

PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, s/n - Ondina
Salvador, BA, Brasil, 40170-110

<http://pgcomp.dcc.ufba.br>
pgcomp@ufba.br

A hard real-time system can be defined as the one that at least one of its tasks must meet its deadlines whenever the system is running. This requirement makes the scheduling algorithm a key element for system correctness. Ideally, the scheduling algorithm employed must both exhibit low overhead (efficiency) and ensure that no task deadline is missed whenever this can be ensured by some scheduling algorithm (optimality). RUN (Reduction to Uniprocessor) is an algorithm capable of efficiently and optimally scheduling a set of strictly periodic tasks on a multiprocessor platform when tasks do not share any resources but processors. Although it has already been shown that RUN is compatible with resource sharing, the only existing solution prevents preemptive access to shared resources. Unlike this approach, which can be considered too restrictive due to its poor schedulability, we used MrsP (Multiprocessor resource sharing Protocol) as a more flexible resource sharing mechanism. Making the rules of both RUN and MrsP compatible to each other was thus our main goal. The derived solution was implemented on Linux Textbed for Multiprocessor Scheduling in Real-Time systems (Litmus-RT), namely a Linux-based real-time operating system. We proposed a new task packaging heuristic and performed experimental evaluations comparing our solution with the existing one. The results showed that the proposed solution presented better results in terms of schedulability, total overhead and number of migrations and preemptions.

Keywords: HARD REAL TIME SYSTEMS, TASK SCHEDULING, RESOURCE SHARING, RUN, MrsP, LITMUS.

Shared Resources in Multiprocessor Real-Time Systems Scheduled by RUN

Ricardo Brasil Teixeira

Dissertação de Mestrado

Universidade Federal da Bahia

Programa de Pós-Graduação em
Ciência da Computação

Dezembro | 2019





Universidade Federal da Bahia
Instituto de Matemática

Programa de Pós-Graduação em Ciência da Computação

**SHARED RESOURCES IN
MULTIPROCESSOR REAL-TIME SYSTEMS
SCHEDULED BY RUN**

Ricardo Brasil Teixeira

DISSERTAÇÃO DE MESTRADO

Salvador
20 de dezembro de 2019

RICARDO BRASIL TEIXEIRA

**SHARED RESOURCES IN MULTIPROCESSOR REAL-TIME
SYSTEMS SCHEDULED BY RUN**

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. George Marconi de Araújo Lima

Salvador
20 de dezembro de 2019

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências e Tecnologias
Prof. Omar Catunda, SIBI - UFBA.

Teixeira, Ricardo Brasil.

Shared Resources in Multiprocessor Real-Time Systems Scheduled by
RUN / Ricardo Brasil Teixeira – Salvador, 2019.

74p.: il.

Orientador: Prof. Dr. George Marconi de Araújo Lima.

Dissertação (Mestrado) – Universidade Federal da Bahia, Instituto de
Matemática, 2019.

1. Sistemas de Tempo Real. 2. Escalonamento de Tarefas. 3. Compartilhamento de Recursos. 4. Algoritmos. 5. Multiprocessadores.. I. Lima, George Marconi de Araújo . II. Universidade Federal da Bahia. Instituto de Matemática. III Título.

CDD – T266

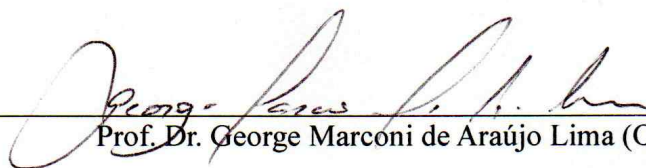
CDU – 004.41

*"SHARED RESOURCES IN MULTIPROCESSOR REAL-TIME
SYSTEMS SCHEDULED BY RUN"*

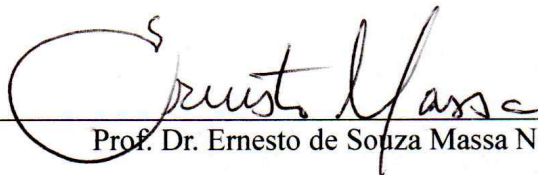
Ricardo Brasil Teixeira

Dissertação apresentada ao Colegiado do Programa de Pós-Graduação em Ciência da Computação na Universidade Federal da Bahia, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

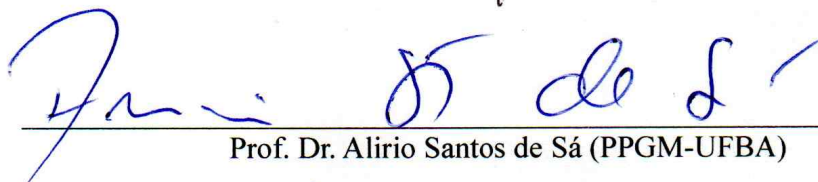
Banca Examinadora



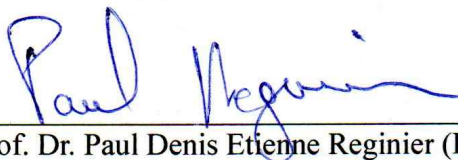
Prof. Dr. George Marconi de Araújo Lima (Orientador-UFBA)



Prof. Dr. Ernesto de Souza Massa Neto (UNEB)



Prof. Dr. Alirio Santos de Sá (PPGM-UFBA)



Prof. Dr. Paul Denis Etienne Reginier (PPGM-UFBA)

Dedico este trabalho aos meus pais, Edinaldo e Solange, por tudo o que fizeram por mim e pelo exemplo de vida; às minhas filhas Julia e Cecília, por serem minha inspiração para encarar desafios e não desanimar; à minha esposa Maria, pelo companheirismo, compreensão e amparo necessários à minha jornada.

ACKNOWLEDGEMENTS

Agradeço a minha família, por estar ao meu lado dando todo o apoio de que precisei antes e durante a realização deste trabalho; ao meu orientador, George, pela enorme parceria, sempre pautada pela excelência e me ajudando a vencer os desafios que surgiram; aos amigos, colegas de trabalho e colegas, professores e funcionários da UFBA, por estarem juntos comigo nesta caminhada em clima de cooperação e pelas palavras de incentivo e ponderação que se fizeram presentes em diversos momentos, especialmente ao longo dos três últimos anos.

Os povos “não são feitos da massa de indivíduos que os compõem, dos territórios que ocupam, das coisas que usam, dos movimentos que executam. Eles são feitos, sobretudo, com as idéias que os indivíduos têm de si mesmos”.

(Durkheim, Émile apud Alves, Rubem. “Entre a ciência e a sapiência. O dilema da educação”, 1999, p. 25)

RESUMO

Um sistema de tempo real crítico é aquele para o qual ao menos uma de suas tarefas deve cumprir os prazos de execução (*deadlines*) enquanto o sistema estiver executando. Este requisito faz do algoritmo de escalonamento um elemento chave para a correção do sistema. Idealmente, o algoritmo de escalonamento deve ser eficiente, para não causar sobrecargas adicionais, e ótimo, garantindo que nenhum *deadline* de suas tarefas seja perdido sempre que isso puder ser assegurado por algum algoritmo de escalonamento. RUN (*Reduction to Uniprocessor*) é um algoritmo capaz de escalonar eficientemente e de maneira ótima um conjunto de tarefas periódicas em uma plataforma com múltiplos processadores, quando as tarefas não compartilham outros recursos que não os processadores. Embora já tenha sido demonstrado que o RUN é compatível com o compartilhamento de recursos, a única solução existente impede preempções no acesso aos recursos compartilhados. Ao contrário desta abordagem, que pode ser considerada muito restritiva devido à sua baixa escalonabilidade, utilizamos o MrsP (*Multiprocessor resource sharing Protocol*) como um mecanismo de compartilhamento de recursos mais flexível. Fazer as regras do RUN e MrsP compatíveis entre si foi, portanto, nosso principal objetivo. A solução derivada foi implementada no Linux Textbed for Multiprocessor scheduling in Real-Time systems (Litmus^{RT}), um sistema operacional de tempo real baseado em Linux. Propusemos uma nova heurística para empacotamento das tarefas e realizamos avaliações experimentais comparando nossa solução com a existente. Os resultados obtidos mostraram que a solução proposta apresentou melhores resultados em escalonabilidade, overhead total e número de migrações e preempções.

Palavras-chave: SISTEMAS DE TEMPO REAL HARD, ESCALONAMENTO DE TAREFAS, COMPARTILHAMENTO DE RECURSOS, RUN, MrsP, LITMUS

ABSTRACT

A hard real-time system can be defined as the one that at least one of its tasks must meet its deadlines whenever the system is running. This requirement makes the scheduling algorithm a key element for system correctness. Ideally, the scheduling algorithm employed must both exhibit low overhead (efficiency) and ensure that no task deadline is missed whenever this can be ensured by some scheduling algorithm (optimality). RUN (Reduction to Uniprocessor) is an algorithm capable of efficiently and optimally scheduling a set of strictly periodic tasks on a multiprocessor platform when tasks do not share any resources but processors. Although it has already been shown that RUN is compatible with resource sharing, the only existing solution prevents preemptive access to shared resources. Unlike this approach, which can be considered too restrictive due to its poor schedulability, we used MrsP (Multiprocessor resource sharing Protocol) as a more flexible resource sharing mechanism. Making the rules of both RUN and MrsP compatible to each other was thus our main goal. The derived solution was implemented on Linux Textbed for Multiprocessor Scheduling in Real-Time systems (Litmus^{RT}), namely a Linux-based real-time operating system. We proposed a new task packaging heuristic and performed experimental evaluations comparing our solution with the existing one. The results showed that the proposed solution presented better results in terms of schedulability, total overhead and number of migrations and preemptions.

Keywords: HARD REAL TIME SYSTEMS, TASK SCHEDULING, RESOURCE SHARING, RUN, MrsP, LITMUS

CONTENTS

Chapter 1—Introduction	1
1.1 Real-Time Systems	1
1.1.1 Scheduling Algorithms	2
1.1.2 Resource Sharing	3
1.2 Motivation	4
1.3 This work	4
Chapter 2—Background	7
2.1 Scheduling Algorithms	7
2.2 Resource Sharing Protocols	9
2.2.1 Non-Hierarchically Scheduled Systems	10
2.2.2 Hierarchically Scheduled Systems	11
2.3 The RUN Scheduling Algorithm	13
2.4 The Target Multiprocessor Resource Sharing Protocol	16
Chapter 3—System Model and Notation	19
3.1 Integrating MrsP and RUN	20
3.2 Resource Share Protocol and its Properties	22
3.3 Illustration of the Proposed Ideas	29
Chapter 4—Processor Demand Evaluation	35
4.1 The Protocol for Comparison - SBLP	35
4.2 Schedulability	36
4.2.1 Ordered Blocking Time Packing	38
4.3 Experimental Assessment	39
4.3.1 Experimental Setup	39
4.3.2 Experimental Results	41
4.3.2.1 CG-SBLP Highest Inflation	43
4.3.2.2 OBT-SBLP Highest Inflation	44
4.3.2.3 Worse Results for OBT-MrsP Compared to OBT-SBLP	44
4.3.2.4 Worse Results for OBT-MrsP Compared to CG-SBLP	46

Chapter 5—Implementation	49
5.1 Litmus ^{RT}	49
5.2 Implementation Details	51
5.2.1 Scheduling Plugin Methods	52
Chapter 6—Overhead Evaluation	59
6.1 Evaluation Preparation	59
6.2 Evaluation Results	62
6.2.1 Overhead	62
6.2.2 Preemptions and Migrations	66
Chapter 7—Conclusion	69

LIST OF FIGURES

2.1	RUN Schedule Example - First Level	14
2.2	RUN Schedule Example - Second Level	15
2.3	RUN Schedule Example - Final Level	15
2.4	Visual Representation of MrsP	17
3.1	Illustration of the Proposed Ideas	30
3.2	RUN Schedule Example	32
3.3	Resource Sharing Examples	33
4.1	Average Number of Generated Servers	41
4.2	Processor Utilization Inflation - All Heuristics	42
4.3	Processor Utilization Inflation - CG-SBLP, OBT-SBLP and OBT-MrsP	42
4.4	CG-SBLP highest processor utilization inflation	43
4.5	Trend for CG-SBLP highest processor utilization inflation	44
4.6	OBT-SBLP highest processor utilization inflation	45
4.7	Trend for OBT-SBLP highest processor utilization inflation	45
4.8	Worse results for OBT-MrsP compared to OBT-SBLP	46
4.9	Trend for worse results for OBT-MrsP compared to OBT-SBLP	47
4.10	Worse results for OBT-MrsP compared to CG-SBLP	47
4.11	Trend for worse results for OBT-MrsP compared to CG-SBLP	48
5.1	State Diagram for the Implementation of RUN with MrsP	58
6.1	Comparative of Overhead	64
6.2	Comparative of Overhead - 99.9 Percentile	65
6.3	Cumulative Overhead Comparison	65
6.4	Distribution of Weighted Overhead by Events	66
6.5	Comparative of Preemptions per Task	67
6.6	Comparative of Migrations per Task	68

LIST OF TABLES

2.1	RUN Off-line Phase	14
3.1	Notations Summary	21
3.2	How MrsP and RUN Would Work	29
3.3	RUN Schedule Example	31
4.1	Schedulability Test Parameters - Tasks Configuration.	40
4.2	Schedulability Test Parameters - Resources Configuration.	40
4.3	Schedulability Test Parameters - CG-SBLP highest inflation.	43
4.4	Schedulability Test Parameters - OBT-SBLP highest inflation.	44
4.5	Schedulability Test Parameters - OBT-MrsP X OBT-SBLP	46
4.6	Schedulability Test Parameters - OBT-MrsP X CG-SBLP	46
6.1	Processor Final Utilization	60
6.2	Servers Demanded by the Experiments Task Sets	61
6.3	Maximum RUN Reduction Tree Level	61

LISTA DE SIGLAS

CG	Coarse Grained	37
EDF	Earliest Deadline First	2
FIFO	First In First Out	16
FG	Fine Grained.....	37
GDB	GNU Debugger	52
JSON	JavaScript Object Notation	61
KDB	Kernel Debugger	52
HRTS	Hard Real Time System	1
MrsP	Multiprocessor resource sharing Protocol	4
OBT	Ordered Blocking Time	38
PCP	Priority Ceiling Protocol	4
QPS	Quasi-Partitioned Scheduling.....	2
RTA	Response Time Analysis.....	16
RTS	Real Time System.....	1
RUN	Reduction to UNiprocessor	2
SBLP	Server Based Locking Protocol	4
SRP	Stack Resource Policy	4
U-EDF	Unfair Earliest Deadline First	2
WCET	Worst Case Execution Time.....	2

INTRODUCTION

This research project looks into the topics of Resource Sharing and Scheduling Algorithms for hard real-time systems. In this chapter we give an overview of the problem we investigated, show the current state of the research in this field, present the motivation for this work and summarize our research goals.

1.1 REAL-TIME SYSTEMS

A Real Time System (RTS) requires that its services be completed within well defined time limits and it is called a Hard Real Time System (HRTS) when its malfunction may lead to serious consequences. Given its criticality, it is necessary to show that a HRTS works correctly through rigorous demonstrations that its temporal behavior is predictable.

A process controller can be considered an example of an HRTS if it is employed as a critical application, in industry or avionics, for instance. It performs monitoring and control functions to keep the controlled variable at the set point (LIU; W. Layland, 1973). The system usually is comprised of a set of tasks (sequence of instructions to be executed). Each task is recurrently released one or more (possibly infinity) times. Each task instance, usually called job, must be executed from its release time until its deadline. This kind of system is present in our lives without us noticing this. Some examples, with different degree of criticality, are systems applied to telecommunication, vehicle braking, electronic fuel injection, flight control, event monitoring, heart rate monitoring. In any case, at least some service of this kind of systems recurrently obtains the current state of what is monitored through the sensors, performs a processing and, if necessary, triggers actuators to reach the planned state. Sampling period, chosen at design time, is ruled by the application needs, it can refer to the time interval between two consecutive data collections of what is sensed. The state of the monitored object is given by several variables (e.g., the rotation speed, frequency, temperature, pressure, etc). Associating tasks to system functions is the role of system designers.

As may have been noticed, some systems contain highly critical functions. Liu (LIU, 2000) gave an example of an automatically controlled train, which cannot stop instantaneously after receiving a stop signal. The controller calculates the time required for the

train to travel the braking distance in a safe condition, which imposes a restriction on the response time of tasks that process the stop signal and activate braking. The deadlines of the tasks “are typically derived from the required responsiveness of the sensors and actuators monitored and controlled by it” (LIU, 2000). Considering several tasks, with their own specified deadlines, there must be a means of feasibly execute all of them. This is the role of real-time scheduling.

1.1.1 Scheduling Algorithms

A task can release a possible infinity set of jobs and a job is a piece of code which should be executed between its release time and deadline. Each job requires a number of processor cycles to finish. In HRTS one must know the upper limit of execution time for all its tasks. That is, each job of a task is known to require no more than its Worst Case Execution Time (WCET). As for the recurrent release of jobs, Aloysius Mok introduced the sporadic task model (MOK, 1983; BRANDENBURG, 2011) as a generalization of the earlier periodic task model (LIU; W. Layland, 1973). In the periodic task model, jobs are released at a regular time interval. In the sporadic task model, jobs are released at intervals not smaller than specified, allowing tasks to become inactive if there are no triggering events. Given a set of tasks implementing an HRTS, their scheduling must preserve their timing constraints and should be carried out as efficient as possible.

An optimal scheduling algorithm is the one which always finds a valid schedule for any system for which some valid schedule exists. A valid schedule is one according to which no task deadline is missed. The Earliest Deadline First (EDF) (LIU; W. Layland, 1973) is an outstanding optimal scheduling algorithm originally designed for uniprocessor systems. At any scheduling instant it selects for execution the job with the earliest deadline. Unfortunately, EDF performs poorly in systems with multiple processors in terms of schedulability.

Although there are many optimal scheduling algorithms for multiprocessor real-time systems, most of them either do not achieve optimality or do so at the expense of high runtime overheads. Low overhead approaches usually partition the system tasks onto different processors forbidding task migrations between processors. However, these *partitioned* scheduling approaches do not achieve optimality and exhibit low schedulability (KOREN; AMIR; DAR, 1998). In contrast, *global approaches*, which do not impose any restrictions on task migration, may achieve optimality. However, most of optimal global scheduling algorithms excessively chop off the tasks’ execution causing unnecessary runtime overhead. More recently, better approaches in terms of reduced overheads have been proposed, namely Reduction to UNiprocessor (RUN) (REGNIER et al., 2011), Quasi-Partitioned Scheduling (QPS) (MASSA et al., 2016) and Unfair Earliest Deadline First (U-EDF) (NELISSEN et al., 2012). These achieve optimality without causing too many preemptions and task migrations between processors as compared to previous algorithms. From these, RUN seems to be the best in terms of generated overhead. Indeed, the authors of an implementation of the RUN algorithm over the Linux Textbed for Multiprocessor Scheduling in Real-Time systems (Litmus^{RT}) (MPI-SWS, Max Planck Institute for Software Systems, 2018) state that “RUN can be efficiently implemented

on top of standard operating system primitives incurring modest overhead and interference, also supporting much higher schedulable utilization than its partitioned and global counterparts” (COMPAGNIN; MEZZETTI; VARDANEGA, 2014). This opens roads for using RUN to support new real-time applications running on multiple processors, which is the current trend in the area.

Indeed, Di Natale (NATALE; SANGIOVANNI-VINCENTELLI, 2010) mentions that “Cost pressure, flexibility, extensibility and the need for coping with increased functional complexity are changing the fundamental paradigms for the definition of automotive and aeronautics architectures”. Microprocessors with poor processing power was previously used for Electronic Control Units (ECU) to support few embedded HRTS applications. With a growing demand for more features with more complexity, there has been a trend to integrate multiple applications into a single processing units and making use of a multiprocessor platform.

In this context, real-time scheduling becomes of paramount importance. Advances in computer architectures have raised the demand for simultaneous applications execution, which increases the concurrency for shared resources (*e.g.*, processor, memory, peripheral devices and others). Scheduling tasks that share such resources requires extra efforts for ensuring temporal isolation in the system. A resource shared by two or more tasks must be locked before modification, possibly causing blocking of other tasks. This kind of interference increases as more software components are executing at the same platform, with possible impact on the system performance, as a consequence of lengthy block chains on shared resources. Designing suitable real-time scheduling algorithms compatible with resource sharing in multiprocessor platforms is an actual need.

1.1.2 Resource Sharing

Race conditions occur when multiple tasks can access and modify the same data item concurrently. The result of these operations will depend on the order in which the operations take place. The critical section of a task is a code segment where it can modify shared objects, such as variables, files, queues, buffers, among others. In order to avoid inconsistencies, a protocol that synchronizes concurrent access to shared resources there must be in place.

Common synchronization primitives (locks, semaphores, and monitors) protect the consistency of shared objects and ensure uninterrupted use of logical or physical resources through mutually exclusive execution of tasks’ critical sections. The direct application of these primitives can, however, “lead to uncontrolled priority inversion, a situation in which a high priority job is indirectly preempted by lower priority jobs for an indefinite period of time” (SHA; RAJKUMAR; LEHOCZKY, 1990). This reduces the possibility of obtaining valid schedules and decreases system predictability. Prolonged duration of blocking can lead to missed deadlines even at low system utilization levels.

The most common approaches to ensuring synchronized accesses to shared resources in real-time environments are busy wait (also known as spinning) and suspension. Both are blocking-based implementations and do not allow concurrent access to objects. As stated in (BRANDENBURG et al., 2008), “suspension-based locking never resulted in

better schedulability than spin-based locking (on the other hand, more processor time may be available to background jobs if suspension locking is used)".

Note that disabling interruptions during critical sections is a possible approach but this can lead to blocking of higher priority jobs that do not compete for the locked resource, increasing the likelihood of deadline misses. In the context of multiprocessor real-time systems, this may mean a too constrained solution. Standard reference protocols created for uniprocessor systems, such as Priority Ceiling Protocol (Priority Ceiling Protocol (PCP)) (SHA; RAJKUMAR; LEHOCZKY, 1990) or Stack Resource Policy (Stack Resource Policy (SRP)) (BAKER, 1990), do not work properly for multiprocessors. Many other solutions have been described recently. Notably, Multiprocessor resource sharing Protocol (MrsP) (BURNS; WELLINGS, 2013) preserves many characteristics found on PCP and SRP and has been implemented and evaluated in different contexts (CATELLANI et al., 2015; ZHAO et al., 2017).

1.2 MOTIVATION

As previously mentioned, RUN is a low overhead optimal scheduling algorithm for multiprocessor real-time system that has already been implemented and evaluated on an actual operating system (COMPAGNIN; MEZZETTI; VARDANEGA, 2014). This makes RUN a natural choice for scheduling even when tasks share resources. Indeed, Server Based Locking Protocol (SBLP) (BONATO; MEZZETTI; VARDANEGA, 2014) is a resource sharing protocol resulted from this research line. It uses RUN as a baseline scheduler and manages critical sections as non-preemptive regions inside servers. However, this kind of solution offers potential disadvantages in terms of system schedulability since higher priority tasks must wait for unrelated lower priority tasks during time periods with disabled preemptions. The use of better resource sharing protocols in line with RUN should thus be investigated.

MrsP is a recently proposed resource sharing protocol for multiprocessor real-time systems, which was inspired by the classical uniprocessor protocols PCP and SRP (BURNS; WELLINGS, 2013). Unlike SBLP, which exhibits a non-preemptive nature, MrsP may offer better system usage, potentially outperforming SBLP in this respect. Although Burns (BURNS; WELLINGS, 2013) states that MrsP works for both fixed- and dynamic-priority scheduling algorithms, to the best of our knowledge MrsP has been only implemented in the context of fixed priority partitioned scheduling (ZHAO et al., 2017; GARRIDO et al., 2017a; BURNS; WELLINGS, 2013; GARRIDO et al., 2017b; CATELLANI et al., 2015). As fixed-priority partitioned systems are not able to fully utilize the processing capacity of a multiprocessor system, as opposed to global dynamic-priority scheduling algorithms like RUN, integrating RUN and MrsP may well improve the state of art. This observation has motivated our research whose goals are described next.

1.3 THIS WORK

Although RUN and MrsP are well defined algorithms, both proved correct in different contexts, putting them together in a single system requires some effort. Both algorithms

establish their own rules for selecting the tasks that must execute at a given time. Their task selecting criteria may be in conflict. For example, the scheduling decisions by RUN are intended to meet deadlines and do not consider which task should execute for the sake of keeping consistency during resource sharing. The main step of our work was to derive adapting rules to make the co-existence of both RUN and MrsP possible.

Once MrsP running over RUN is proved correct, our secondary goal was to provide an implementation of our solution on Litmus^{RT} (MPI-SWS, Max Planck Institute for Software Systems, 2018), a Linux-based operating system designed for evaluating and testing real-time scheduling algorithms and resource sharing protocols. As there are already implementations of RUN (COMPAGNIN; MEZZETTI; VARDANEGA, 2014) and MrsP (ZHAO et al., 2017; CATELLANI et al., 2015) on Litmus^{RT}, experimentally comparing our solution to previous ones (BONATO; MEZZETTI; VARDANEGA, 2014) was a natural choice.

Resource-sharing protocols in general require an increased reserve of processor capacity to handle blocking times that tasks suffer. Some heuristics have been proposed in previous work to minimize the increase in demand for processor capacity. We have seen in this research project the opportunity to obtain better results than existing heuristics, so we also present Ordered Blocking Time (OBT), which obtained significant advantage over other heuristics.

In the remainder of this text we will provide more detailed information to base the defined research goals. Chapter 2 gives background information on scheduling and resource sharing protocols in real-time systems. Chapter 3 explains the model of the proposed system and proves its correctness. Chapter 4 describes the experiment performed to evaluate the performance of SBLP and MrsP in relation to the demand for processor time reservation. Chapter 5 explains the implementation of the MrsP protocol in the RUN plugin. Chapter 6 presents and analyzes the results of the experiments to measure the overhead produced by each protocol. Our final comments are given in Chapter 7.

BACKGROUND

The purpose of this chapter is to give an overview of the work that gave support to this research project, so as to better contextualize it. In the following sections we will introduce the task scheduling for real-time systems and give a brief overview on optimal scheduling algorithms and on resource sharing protocols for RTS. Then we will show work related to the RUN algorithm and the MrsP protocol, which were the chosen algorithms in the context of this work.

2.1 SCHEDULING ALGORITHMS

Real-time scheduling for uniprocessor real-time systems usually follows two classic approaches, based on assigning either fixed or variable (*i.e.*, dynamic) priorities to tasks according to some criteria. In both cases, the scheduler selects, at each scheduling instant, the highest priority task/job to execute. Well known priority assignments are Rate Monotonic (RM) and EDF for fixed- and dynamic-priority scheduling, respectively (LIU; W. Layland, 1973). According to the former, the lower the period of the tasks, the higher their priorities. This means that different jobs of same task have the same priority. EDF assigns priority to jobs instead. The earlier the deadline of a job, the higher its EDF priority. Although fixed-priority scheduling is simpler to implement, it usually leads to lower bounds on utilizing the system. For example, a set of n independent preemptive periodic tasks scheduled by EDF can use 100% of the processor without jeopardizing any of their deadlines. The bound for RM is close to 69%. Most approaches seeking better utilization of the system tend to use rely on dynamic-priority scheduling. In this work, we follow this trend.

Scheduling can also be designed for a system hierarchically using the concept of *server*. Servers work like bandwidth reservation schemes according to which their processing demands are confined within specific periods. For example, a server can be specified for reserving C processing cycles at every time period of T , meaning that C/T of a processor is reserved to this server. This reserve can be used to schedule a sub-set of tasks in the system, providing a *hierarchical* scheduling mechanism. This server-based scheme

is usually applied to control the execution of some tasks, protecting the system against possible overruns, for instance. If a task executes in the context of a server and its budget is depleted, the task must wait for resuming its execution later on, when its server budget is replenished. Task overruns, if any, do not affect the timeliness of other tasks in the system, a property known as *temporal isolation* (BIONDI et al., 2014). Servers, in turn, may use their own scheduling criteria, based on fixed or dynamic priorities.

Real-time scheduling on multiprocessor platforms is more challenging. Even EDF performs poorly when more than one processor is considered. For example, let three identical tasks requiring $1 + \epsilon$ time units out of 2 time units (deadlines equal to 2) with ϵ a small positive constant. If all tasks released their jobs at the same instant t , they would have the same priority by EDF and two of them would be selected to execute. After their execution, there will be only $1 - \epsilon$ time unit left to be used from time $t + 1 + \epsilon$. This is not enough to complete the remainder job. This approach applies EDF in a global manner since tasks are being scheduled globally (a unique priority queue) in the system. Another attempt would be to statically allocate the system tasks onto processors and apply EDF independently on each of them. This approach, called partitioned, would also fail in the given example since any two out of three tasks jointly require more than one processor. Note that under the partitioned approach, a valid schedule does not exist in this case. That is, if one is interested in optimality, applying a global scheduling policy is necessary.

There are a number of optimal priority scheduling algorithms for multiprocessor real-time systems, *e.g.*, Pfair (BARUAH et al., 1996), PD (BARUAH; GEHRKE; PLAXTON, 1995), PD² (ANDERSON; SRINIVASAN, 2000), DP-wrap (LEVIN et al., 2010). Most of them are based on implementing some notion of fairness during the scheduling (BARUAH et al., 1996). The idea is to divide time into slots across the processors and assign such slots to all tasks in the system proportional to their required execution rate. The slot sizes depend on the distance between consecutive deadlines in the system; job deadlines are taken as synchronization points by these approaches. Optimality is thus obtained at the expense of possibly generating too many preemption points and forcing task migration too often during execution. This is because, when consecutive deadlines are too close to each other, the generated time slots may be too small. Due to this side effects, fairness-based approaches will not be further considered in this work. Readers can refer to other sources for an overview on the subject (LIMA; REGNIER; MASSA, 2019; DAVIS; BURNS, 2011).

More recently, three non-fairness-based scheduling approaches have been described as a means of obtaining optimality with reduced runtime overheads, namely Reduction to UNiprocessor (RUN) (REGNIER et al., 2011), Quasi-Partitioning Scheduling (QPS) (MASSA et al., 2016), and Unfair EDF (U-EDF) (NELISSEN et al., 2012). RUN transforms the problem of multiprocessor scheduling into an equivalent series of uniprocessor scheduling problems. EDF-based servers is the basis for solving each of them, whose solutions are transformed back at runtime into a multiprocessor schedule. Only periodic tasks are dealt with by RUN. QPS resembles RUN in some aspects (MASSA; LIMA; REGNIER, 2014) but uses a set of more complex mechanisms to deal with sporadic tasks. U-EDF is a modification of global EDF. It assigns jobs to processors at each scheduling instant. When a set of jobs is not feasible on a processor, a job is split into two pieces

and the pieces are suitably scheduled by EDF on each processor so that their execution does not overlap in time. Although U-EDF does not generate too many preemptions and task migrations, it requires $O(n \times m)$ operations per scheduling instants, with n and m denoting the numbers of task and processors respectively. This may be a point of concern for some large systems in terms of runtime overhead.

In this work we will focus on the RUN algorithm. This choice is to avoid both the extra complexity of QPS as well as keeping a low runtime overhead. RUN will be better explained in Section 2.3.

2.2 RESOURCE SHARING PROTOCOLS

Actual real-time applications make access to resources that need to be shared among different tasks. Resources can be memory, files, devices, queues, buffers *etc.*. Usual solutions designed to general purpose systems do not apply in the real-time domain due to the timeliness requirements. Possible problems to be avoided are

- *Uncontrolled priority inversion*, which occurs when a low priority job accesses resources that are used by a high priority one. The latter can be blocked by both the locking lower priority job and all other jobs that may preempt it.
- *Deadlocks*, possibly caused when two or more jobs need to lock a different set of resources and they do so concurrently and in a reversed order. One job can be forced to wait for another indefinitely.
- *Lengthy blocking chains*, formed when a high priority job J suffers successively priority inversions by several lower priority jobs which had locked resources accessed by J but before J started its execution.

These problems are avoided by resource sharing protocols. In general, they work by accelerating the execution of low priority jobs that lock resources shared with high priority jobs. A common mechanism to do so is via temporally increasing the priority of the locking job. There are a number of solutions to do so. For example, in uniprocessor systems, solutions like Priority Ceiling Protocol (PCP) (SHA; RAJKUMAR; LEHOCZKY, 1990), designed for fixed-priority scheduling, work as follows. Each resource has a ceiling priority, which is equal to the priority of the highest priority task that can access this resource. When a job accesses a shared resource, its priority is raised to the priority ceiling of that resource, returning to its previous priority upon unlocking the corresponding resource. Also, any job can lock any of its resources at time t only if its priority is higher than the ceiling priority of all blocked resources at t . Stack Resource Policy (SRP) (BAKER, 1990) is an extension of the same principles but aiming at EDF-scheduled systems.

The aforementioned solutions do not apply to the multiprocessor case since raising priorities on a processor does not necessarily cause any interference on the scheduling decisions taken on another processor. For partitioned scheduled systems, when a resource can be referenced by tasks allocated to more than one processor, it is labeled as *global*. Otherwise, it is a *local* resource. If a task/job accesses only local resources, interferences

due to possible lockings are confined to the processor this resources is associated with. The effects of accessing global resources may be propagated on other processors. For systems scheduled by global algorithms, there are only global resources. Multiprocessor resource sharing protocols must take into consideration which type of resource is being required, local or global.

Another source of complication for resource sharing protocols in multiprocessor is whether or not the scheduling algorithm is hierarchical, employing some server-based scheme. In this case, a task (or server) may hold resources between server budget replenishments possibly causing too much blocking for other tasks requesting the same resources.

Resource sharing protocols that are based on blocking tasks that require access to resources already locked by other tasks may use two types of blocking schemes, *suspension* or *busy-waiting* (*i.e.*, spinning). In both schemes, an ordered (by some criteria) queue is associated with each shared resource and blocked tasks (or servers in a hierarchically scheduled system) are kept in the queue while waiting for their required resources. The suspension scheme blocks the task that cannot gain access to a locked resource by releasing the processor on which it was executing. The busy-waiting scheme, on the other hand, executes a dummy code on the processor the blocked task was executing during the time it is waiting for the requested resource. Both schemes can also apply to servers. Busy-waiting may result in an extended blocking period for the waiting tasks on others processors if the task which is locking resources undergoes preemption (BURNS; WELLINGS, 2013).

The following two subsections give some brief information about the protocols most related to this work. Some of them were designed for hierarchically scheduled systems whereas others do not offer any special mechanism for server-based scheduling.

2.2.1 Non-Hierarchically Scheduled Systems

MSRP - Multiprocessor Stack Resource Policy. MSRP (GAI; LIPARI; Di Natale, 2001) is a busy-waiting-based resource sharing protocol that can be used for partitioned fixed-priority scheduling or EDF. Its non-preemptive characteristic can lead to long blocking time for high priority tasks, which may make the protocol unacceptable for some workloads. MSRP limits the cost of accessing global shared resources by restricting cumulative requests per processor. In order to do this the protocol raises the priority of the task requesting the resource and the global resources are served through a FIFO queue. The task that cannot gain access to the resource is blocked and busy-waits until the access is granted. When the accessed resource is local the task can only be blocked once for that resource and the blocking will occur before the task starts executing.

DPCP - Distributed Priority Ceiling Protocol. Rajkumar (RAJKUMAR; SHA; LEHOCZKY, 1988) presented the Distributed Priority Ceiling Protocol (DPCP) that does not use shared memory for synchronizing access to shared resources, so the protocol can be used in multiprocessor real-time systems and in distributed systems. DPCP provides local agents, located on the processors where the resources will be served, to take care of synchronization in accessing shared resources. A job makes a request to the agent

and suspends itself while waiting for the agent to respond, after receiving the response, it resumes its execution (BRANDENBURG; ANDERSON, 2008). The protocol was originally proposed considering that each resource is assigned to a single synchronization processor, a processor which controls the accesses to shared resources. This constraint has been removed to allow for multiple synchronization processors (BURNS; WELLINGS, 2013).

MPCP - Multiprocessor Priority Ceiling Protocol. MPCP, described in (RAJKUMAR, 1990), uses semaphores to synchronize access to shared resources and uses suspension when the requested resource is locked. By the MPCP rules, a job runs at the highest priority on its processor when it is accessing global resources. Deadlocks are avoided by not allowing global resources to be accessed in a nested way (BRANDENBURG; ANDERSON, 2008).

FMLP - Flexible Multiprocessor Locking Protocol. FMLP, as described in (BLOCK et al., 2007), is a resource sharing protocol that can be used for systems scheduled by partitioned or global approaches. It uses busy-waiting or suspension for blocking tasks. Resources are classified as short or long and are grouped into these classes accordingly. The short resource group is accessed through a non-preemptive FIFO queue and tasks spin when blocked. The long resource group is accessed through a semaphore and task undergoes suspension in a FIFO queue when blocked. Non-nested resources are individually grouped.

MrsP - Multiprocessor resource sharing Protocol. MrsP has been designed in (BURNS; WELLINGS, 2013) to be a very flexible protocol in terms of which type of scheduling mechanism is employed. Although the protocol is similar to MSRP in some aspects, according to the authors, MrsP is “applicable to partitioned, semi-partitioned, or global scheduling with fixed priority, EDF or other form of urgency designation”. MrsP exhibits the same properties of the uniprocessor protocols PCP (SHA; RAJKUMAR; LEHOCZKY, 1990) and SRP (BAKER, 1990). Like SRP, it can also be used by any system whose priority ceilings can be assigned off-line. Although MrsP has originally not targeted at hierarchically scheduled systems neither is there up to now any implementation of the protocol for dynamic-priority systems, we believe that its flexibility allows for applying the protocol to systems scheduled by RUN. This observation has motivated us to choose MrsP for our research. Section 2.4 will give more details about the protocol.

2.2.2 Hierarchically Scheduled Systems

HSRP - Hierarchical Stack Resource Policy. Davis (DAVIS; BURNS, 2006) have proposed HSRP, an extension of SRP for hierarchical systems. According to HSRP, the SRP rules are applied to local resources, and the ceiling level is given by the highest priority task of the same server that can request the resource. While accessing a local resource, the priority of the task is increased. However, when the server’s budget is exhausted, its execution is suspended. The authors have also defined a ceiling priority for each global resource, which is given by the highest priority server that can use it. A task will execute non-preemptively while blocking a global resource and its server will have the priority increased. The server will continue its execution on budget exhaustion only if

this occurs within a critical section (optionally the budget exceeded can be compensated). The use of non-preemptive access to global resources limits the excess of budget that the server can request. Prolonged access to shared resources in hierarchically scheduled systems reduces schedulability. The reimbursement mechanism can help schedulability, but increases the complexity of the solution.

BROE - Bounded-Delay Resource Open Environment. This is another extension to SRP, described by Biondi (BIONDI; BUTTAZZO; BERTOIGNA, 2013). The protocol uses servers for employing a reservation mechanism. When the critical section of the running task does not fit into the budget of the its active server, there is an anticipated budget replenishment. This is done without jeopardizing schedulability. According to the rules of the protocol, each resource receives, per server, a static ceiling level, which equals the highest priority level of the tasks allocated to the server that use it. A task can only preempt another task of the same server if its level is greater than the server ceiling level. The system's ceiling level (dynamic) is the highest ceiling level of the resources that are being used at any given time and a server is allowed to preempt another server only when its preemption level is greater than the system's ceiling level.

SBLP - Server Based Locking Protocol Bonato et al. (BONATO; MEZZETTI; VARDANEGA, 2014) implemented SBLP based on RUN using servers to group and manage collaborative tasks (*i.e.*, tasks which share logical resources) with the intention to decrease the maximum degree of parallelism at resource usage. Their work generalized the properties of PCP and SRP protocols to multiprocessor systems embedding pessimism needed to accommodate blocking situations not foreseen by the precursor protocols, which were designed for single processor systems. They also produced a simple scheduling test that takes into consideration the effects of SBLP on the schedule.

An important feature of this protocol is that SBLP critical sections are non-preemptive from the point of view of a server. From the moment a task requests a resource until it completes its critical section, the task cannot be preempted by another task of the same server. However, a task may undergo preemption even within its critical section if its server is preempted.

Resources that can be accessed by tasks belonging to different servers are called global resources. To prevent a task from being indefinitely waiting for a global resource blocked by another server that is not running, SBLP servers can give way part of their budgets to the server whose task is locking a resource under dispute (BONATO; MEZZETTI; VARDANEGA, 2014).

A dedicated FIFO queue is allocated for each global resource. The maximum size of the queue of a global resource corresponds to the number of servers containing tasks which can access such a resource. Under SBLP the maximum time a task can wait for resources accessed by two or more servers is bounded by the size of the queue and the maximum size of the resource critical section. If a task requests a resource, it must busy wait while it is not at the head of the FIFO queue.

2.3 THE RUN SCHEDULING ALGORITHM

RUN is an optimal scheduling algorithm originally designed for independent implicit deadline periodic tasks. That is, it is assumed that tasks are strictly periodic, their deadlines are equal to the respective periods, and they do not share any resource but the processors. According to (COMPAGNIN; MEZZETTI; VARDANEGA, 2014), “the algorithm represents an original approach to multiprocessor scheduling that exhibits the prerogatives of both global and partitioned algorithms without incurring the respective drawbacks”.

Instead of formally explaining RUN, we will illustrate how it works with a simple example. Let Γ be a set of 5 tasks to be scheduled on 3 identical processors. Each task τ_i requires C_i time units of execution at each period of T_i . This means that each task requires a processor execution rate of C_i/T_i . The utilization of τ_i , defined as $U_{\tau_i} = C_i/T_i$ represents this requirement. In our illustrative example, the set of Γ 's tasks utilization is $[0.7; 0.6; 0.7; 0.5; 0.5]$, for $i = 1, 2, \dots, 5$ and their periods are as follows: $[60; 40; 60; 40; 60]$. These tasks can be represented as tuples $\tau_i(U_{\tau_i}, T_i\mathbb{N}^+)$ meaning that the τ_i requires $C_i = U_{\tau_i}T_i$ at every multiple of T_i .

RUN first performs a sequence of operations to transform the set of tasks into a set of servers to be scheduled in single processor systems. This is carried out off-line. Then the information generated off-line is used on-line to schedule the original system via scheduling the transformed systems. For the given example, during the off-line phase, RUN packs the set of 5 tasks into servers. This is called PACK operation. As each server cannot deal with a set of clients requiring more than one processor, only one task can be packed into a single server for the tasks τ_1 , τ_2 and τ_3 , while tasks τ_4 and τ_5 can be grouped into a single server. Thus the first PACK operation returns 4 servers in the format: $\sigma_1(0.7, 60\mathbb{N}^+)$; $\sigma_2(0.6, 40\mathbb{N}^+)$; $\sigma_3(0.7, 60\mathbb{N}^+)$; $\sigma_4(1.0, 60\mathbb{N}^+, 40\mathbb{N}^+)$. Note that servers in RUN are denoted by its utilization, which equals the sum of its clients utilizations, and the instants it releases a job, which corresponds to when its clients release a job. Note also that σ_4 is a unit server, *i.e.*, its utilization equals to one and will require one virtual processor for scheduling its tasks. σ_4 will not be considered in subsequent operations as it is a unit server and will be scheduled separately as on a semi-partitioned clustered system.

In the second step, RUN carries out the DUAL operation on the set of generated servers. Let $\{T_{\sigma_i}\mathbb{N}^+\}$ be the set of release times of σ_i . A dual server of $\sigma_i(U_{\sigma_i}, \{T_{\sigma_i}\mathbb{N}^+\})$ is one with complementary utilization and the same release instants in $\{T_{\sigma_i}\mathbb{N}^+\}$, *i.e.*, $\sigma_i^*(1 - U_{\sigma_i}, \{T_{\sigma_i}\mathbb{N}^+\})$. In the illustration, this operation will give rise to three new servers, denoted as $\sigma_1^*(0.3, 60\mathbb{N}^+)$; $\sigma_2^*(0.4, 40\mathbb{N}^+)$ and $\sigma_3^*(0.3, 60\mathbb{N}^+)$. Each σ_i^* is called the dual of its primal counterpart σ_i . RUN aims at reaching sets of servers that require a single processor. In this case, another phase is not necessary, as indicated in Table 2.1. As can be seen, each set of servers identified as Γ^j corresponds to the set of packed servers produced by previous DUAL operation, $j > 0$. Also, note that the set of release instants of a server is the union of the release instants sets of its clients. The REDUCE operator is a combination of the operations PACK and DUAL. It is also worth noticing that the off-line phase of RUN produces one or more trees, called *reduction tree* with the root defined to be the single processor system, σ_5 and σ_4 in the example.

Table 2.1 The result of the off-line phase in RUN. The original set of tasks is transformed into a system to be scheduled on a single processor.

Γ	$\tau_1(0.7, 60\mathbb{N}^+)$	$\tau_2(0.6, 40\mathbb{N}^+)$	$\tau_3(0.7, 60\mathbb{N}^+)$	$\tau_4(0.5, 60\mathbb{N}^+)$	$\tau_5(0.5, 40\mathbb{N}^+)$
$\Gamma^0 = \text{PACK}(\Gamma)$	$\sigma_1(0.7, 60\mathbb{N}^+)$	$\sigma_2(0.6, 40\mathbb{N}^+)$	$\sigma_3(0.7, 60\mathbb{N}^+)$	$\sigma_4(1.0, 60\mathbb{N}^+, 40\mathbb{N}^+)$	
$\Gamma^{0*} = \text{DUAL}(\Gamma^0)$	$\sigma_1^*(0.3, 60\mathbb{N}^+)$	$\sigma_2^*(0.4, 40\mathbb{N}^+)$	$\sigma_3^*(0.3, 60\mathbb{N}^+)$		
$\Gamma^1 = \text{PACK}(\Gamma^{0*})$	$\sigma_5(1.0, 40\mathbb{N}^+, 60\mathbb{N}^+)$				

Servers in RUN use EDF to schedule their clients. That is, when a server σ is scheduled to execute, it selects the earliest deadline client to execute. Only one client can execute at a given time within a server. If d is the earliest deadline of a client, the budget of a server σ is defined at its release instant r as $U_\sigma(d - r)$. If the budget of a server is depleted, the server, and hence its clients, cannot be selected to execute until the budget is replenished.

The on-line phase of RUN makes use of the reduction tree, the server behavior and the duality principle, which states that a server runs if and only if its dual does not run. Hence, RUN selects first the server to execute at the root of the tree. In the example σ_2^* would be assigned to a virtual processor identified as V_1 . This server has the earliest deadline of its unit server (σ_5) at time 0. After having its budget depleted, another server is selected via EDF and so on. These servers actually do not execute since they are *virtual* servers. The processors they are assigned to in fact do not exist. Defining virtual processors for executing all servers generated after the first DUAL operation provides a useful abstraction, though. Figure 2.1 illustrates how the set of dual servers could be scheduled by RUN, all of these servers belong to the σ_5 unit server.



Figure 2.1 RUN schedules servers at the root of the reduction tree by applying EDF. Time scale is indicated at the bottom.

By applying the duality principle, RUN decides which server must run at the second level of the reduction tree. For example, while σ_2^* runs within time interval $[0, 16]$ σ_2 must not run. This means that during this interval, both σ_1 and σ_3 should be scheduled. The same reasoning applies to all time instants. RUN thus would produce the following schedule at the second reduction level, as illustrated in Figure 2.2. It can be seen from the figure that virtual processor V_1 is related to virtual processors V_3 and V_4 after performing the DUAL operation. σ_4 now appears as it is the second unit server in the system and its level in the tree is reduced compared to the level of unit server 5. It must also execute at time 0 in a virtual processor identified as V_2 .

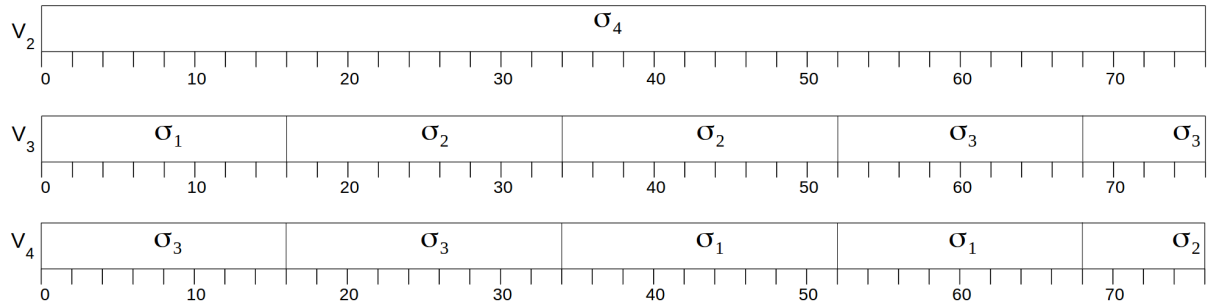


Figure 2.2 RUN schedules the servers at the second level of the reduction tree by applying the DUAL operation. Time scale is indicated at the bottom.

Based on the servers running at the second reduction level, RUN selects the tasks to be scheduled at the next level by applying EDF. The result is shown in Figure 2.3. Recall that servers σ_1 , σ_2 and σ_3 contain only one task each, so the association is straightforward. Server σ_4 , in turn, has two tasks, so EDF scheduling was applied, as can be seen from figure 2.3, where task τ_5 , which is the closest deadline at time 0, comes first. Note that virtual processors have been replaced by physical processors P_1 , P_2 , and P_3 , indicating that this is the final scheduling step.

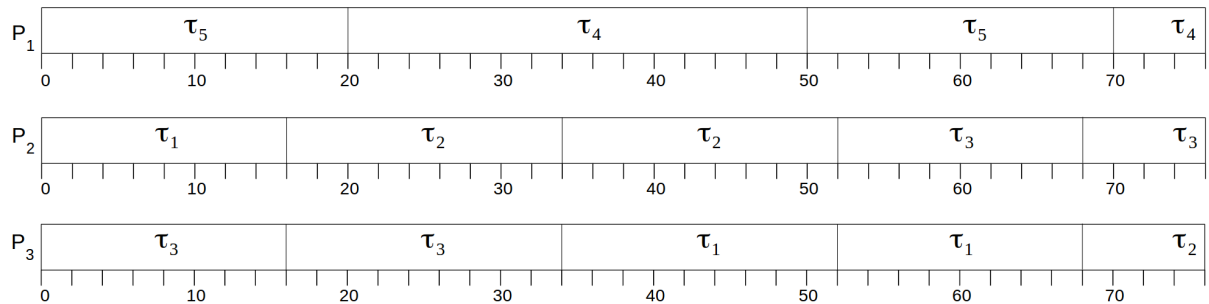


Figure 2.3 RUN schedules tasks at the final level of the reduction tree by applying EDF. Physical processors are present in place of virtual processors. Time scale is indicated at the bottom.

We note that RUN starts by scheduling a virtual system at the root of the reduction tree. It needs that this system has utilization equal to 1. This means that there is always some server running at the last level in the reduction hierarchy. Fully utilized systems at the root of the tree are produced when the original system is also fully utilized. In the example of this section, the five tasks jointly utilize 100% of the 3 processors. When this is not the case, one needs to create dummy tasks to complete the system, as described in (REGNIER et al., 2011).

2.4 THE TARGET MULTIPROCESSOR RESOURCE SHARING PROTOCOL

In this section we will describe MrsP, giving the necessary information to justify the choice made in this project. Although this protocol was not specifically designed for hierarchically scheduled systems and RUN makes use of servers, they can be made compatible as will be seen in the next chapter. This section is to give some intuition on this goal.

MrsP was based on MSRP (GAI; LIPARI; Di Natale, 2001). The main difference between them is that MrsP uses what is known as the *helping mechanism*. Other protocols also make use of this mechanism, *e.g.*, SPEPP (TAKADA; SAKAMURA, 1997) and SBLP protocol (BONATO; MEZZETTI; VARDANEGA, 2014). This mechanism is to allow a task τ_i to give away its processor in favor of another task τ_j (allocated to a different processor) if τ_j is locking a resource ψ and τ_i is busy-waiting for ψ while τ_j is being preempted by higher priority tasks on its processor. Task τ_i in this case gives way its processor so that its busy-waiting time can be used for accelerating the execution of preempted tasks. By helping other tasks in this way τ_i is actually helping itself since the resources it waits for will be released earlier. Recall that all tasks requesting an already locked resource are enqueued in the waiting FIFO queue of that resource. The helping mechanism decreases the total waiting time τ_i waits in the queue.

According to Burns (BURNS; WELLINGS, 2013), when it comes to local resources, the following properties are inherited from PCP/SRP: a job can be blocked at most once after release; the blocking will happen before the job starts executing; once a job is running it will have access to all required resources; and deadlocks are prevented. In addition to meeting the properties related to resource sharing for uniprocessor systems, MrsP also provides bounded duration of blocking on multiprocessors and consequently bounded response time for the system tasks. To show this, the authors used Response Time Analysis (RTA) (JOSEPH; PANDYA, 1986), technique commonly used for fixed-priority uniprocessor systems. In the MrsP context with tasks being scheduled in a partitioned fixed-priority basis, they have shown that by increasing the resource access cost to account for parallel resource accesses (global resources), RTA is kept almost the same as in the case of uniprocessor system.

These statements by Burns (BURNS; WELLINGS, 2013) summarize the definition of MrsP:

- Each resource is associated to a set of ceiling priorities, one priority per resource for each processor that uses it.
- A request for a resource access by a task raises its priority to the local resource ceiling.
- Resource accesses are treated in First In First Out (FIFO) order in the resource global queue.
- While busy-waiting for (or using) a resource, a task remains active and executes with priority equal to the local resource ceiling.

- A busy-waiting task τ_i blocked for the resource ψ must give away its processor to another task which is locking ψ but has been preempted.
- A collaborative task (which gives way its processor) must do so and allowing for execution of other tasks following the FIFO order of requests.

Figure 2.4 gives a visual representation of the MrsP protocol for systems scheduled by some partitioned policy. Each processor P_i , $1 \leq i \leq m$, has a set of tasks statically assigned to it. There is a FIFO queue for each resource ψ . The length of the FIFO queue is equal to the number of processors having tasks accessing ψ . When a task τ_y is executing on the processor P_m and tries to access ψ , if the resource is already locked by a task assigned to other processor, then τ_y will spin at the local ceiling of ψ on processor P_m .

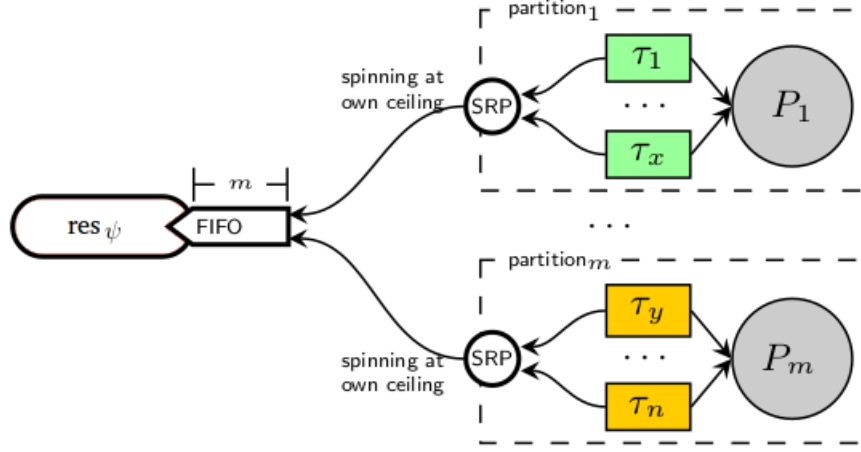


Figure 2.4 Visual Representation of MrsP (CATELLANI et al., 2015)

For minimizing the number of global resources, Burns (BURNS; WELLINGS, 2013) have suggested that mappings could be used by assigning tasks that access the same resources to the same processor cluster. Although this practice has been suggested in the context of partitioned scheduling, it is not difficult to imagine this strategy being used in global scheduling. For example, Huang (HUANG; YANG; CHEN, 2017) have shown that “the proposed resource-oriented partitioned scheduling using PCP combined with a reasonable allocation algorithm can achieve a non-trivial speedup factor guarantee”.

Interestingly, despite the fact that RUN is a global scheduling algorithm, integrating RUN and MrsP could be thought similarly to applying MrsP in systems scheduled by some partitioned approach. The set of servers generated by the first PACK operation works as if each server was a processor with reduced capacity. Thinking of these servers as “processors” is the base of our research. More details of idea will be given in Chapter 3.

SYSTEM MODEL AND NOTATION

We assume a system composed of a set Γ of periodic real-time tasks with implicit deadlines to be scheduled on a multiprocessor platform comprised of m identical processors. Period and computation time of task $\tau_i \in \Gamma$ is denoted by T_i and C_i , respectively. Utilization of task τ_i is defined as $U_{\tau_i} = \frac{C_i}{T_i}$ and the initial system utilization is obtained through $\sum_{\tau_i \in \Gamma} U_{\tau_i}$. Each task $\tau_i \in \Gamma$ has a preemption level, denoted Π_{τ_i} . This is a value defined such that for any two tasks τ_i and τ_j in Γ , if $T_i < T_j$, then $\Pi_{\tau_i} > \Pi_{\tau_j}$.

A server σ is an execution entity that can serve (*i.e.*, schedules) one or more tasks when it executes. A task τ_i , if served by a server σ , is said to belong to the client set of σ , denoted Γ_σ . Rules governing the server behavior will be detailed shortly. The server σ on which a task τ_i is allocated is denoted as δ_{τ_i} .

A special server set is referenced as Γ^0 which corresponds to the set of servers generated after the first RUN pack operation. For short, all references to σ denote a server in the set Γ^0 , except when explicitly said otherwise.

The system contains a resource set Ψ , which can be shared by the system tasks. The set of tasks in Γ that may access some resource ψ is denoted as Γ^ψ . Servers in Γ^0 which have some task in Γ^ψ is denoted as the set δ^ψ . We refer to Γ_σ^ψ as the task set served by σ that may access resource ψ . More precisely, $\Gamma_\sigma^\psi = \Gamma_\sigma \cap \Gamma^\psi$. We also identify the resource set used by a task τ_i as Ψ_{τ_i} whereas the resource set that may be accessed by some client tasks of a server σ is denoted as Ψ_σ . More formally, $\Psi_\sigma = \{\psi \in \Psi \mid \exists \tau_i \in \Gamma_\sigma^\psi\}$. The local task set with preemption level greater than that of task τ_i is given by $hpl(\tau_i)$. Formally stated,

$$hpl(\tau_i) = \bigcup_{\tau_j \in \Gamma_{\delta_{\tau_i}} \mid \Pi_{\tau_j} > \Pi_{\tau_i}} \{\tau_j\}.$$

We define predicates to indicate whether at a given instant t some resource ψ is locked or there is some task or server awaiting locking it. Locking predicate for a task is defined as

$$\text{lock}(\psi, \tau_i, t) = \text{true} \Leftrightarrow \psi \text{ is locked by } \tau_i \text{ at time } t$$

For a server, the predicate is cast as

$$\text{lock}(\psi, \sigma, t) = \text{true} \Leftrightarrow \exists \tau_i \in \Gamma_\sigma \text{ such that } \text{lock}(\psi, \tau_i, t)$$

Similarly,

$$\text{wait}(\psi, \tau_i, t) = \text{true} \Leftrightarrow \tau_i \text{ waits for locking } \psi \text{ at time } t$$

and

$$\text{wait}(\psi, \sigma, t) = \text{true} \Leftrightarrow \exists \tau_i \in \Gamma_\sigma \text{ such that } \text{wait}(\psi, \tau_i, t).$$

In this work we do not address nested resources and assume that the maximum execution cost of a critical section on a shared resource ψ is known and represented as C^ψ .

For convenience, we say the maximum of elements in a set is null if the set is empty, i.e., $\max\{\} = 0$.

3.1 INTEGRATING MrsP AND RUN

This section states the properties that should hold in a correct integration of MrsP and RUN. We first observe that, according to Baker (BAKER, 1990, 1991), SRP can be used by any system which accepts a static way to define task eligibility, *i.e.*, task's preference to execute can be defined off-line. This is a crucial question to determine the MrsP compatibility with a global scheduling algorithm like RUN. Tasks priorities in RUN's servers are determined by EDF, which is dynamic, and thereby cannot be determined off-line. Thus, as suggested by Baker (BAKER, 1990) for SRP, for each task τ_i we intend to make use of its preemption level Π_{τ_i} , gathered off-line, in combination with its EDF priority. Combining both properties is necessary because EDF is a dynamic-priority policy and SRP-like systems have to provide a static eligibility criterion for defining an order of preference in resource sharing.

The second observation on which this work is based is related to the servers in Γ^0 , created by the first step of packing in the off-line phase of RUN. These servers can be seen as “logical” processors and they can have an analogous role the physical processors have from the MrsP's perspective. Interestingly, this interpretation is not original, as it has been explored in the design of SBLP, a protocol specifically designed for RUN (BONATO; MEZZETTI; VARDANEGA, 2014).

By interpreting servers in Γ^0 as logical processors, we can start modifying MrsP original rules accordingly. MrsP treats a resource as global when it is used by tasks allocated to different processors. Here we intend to treat a resource as global when it is used by tasks allocated to different servers. According to this approach, the system composed of servers in set Γ^0 can be thought of as similar to a system scheduled by the partitioned EDF policy.

Although the analogy between servers and processors is very intuitive, the integration of MrsP and RUN must be formally proved as correct. Some subtle differences between servers and processors exist and must be accounted for in such a proof. For example, the interference tasks may suffer due to blocking differ from the original scenarios MrsP has been designed for. Indeed, servers can be preempted during their execution, as previously

Table 3.1 Notations Summary

Γ	Task set or server set
Γ^0	Packed server set generated by the first packing phase of RUN
τ_i	Task with index i
σ, σ_i	Generic server, server with index i
C_i, T_i	WCET and period of τ_i
\hat{C}_i	Inflated WCET of τ_i
$U_{\tau_i} = C_i/T_i$	Utilization of τ_i
U_σ	Utilization of σ
Π_{τ_i}	Preemption level of τ_i
Γ_σ	Client set of σ
δ_{τ_i}	Server τ_i is assigned to
Ψ	System's resource set
ψ, ψ_i	Generic resource, resource with index i
Γ^ψ	Task set that may access a resource ψ
Γ_σ^ψ	Tasks served by σ that may access a resource ψ
δ^ψ	Servers containing some client in Γ^ψ
Ψ_{τ_i}	Resource set used by τ_i
Ψ_σ	Resource set that may be accessed by some client of σ
Π_σ^ψ	ψ 's ceiling level for σ
$\Pi_\sigma(t)$	σ 's ceiling level at a given instant t
C^ψ	Maximum execution cost of a critical section on ψ
B^ψ	Maximum blocking time for an attempt to access ψ
$N_{\tau_i}^\psi$	Number of times a task τ_i can use ψ
$\text{hpl}(\tau_i)$	Set of tasks in the same server as τ_i and with preemption level greater than or equal to that of τ_i
$\text{lpl}(\tau_i)$	Set of tasks in the same server as τ_i and with preemption level lower than that of τ_i
$\text{lock}(\psi, \tau_i, t)$	True if and only if ψ is being locked by τ_i at a given time instant t
$\text{lock}(\psi, \sigma, t)$	True if and only if ψ is being locked by a client of σ at a given time instant t
$\text{wait}(\psi, \tau_i, t)$	True if and only if τ_i is waiting for ψ at a given time instant t
$\text{wait}(\psi, \sigma, t)$	True if and only if a client of σ is waiting for ψ at a given time instant t
$\text{lblock}(\tau_i)$	Maximum local blocking time τ_i can be exposed
$\text{gblock}(\tau_i)$	Maximum global blocking time τ_i can be exposed

mentioned. If some of their clients hold resources during intervals the respective servers are not running, extra blocking interferences can be in place. Moreover, RUN selects which servers are to run using duality from one reduction level to another. This behavior, not present in partitioned scheduling algorithms, must be taken into consideration when integrating MrsP and RUN. First, we list the main properties that should hold so as to make MrsP and RUN compatible with each other and then we present the definitions and rules necessary to substantiate the proof of the correctness of the proposed solution.

1. A task only can start executing if all resources it needs are not locally blocked by another task of the same server. See Lemma 1 and Rule 1.
2. A higher priority task τ_i can only preempt another task, client of the same server, if τ_i 's preemption level is higher than the server's ceiling. This property is specified in Rule 1.
3. After starting execution, a task cannot be locally blocked by another task of the same server. Lemma 2 defines this property.
4. The local blocking time (*i.e.*, the time during which a task is blocked by another task client of the same server) on resource ψ can be bounded and the bound is known. This is related to Lemma 2.
5. The maximum potential blocking cost a task can experience in one attempt to access a shared resource is upper bounded and can be derived. Lemma 3 proves it.
6. Priority inversion may occur but its effect is bounded. This property has its definition given by Lemma 2.
7. As global resource access is controlled by one FIFO queue per resource, then no server is able to interfere more than once in another server in a dispute over a shared global resource. One can see Rule 2 and Lemma 3.

3.2 RESOURCE SHARE PROTOCOL AND ITS PROPERTIES

Definition 1 (Resource Ceiling Level). *Each resource $\psi \in \Psi$ has a ceiling level associated with each server σ , denoted Π_σ^ψ , which corresponds to the maximum of the preemption levels of σ 's clients that have access to resource ψ . More formally,*

$$\Pi_\sigma^\psi = \max_{\tau_i \in \Gamma_\sigma^\psi} \Pi_{\tau_i}$$

Definition 2 (Server Ceiling). *At a given instant t , the ceiling level of a server σ represents the maximum ceiling level of resources either locked or being required by some of σ 's clients at t . If no resources are being locally locked or requested by any client of σ , its ceiling level is null. That is,*

$$\Pi_\sigma(t) = \max\{\Pi_\sigma^\psi \mid \psi \in \Psi_\sigma, \text{lock}(\psi, \sigma, t) \vee \text{wait}(\psi, \sigma, t)\}$$

Definition 3 (Locally Available Resource). *A resource is said to be locally available within a server at some instant t when none of its clients is locking or is waiting for such a resource at t .*

Definition 4 (Globally Available Resource). *A resource ψ is said to be globally available at some instant t when none of the tasks $\tau_i \in \Gamma^\psi$ is locking or is waiting for ψ at t .*

Lemma 1. *Let σ be a server and consider some task $\tau_i \in \Gamma_\sigma$. If at a given instant t , $\Pi_{\tau_i} > \Pi_\sigma(t)$, then all resources in Ψ_{τ_i} are locally available at t .*

Proof. By contrapositive. Suppose that $\psi \in \Psi_{\tau_i}$ is a resource requested or locked by another task $\tau_j \in \Gamma_\sigma$ at some time t . From Definition 1, it is known that $\Pi_\sigma^\psi \geq \Pi_{\tau_i}$. Definition 2 tells that $\Pi_\sigma(t) \geq \Pi_\sigma^\psi$ at instant t , implying that $\Pi_{\tau_i} \leq \Pi_\sigma(t)$. ■

From Lemma 1 it is clear that a task can have access to its required resources if its preemption level is greater than its server ceiling, leading to the following rule:

Rule 1 (Local Blocking). *Whenever a server σ in Γ^0 runs, it schedules its client tasks by EDF but a client τ_i may only be scheduled to execute at a given time t if $\Pi_{\tau_i} > \Pi_\sigma(t)$.*

Rule 1 establishes a simple blocking criterion and comes from classical protocols such as SRP (BAKER, 1990) and is also present in MsrP (BURNS; WELLINGS, 2013). The only difference here is that instead of real processors, the concept of logical processors (*i.e.*, servers) are being used. When a task is blocked due to Rule 1, we say that it suffers a local blocking or equivalently is locally blocked.

Lemma 2. *A job of any task is locally blocked due to at most one critical section and the local blocking period occurs before the job starts its execution.*

Proof. For a job J of a task $\tau_i \in \Gamma_\sigma$ to start its execution and then be locally blocked at some time t , it is necessary that some other job from a client of the same server as τ_i requests a resource at time t' before t and this resource has a ceiling higher than or equal to the preemption level of τ_i . However, this scenario is prevented by Rule 1 since the server preemption level would be increased at t' to at least Π_{τ_i} (recall Definitions 1 and 2).

In order to see that the local blocking period suffered by J is due to a single critical section note that J would suffer more than one local blocking only if lower priority jobs must have requested at least two resources used either by τ_i or by higher preemption level tasks, *i.e.*, tasks in $\text{hpl}(\tau_i)$. However, Lemma 1 (*i.e.*, its contrapositive statement) implies that the first request of such a kind of resource by any of these lower priority jobs makes σ 's preemption level raise to at least Π_{τ_i} . As other lower priority jobs do not start their execution until σ 's preemption level is decreased accordingly and J has a higher priority, no other local blocking of J will be possible. ■

Rule 2 (Global resource FIFO service). *If a resource can be accessed by clients of different servers then it is called a global resource. There is one queue associated with each global resource and resources are served in FIFO order. When a task $\tau_i \in \Gamma_\sigma$ requests a global*

resource ψ and ψ is locked by a task $\tau_j \notin \Gamma_\sigma$, the request is enqueued in ψ 's FIFO queue and τ_i busy-waits whenever its server σ schedules τ_i for execution. If the resource is not locked, ψ is immediately locked by τ_i . The request at the first position of the FIFO queue is dequeued when the task which is locking the corresponding resource unlocks it.

Lemma 3. *Any job that requests access to a global resource ψ suffers at most $|\delta^\psi| - 1$ global blockings before accessing ψ .*

Proof. A server cannot issue two concurrent requests for a resource. Indeed, consider two jobs by tasks τ_i and τ_j , clients of the same server, and assume that they share ψ . Further, assume that one of them requests ψ before the other. Upon the first request, their server ceiling level rises to at least $\max(\Pi_{\tau_i}, \Pi_{\tau_j})$. Rule 1 then prevents a second request from happening until at least the job that locked ψ first releases the corresponding lock. As by Rule 2 a request is removed from the queue upon unlocking, the two requests cannot be in the queue at the same time. Thus, the queue has at most $|\delta^\psi|$ requests at any given time. Since it is a FIFO queue, *i.e.*, new requests are always enqueued at the end of the queue, there are at most $|\delta^\psi| - 1$ requests to be serviced before the last one is processed. This leads to the maximum number of global blockings stated in the lemma. ■

Lemma 4. *Under Rules 1 and 2, there can be no deadlock during concurrent access to resources provided that no nested resources are allowed.*

Proof. From Rule 1 and Lemma 2 we know that any job that does not access global resources is blocked at most once and this blocking occurs before the job starts its execution. In this case, deadlock cannot occur as the local resources for each executing job are not blocked by other jobs.

As for global resource sharing, let τ and τ' be tasks of distinct servers sharing global resources. According to Rule 2, their jobs are put in the global queue of the resources upon their request. Without loss of generality, assume that at a given instant in time the job by τ comes before that of τ' in the queue of some of the resources they share, namely ψ . Let these jobs be J and J' , respectively. Since both jobs are in the queue of ψ , they have released all resources they have locked before requesting ψ as there is no nested resource. Hence, when J gains access to ψ , J' is kept awaiting accessing ψ while J is allowed to execute in the critical region of ψ . Thus, no resource waiting cycle takes place for any two jobs and so no deadlock can occur. ■

Avoiding the use of nested resources is a decision that prevents the formation of deadlocks. Instead of prohibiting nested resources, deadlocks could be avoided by grouping resources or ensuring that resources were always accessed in a given order. The main advantage in avoiding the use of nested resources is to simplify the application of MrsP protocol, since nested resources increase the complexity of the solution, as discussed in Garrido et al (GARRIDO et al., 2017b).

Rule 3 (Helping Mechanism). *This mechanism acts on accelerating the execution of client jobs served by distinct servers. Let J and J' be jobs released by client tasks of two distinct servers, σ and σ' , respectively, and assume that J and J' share a (global)*

resource ψ . Whenever J requests ψ and ψ is being locked by J' , if J' is not running (due to preemption, for example), then J' is brought to execute in the context of server σ as long as σ is executing and J is its highest priority job. In this case J and J' are called *helping* and *helped jobs*, respectively. Any job busy-waiting for ψ may be selected to execute as a helping job on behalf of the locking job J' but there is at most one helping job on the same resource at a given time.

Lemma 5. *Provided that there is no nested accesses to resources, the maximum blocking time before accessing a global resource ψ is given by*

$$B^\psi = (|\delta^\psi| - 1) C^\psi, \quad (3.1)$$

and any job of a task τ_i cannot be globally blocked for more than

$$gblock(\tau_i) = \sum_{\psi \in \Psi_{\tau_i}} N_{\tau_i}^\psi B^\psi, \quad (3.2)$$

with $N_{\tau_i}^\psi$ representing the number of times τ_i requests resource ψ .

Proof. The maximum number of global blocking comes directly from Lemma 3. As by Lemma 4 no deadlock occurs, each global blocking may make the job which is requesting a global resource ψ wait for bounded amount of time. Let J be a job requesting the already locked resource ψ . Two scenarios may occur at the time J requests ψ , depending on whether or not the blocking job is executing. If it is, it takes at most C^ψ time units for the resource be released since C^ψ is the maximum length of the critical sections of ψ . If it is not executing, due to the helping mechanism, the server where J executes starts executing the locking job and this also takes at most C^ψ time units. In other words, each job that locks ψ before J interferes at most C^ψ time units on J 's execution due to blocking. Therefore, the maximum blocking time due to global resource access cannot exceed B^ψ .

Furthermore, since B^ψ is the maximum blocking time on resource ψ and considering that a job of task τ_i accesses resource ψ at most $N_{\tau_i}^\psi$ times, the maximum interference each job of τ_i may suffer due to global blocking is $N_{\tau_i}^\psi B^\psi$. Summing this interference for all resources in Ψ_{τ_i} yields the maximum blocking time $gblock(\tau_i)$. \blacksquare

Lemma 5 is not valid when nested resources are allowed. The research done by Garrido *et al.* (GARRIDO *et al.*, 2017b) explored the issue of nested resources for MrsP protocol in depth, although that work has been limited to fixed priority partitioned scheduling.

The stated results up to here imply that a task waits for resources a bounded amount of time. However, this does not suffice to prove that the whole system works correctly. For example, a task may spin while waiting for resources and this spinning time should be accounted for in the budget its server provides. Lemma 6 bounds the maximum time a task may take to finish its execution, called inflated execution time. Another aspect is related with the utilization that must be set to a server to account for both the inflated execution time and the local blocking its client tasks may suffer. This aspect is addressed in Lemma 6 and Theorem 1.

Lemma 6 (Task execution time inflation). *Consider a set of shared resources Ψ and a set of tasks Γ that are packed into a set of servers Γ^0 . Each task τ_i in Γ executes for at most*

$$\hat{C}_i = C_i + gblock(\tau_i) \quad (3.3)$$

and its inflated utilization is $\hat{U}_{\tau_i} = \frac{\hat{C}_i}{T_i}$.

Proof. The lemma follows from the fact that task τ_i executes for its worst-case execution time C_i but according to Rule 2 may busy-wait when globally blocked. By Lemma 5 the maximum global blocking time is given by $gblock(\tau_i)$, leading to the maximum time of \hat{C}_i and the corresponding inflated utilization \hat{U}_{τ_i} . \blacksquare

Note that when a job of task τ_i is locally blocked, this blocking can be direct or indirect since condition $\Pi_{\tau_i} \leq \Pi_{\sigma}(t)$ can be true due to the execution of lower priority jobs independently of whether their tasks share resources with τ_i . For example, consider a job of a task τ_j such that $\Pi_{\tau_j} \geq \Pi_{\tau_i}$, *i.e.*, $\tau_j \in \text{hpl}(\tau_i)$. In this case, the jobs of tasks that access resources in Ψ_{τ_j} but do not access those in Ψ_{τ_i} may indirectly block τ_i 's jobs since their execution may make $\Pi_{\tau_i} \leq \Pi_{\sigma}(t)$. Direct blocking, on the other hand, concerns tasks that share the same resources. In any case, the local blocking of τ_i , denoted $lblock(\tau_i)$, can be defined as the maximum blocking time (direct or indirect) suffered by τ_i due to tasks of lower preemption levels accessing resources whose preemption level is higher than or equal to that of τ_i . More formally,

$$lblock(\tau_i) = \max\{B^{\psi} + C^{\psi} \mid \psi \in \Psi_{\delta_{\tau_i}}, \exists \tau_k \in \Gamma_{\delta_{\tau_i}}^{\psi}, \Pi_{\tau_k} < \Pi_{\tau_i}, \Pi_{\delta_{\tau_i}}^{\psi} \geq \Pi_{\tau_i}\}.$$

Theorem 1 (Inflated server utilization). *If any server $\sigma \in \Gamma^0$ has its utilization set to*

$$\hat{U}_{\sigma} = \sum_{\tau_i \in \Gamma_{\sigma}} \hat{U}_{\tau_i} + \max_{\tau_i \in \Gamma_{\sigma}} \left(\frac{lblock(\tau_i)}{T_i} \right), \quad (3.4)$$

then no client of σ misses any of its deadlines provided that $\hat{U}_{\sigma} \leq 1$ and that σ meets all of its own deadlines.

Proof. We show the theorem by proving that if there is a client of σ that misses a deadline, then σ 's utilization must be lower than what is given by (3.4).

Since σ is a server, its utilisation must be necessarily set to a value not greater than 1. It is thus convenient to think of σ being scheduled on a single processor even if in the actual schedule σ migrates between processors. This does not restrict the proof. The single processor schedule can be taken as a projection of the system schedule on a single processor. Apart from context-switches and migrations, σ ' timeliness is not compromised in this uniprocessor schedule of σ . Hence, at any time, the processor is idle in the uniprocessor schedule either when σ does not execute or when it does but selects no job to execute. And when the processor is not idle, σ is executing some job.

Assume that some client of σ misses its deadline at instant d . Without loss of generality, let d be the first missed deadline. Let $r \geq 0$ be the latest instant before d such

that no job by σ 's clients released before r with deadline by d is waiting for execution. By construction, r and d are also release time and deadline of σ , respectively. This is because every client of σ is an implicit-deadline periodic task and the deadline of σ is the minimum deadline of its clients.

By the definitions of r and d , there is always some pending job by clients of σ to execute during $[r, d)$. None of these pending jobs could be left awaiting execution leaving the processor idle when σ executes during $[r, d)$. Recall from Rules 2 and 3 that jobs globally blocked busy-waits and so occupy the processor. And if there are locally blocked jobs at some time t , Rule 1 tells us that the preemption level $\Pi_\sigma(t)$ is higher than that of the locally blocked job. This means that some other job by a client of σ is executing and locking a resource as stated in Definition 3. Thus, there is no idle time during intervals σ executes in $[r, d)$. As σ meets all its deadlines, it provides a total of

$$(d - r)\hat{U}_\sigma \quad (3.5)$$

time units during $[r, d)$ to execute its clients. We now compute all demand by the clients of σ whose jobs have deadlines up to d . Jobs with deadlines greater than d cannot be scheduled by σ within $[r, d)$ due to EDF order. The number of relevant jobs of $\tau_i \in \Gamma_\sigma$ is bounded by $\lfloor \frac{d-r}{T_i} \rfloor$. Thus, the maximum demand of such jobs is bounded by

$$\sum_{\tau_i \in \Gamma_\sigma} \left\lfloor \frac{d-r}{T_i} \right\rfloor (C_i + gblock(i)) \leq (d-r) \sum_{\tau_i \in \Gamma_\sigma} \hat{U}_{\tau_i}, \quad (3.6)$$

since we know from Lemma 6 that each task $\tau_i \in \Gamma_\sigma$ requires at most $\hat{C}_i = C_i + gblock(i)$.

If some client of σ is locally blocked during $[r, d)$, σ executes low priority jobs J holding a local resource in some interval during $[r, d)$. We need to determine the computation demand caused by the execution of blocking jobs J . Shortly we will see that only one such a blocking job J is relevant to be considered.

By the definition of r , no blocking job J with deadline less than d could start its execution before r . Also, from the EDF priority order and from the fact that there are pending jobs with deadlines less than or equal to d , no blocking job J with deadline greater than d can start its execution at or after r . Further, (3.6) already accounts for every blocking job J that starts its execution at or after r and have deadlines no later than d . Therefore, only blocking jobs that start their execution before r and finish their execution after r are relevant for increasing the demand to be executed within $[r, d)$. However, there can be only one such a job. If there were more than one job J requesting resources at some time before r , Lemma 2 tells that only one of them would be able to lock the resources and so the others would not be able to start execution within $[r, d)$ due to EDF priority order. From r until the moment J releases the resources, σ executes J and the maximum blocking due to J within $[r, d)$ is bounded from above by

$$\max_{\tau_i \in \Gamma_\sigma} (lblock(\tau_i)) \quad (3.7)$$

Since d is a missed deadline, this service time, given by (3.5), must be lower than the

total demand by clients of σ (3.6) summed with maximum local blocking (3.7),

$$(d - r)\hat{U}_\sigma < (d - r) \sum_{\tau_i \in \Gamma_\sigma} \hat{U}_{\tau_i} + \max_{\tau_i \in \Gamma_\sigma}(\text{lblock}(\tau_i))$$

Dividing by $d - r$,

$$\hat{U}_\sigma < \sum_{\tau_i \in \Gamma_\sigma} \hat{U}_{\tau_i} + \max_{\tau_i \in \Gamma_\sigma} \left(\frac{\text{lblock}(\tau_i)}{d - r} \right)$$

As $d - r \geq T_i$ for any $\tau_i \in \Gamma_\sigma$, the above relation contradicts (3.4). ■

Theorem 1 determines the server's inflated utilization to accommodate its tasks, as well as the time reserves required for the worst case global and local blocking. The theorem also ensures that all server client tasks will meet their deadlines. It's possible to see that the RUN theorem 3.1, transcribed below and adapted to the notations of this work, applies perfectly to this solution.

The EDF server $\sigma = \text{PACK}(\Gamma)$ of a set of servers Γ produces a valid schedule of Γ when $U_\Gamma \leq 1$ and all jobs of σ meet their deadlines. (REGNIER et al., 2011)

In order to adapt the RUN theorem 3.1 to the solution proposed in this work, the utilization of the servers of Γ^0 must be inflated according to (3.4). After we have defined the inflated utilization of each of the servers σ , being $\sigma \in \Gamma^0$, the RUN rules for building the reducing tree remains the same as in RUN. Since it has been ensured that each server will respect its processor time reservation limits, it is possible to apply the DUAL and PACK operations leading to the construction of the RUN reduction tree. For every server $\sigma \in \Gamma^0$ we can create a σ' server with utilization equal to \hat{U}_σ and use σ' for the application of the RUN rules, instead of considering σ . Thus, there is a valid application of RUN theorems 4.1 and 4.3, transcribed below and adapted to the notations of this work. Let Σ be a schedule of a set of tasks or servers and $\{\Delta^i\}$ the i -th REDUCE operation, then

Let Γ be a set of $n = m + k$ servers on m processors, scheduled by Σ , with $k > 1$ and $U_\Gamma = m$. Let Γ^* and Σ^* be their duals. Then $U_{\Gamma^*} = k$, and so Γ^* is feasible on k processors. Further, Σ is valid if and only if Σ^* is valid. (REGNIER et al., 2011)

If Γ is a proper set under the reduction sequence $\{\Delta^i\}_{i \leq p}$, then the RUN algorithm produces a valid schedule Σ for Γ . (REGNIER et al., 2011)

The proof for these theorems can be found in the RUN paper (REGNIER et al., 2011). Since the RUN algorithm can be considered perfectly adapted to the MrsP protocol, let us look at the Algorithm 1 which allows us to get an overview of RUN adapted to incorporate the resource sharing approach. An illustration involving also the on-line phase is given in the next section.

Algorithm 1: RUN off-line phase**Require:** Γ ; {Set of tasks}

- 1: $i \leftarrow 0$;
- 2: Generate Γ^i servers for Γ tasks according to some suitable package strategy;
- 3: $\hat{U}_\Gamma \leftarrow \sum_{\sigma \in \Gamma^i} \hat{U}_\sigma$ {Calculate the total inflated utilization of the system}
- 4: **if** $(\hat{U}_\Gamma - \lfloor \hat{U}_\Gamma \rfloor) > 0$ **then**
- 5: Generate dummy task τ ; $\{U_\tau = (1 - (\hat{U}_\Gamma - \lfloor \hat{U}_\Gamma \rfloor))\}$
- 6: $\Gamma \leftarrow \Gamma \cup \{\tau\}$;
- 7: **end if**
- 8: **repeat**
- 9: $i \leftarrow i + 1$;
- 10: $\Gamma^i \leftarrow \text{PACK}(\Gamma^{i-1})$;
- 11: $\Gamma^{i*} \leftarrow \text{DUAL}(\Gamma^i)$;
- 12: **until** $\nexists \sigma \in \Gamma^i | \sigma$ is not unit server;
- 13: **return** $\Gamma^{0..i}$; {The RUN reduction tree}

3.3 ILLUSTRATION OF THE PROPOSED IDEAS

This section provides some illustration on how MrsP and RUN could jointly be used to support resource sharing in HRTS. Consider a system with 2 processors, composed of a set Γ with 4 tasks, as specified in Table 3.2. Resources are shared as indicated in the table. Task τ_1 shares ψ_1 with τ_3 and ψ_3 with τ_2 whereas resource ψ_2 is shared by τ_3 and τ_4 . Note that Table 3.2 also shows the inflated utilization of the tasks. For example, the nominal value of $U_{\tau_1} = 0.50$ but upon packing this value goes to 0.60. The maximum execution time for the critical section of each resource is represented in the table by C^{ψ_i} .

Table 3.2 Example to illustrate how MrsP and RUN would work.

Γ	C_i	T_i	U_{τ_i}	\hat{U}_{τ_i}	Accesses
τ_1	15	30	0.50	0.60	ψ_1, ψ_3
τ_2	22	40	0.55	0.60	ψ_3
τ_3	04	20	0.20	0.25	ψ_1, ψ_2
τ_4	59	120	0.49	0.49	ψ_2
$C^{\psi_1} = 1, C^{\psi_2} = 1.2, C^{\psi_3} = 2$					

As there is no constraint on how RUN selects tasks in Γ to pack them, assume that this pack operation produces three servers in Γ^0 , namely σ_1 , σ_2 and σ_3 , as indicated in Fig. 3.1. Servers σ_1 and σ_2 have a single client each, respectively, τ_1 and τ_2 whereas tasks τ_3 and τ_4 are clients of σ_3 . Due to this packing, resources ψ_1 and ψ_3 are global while resource ψ_2 is local as can be seen in Fig. 3.1.

Lets see how the inflation of the tasks was calculated. For example, τ_1 can be blocked by ψ_1 and ψ_3 , so its utilization must be increased considering these global blocking times. τ_4 on the other hand does not access global resources, so its inflated utilization is equal

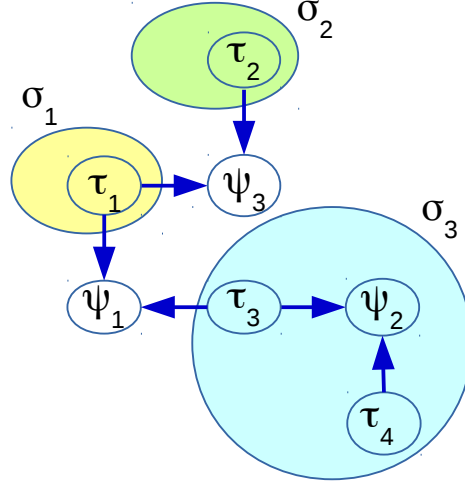


Figure 3.1 Illustration of the Proposed Ideas

to its original utilization. Below are the formulas for the inflated utilization of the tasks of our example:

$$\begin{aligned}\hat{U}_{\tau_i} &= (C_i + gblock(\tau_i))/T_i, \\ \hat{U}_{\tau_1} &= (15 + 3)/30 = 0.60 \\ \hat{U}_{\tau_2} &= (22 + 2)/40 = 0.60 \\ \hat{U}_{\tau_3} &= (4 + 1)/20 = 0.25 \\ \hat{U}_{\tau_4} &= (59 + 0)/120 = 0.49\end{aligned}$$

Server utilization inflation should consider the local blocking time. Only τ_3 can be locally blocked in our example. Although τ_4 can also access a local resource, it cannot be locally blocked due to its low preemption level. τ_1 and τ_2 do not access local resources, so σ_1 and σ_2 utilizations are equal to the sum of the inflated utilizations of its clients. Next are the formulas to calculate the inflated utilization of the servers.

$$\begin{aligned}\hat{U}_{\sigma} &= \sum_{\tau_i \in \Gamma_{\sigma}} \hat{U}_{\tau_i} + \max_{\tau_i \in \Gamma_{\sigma}} \left(\frac{lblock(\tau_i)}{T_i} \right), \\ \hat{U}_{\sigma_1} &= 0.60 \\ \hat{U}_{\sigma_2} &= 0.60 \\ \hat{U}_{\sigma_3} &= 0.25 + 0.49 + 1.2/20 = 0.80\end{aligned}$$

In particular, note that task τ_1 requests two global resources. Thus, it could be blocked twice (once per resource) according to MrsP rules. Task τ_3 , on the other hand, shares ψ_2 with τ_4 , which is a local resource. The maximum local blocking τ_3 may experience is bounded by C^{ψ_2} , since it is the only local resource of σ_3 which is also used by another lower

priority task. Task τ_3 must also account for a possible blocking time on the global resource ψ_1 . It is important to note in Table 3.3 that tasks τ_1 , τ_2 and τ_3 show inflated utilizations due to the application of Lemma 6, while task τ_4 has its final utilization unchanged. Another noteworthy observation is that the utilization of server σ_3 corresponds to the sum of the utilization of its client tasks plus the sum of the longest local blocking time that the tasks of this server can suffer, by applying (3.4)¹.

After producing a packing result Γ^0 , the remainder procedure to reduce the system into a virtual uniprocessor system follows the same original rules of RUN, as can be seen in Table 3.3. The scheduling steps by RUN are also kept unchanged. From RUN’s perspective, it is as if a set of inflated tasks are to be scheduled.

Table 3.3 RUN Schedule Example

Γ	$\tau_1(0.6, 30\mathbb{N}^+)$	$\tau_2(0.6, 40\mathbb{N}^+)$	$\tau_3(0.25, 20\mathbb{N}^+)$	$\tau_4(0.49, 120\mathbb{N}^+)$
$\Gamma^0 = \text{PACK}(\Gamma)$	$\sigma_1(0.6, 30\mathbb{N}^+)$	$\sigma_2(0.6, 40\mathbb{N}^+)$	$\sigma_3(0.8, 20\mathbb{N}^+)$	
$\Gamma^{0*} = \text{DUAL}(\Gamma^0)$	$\sigma_1^*(0.4, 30\mathbb{N}^+)$	$\sigma_2^*(0.4, 40\mathbb{N}^+)$	$\sigma_3^*(0.2, 20\mathbb{N}^+)$	
$\Gamma^1 = \text{PACK}(\Gamma^1)$	$\sigma_4(1.0, 20\mathbb{N}^+, 30\mathbb{N}^+)$			

The schedule produced by RUN for the illustrative example is shown in Figure 3.2. The EDF schedule generated by σ_4 was omitted since it is irrelevant for the discussion here. The figure shows only the schedules for Γ^0 (packed servers) and Γ (system tasks). Note that all tasks are preempted during execution at some time in the schedule, illustrating that the proposed solution is not based on disabling interruptions. Even τ_3 , which has the highest preemption level, is prevented from execution because its server hands over the processor at time 84. Such a characteristic of our solution favors schedulability as compared to SBLP, which is the only resource sharing protocol that deals with RUN scheduled systems.

Some relevant observations regarding the behavior of the proposed solutions are highlighted in Figure 3.3 and commented below.

1. Effects due to the helping mechanism. Figure 3.3(a) illustrates how the helping mechanism would work in the proposed solution. Task τ_1 blocks global resource ψ_1 at time 4 just before its server σ_1 releases the processor, as indicated in Figure 3.2(a). At time 8, when task τ_3 attempts to access ψ_1 , it gets blocked. Since τ_1 is not executing, τ_3 will give way the “logical” processor (*i.e.*, server) in favor of τ_1 so as to accelerate its execution during its critical section. This is indicated in the figure 3.3(a) by the grey area within the context of σ_3 ’s execution, which is shown as a green area. At the end of the τ_1 ’s critical section, τ_3 will resume the execution. This means that σ_3 ’s budget is being used to execute part of τ_1 (a non-client task). However, this extra execution σ_3 is taking care of has been

¹We used rounded values for illustration purposes, but this example would not be schedulable on two processors without rounding.

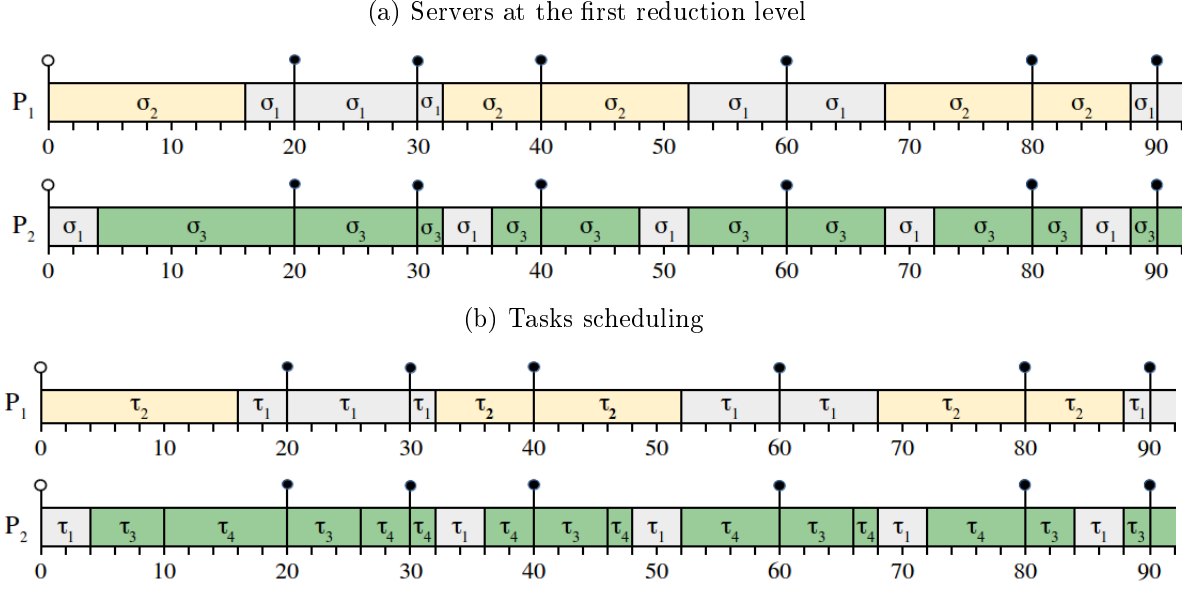


Figure 3.2 RUN Schedule Example for Table 3.3

already taken into consideration by the blocking term $gblock(\tau_i)$. As consequence, the utilization of σ_3 has been inflated accordingly. Since the off-line phase of RUN has finished successfully, the rules of RUN hold and system schedulability is thus ensured.

2. Effects due to local blocking. As illustrated in Figure 3.3(b), task τ_4 is holding ψ_2 at time 20. At this point in time, the ceiling of server σ_3 is equal to the preemption level of τ_3 , which is the highest of the preemption levels of the σ_3 's client tasks that access ψ_2 . When τ_3 attempts to start its execution at time 20 it will get blocked, because its preemption level is not greater than the σ_3 's ceiling level. Then τ_4 will continue to execute. The blocking will be maintained as long as τ_4 executes its critical section. Immediately after the end of the critical section of τ_4 , τ_3 can start to execute. Again, the blocking suffered by τ_3 has been already account for when computing \hat{U}_{σ_3} (3.4) and so this case does not impair system schedulability.
3. Effects due to global blocking. Task τ_1 locks the global resource ψ_3 at time 69, as showed in the Figure 3.3(c). When τ_2 attempts to access the same global resource at time 70, it gets blocked. Since τ_1 is executing at this moment, τ_2 busy-waits for the releasing of ψ_3 , represented in the figure as a black area. At the end of the critical section of τ_1 , τ_2 can resume its execution.

Form these three scenarios in the illustrative example, it is clear that WCET of tasks and utilization of servers must be increased to accommodate possible blocking. In short, the scenarios have highlighted that: a task may give way its processor in favor of a

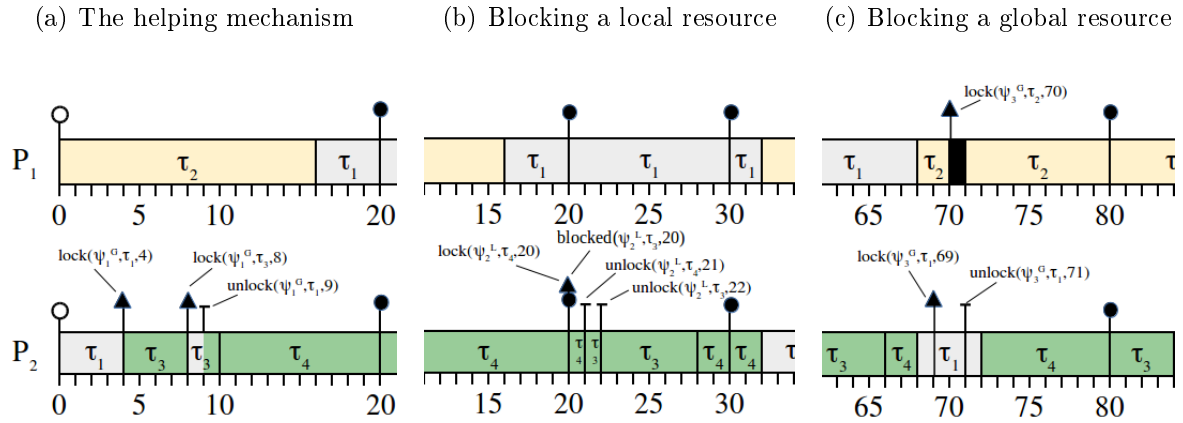


Figure 3.3 Resource Sharing Examples

task from another server, through the helping mechanism; before starting to execute, a task may get blocked if a local resource it needs is already locked; and while attempting to access a global resource, a task may get blocked and stay in busy waiting on its processor/server.

We demonstrated how MrsP works in combination with RUN proving the correctness of this solution. We will present in the next chapter the results of the schedulability testing experiments.

PROCESSOR DEMAND EVALUATION

The experiments carried out in this work had as objective to demonstrate the viability of the proposed solution and to evaluate the benefits of applying MrsP to the RUN algorithm in comparison to SBLP (BONATO; MEZZETTI; VARDANEGA, 2014). In addition, different heuristics for task allocation to servers were evaluated. The most important aspects observed here were (i) the processor time demand that each solution requires for the tasks to be scheduled under the worst-case resource sharing condition and (ii) the overhead that each solution presented in the execution of the basic primitives of scheduling and resource sharing. We will explore the first aspect in this chapter and in the next chapters, the second.

The hypothesis which motivated this work is that MrsP offers advantages in terms of schedulability, since the protocol is based on SRP, which has the clear benefit of not preventing the execution of highest priority tasks if the resources demanded by these had not been blocked inside their respective servers. This premise allowed this work to explore new possibilities of aggregation of tasks into servers that had not been exploited by the precursor work (BONATO; MEZZETTI; VARDANEGA, 2014), because our solution is not significantly affected if unrelated tasks are grouped together into the same server. In the following subsections, we will detail the protocol used for comparison, explain how the amount of processor time required by each protocol was measured and evaluated and analyze the results.

4.1 THE PROTOCOL FOR COMPARISON - SBLP

In order to adapt RUN to SBLP, the off-line phase of RUN (when the reduction tree is created) was modified so that the worst-case execution times of the tasks are increased in order to account for the need to serialize parallel access to shared resources. Consider that task τ_i can access a resource ψ at most $N_{\tau_i}^{\psi}$ times and that this resource can be accessed by at most $|\delta^{\psi}|$ servers. Let C^{ψ} be the maximum execution cost of a critical section for this resource. The maximum global blocking time, defined as the maximum time a task τ_i waits in ψ 's queue is thus $N_{\tau_i}^{\psi}B^{\psi}$ (3.1) as τ_i 's request can be the last one in the queue.

Taking into consideration all resources accessed by τ_i , denoted by Ψ_{τ_i} , it waits at most $gblock(\tau_i)$ (3.2) time units in resources' queues. This also implies that τ_i 's execution time must be inflated, accounting for $gblock(\tau_i)$ since during this time it busy-waits, that is, it occupies the processor while waiting for resources. Hence, its inflated execution time can be expressed as \hat{C}_i (3.3).

There must also be an increase in servers' utilization. This increase comes from two aspects. The first is the need to incorporate the inflated tasks' execution times. The second is related with the time that a high-priority task may have to wait before starting its execution if a resource is locally blocked by another task of the same server. RUN servers are modified in SBLP so as to implement non-preemptive execution of tasks requiring resources. The first aspect is accounted for by summing up all inflated task utilization,

$$\sum_{\tau_i \in \Gamma_\sigma} \frac{\hat{C}_i}{T_i},$$

with Γ_σ representing the set of σ 's client tasks. A bound for accounting for the second aspect has been derived in (BONATO; MEZZETTI; VARDANEGA, 2014) by considering that the highest frequent task in the server may be waiting for the maximum time a lower priority task can be in a resource queue, including its own critical section and limited by only one attempt to access a resource (BAKER, 1991). Summing up the effect of the two aforementioned aspects yields

$$\hat{U}_\sigma = \sum_{\tau_i \in \Gamma_\sigma} \frac{\hat{C}_i}{T_i} + \frac{\max_{\psi \in \Psi_\sigma} (|\delta^\psi| C^\psi)}{\min_{\tau_j \in \Gamma_\sigma} (T_j)} \quad (4.1)$$

with Ψ_σ denoting the resources accessed by tasks that are clients of σ . Although pessimistic, the bound expressed by (4.1) serves to the purpose of preserving schedulability within each server.

4.2 SCHEDULABILITY

As is in the SBLP definition, our solution also requires increasing the tasks' WCET and servers' utilization. This modification occurs during the off line phase of RUN. The increments are necessary to avoid deadline misses for the scenarios in which a task may suffer global or local blocking. The detailed calculation of the increment factors for tasks' WCET and servers' utilization can be observed in the Section 3.2. We observed that the formula for dealing with global blocking is similar in both protocols, however, there is a difference in the calculation formula for increasing the server utilization to deal with local blocking that confers a advantage to the MrsP protocol. This advantage is due to the fact that MrsP allows preemption within the server while there are blocked resources. SBLP does not offer this possibility, so it introduce a more pessimistic increase in the server's utilization rate.

To carry out the experiments, we generated synthetic task sets using the same procedures as the ones used for evaluating SBLP (BONATO; MEZZETTI; VARDANEGA,

2014)¹. This allowed us to precisely mimic the experiments originally published for a fair comparison. The difference in performance of the used packing heuristics and protocols are given in the necessary processor utilization inflation demanded by each generated task set, that is by how much the values given by (4.1) (for SBLP) or (3.4) (for MrsP) are higher when compared to the original task set utilization. The utilization inflation obtained for each protocol is the main evaluation parameter.

Before this work, two heuristics were proposed for packing tasks into servers (BONATO; MEZZETTI; VARDANEGA, 2014), named Fine Grained (FG) and Coarse Grained (CG). The former distributes tasks to servers so that the tasks of the same server uses exactly the same combination of resources. It is intended to reduce the local blocking time avoiding unrelated contentions. However, it may increase the waiting time for global resources due to the potential increase in the size of the resources' queues as there will be more servers in a race for the resources. The following example illustrates the application of this heuristic.

Example 1. *Let Γ be a set of 3 tasks indicated by τ_i , $i = 1, 2, 3$. The system has two resources, ψ_1 and ψ_2 , with ψ_1 being accessed by τ_1 and τ_2 whereas ψ_2 can be accessed by τ_2 and τ_3 . According to FG, the tasks must be packed into 3 distinct servers, σ_j , where $j = 1, 2, 3$, since $\{\psi_1\}$, $\{\psi_1, \psi_2\}$ and $\{\psi_2\}$ are distinct subsets. As can be seen, the queue sizes of both ψ_1 and ψ_2 are equal to 2 because each of these resources can be accessed by two servers.*

The CG packing heuristic seeks to reduce the size of the resource queues with a possible side effect of increasing local blocking time due to unrelated contentions within the same server. The heuristic specifies that “Servers for collaborative tasks are created upon the connected components of the graph in which the tasks are the nodes, and the edges represent the fact that two tasks share at least one resource” (BONATO; MEZZETTI; VARDANEGA, 2014), but is not clear about the criterion of distribution. Example 2 illustrates the CG heuristic.

Example 2. *Consider exactly the same system defined in Example 1. CG would pack the tasks into 2 distinct servers, σ_1 and σ_2 . Server σ_1 may contain tasks τ_1 and τ_2 or only τ_1 and σ_2 would server the tasks not contemplated by the other server. It can be seen that the queue size of one resource would be 1, contrasting with the FG heuristic in Example 1 where both resources queues have sizes of 2.*

From Examples 1 and 2, it can be seen that the interference between tasks in different servers tend to be lower in CG when compared to FG if collaborative tasks cannot be all packed together. CG, however, does not take into account the size of the critical section of each resource or the number of servers that can be generated during the packing of the tasks. We next describe a new heuristic that takes these factors into account.

¹The *Python* scripts were handed to us by Dr. Enrico Mezzetti, one of the authors of SBLP.

4.2.1 Ordered Blocking Time Packing

In this section we describe the packing heuristic proposed for the protocols used in this work, named Ordered Blocking Time (OBT). This heuristic takes the demand for processor time to obtain better results when compared to the original heuristics previously presented. OBT groups tasks into servers according to the following steps:

1. Build an ordered list Ψ of resources. In this step, all available resources are stored in the list Ψ sorted in descending order of maximum size of resources' critical sections multiplied by the number of tasks that can use it minus 1, *i.e.*, the ordering factor of each resource ψ is given by $C^\psi(L^\psi - 1)$, with L^ψ denoting the number of tasks that can use ψ . More formally,

$$\forall \psi_i, \psi_j \in \Psi, (i) < (j) \Rightarrow C^{\psi_i}(L^{\psi_i} - 1) \geq C^{\psi_j}(L^{\psi_j} - 1),$$

with (i) hereafter representing the position a resource ψ_i appears in the list Ψ . Note that for resources ψ for which $L^\psi = 1$, $C^\psi(L^\psi - 1) = 0$ independently of the value of C^ψ . Hence, such resources, if any, will be in the last positions in Ψ representing the fact that there is no collaborative tasks using them.

2. Set up task groups according to the resource list Ψ . To do so, for each resource $\psi_i \in \Psi$, create a task group $g_{(i)}$ containing all tasks that use ψ_i but does not use any resource ψ_k with $(k) < (i)$. The definition of $g_{(i)}$ can be stated as

$$g_{(i)} = \{\tau \in \Gamma \mid \forall (k) < (i), \tau \notin g_{(k)} \wedge \psi_i \in res(\tau)\}$$

Note that a task belongs to only a group and that if a group contains more than one task, they are collaborative. Groups with a single task is possible when a resource is accessed by only a task.

Unlike FG and CG, grouping tasks according to the order in Ψ tends to isolate collaborative tasks suffering highest interference in the same group. These groups are then used to set up the servers.

3. Create servers for collaborative tasks. For each defined group g_i , create k_i servers having tasks in g_i as clients. Each of the k_i servers must have their utilization, as computed using (4.1), lower than or equal to 1.
4. Merge servers. This step is to reduce the concurrency on shared resources by distinct servers. As the previous step may create small servers, some of them may be merged if there are resources in common. If two servers are merged, one of them is discarded and the one that is kept is set to serve all client tasks of both of them. Merging servers is subject to the constraints imposed by (4.1).
5. Merge unrelated servers. This step was applied to the MrsP protocol only, as it best handles the presence of unrelated tasks on the same server, unlike the SBLP protocol. It consists of merging servers even if they do not have tasks sharing resources in common.

6. Create servers for isolated tasks. This final step is to pack tasks not requesting any resources into new servers. These servers are not subject to (4.1) and the packing followed in this step can be the usual carried out by the off-line phase of RUN.

The OBT heuristic is now illustrated taking Example 3.

Example 3. Consider the same system as in Example 1 and complete its specification with the following additional information. Let the maximum size for the critical sections be $C^{\psi_1} = 1$ and $C^{\psi_2} = 2$ and define the system tasks as

$$\Gamma = \{\tau_1(0.4; 40N^+), \tau_2(0.4; 20N^+), \tau_3(0.3; 30N^+)\}.$$

After carrying out step 1, the resulting ordered list $\Psi = \{\psi_2, \psi_1\}$ since $L^{\psi_1} = L^{\psi_2} = 2$ and $C^{\psi_2} > C^{\psi_1}$. Then two task groups are generated according to step 2, namely $g_1 = \{\tau_2, \tau_3\}$ and $g_2 = \{\tau_1\}$. Step 3 is able to define one server for each group since according to (4.1), setting up a server σ_1 for the tasks in g_1 would give $\hat{U}_{\sigma_1} = 0.85 < 1$. Group g_2 containing a single task give rise to server σ_2 with $\hat{U}_{\sigma_2} = 0.425$. Step 4 is not able to merge the two servers since a single server for the three tasks would have utilization greater than 1.

It is worth observing in Example 3 that the initial utilization of the system is 1.1 and the total system utilization after carrying out the OBT heuristic for SBLP is 1.275 and it is equal for MrsP. If FG had been used, the total system utilization would be 1.342 and in the case of CG, it would be 1.32 considering that τ_1 and τ_2 would have been packed together into single server. That is, the inflation imposed by OBT tends to be lower when compared to FG and CG. This relevant characteristic of OBT is experimentally evaluated next.

4.3 EXPERIMENTAL ASSESSMENT

In this section we present results from experiments that compared the performance of the OBT heuristic and the MrsP protocol with the ones originally developed for the SBLP protocol.

4.3.1 Experimental Setup

As previously mentioned, we generated synthetic task sets using the same procedures as the ones used for evaluating SBLP (BONATO; MEZZETTI; VARDANEGA, 2014). Thus, it was possible to perform experiments similar to those of the referenced work and compare the processor utilization inflation demanded by each generated task set. Each value of inflated utilization obtained during experiments represents an average of 100 experiment runs each of which taking sets with 40 tasks. There were 20 or 30 resources in the system to be distributed between tasks. Each task could access 3 or 6 resources and the resources were associated with its respective tasks in a random manner. The size of the critical section for each resource was chosen randomly from the ranges 10 to 100 μ s or 50 to 200 μ s. Four task period ranges, four system utilization ranges, two values for the number of resources per task, two values for the number of resources in the system

and two ranges for the size of the critical section were considered. This gave a total of 128 settings. Period and utilization for each task were generated according to a uniform distribution within the respective ranges for each considered setting in Table 4.1.

Table 4.1 Schedulability Test Parameters - Tasks Configuration.

#	Period (ms)	Utilization
1	[50, 150]	[0.01, 0.1]
2	[50, 150]	[0.1, 0.3]
3	[50, 150]	[0.3, 0.5]
4	[50, 150]	[0.01, 0.5]
5	[150, 500]	[0.01, 0.1]
6	[150, 500]	[0.1, 0.3]
7	[150, 500]	[0.3, 0.5]
8	[150, 500]	[0.01, 0.5]
9	[500, 2000]	[0.01, 0.1]
10	[500, 2000]	[0.1, 0.3]
11	[500, 2000]	[0.3, 0.5]
12	[500, 2000]	[0.01, 0.5]
13	[50, 2000]	[0.01, 0.1]
14	[50, 2000]	[0.1, 0.3]
15	[50, 2000]	[0.3, 0.5]
16	[50, 2000]	[0.01, 0.5]

The options available for the number of resources per task, the number of resources in the system and the size of the critical section are shown in Table 4.2.

Table 4.2 Schedulability Test Parameters - Resources Configuration.

#	Resources/task	Resources/system	Critical section's (μ s)
1	3	20	[10, 100]
2	3	30	[10, 100]
3	6	20	[10, 100]
4	6	30	[10, 100]
5	3	20	[50, 200]
6	3	30	[50, 200]
7	6	20	[50, 200]
8	6	30	[50, 200]

For each setting consisting of a pair of rows, one row from Table 4.1 and another from Table 4.2, the degree of collaboration was made to vary from 0 to 100% in steps of 5%. Collaboration degree of 0 means that tasks were not referring to any shared resource and a collaboration degree of 5% represents scenarios were 2 out of 40 tasks access resources.

4.3.2 Experimental Results

The FG and CG heuristics were applied only to SBLP in this work, so we mention FG-SBLP and CG-SBLP for the combination of the respective heuristic with SBLP. The OBT heuristic was applied to the SBLP and MrsP protocols; therefore, we named OBT-SBLP for the heuristic applied to SBLP and OBT-MrsP for MrsP.

The experiments were performed for all possible row combinations of Tables 4.1 and 4.2, *i.e.*, line pairs where one line belongs to Table 4.1 and another line belongs to Table 4.2. Obtained results showed that FG-SBLP was worse than CG-SBLP, OBT-SBLP and OBT-MrsP. This behavior is due to the fact that FG-SBLP tends to generate more servers and so it imposes more concurrency at the server level. This, according to (3.3), is likely to over-inflate the tasks' execution times for all tasks that use a global resource. For all scenarios, the processor demand augmentation of FG-SBLP was worst than the ones of OBT-MrsP. For one of the 128 scenarios FG-SBLP performs better than OBT-SBLP and for few scenarios FG-SBLP performs better than CG-SBLP (about 13%). For higher task utilization settings, FG-SBLP and CG-SBLP performed closer due to the greater number of servers that both heuristics generate, a characteristic already observed in (BONATO; MEZZETTI; VARDANEGA, 2014). The comparison of the number of servers that each heuristic / protocol obtained during the experiments can be observed in Figure 4.1.

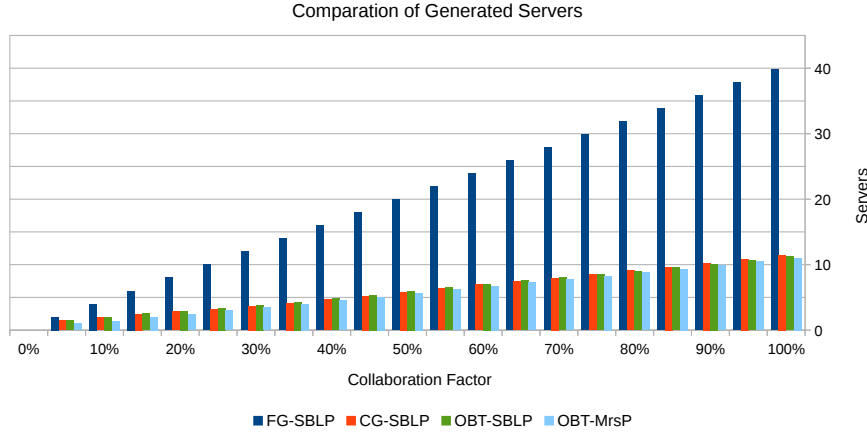


Figure 4.1 Average number of generated servers for the scenarios in Tables 4.1 and 4.2

Fig. 4.2 shows us the comparison of the results obtained by all heuristics used in this work for all scenario combinations described in Tables 4.1 and 4.2.

Note that FG-SBLP performed worse than the others, even compromising the visualization of the graph scale, so we observe in Fig. 4.3 the results obtained excluding this heuristic. As FG was shown to behave consistently worse than CG in almost all evaluated settings and worse than OBT in all scenarios but one, FG will not be further considered hereafter.

From Fig. 4.3, it is possible to see that CG-SBLP obtained slightly larger augmentation than the OBT heuristic applied to both protocols, and OBT-SBLP and OBT-MrsP presented equivalent augmentation in this graph.

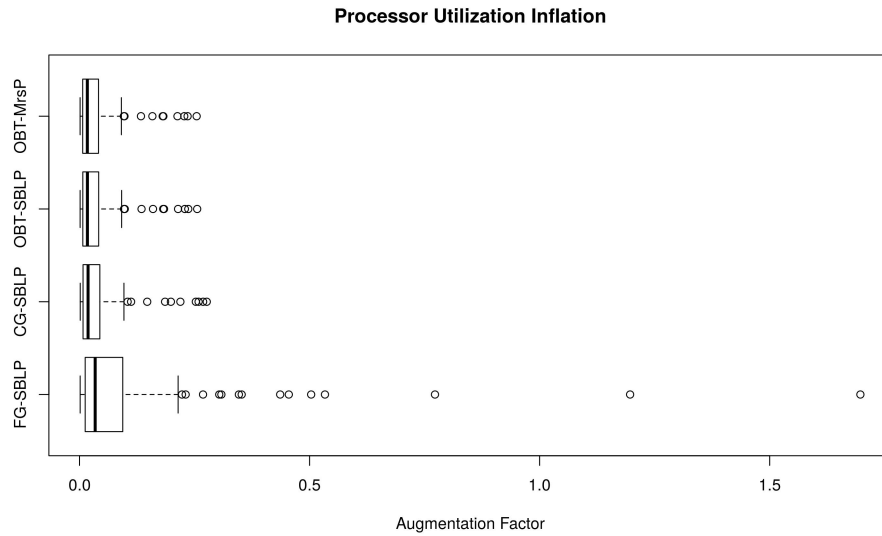


Figure 4.2 Comparison of processor utilization inflation for all heuristics used in this work for all scenarios defined in Tables 4.1 and 4.2.

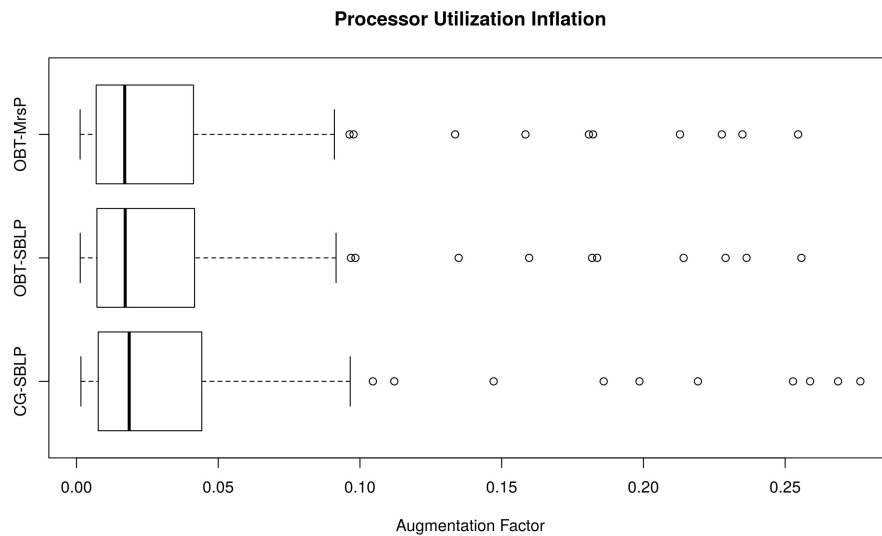


Figure 4.3 Comparison of processor utilization inflation for CG-SBLP, OBT-SBLP and OBT-MrsP for all scenarios defined in Tables 4.1 and 4.2.

We identified therefore the need to evaluate the worst scenarios achieved by each of the heuristics explored in this project, since the Fig. 4.3 does not show great differences between the evaluated solutions, as it shows the averages of all possible configurations of the experiments. Let us now examine in detail the behavior of each protocol by applying heuristics and varying the following parameters: task utilization and period, number of resources accessed by each task, available system resources, and size of critical sections.

4.3.2.1 CG-SBLP Highest Inflation We observed the highest inflation for CG-SBLP was obtained for the parameters defined in Table 4.3.

Table 4.3 Schedulability Test Parameters - CG-SBLP highest inflation.

Number of resources per task	6
Number of system resources	20
Critical section size	range between 50 and 200 μ s
Period	range 50 to 150 ms
Utilization	range 0.1 to 0.3

It is possible to see in Fig. 4.4 the comparison of the results obtained by CG-SBLP, OBT-SBLP and OBT-MrsP for this configuration. The figure represents the results gathered by performing 100 experiments at the highest level of concurrency, *i.e.*, when all tasks access resources.

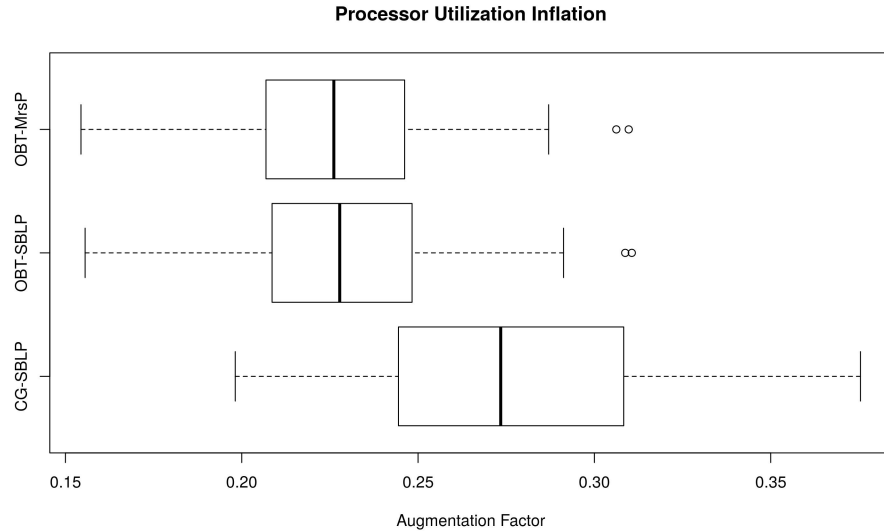


Figure 4.4 CG-SBLP highest processor utilization inflation

Fig. 4.5 illustrates the average run of 100 experiments for the configuration of Table 4.3. The graph shows in the horizontal axis the increments in the system concurrency. 0% stands for none of the tasks accessing resources and 100% for all tasks accessing resources. The right vertical axis reports the percentage increase in system utilization obtained by each heuristic / protocol. The graph shows the highest average inflation for all the experiments disregarding FG-SBLP. This is due to a shorter period of the tasks and the higher blocking time related to the number of resources per task, number of resources in the system and length of the critical section. The figure also shows that OBT-MrsP had results quite close to OBT-SBLP in this case.

Considering not only the average of 100 experiments, but individual results, the worst case of CG-SBLP obtained a difference of 9.34% when comparing to OBT-MrsP, the former presented a inflation of 35.51% where the latter presented 26.16%, which represents a disadvantage ratio of 35.7% for CG-SBLP.

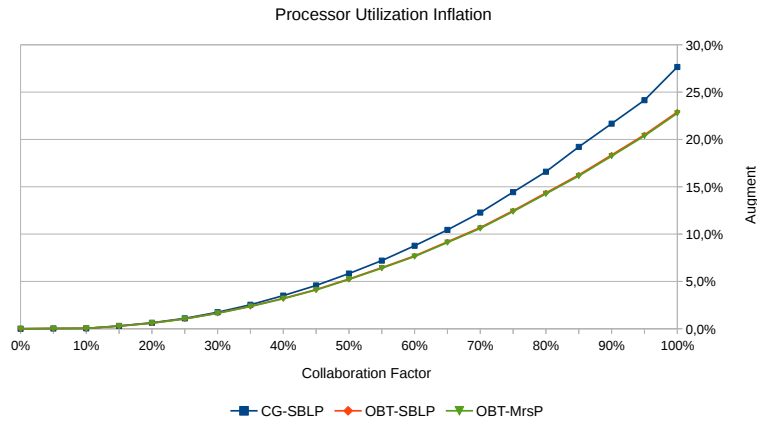


Figure 4.5 Trend for CG-SBLP highest processor utilization inflation

4.3.2.2 OBT-SBLP Highest Inflation The highest inflation for OBT-SBLP was obtained for the parameters defined in Table 4.4.

Table 4.4 Schedulability Test Parameters - OBT-SBLP highest inflation.

Number of resources per task	3
Number of system resources	30
Critical section size	range between 50 and 200 μ s
Period	range 50 to 150 ms
Utilization	range 0.01 to 0.1

Let us observe through Fig. 4.6 the comparison of the results obtained by CG-SBLP, OBT-SBLP and OBT-MrsP at the highest level of concurrency of the tasks for the configuration of Table 4.4.

The figure represents the results gathered by performing 100 experiments for this configuration set. The advantage obtained by OBT-MrsP is not so great. This leads us to conclude that the greatest reduction in inflation was achieved by applying the heuristic rather than the MrsP protocol itself.

Fig. 4.7 represents the results gathered for the configuration described in Table 4.4 for the average execution of 100 experiments. From the figure it is possible to see that there is also high inflation, however, lower than in the previous scenario. This is due to small WCET and period of tasks combined with the larger size of critical sessions which have considerable impact on blocking times. In this example, however, tasks use fewer resources than in the scenario described in Table 4.3 and there are more resources in the system to be distributed between tasks, which reduces concurrency.

4.3.2.3 Worse Results for OBT-MrsP Compared to OBT-SBLP Looking at other configurations, the worse scenarios for OBT-MrsP compared to OBT-SBLP were a tie and had periods ranging between 500 and 2000 (the longest periods of the experiments), utilizations ranging from 0.3 to 0.5 (the highest of the experiments) and size of the critical sections ranging from 10 to 100 μ s (the shortest of experiments). This

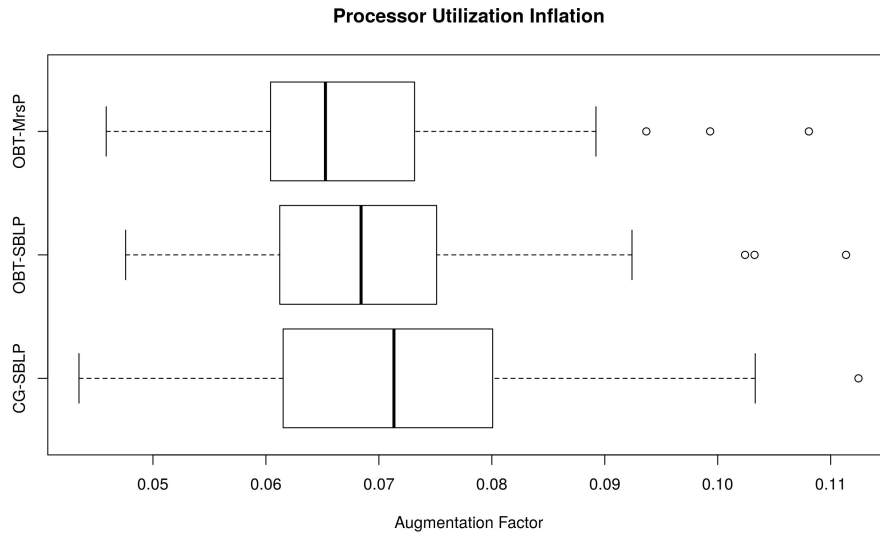


Figure 4.6 OBT-SBLP highest processor utilization inflation

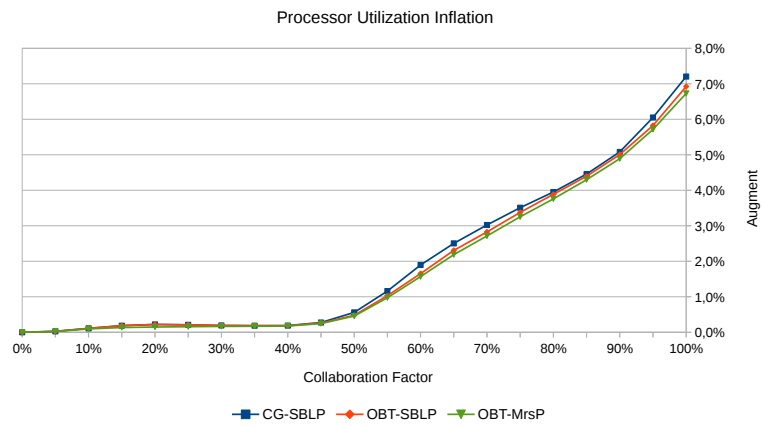


Figure 4.7 Trend for OBT-SBLP highest processor utilization inflation

configuration represents low processor utilization inflation and can be viewed in table 4.5.

Table 4.5 Schedulability Test Parameters - OBT-MrsP X OBT-SBLP

Number of resources per task	6
Number of system resources	20
Critical section size	range between 10 and 100 μ s
Period	range 500 to 2000 ms
Utilization	range 0.3 to 0.5

We can see in Fig. 4.8 that the result for the researched solutions was very close and that the processor inflation was very low.

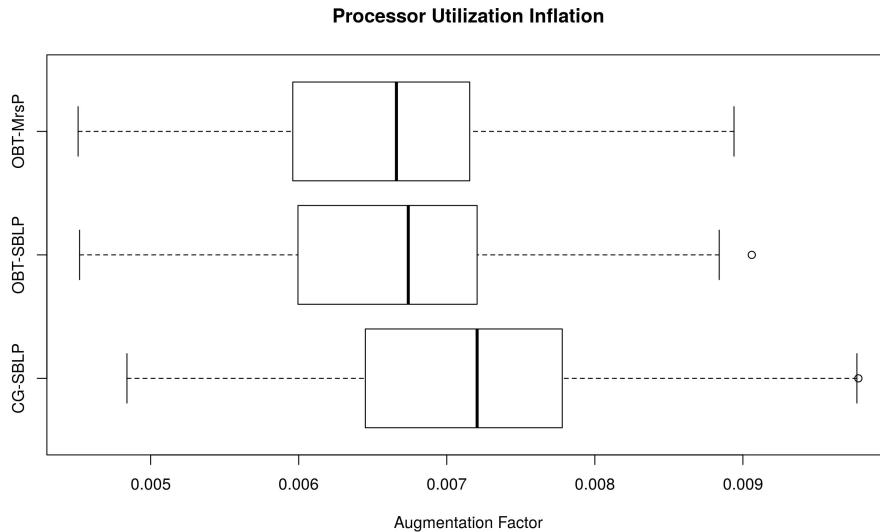


Figure 4.8 Worse results for OBT-MrsP compared to OBT-SBLP

Fig. 4.9 also confirmed the results obtained in Fig. 4.8. It can be seen that the percentages of processor time reservation inflation are low and that the results obtained by OBT-SBLP and OBT-MrsP are almost equal, while CG-SBLP has minor disadvantage.

4.3.2.4 Worse Results for OBT-MrsP Compared to CG-SBLP The worse results for OBT-MrsP compared to CG-SBLP were collected for the parameter combination described in Table 4.6.

Table 4.6 Schedulability Test Parameters - OBT-MrsP X CG-SBLP

Number of resources per task	6
Number of system resources	20
Critical section size	range between 50 and 200 μ s
Period	range 150 to 500 ms
Utilization	range 0.01 to 0.1

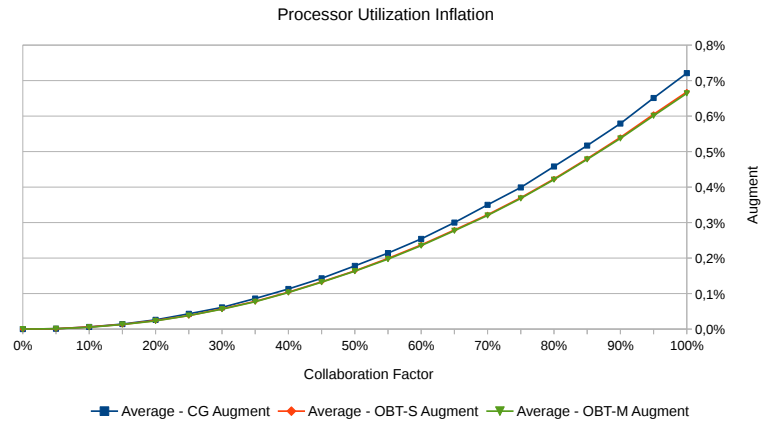


Figure 4.9 Trend for worse results for OBT-MrsP compared to OBT-SBLP

Observing Fig. 4.10 it is possible to realize a very small advantage for the CG-SBLP protocol compared to the others and comparing this figure with Fig. 4.8 we can see that the augmentation factor is 10 times higher.

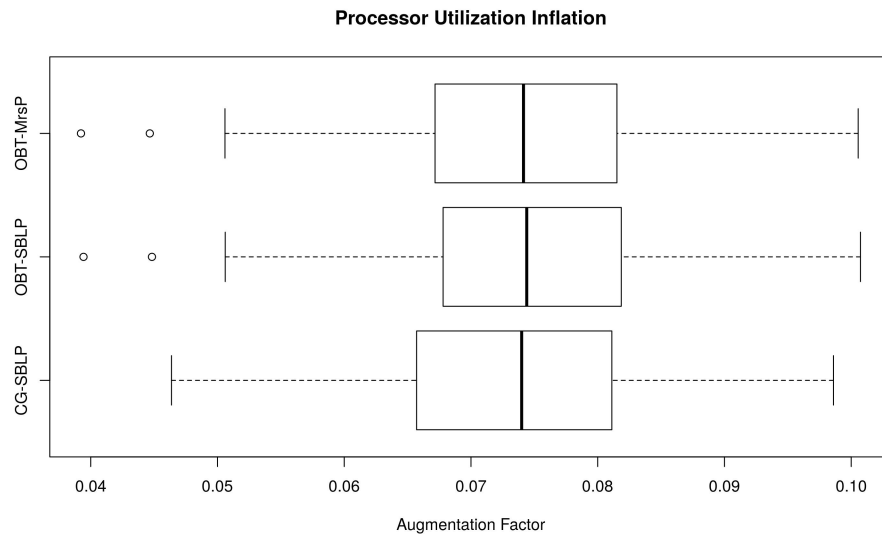


Figure 4.10 Worse results for OBT-MrsP compared to CG-SBLP

Finally, we have Fig. 4.11 representing the trend of processor time reservation as the system concurrency increases for the solutions evaluated according to the configuration given in Table 4.6. We can again see very close results for the heuristics / protocols, which makes us realize that the OBT-MrsP solution has no losses for the other solutions in the observed scenarios and in some scenarios has advantages, the most significant being the advantage over the heuristic CG-SBLP.

Considering that the MrsP protocol obtained positive results when compared to the SBLP protocol with the CG heuristic, but obtained not much better results when com-

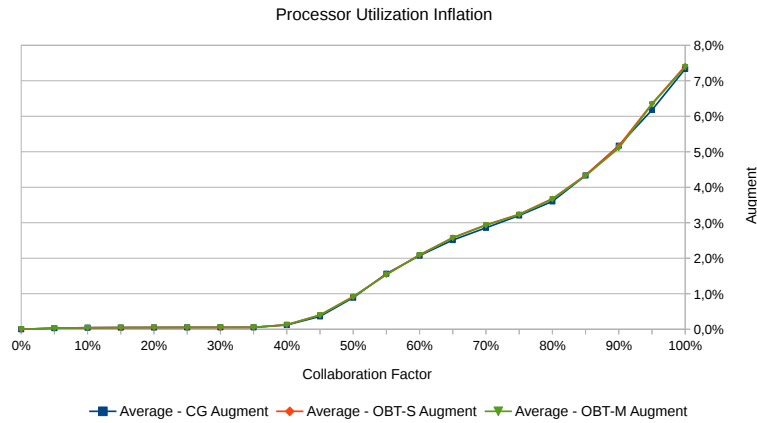


Figure 4.11 Trend for worse results for OBT-MrsP compared to CG-SBLP

pared to SBLP with the OBT heuristic, we verified the importance of observing the behavior of these protocols in a real operating system environment and compare the overhead produced by each of the solutions. In the next chapter, we describe how this solution was implemented in the Litmus^{RT} platform and then present the results obtained by comparing the two solutions.

IMPLEMENTATION

In order to make feasible the experimental evaluation of the MrsP protocol and the comparison of the results with those obtained by the SBLP protocol, we selected Litmus^{RT} platform, a Linux-based operating system designed for evaluating and testing real-time scheduling algorithms and resource sharing protocols (MPI-SWS, Max Planck Institute for Software Systems, 2018).

The following section shows some details of Litmus^{RT} and its relation to our protocol implementation and experimentation. In Section 5.2, we will explain in more detail how we implemented MrsP based on an existing RUN implementation ¹. In Section 6.1, we will present the preparatory procedures for the final experiments, which demonstrated the correctness of the solution proposed in this work and compared the results obtained for the two protocols. Finally, the results obtained by the experiments will be presented and explained.

5.1 LITMUS^{RT}

Litmus^{RT} is an extension of the Linux kernel that makes it easy to deploy and evaluate real-time systems. In this platform, the scheduling algorithms can be implemented through plugins. There are previously defined methods that can be specialized by the plugins to implement the necessary functionality to obtain the expected behavior of the algorithms being developed. Similarly, there are also methods that enable the implementation of resource sharing protocols. This platform has been maintained since 2006 and its current version is 2017.1, which is based on Linux 4.9.30. Some factors were observed by us and determined the choice for this platform, among them:

- Litmus^{RT} is well accepted and widely used in research works;
- It is possible to implement scheduling algorithms and resource sharing protocols as plugins;

¹The implementation is available at https://gitlab.com/ricardo.btxr/RUN_MrsP

- There is a previous implementation of SBLP and RUN for this platform;
- There is an implementation of MrsP for partitioned systems with fixed priority for Litmus^{RT};
- It facilitates the collection of statistics of overhead and scheduling decisions;
- There is a mailing list with active participation of the collaborators;
- Documentation and website with updated information are available;

Liblitmus is an interface to the system kernel functionalities that is provided together with Litmus^{RT}. It contains a tool called *rtspin* which is a simple program that is executed for the simulations required by this research project as a representation of the tasks to be scheduled by the system. This tool runs a loop for a parameterized period of time, which is the duration of the experiments. The launch of real-time tasks is done through the *rt_launch* tool, so we used it to launch the tasks which runs *rtspin*. The parameters to be used when launching a task can vary from protocol to protocol.

The most common parameters are WCET and period of the tasks. In addition to these parameters, our implementation also used the following: (i) identification of the server associated with the task, (ii) the resource sharing protocol used for synchronization of the tasks, (iii) identification of the resource referenced by the task and (iv) duration of the resource's critical session. Some tasks did not use the last three parameters when they do not share resources. Our experiments also used the tool called *release_ts* which allows us to set up and trigger synchronous tasks releases. To enable the execution of tasks in an operating system, it was necessary to detect and correct all errors found in the plugin developed in this work. Detecting and correcting errors in software development that perform the core functions of an operating system is a separate challenge. "However, experience over the last five years show that novice developers still struggle with the complexity of programming in a kernel even after much of Linux's complexity has been hidden - realistically, a certain degree of "kernel hacking" aptitude is required to work with Litmus^{RT}." (BRANDENBURG, 2011)

Recording tracers is an effective technique for analyzing the algorithm behavior and correcting the implementation errors. Linux developers often use *printk()* function to issue logs that can aid in debugging the software, however using this function can lead to deadlocks when it is called inside Litmus^{RT} plugin functions. To circumvent this situation, some macros were developed by University of North Carolina to enable the collection of tracers without impacting the execution of the system. These tracers are usually turned off in the executions of the experiments when the purpose is to collect the workload data, avoiding additional overhead. The macros that generate tracers containing the data for software debugging do not apply to the purpose of overhead data collection and scheduling statistics, since they use strings instead of binary format.

In order to enable the collection of data, there is a library containing a set of tools, scripts, and macros that enable overhead data collection and job execution statistics, including scheduling decisions, preemptions and migrations. This library is called Feather

Trace Tools and was thought to produce low impact in the execution of the Litmus^{RT} kernel, avoiding greater impacts on the overhead as a whole. In the implementation of the plugin methods it is possible to define new macros to collect more specific overhead data, but there are macros previously defined in the platform that collect overhead data for the events of (i) context switching, (ii) lock, (iii) unlock, (iv) schedule decision, (v) release and (vi) release latency. For the implementation of the protocols evaluated in this work, we also collected data about the operations related to the update of the RUN Reduction Tree. There is also the need to collect and analyse schedule data.

Feather Trace Tools have macros that allow the plug-in developer to define and collect all the scheduling data tracers needed for an analysis of a problem. These macros are designed to save data in a location other than where the overhead data is saved. Some tracers are generated by Litmus^{RT} methods, so they do not require additional implementation in the plugin code, e.g., a job release and its completion. From this data, we can generate statistics and evaluate whether deadlines were missed, as well as the number of migrations and preemptions that a job has suffered. It is also possible to configure other types of data related to scheduling and synchronization decisions. For this work tracers also collected data regarding the helping mechanism, as well as spinning time.

Compiling the Linux kernel is the first step in order to run the system configured with the scheduling plugin. The web site (MPI-SWS, Max Planck Institute for Software Systems, 2018) contains basic information about how to install Litmus^{RT} and compile the Linux kernel. In the first compilation it is necessary to take into consideration all the Linux modules, as well as the kernel, which is a time-consuming operation, but in the following compilations it is possible to compile only the kernel and this significantly reduces the time spent for compiling, except when there are kernel configuration changes. The development of plugins in Litmus^{RT} is not a trivial service, since Linux kernel development knowledge and skills are necessary for this objective. For a better understanding of how this process was conducted in this work, the details of the implementation of MrsP for RUN are discussed in the following section.

5.2 IMPLEMENTATION DETAILS

The development of plugins for Litmus^{RT} requires that the Linux kernel hacking skills are present. These skills were intensively explored during the stages of implementation, testing and experimentation for this work. In this chapter we will describe some of the tools that have supported this development process as well as explain the functions implemented to meet the requirements for adapting the MrsP protocol to the RUN algorithm.

Virtualization technique provides essential support for debugging a scheduling algorithm that runs within an actual operating system. Without such a tool it would be unfeasible to detect system crash situations where no information about the error is presented in the screen. For example, when the software stops responding without any trail as to what motivated nonresponsive behavior. QEMU was the software chosen in this work for virtualization (QEMU, 2019). Its ease of use, stability and abundant documentation favored the choice of this tool. In conjunction with virtualization, a debugging tool becomes extremely useful.

The tools selected for debugging the code were GNU Debugger (GDB) and Kernel Debugger (KDB) (ALVES et al., 2019; WESSEL, 2019). Both are widely disseminated among Linux kernel developers, but there are a variety of tools of a similar nature. Through GDB it was possible to gain access to the tracers generated by our plugin in situations where the system crashed. KDB complemented the functions of GDB, allowing access to the system that crashed to obtain important information about the state of the system. Step-by-step debugging techniques have not been shown to be effective for real-time systems. The best approach is to generate tracers and analyze them after running the test cases. Another tool that gave important support to this work was the version control tool. We selected the GitLab web tool motivated by the proposal to allow free use, but maintaining a restricted access control to our code (GITLAB, 2019). After gaining knowledge in using these tools, the next step was to begin implementation.

As we were aware, there is an implementation of the SBLP protocol (BONATO; MEZZETTI; VARDANEGA, 2014) for the Litmus^{RT} platform based on an implementation of the RUN algorithm²(COMPAGNIN; MEZZETTI; VARDANEGA, 2014). As the available implementation of the protocol had been made for an older version of Litmus^{RT} (v2012.3), so we had to adapt this code to the newer version of Litmus^{RT}, since the old version is no longer supported and contains bugs that were already fixed. We did the adaptation of the code avoiding to modify the existing functionalities. Then we made a copy of the whole code (which includes the Linux and Litmus^{RT} code) and started the implementation of the new plugin which implements the rules of the MrsP protocol and the RUN algorithm.

For the development of a new plugin we followed the steps proposed in the documents that describes the creation of plugins for Litmus^{RT}(MPI-SWS, Max Planck Institute for Software Systems, 2018). In addition, we referenced Chapter 3 of Brandenburg’s thesis (BRANDENBURG, 2011) which describes the architecture of this platform and explain in detail the methods available for creating plugins. In the following, we will describe some of these methods and the details of their implementation.

5.2.1 Scheduling Plugin Methods

For a better understanding of the implementation of our scheduling plugin, we will follow with an explanation of the main methods available in the Litmus^{RT} plugin interface that were coded in this project.

The *schedule()* method of a Litmus^{RT} plugin is invoked to determine what the next task will be selected to run on a processor. By the original RUN implementation for Litmus^{RT}, when there is no server associated with the processor being scheduled, no task is selected to execute and Litmus^{RT} returns a NULL pointer to the Linux operating system schedule function, indicating that it can select a non-real-time task for execution.

Whenever there is a server associated with the processor being rescheduled, a verification occurs inside the *schedule()* function to determine whether the highest priority task of the current server’s ready queue can take the place of the task of the same server that was previously running. At this time, EDF priority is considered, but it is also checked

²The code was handed to us by Dr. Enrico Mezzetti, one of the authors of SBLP.

whether the preemption level of the high priority ready task is higher than the server's ceiling level. This is done to ensure the inherited property of the SRP protocol.

The ready queue contains tasks that are ready to run, and the release queue contains tasks that will be released to execute at a future time. Whenever a task is selected to run, it must be removed from the ready queue and the *schedule()* function ensures consistency between ready and release queues. In cases where a task is leaving the processor, the task will be stored in one of the available queues (*i.e.*, ready queue or release queue) according to its status.

The helping mechanism can be activated or deactivated according to the task transition, *i.e.*, the helping mechanism should be deactivated whenever a task that was helping another is not selected for execution. Similarly, if a task is selected to execute and the resource it is requesting is blocked by a task which is not running, so the helping mechanism must be activated and the selected task must give way its processor to the blocking task. Each time the helping mechanism is activated, the task which is receiving help is removed from the ready queue of its server, in order to prevent the task from running on two servers simultaneously. In the same way, when the helping mechanism is deactivated, both the helper task and the task being helped are returned to the ready queue of their respective servers in order to allow them to be rescheduled.

Algorithms 2 and 3 give a general understanding of servers allocation to processors and the RUN task scheduling mechanism³.

According to multiprocessor resource sharing protocols based on PCP / SRP, each resource must have a ceiling level for each processor that has tasks that use the resource. In adapting the MrsP protocol to the RUN algorithm in this work, it was established that servers would be an analogy of processors for the purpose of managing concurrent access to resources, so each resource must have a ceiling level for each server that has tasks that use it. In our implementation of the MrsP protocol, when a task uses resources, it must initially invoke the *open()* method for each resource it uses. This method determines the resource ceiling level for the server to which the task belongs. The ceiling level of the resource is given by the shortest period of tasks that use it for a given server.

The *lock()* method is invoked whenever a task needs to access a shared resource. In this method it is checked if the resource is already locked by a task of another server, in this case the current task will busy wait if the resource holder task is running, or will help the resource holder task otherwise. If the requested resource is not locked, the task get the resource. The server ceiling level can be increased in this function if the resource ceiling level for the server is higher than the current server level. A stack is used for returning to the previous ceiling level after the lock be released. It is important to remember that resource access requests are served in FIFO order and this work does not deal with nested resources. See Algorithm 4 for a better understanding of the lock event⁴.

When a task is no longer accessing a resource, it must call the *unlock()* method. This method restores the server ceiling level to the level prior to the corresponding lock

³Some details have been omitted for ease of understanding

⁴Some details have been omitted for ease of understanding

Algorithm 2: RUN-MrsP schedule servers event

Require: $\Gamma^{0..i}$; {the RUN reduction tree with i levels}
Require: $P^{0..m}$; {list of m processors}

- 1: **for all** $\sigma \in \Gamma^{0..i}$ **do**
- 2: update deadline of σ ;
- 3: **end for**
- 4: **for all** $\sigma \in \Gamma^{i..0}$ **do**
- 5: check or uncheck σ for execution; {the tree is traversed from root to leaf}
- 6: **end for**
- 7: **if** $\sigma \in \Gamma^0$ is unchecked **then**
- 8: $\tau \leftarrow \sigma$'s current task;
- 9: enqueue τ into σ 's ready queue;
- 10: **if** τ is helping a task τ' **then**
- 11: enqueue τ' into its server's ready queue;
- 12: **end if**
- 13: **end if**
- 14: $\Gamma' \leftarrow$ set of servers checked for execution, $\Gamma' \subset \Gamma^0$;
- 15: **for all** $p \in P^{0..m}$ **do**
- 16: assign σ to p , $\sigma \in \Gamma'$;
- 17: $\Gamma' \leftarrow \Gamma' \setminus \{\sigma\}$;
- 18: schedule p ;
- 19: **end for**

Algorithm 3: RUN-MrsP schedule processor event

Require: σ ; {Server assigned to the processor being scheduled}
Require: τ ; { σ 's current task}

- 1: **if** $\sigma = 0$ **then**
- 2: **return** 0;
- 3: **end if**
- 4: $t \leftarrow$ current instant;
- 5: $\tau' \leftarrow$ dequeue highest priority task of σ 's ready queue;
- 6: **if** τ is completed **or** ($priority(\tau') > priority(\tau)$ **and** $\Pi_{\tau'} > \Pi_{\sigma}(t)$) **then**
- 7: enqueue τ into release queue or σ 's ready queue;
- 8: **if** τ is helping a task τ'' **then**
- 9: enqueue τ'' into its server's ready queue;
- 10: **if** τ'' 's server is assigned to a processor p' **then**
- 11: schedule p' ;
- 12: **end if**
- 13: **else if** $lock(\psi, \tau, t)$ **then**
- 14: **for all** $\tau'' \in \psi$'s queue **do**
- 15: **if** τ'' is running **then**
- 16: schedule the processor τ'' is running; {to enable τ to receive help}
- 17: **end if**
- 18: **end for**
- 19: **end if**
- 20: σ 's current task $\leftarrow \tau'$;
- 21: **return** τ' ;
- 22: **else if** $wait(\psi, \tau, t)$ **and** $lock(\psi, \tau''', t)$ **and** τ''' is not running **then**
- 23: dequeue τ''' from its server's ready queue;
- 24: **return** τ''' ;
- 25: **else**
- 26: **return** τ ;
- 27: **end if**

Algorithm 4: RUN-MrsP lock event

Require: p ; {Current processor}
Require: τ ; {Current task}
Require: σ ; { τ 's server}
Require: ψ ; {Requested resource}

- 1: $t \leftarrow$ current instant;
- 2: store σ 's current ceiling level into σ 's stack;
- 3: increase σ 's current ceiling level to the ψ 's ceiling level for σ , if necessary;
- 4: **if** lock(ψ, τ', t) **then**
- 5: enqueue τ into ψ 's queue;
- 6: τ waits for ψ ;
- 7: **if** τ' is not running **then**
- 8: schedule p ;
- 9: **end if**
- 10: **while** lock(ψ, τ', t) **do**
- 11: τ spin;
- 12: **end while**
- 13: τ blocks ψ ;
- 14: **else**
- 15: τ blocks ψ ;
- 16: **end if**

operation. If the running task was receiving help from another task, it is necessary to deactivate the helping mechanism and invoke the rescheduling operation on both processors which are hosting the server of the helping task and the server of the task being helped. If the running task is not getting help, then releasing the lock should invoke a rescheduling operation on the respective processor, as the server ceiling level may have dropped and some locally blocked task may start running.

After the resource lock is released, the global blocking queue for the resource is traversed to check for any tasks waiting to access it. If so, the first task in the queue gets the access to the resource. If the task is not running and there are tasks busy waiting for the resource, then the helping mechanism is activated. In this case, the spinning task will give up the place on the processor for the task that got the lock on the resource and, as a result, a reschedule will be required on the spinning task processor. Algorithm 5 helps to understand the *unlock()* method⁵.

Fig. 5.1 represents the state diagram for the implementation of RUN with MrsP. It is possible to observe the state transitions and the events that trigger each transition. We can see that a task only gets help if it is ready to execute and has a lock on a resource. Similarly, a task only provides help if it is running in busy waiting mode. Other state transitions are common to other scheduling and resource sharing algorithms, so they will not be explained here. In the next chapter we will discuss the overhead measurement

⁵Some details have been omitted for ease of understanding

Algorithm 5: RUN-MrsP unlock event

Require: p ; {Current processor}
Require: τ ; {Current task}
Require: σ ; { τ 's source server}
Require: ψ ; {Blocked resource}

- 1: τ unlock ψ ;
- 2: restore σ 's current ceiling level from σ 's stack;
- 3: **if** τ was being helped **then**
- 4: enqueue τ into σ 's ready queue;
- 5: **if** σ is allocated to processor p' **then**
- 6: schedule p' ;
- 7: **end if**
- 8: **end if**
- 9: schedule p ;
- 10: **if** ψ 's queue is not empty **then**
- 11: dequeue τ' from ψ 's queue;
- 12: τ' blocks ψ ;
- 13: **if** τ' is not running **then**
- 14: **for all** $\tau'' \in \psi$'s queue **do**
- 15: **if** τ'' is running on processor p'' **then**
- 16: schedule p'' ; {to enable τ' to receive help}
- 17: **break**
- 18: **end if**
- 19: **end for**
- 20: **end if**
- 21: **end if**

OVERHEAD EVALUATION

The comparative study of overhead originating from the MrsP and SBLP protocols, both applied in conjunction with the RUN algorithm, will be detailed in this chapter. The experiments performed for this comparison were important because they demonstrated how each protocol behaved in applying the basic task scheduling and resource sharing primitives in a real-time operating system environment. In the next session, we will explain how the experiments were configured for the mentioned protocols and the heuristics applied to each one. In the following, we will present the results obtained and explain them.

6.1 EVALUATION PREPARATION

Aiming at sequencing and configuring the use of the tools described in Section 5.1 and enabling future replications of the experiments, we used sets of scripts that were provided to us along with SBLP and RUN implementations¹. Among other things, these scripts do (i) activation of the scheduling plugin, (ii) activation of the tracers for data collection of overheads, scheduling decisions, lateness, preemptions, migrations, deadline misses and other informations for debugging purposes, (iii) loading the parameters for executing the tasks, (iv) loading and configuring the RUN reduction tree, (v) execution of the tasks, (vi) shutdown of the tracers and copy resulting files to the experiment folder and (vii) deactivation of the plugin. Before carrying out the experiments, it was first necessary to prepare a mass of data to enable the comparison of the protocols.

This work used 8 parameter combinations to prepare the data mass for the execution of the experiments. For each of these combinations, 100 task sets were randomly generated, totaling 800 algorithm execution cases, obtaining a significant set of scenarios for the comparison of protocols and heuristics. The experiments were performed on a 4-core Intel (R) Core (TM) i5-6200U CPU @ 2.30GHz notebook with L1, L2 and L3 caches of 32K, 256K and 3072K, respectively. Power scaling and power management functions

¹The *Python* scripts were handed to us by Dr. Enrico Mezzetti, one of the authors of SBLP.

were intentionally disabled. Since there were a limited number of cores available, the task sets generated for the experiments were compatible with this feature, as we will see ahead.

We set the initial total utilization of each set of tasks to 3.5, as there were 4 processor cores to perform the tasks. The difference between the total hardware capacity and the total initial utilization of the system was designed to accommodate the necessary inflation regarding blocking times that tasks may suffer from concurrent access to resources. The utilization of each task was generated randomly through a uniform distribution within the range 0.075 to 0.35. The number of tasks for each experiment could vary and was not defined as a parameter for experiments. The period of each task, in turn, was chosen randomly through a uniform distribution within the following set of values: [100, 125, 150, 175, 200, 250, 300, 400]. The task periods were set in milliseconds. Each task had its maximum execution time limited to 95% from its WCET, in order to accommodate the operating system overhead. The number of resources to be distributed among tasks was set at 4 resources but each task could access only 1 resource. Now we talk about the parameters that varied between the execution of the experiments, forming the 8 configurations mentioned.

Resource distribution is the parameter that defines the level of concurrency between tasks. This parameter defines how many tasks can access available resources, 4 options were used for it: 0.5, 0.65, 0.8 and 1.0. This parameter set to 0.5 means that 5 out of 10 tasks accesses resources, while 1.0 indicates all tasks accesses resources. The critical section length of a resource has been set to 4% or 6% of the smallest WCET of tasks using the resource, but fractional values were rounded down (*floor* function). Each set of tasks was configured to run for 30 seconds.

After the generation of the task sets for the experiments, we observed the final utilization obtained by the application of the formula used to calculate the processor utilization inflation demanded by the protocols. This result can be seen in Table 6.1. It is noticed that the protocols presented similar inflation. The smallest inflations were obtained for Resource Distribution 0.5 and resource critical section 4%, while the largest inflations were obtained for these same factors set at 1.0 and 6%. These values can be seen in Table 6.1 through the Minimum and Maximum columns, respectively. This is reasonable because the lowest inflations were obtained for the least concurrent task sets.

Table 6.1 Processor final utilization for CG-SBLP, OBT-SBLP, and OBT-MrsP

Heuristic-Protocol	Average	Minimum	Maximum
CG-SBLP	3.57	3.52	3.73
OBT-SBLP	3.57	3.52	3.74
OBT-MrsP	3.57	3.51	3.74

We also compared the number of servers generated by the scripts for the task sets of CG-SBLP, OBT-SBLP, and OBT-MrsP. The result of this comparison can be seen in Table 6.2. It is clear that the heuristics applied to SBLP obtained more servers, they generated 14% more servers than MrsP. Regarding OBT, the difference is due to the non-application of the heuristic step 5 for the SBLP protocol (see Subsection 4.2.1),

since the application of this step would not be advantageous for the protocol in terms of schedulability as we can see in the example below.

Example 4. Consider tasks τ_i and τ_j with periods set at 10ms and 1000ms, respectively, and being able to access resources ψ_i and ψ_j , respectively. Under SBLP, if both tasks were on the same server, τ_i could be prevented from running while τ_j was accessing or waiting for resource ψ_j . This condition would make the system unschedulable if the length of the critical session of ψ_j is equal to or greater than 10ms and this is an undesirable situation since τ_i and τ_j do not compete for the same resources. This is a problem that does not occur with the MrsP protocol.

As in this part of the experiments each task was limited to accessing only 1 resource, there were more servers for SBLP, as the heuristics applied to this protocol did not join sets of tasks that did not have common resources on the same server.

Table 6.2 Servers demanded by the experiments task sets for CG-SBLP, OBT-SBLP and OBT-MrsP. The Total and Average columns represent the number of servers generated for all experiments and the average number of servers obtained for each experiment, respectively.

Heuristic-Protocol	Total	Average	Standard Deviation
CG-SBLP	4495	5.62	0.53
OBT-SBLP	4523	5.65	0.53
OBT-MrsP	3938	4.92	0.46

Finally, we look at the RUN reduction tree level for each of the solutions. Table 6.3 contains the averages of the tree levels for all tasks sets of the experiments. From the table it is possible to see that there is a reasonable advantage to the OBT-MrsP solution. As we can see from the number of servers generated for each solution, there is also influence of step 5 of the OBT heuristic (Subsection 4.2.1) on the level of the tree.

Table 6.3 Maximum RUN reduction tree level for CG-SBLP, OBT-SBLP and OBT-MrsP

Heuristic-Protocol	Total
CG-SBLP	2.60
OBT-SBLP	2.63
OBT-MrsP	2.07

At running the data generation scripts, 3 files are generated for each task set: `params.py`, `sched.py`, and `tree.json`. The first file contains the parameters used for the experiment setup, the second file contains the tasks and their settings (server, identification of the resource accessed, size of the resource critical section, WCET and period). Finally, the last file contains the RUN reduction tree in JavaScript Object Notation (JSON) format (ECMA, 2019). These files are input for running the experiments to collect overhead statistics and scheduling decisions.

During the execution of the experiments, each lasting 30 seconds, one round for each of the 800 task sets, two tracers collected plug-in execution statistics, as described in

Section 5.1. One tracer collected overhead statistics for (i) context switching, (ii) lock, (iii) unlock, (iv) schedule decision, (v) release and (vi) release latency. The other tracer was responsible for collecting (i) preemption, (ii) migrations, (iii) deadline losses, (iv) execution times of each job. Both tracers produce data in binary files and complementary tools allow us to access data from these files and collect the statistics of the experiments. In the next section we will discuss the results and analyze the protocols.

6.2 EVALUATION RESULTS

In order to analyze the performance of the protocols with their respective heuristics, we start by looking at the overhead produced by each one in performing the basic scheduling and resource sharing primitives in the following session. After we then characterize overheads in terms of the number of migrations and preemptions suffered by the tasks in the execution of the experiments within the Litmus^{RT} environment.

6.2.1 Overhead

The comparison of the overhead obtained by each of the protocols studied in this work was based on the collection of a timestamp at the beginning of an event and the collection of a timestamp at the end of the code execution for the same event, thus obtaining an interval of time in nanoseconds which represents the code execution of an implementation that is part of the operating system primitives for a real-time system, such as those related to task scheduling and resource sharing.

The time measurements collected in this work were (i) context switching: time spent for one task leaving the processor and another task occupying it; (ii) lock: measures the time interval between requesting a lock operation on a resource and obtaining the lock. Time spent in busy waiting activity is not considered in this measurement; (iii) unlock: time interval between the request for the resource release and the effective release of the resource; (iv) schedule decision: every time a scheduling decision is made to a processor, the time spent on that decision is collected. The measurement is divided into two parts, one of which refers to task selection and the other refers to post-context-switch cleanup and management activities; (v) release: measures how much time is spent to enqueue a newly-released job in the ready queue; (vi) release latency: release latency is the difference between the time a task release should have taken place and the time it actually took place; (vii) Tree: represents the time spent by the system maintaining the state of the RUN reduction tree.

Fig. 6.1 shows the comparison of the average of the results obtained during the experiments performed. Let us now analyze the differences obtained by the protocols.

(i) Context switch (CXS): The difference between the MrsP and SBLP protocols was up to 5.2% for the CG-SBLP composition and up to 3.8% for OBT-SBLP. This result means that MrsP performed worse. We found no clear motivation for the difference between the protocols when analyzing their implementations, except that we added extra information to MrsP's task data structures, which could justify a little additional overhead for the protocol.

(ii) Lock: As with context-switching events, we also had a difference in measurements made for SBLP and MrsP for resource blocking events that was unfavorable to MrsP. The result was up to 4.2% higher for MrsP when compared to CG-SBLP and 1.7% higher when compared to OBT-SBLP. MrsP uses a stack to store server's ceiling level and maintaining it may have affected the time taken to acquire the lock. The SBLP plugin does not need to implement this stack, as the protocol is non-preemptive inside a server whenever a task is requesting or locking a resource. In addition, the MrsP plugin helping mechanism can be configured during the resource lock and in doing so our plugin removes the task that receives the help from the ready queue. This does not happen with the SBLP plugin.

(iii) Unlock: The MrsP plugin restores the server ceiling level from the stacked level. In this event, the helping mechanism can be deactivated and in this case the task receiving help is reinserted into the ready queue. MrsP can also set up a new helping mechanism and in doing so removes the task that gets help from the ready queue. SBLP can configure the helping mechanism during this event, but does not remove the task that gets help from the ready queue, as the task requesting the resource cannot be preempted by another task from its source server. The difference in time spent on this primitive was unfavorable to MrsP up to 6.5% when compared to CG-SBLP and up to 5.6% for OBT-SBLP.

(iv) Schedule decision (SCHED and SCHED2): The difference for the SCHED event was 12.9% and 12.6% disadvantageous for MrsP when compared with CG-SBLP and OBT-SBLP, respectively, and it was 1.9% and 1.6%, respectively, for SCHED2. In MrsP, the `run_schedule()` method may include in the ready queue of the servers both tasks which offers and receives help by undoing the helping mechanism; This method can also set up a new helping mechanism and, when this occurs it removes the task that gets help from its server's ready queue. In SBLP, when the server is blocking or requesting a resource, the `run_schedule()` method automatically selects the previously running task (`hp_task`), not undoing the helping mechanism when the task stops running. This check is now done within the `resched_servers()` method because when the server leaves the processor it needs to undo the helping mechanism. We believe that adding or removing tasks in the ready queue can negatively influence the MrsP protocol runtime.

(v) Release: MrsP spent up to 10.6% and 15.5% less time than CG-SBLP and OBT-SBLP, respectively, to perform this operation. The implementation of both plugins goes through the RUN reduction tree to update its status with deadline information of released tasks. The smaller number of servers influenced the release-related overhead in favor of the MrsP protocol. Both protocols check to see if there is a need to undo the helping mechanism, but only the MrsP protocol includes in the ready queue of the servers the tasks that offers and receives help, which slightly penalizes this protocol.

(vi) Release latency: It was not very clear to us what factors in the implementation of the plugins could have influenced this metric. We observed that MrsP delayed up to 32% more than CG-SBLP to release the tasks and up to 20.3% more than OBT-SBLP. It is not trivial to find the reason for the difference in this measurement. When a job is created but not in its release time (*e.g.*, when a job completes a new job is created to be released at the next execution interval), it is placed in the release heap and a Linux timer is set to fire at release time. When this timer reaches its time, the system executes the method that generates the trace for Release Latency. Unlike other tracers, this tracer

is a single record (the other tracers have two records, one for the beginning and one for the end of the event). In the generation of this trace, the job release time is passed and the difference between the time and the current time gives the measurement for Release Latency. Understanding why the timer is being delayed is not straightforward, as the source may be the blocking of interruptions or some lock on the operating system semaphore that may be causing the delay. The origin of this delay can be at several points in the code.

It should be noted that the maximum delay on this one was similar for both protocols. Fig. 6.2 shows the results considering 99.9 percentile. It is also important to note that the median for both protocols presented low and very close values, around 0.6 microseconds.

(vi) Tree: The *tree()* method traverses the RUN reduction tree to update its status with deadline information for released tasks. The smaller number of servers certainly influenced the overhead related to this operation favorably to the MrsP protocol. Scanning this tree may enable or disable the help engine for both protocols. This may include adding or removing tasks from the ready queue of their respective servers which affect only MrsP.

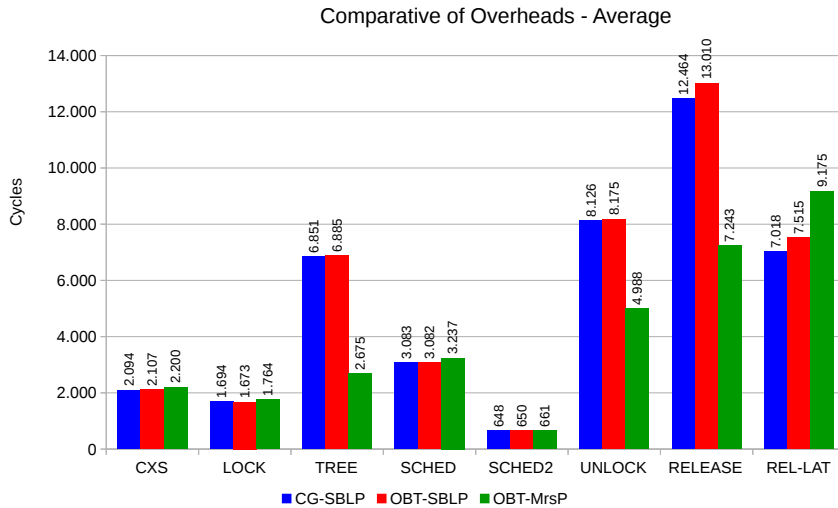


Figure 6.1 Comparative of overhead for CG-SBLP, OBT-SBLP and OBT-MrsP

As we can see from these results, the MrsP protocol obtained better results in the methods in which the RUN reduction tree is scanned due to the smaller size of its tree, but was penalized in the methods in which the helping mechanism is activated or deactivated, due to the need for more job additions and removals in the ready queue.

For a fairer comparison, we need to measure the overheads taking into consideration the number of events that occurs on the processors during the experiments execution. Doing so, we weight the time spent handling each event by the number of occurring events. For example, as the reduction tree for each protocol differs, number of events to be handled by them is unlikely to be the same. Fig. 6.3 plots a graphs comparing the weighted overhead. As can be seen the total weighted overhead achieved by OBT-SBLP and CG-SBLP was greater than OBT-MrsP by 3.24% and 8.48%, respectively.

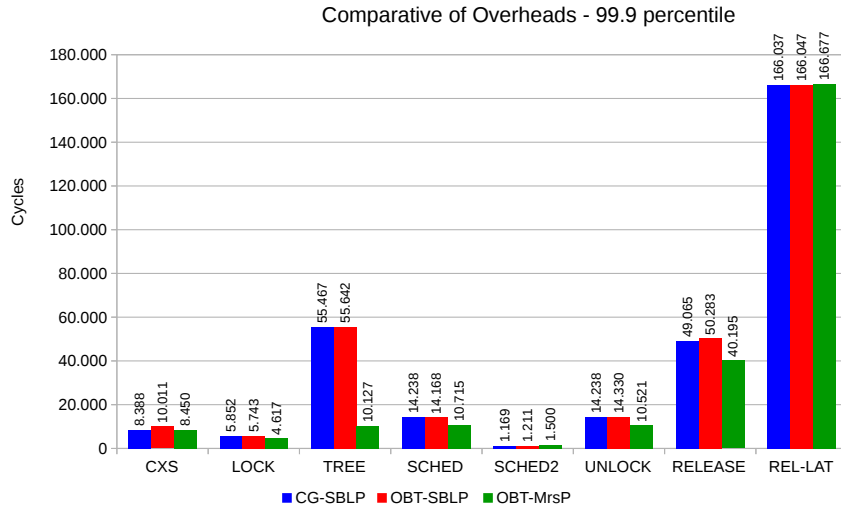


Figure 6.2 Comparative of overhead for CG-SBLP, OBT-SBLP and OBT-MrsP - 99.9 Percentile

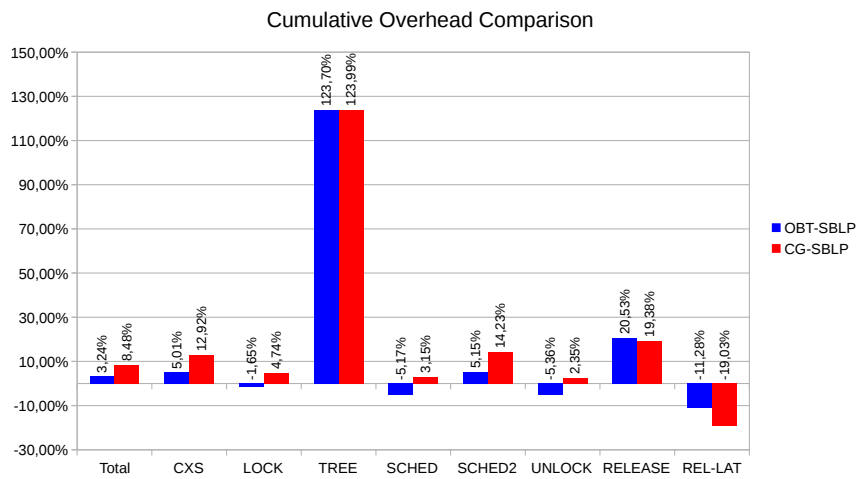


Figure 6.3 Comparative of weighted overhead for CG-SBLP and OBT-SBLP in relation to OBT-MrsP

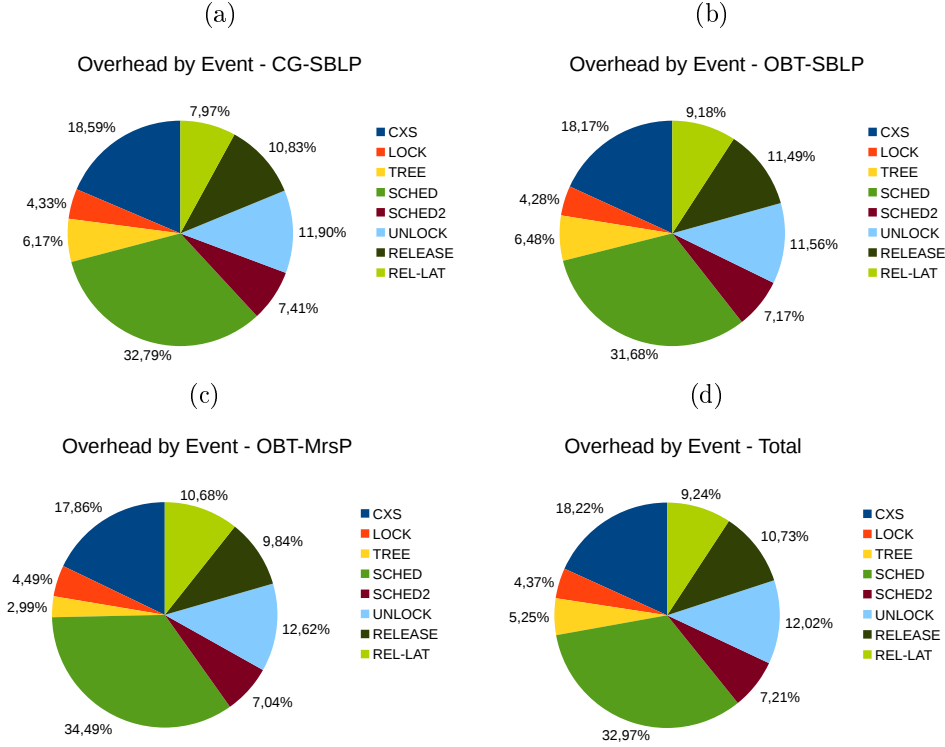


Figure 6.4 Distribution of Weighted Overhead by Events

Fig. 6.4 shows the total weighted overhead distributed between events. Fig. 6.4(d) shows the general distribution of weighted overhead between events for all solutions evaluated in this work. It can be seen that the Schedule event produces the greatest overhead, followed by the Context Switch.

In the next section we will discuss the results obtained by the protocols related to the number of preemptions and migrations.

6.2.2 Preemptions and Migrations

Gathering migration and preemption statistical data is already part of the standard Litmus^{RT} toolkit, so no additional implementation of our plugin was required to enable this data collection. After each round of the experiment, the Litmus^{RT} libraries generate a binary file containing the following information: (i) Task ID, (ii) JOB Number, (iii) Task Period, (iv) Time to complete JOB (v) Flag indicating deadline loss, (vi) Difference between JOB completion time and its deadline, (vii) Delay to deadline, (viii) Flag indicating whether conclusion was forced due to the completion of the experiments, (ix) JOB execution time, (x) Number of preemptions and (xi) Number of migrations.

Regarding the number of preemptions, we observed that the MrsP protocol performed better in most scenarios evaluated, averaging 12.5% less preemption, but there is a scenario where the reduction in the number of preemptions reached 26% lower for the protocol. The main reason for this performance is related to the smaller number of servers, which implies less preemption due to rescheduling of servers. Fig. 6.5 illustrates this

result, the x-axis represents pairs of the parameters concurrency factor and size of the critical section. The Y-axis represents the averages of the number of preemptions suffered by each job. As we can see, MrsP performed better in almost all scenarios except when the concurrency and critical section size parameters were set to 0.65 and 0.06, respectively, when the SBLP protocol obtained very close results. The reason for this close result between protocols has not been fully understood by us, but it probably has to do with the formation of the servers.

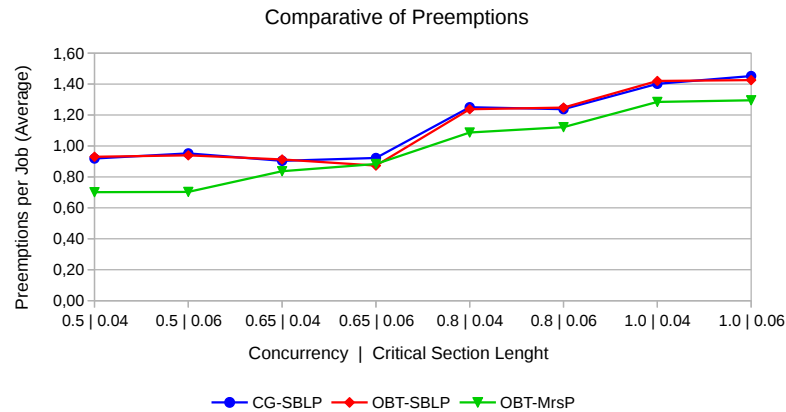


Figure 6.5 Comparative of the number of preemptions per task for CG-SBLP, OBT-SBLP and OBT-MrsP

The results obtained for the migrations were also favorable to the MrsP protocol. We observed that this protocol had an advantage of up to 67.8% in the number of migrations for one of the observed scenarios, while the average of this protocol was 36% less migrations when compared to CG-SBLP and 33.5% less migrations for the comparison with OBT-SBLP. The reasons for fewer migrations are the same as those for preemptions. With fewer servers, there are less task migrations. Fig. 6.6 illustrates this result. It can be observed that only one of the scenarios produced better results for OBT-SBLP, in this case the advantage was 10.3% less in the number of migrations to this protocol.

As we can see in Figs. 6.5 and 6.6, there is a clear advantage obtained by the MrsP protocol due to the smaller number of servers and this advantage was provided by the protocol's ability to join tasks that do not compete for the same resources on the same server without impacting execution of high-priority tasks allocated to servers. As demonstrated in Example 4, the SBLP protocol can reduce the system's schedulability level when tasks that do not compete for the same resources are placed on the same servers and these tasks have widely varying time periods.

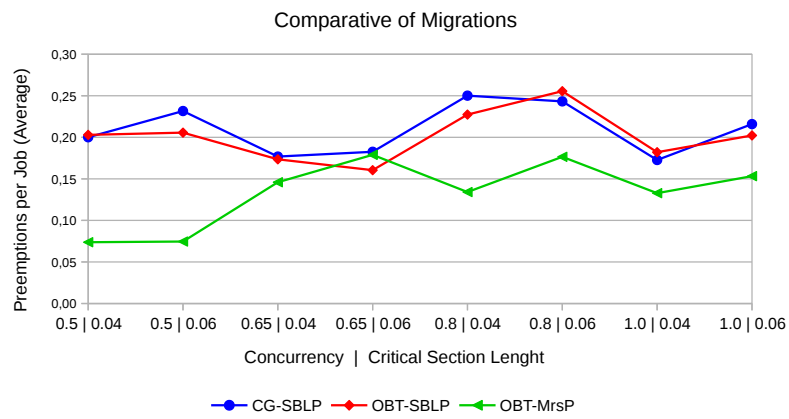


Figure 6.6 Comparative of the number of migrations per task for CG-SBLP, OBT-SBLP and OBT-MrsP

CONCLUSION

We have identified a specific research problem, namely the integration of RUN and MrsP so that system utilization can be improved, as compared to related work, and scheduling can be carried out efficiently in the context of resource sharing for multiprocessor real-time systems. As we have pointed out, MrsP has recently attracted researchers' attention due to its potentials. Likewise, RUN has been shown to be an efficient real-time scheduling algorithm. Further, the only attempt to implement resource sharing for RUN scheduled systems, namely SBLP, can be considered too restrictive for some application domains since such a solution is based on preventing preemptions within RUN servers while resources are blocked or requested by their tasks. By using MrsP, we intend to design a more flexible and efficient solution.

Integrating RUN and MrsP requires adapting their rules and proving it correct. We worked to preserve most characteristics of both RUN and MrsP, algorithms that have shown to perform very well by previous work. We kept modifications to their rules at a minimum. We defined the rules for this new proposed solution, based on the rules of MrsP and RUN and derived the properties and rules to enable both to adapt and prove their correctness.

After proving the feasibility of the integration between MrsP and RUN, we used scripts to perform experiments to measure how well our solution has achieved in terms of increasing processor demand and compare with the results obtained by the the SBLP protocol. Besides the comparison between the protocols, we also proposed a new heuristic namely OBT and observed the results obtained by OBT in relation to FG and CG heuristics. We observed that OBT presented better results. The MrsP protocol obtained slightly better results than SBLP, up to 93% of the increment reached, both under the OBT heuristic. A smaller increment achieved by the protocol indicates that its performance is better as the processor capacity reserve will be smaller. When comparing the results obtained by MrsP with those of SBLP with the other heuristics, the results were even better for MrsP, which obtained up to 79% and 12% of the increment obtained by CG-SBLP and FG-SBLP, respectively.

Next, we implemented the solution derived from the integration of MrsP with RUN in Litmus^{RT} environment and performed experiments for several sets of tasks generated according to the parameters defined in this work. We observed the overhead results obtained by each of the solutions, namely OBT-MrsP, OBT-SBLP and CG-SBLP. MrsP obtained up to 74% of the number of preemption and up to 35% of the number of migrations obtained by SBLP, which means a significant reduction in overhead. When comparing average times obtained in performing the basic task scheduling and resource sharing primitives, we observed that there were alternations in the advantages between the protocols. The *Tree* and *Release* primitives were significantly advantageous for MrsP, while *Context Switch*, *Lock*, *Unlock*, *Schedule* and *Release Latency* were eminent for SBLP, highlighting *Schedule* and *Release Latency*. When considering the total overhead achieved by the solutions, we observed that OBT-SBLP and CG-SBLP presented an overhead higher than that of MrsP by 3.24% and 8.48%, respectively.

This research work can motivate further developments. We did not address scenarios with nested resources. Removing this constraint is certainly necessary for broadening the application of the obtained results in this work. Further, as we have mentioned in Chapter 4, system schedulability may depend on how tasks are assigned to the RUN servers. Seeking for task assignment policies aiming at maximizing schedulability is a problem to be further investigated. Regarding implementation, the results of our work benefit from an efficient implementation of the RUN reduction tree structure. Since the tree is managed at any scheduling instant, all the scheduling decisions can be taken in a more efficient way if the tree implementation is optimized. In this work we did not go into such implementation aspects. Another interesting topic to be considered in the future is integrating MrsP with other global scheduling policies. QPS is a natural choice. Although more complex to enable this integration with QPS, it performs similarly to RUN. Since QPS can deal with sporadic tasks, having such an integration can be useful for enlarging the application domain of MrsP.

In summary, by addressing a relevant research topic, we believe that the solution that this work provided brings about both advances in the state of the art and starting points for further developments in the field.

BIBLIOGRAPHY

- ALVES, P. et al. *GDB: The GNU Project Debugger*. 2019. Disponível em: <<https://www.gnu.org/software/gdb/>>.
- ANDERSON, J.; SRINIVASAN, A. Early-release fair scheduling. *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, p. 35–43, 2000.
- BAKER, T. P. A stack-based resource allocation policy for realtime processes. *Proceedings - Real-Time Systems Symposium*, n. January 1991, p. 191–200, 1990. ISSN 10528725.
- BAKER, T. P. Stack-based scheduling of realtime processes. *Real-Time Systems*, v. 3, n. 1, p. 67–99, 1991. ISSN 09226443.
- BARUAH, S. K. et al. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, v. 15, n. 6, p. 600–625, 1996.
- BARUAH, S. K.; GEHRKE, J. E.; PLAXTON, G. C. Fast scheduling of periodic tasks on multiple resources. *Proceedings of 9th International Parallel Processing Symposium*, p. 280–288, 1995.
- BIONDI, A.; BUTTAZZO, G.; BERTOGNA, M. Schedulability Analysis of Hierarchical Real-Time Systems under Shared Resources. *Retis.Sssup.It*, v. 65, n. 5, p. 1–17, 2013. ISSN 0018-9340.
- BIONDI, A. et al. Optimal design for reservation servers under shared resources. *Proceedings - Euromicro Conference on Real-Time Systems*, n. Ecrts, p. 153–164, 2014. ISSN 10683070.
- BLOCK, A. et al. A flexible real-time locking protocol for multiprocessors. *Proceedings - 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2007*, n. Rtcsa, p. 47–56, 2007. ISSN 1533-2306.
- BONATO, L.; MEZZETTI, E.; VARDANEGA, T. Supporting Global Resource Sharing in RUN-scheduled Multiprocessor Systems Our goal. In: *International Conference on Real-Time Networks and Systems*. Versaille, France: ACM New York, NY, USA, 2014. p. 109–118. ISBN 9781450327275.
- BRANDENBURG, B. B. *Scheduling and locking in multiprocessor real-time operating systems*. 598 p. Tese (Doutorado) — University of North Carolina at Chapel Hill, 2011.

BRANDENBURG, B. B.; ANDERSON, J. H. A comparison of the M-PCP, D-PCP, and FMLP on LITMUSRT. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 5401 LNCS, p. 105–124, 2008. ISSN 03029743.

BRANDENBURG, B. B. et al. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, p. 342–353, 2008. ISSN 15453421.

BURNS, A.; WELLINGS, A. J. A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP. *Proceedings - Euromicro Conference on Real-Time Systems*, p. 282–291, 2013. ISSN 10683070.

CATELLANI, S. et al. Challenges in the implementation of MrsP. p. 1–22, 2015.

COMPAGNIN, D.; MEZZETTI, E.; VARDANEGA, T. Putting RUN into practice: Implementation and evaluation. *Proceedings - Euromicro Conference on Real-Time Systems*, p. 75–84, 2014. ISSN 10683070.

DAVIS, R. I.; BURNS, A. Resource sharing in hierarchical fixed priority pre-emptive systems. *Proceedings - Real-Time Systems Symposium*, p. 257–267, 2006. ISSN 10528725.

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, v. 43, n. 4, p. 1–44, 2011. ISSN 03600300. Disponível em: <<http://dl.acm.org/citation.cfm?doid=1978802.1978814>>.

ECMA, I. *The JSON Data Interchange Syntax*. 2019. Disponível em: <<https://www.ecma-international.org/publications/standards/Ecma-404.htm>>.

GAI, P.; LIPARI, G.; Di Natale, M. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. *Proceedings - Real-Time Systems Symposium*, p. 73–83, 2001. ISSN 10528725.

GARRIDO, J. et al. Evaluating msrp and mrsp with the multiprocessor ravenscar profile. In: BLIEBERGER, J.; BADER, M. (Ed.). *Reliable Software Technologies – Ada-Europe 2017*. Cham: Springer International Publishing, 2017. p. 3–17. ISBN 978-3-319-60588-3.

GARRIDO, J. et al. Supporting nested resources in mrsp. In: BLIEBERGER, J.; BADER, M. (Ed.). *Reliable Software Technologies – Ada-Europe 2017*. Cham: Springer International Publishing, 2017. p. 73–86. ISBN 978-3-319-60588-3.

GITLAB. *The DevOps Lifecycle with GitLab*. 2019. Disponível em: <<https://about.gitlab.com/>>.

HUANG, W. H.; YANG, M.; CHEN, J. J. Resource-Oriented Partitioned Scheduling in Multiprocessor Systems: How to Partition and How to Share? *Proceedings - Real-Time Systems Symposium*, p. 111–122, 2017. ISSN 10528725.

JOSEPH, M.; PANDYA, P. Finding response times in a real-time system. *The Computer Journal*, v. 29, n. 5, p. 390–395, 1986.

KOREN, G.; AMIR, A.; DAR, E. The Power of Migration in Multi-Processor Scheduling of Real-Time Systems. v. 30, n. 2, p. 226–235, 1998.

LEVIN, G. et al. DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. *2010 22nd Euromicro Conference on Real-Time Systems*, p. 3–13, 2010.

LIMA, G.; REGNIER, P.; MASSA, E. Practical Considerations in Optimal Multiprocessor Scheduling. In: TIAN, Y.; Levy Charles, D. (Ed.). *Handbook of Real-Time Computing*. 1. ed. [S.l.]: Springer Singapore, 2019. cap. 8.

LIU, C. L.; W. Layland, J. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment Scheduling Algorithms for Multiprogramming. *Journal of the Association for Computing Machinery*, v. 20, n. 1, p. 46–61, 1973. ISSN 0004-5411.

LIU, J. W.-S. *Real-Time systems*. [S.l.: s.n.], 2000. 610 p. ISBN 978-0-13-099651-0.

MASSA, E.; LIMA, G.; REGNIER, P. Revealing the secrets of RUN and QPS: new trends for optimal real-time multiprocessor scheduling. *Proceedings of the Brazilian Symposium on Computing Systems Engineering 2014*, p. 1–6, 2014.

MASSA, E. et al. Quasi-partitioned scheduling: optimality and adaptation in multiprocessor real-time systems. *Real-Time Systems*, Springer US, v. 52, n. 5, p. 566–597, 2016. ISSN 15731383.

MOK, A. H. L. *Fundamental Design Problems of Distributed Systems for the Hard-Real time Environment*. 1983.

MPI-SWS, Max Planck Institute for Software Systems. *LITMUS-RT Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*. 2018. Disponível em: <<http://www.litmus-rt.org>>.

NATALE, M. D.; SANGIOVANNI-VINCENNELLI, A. L. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proceedings of the IEEE*, v. 98, n. 4, p. 603–620, 2010. ISSN 00189219.

NELISSEN, G. et al. U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In: *Proceedings - Euromicro Conference on Real-Time Systems*. [S.l.: s.n.], 2012. p. 13–23. ISBN 9780769547398. ISSN 10683070.

QEMU. *QEMU*. 2019. Disponível em: <<https://www.qemu.org/>>.

RAJKUMAR, R. Real-time synchronization protocols for shared memory multiprocessors. *Proceedings., 10th International Conference on Distributed Computing Systems*, 1990.

RAJKUMAR, R.; SHA, L.; LEHOCZKY, J. P. Real-time synchronization protocols for multiprocessors. *Proceedings. Real-Time Systems Symposium*, p. 259–269, 1988. Disponível em: <<http://ieeexplore.ieee.org/document/51121/>>.

REGNIER, P. et al. RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: *Proceedings - 32th Real-Time Systems Symposium*. [S.l.: s.n.], 2011. p. 104–115. ISBN 9780769545912. ISSN 10528725.

SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, v. 39, n. 9, p. 1175–1185, 1990. ISSN 00189340.

TAKADA, H.; SAKAMURA, K. A Novel Approach to Multiprogrammed Multiprocessor Synchronization for Real-Time Kernels. *Proceedings Real-Time Systems Symposium*, p. 134–143, 1997. ISSN 10528725. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=641276>>.

WESSEL, J. *Using kgdb, kdb and the kernel debugger internals*. 2019. Disponível em: <<https://www.kernel.org/doc/html/v4.14/dev-tools/kgdb.html>>.

ZHAO, S. et al. New schedulability analysis for MrsP. *RTCSA 2017 - 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2017.