 169-184. https://doi.org/10.1142/S0129626401000506

[5] Tan, Q., Xiao, K., He, W., Lei, P., Chen, L. (2021). A global dynamic load balancing mechanism with low latency for micokernel operating system. In 2021 7th International Symposium on System and Software Reliability (ISSSR), Chongqing, China, pp. 178-187. https://doi.org/10.1109/ISSSR53171.2021.00026

[6] Panwar, P., Kaushal, C., Singla, A., Rattan, V. (2022). Load balancing in multiprocessor systems using modified real-Coded genetic algorithm. In Innovations in Computational Intelligence and Computer Vision: Proceedings of ICICV 2021. Singapore: Springer Nature Singapore, pp. 201-210. https://doi.org/10.1007/978-981-19-0475-2_18

[7] Nirmala, H., Girijamma, H.A. (2022). Fllbhgats: Efficient load balancing and task scheduling algorithm for real-time multiprocessor. Journal of Algebraic Statistics, 13(3): 433-450.

[8] Ali, A., Suleman, M. (2024). Implementing dynamic load balancing for real-time user requests in distributed systems. ResearchGate: Distributed Computing-Load Balancing. https://doi.org/10.13140/RG.2.2.29397.64484

[9] Khawatreh, S.A. (2018). An efficient algorithm for load balancing in multiprocessor systems. International Journal of Advanced Computer Science and Applications, 9(3): 160-164. https://doi.org/10.14569/IJACSA.2018.090324

[10] Leinberger, W., Karypis, G., Kumar, V., Biswas, R. (2000). Load balancing across near-Homogeneous multi-Resource servers. In Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556), Cancun, Mexico, pp. 60-71. https://doi.org/10.1109/HCW.2000.843733

[11] Ghomi, E.J., Rahmani, A.M., Qader, N.N. (2017). Load-balancing algorithms in cloud computing: A survey. Journal of Network and Computer Applications, 88: 50-71. https://doi.org/10.1016/j.jnca.2017.04.007

[12] Yagoubi, B., Meddeber, M. (2010). Distributed load balancing model for grid computing. Revue Africaine de Recherche en Informatique et Mathématiques Appliquées, 12: 43-60. https://doi.org/10.46298/arima.1931

[13] Djennane, N., Aoudjit, R., Bouzefrane, S. (2018). Energy-efficient algorithm for load balancing and VMs reassignment in data centers. In 2018 6th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), Barcelona, Spain, pp. 225-230. https://doi.org/10.1109/W-FiCloud.2018.00043

[14] Piel, É., Marquet, P., Soula, J., Dekeyser, J.L. (2006). Asymmetric scheduling and load balancing for real-time on Linux SMP. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds) Parallel Processing and Applied Mathematics. PPAM 2005. Lecture Notes in Computer Science, Springer. https://doi.org/10.1007/11752578_108

[15] Robilliard, D., Marion-Poty, V. Fonlupt, C. (2009). Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines, 10: 447-471. https://doi.org/10.1007/s10710-009-9092-3

[16] Schopf, J.M. (2004). Ten Actions When Grid Scheduling. In: Nabrzyski, J., Schopf, J.M., Węglarz, J. (eds) Grid Resource Management. International Series in Operations Research & Management Science, Springer, Boston, USA. https://doi.org/10.1007/978-1-4615-0509-9_2

[17] Morchid, A., Jebabra, R., Ismail, A., Khalid, H.M., El Alami, R., Qjidaa, H., Jamil, M.O. (2024). IoT-enabled fire detection for sustainable agriculture: A real-time system using flask and embedded technologies. Results in Engineering, 23: 102705. https://doi.org/10.1016/j.rineng.2024.102705

[18] Mehalaine, R., Djezzar, M., Nessah, D., Saiad, Z., Saidi, A. (2024). Watchdog timer for fault tolerance in embedded systems. Journal Européen des Systèmes Automatisés, 57(6): 1713-1720. https://doi.org/10.18280/jesa.570619

[19] Mehalaine, R., Boutekkouk, F. (2020). A new intelligent biologically-inspired model for fault tolerance in distributed embedded systems. International Journal of Embedded and Real-Time Communication Systems, 11(3): 22-47. https://doi.org/10.4018/IJERTCS.2020070102

[20] Mayank, J., Mondal, A. (2019). An integer linear programming framework for energy optimization of non-preemptive real time tasks on multiprocessors. Journal of Low Power Electronics, 15(2): 162-167. https://doi.org/10.1166/jolpe.2019.1604

[21] Nadeem, A., Rizvi, A.A., Noor, M.Y. (2024). Applying a higher number of output membership functions to enhance the precision of a fuzzy system. IEEE Transactions on Emerging Topics in Computational Intelligence, 9(1): 394-405. https://doi.org/10.1109/TETCI.2024.3425309