Robust Partitioning for Real-Time Multiprocessor Systems with Shared Resources
Frédéric Fauberteau, Serge Midonnet

To cite this version:
Frédéric Fauberteau, Serge Midonnet. Robust Partitioning for Real-Time Multiprocessor Systems with Shared Resources. RACS 2011, Nov 2011, Miami, United States. pp.71-76, 10.1145/2103380.2103394. hal-00668729

HAL Id: hal-00668729
https://hal.archives-ouvertes.fr/hal-00668729
Submitted on 10 Feb 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Robust Partitioning for Real-Time Multiprocessor Systems with Shared Resources

Frédéric Faubertea
Université Paris-Est
LIGM, UMR CNRS 8049
5, bd Descartes, Champs-sur-Marne
77454 Marne-la-Vallée CEDEX 2, FRANCE
frederic.fauberteau@univ-paris-est.fr

Serge Midonnet
Université Paris-Est
LIGM, UMR CNRS 8049
5, bd Descartes, Champs-sur-Marne
77454 Marne-la-Vallée CEDEX 2, FRANCE
serge.midonnet@univ-paris-est.fr

ABSTRACT
In this paper, we focus on the temporal robustness in the hard real-time multiprocessor systems. This robustness is the capacity to tolerate faults in such a way that no deadlines are missed. A model of sporadic and dependent tasks is considered. Our contribution is to propose a partitioning algorithm which assigns the tasks to processors in order to maximize the robustness of the system to Worst Case Execution Time (WCET) overruns of faults or Minimum Inter-arrival Time (MIT) violations.

Keywords
Real-time Scheduling, Multiprocessor Systems, Robustness, Partitioning Algorithms, Shared Resources.

1. INTRODUCTION
We consider that an application consists of a set of tasks where each task is recurring jobs. One of the main problems in real-time scheduling is to guarantee that no task misses its deadline. Uniprocessor scheduling attempts to solve the priority problem: when, and in what order should each job execute. In multiprocessor scheduling, to the priority problem is added the allocation problem: on which processor a job should execute. Multiprocessor scheduling algorithms can be classified according theses two problems.

We focus on the static-priority approach (for the priority problem) for which each task has a single static priority applied to all of its jobs and on the partitioned approach (for the allocation problem) for which each task is allocated on a processor and no migration is allowed. The partitioned scheduling approach consists in partitioning a taskset among processors. Each subset of task is scheduled on in own processor independently of others. The partitioning problem can be seen as BIN-PACKING problem which is NP-hard in the strong sense. Hence no optimal algorithm exists unless \( P = NP \). But several approaches are inspired by techniques for BIN-PACKING and enable to solve partitioning problem.

1.1 Related work
One of the most prevalent strategies to provide fault tolerance is replication. Qin and Hong has proposed an algorithm which generates distributed static schedules to handle processor failures [14]. Emberson and Bate have proposed a task allocation algorithm which takes into account static redundancy to dispatch the tasks among the processors [10]. Our paper is concerned by a strategy of margin on the temporal constraints of the tasks. Instead to replicate tasks to deal with faults, we give them the possibility to continue their execution beyond their WCET.

Echtle and Eusgeld used a genetic algorithm to find fault-tolerant partitions [9]. However, their work does not focus on real-time systems. Tindell, Burns and Wellings have proposed a partitioning algorithm based on simulated annealing to allocate tasks on a multiprocessor distributed systems [17]. Di Natale and Stankovic have also used this technique to deal with systems of tasks with jitter [8]. These two works are aimed at real-time systems but the fault-tolerance is not taken into account.

Oh and Son discuss the need to consider schedulability and fault-tolerance simultaneously [13]. They prove that finding a schedule to handle a single processor failure is a NP-hard problem. Their model does not include dependency constraints.

In this paper, we propose a partitioning algorithm which is fault-tolerant in such a way that it dispatches the tasks in order to maximize the margin (on WCET overruns or on frequency increasing).

1.2 Terminology
We consider a set \( \tau = \{\tau_1, \tau_2, \ldots, \tau_n\} \) of \( n \) sporadic real-time tasks. A sporadic task is a recurring task for which only a lower bound on inter-arrival time of the jobs is known. Each task \( \tau_i \) is characterized by a minimum inter-arrival time \( T_i \) (also denoted period), a WCET \( C_i \), and a relative deadline \( D_i \). The considered tasks are tasks with constrained deadlines (i.e. \( D_i \leq T_i \)). The processor utilization of \( \tau_i \) is denoted \( U_i \) and is defined as \( U_i = C_i / T_i \). This application runs on a platform \( \Pi = \{\tau_1, \tau_2, \ldots, \tau_m\} \) of \( m \) identical processors (homogeneous case). A job \( J_i \) is characterized by its release time \( r_i \), its execution time \( e_i \) (observed at run time) and its absolute deadline \( D_i \). The priorities of the tasks are taken in an increasing order (i.e. \( \tau_i \) is higher than the priority of \( \tau_{i+1} \)). The set of tasks with priorities lower (resp. higher) than the priority of \( \tau_j \) is denoted \( U(\tau_j) \). We consider a static-priority scheduling on each processor. The priorities of the tasks are taken in an increasing order (i.e. the priority of \( \tau_{i+1} \) is higher than the priority of \( \tau_i \)). The set of tasks with a priority lower (resp. higher) than the priority of \( \tau_i \) is denoted \( hp(\tau_i) \) (resp. \( lp(\tau_i) \)). In this paper, we denote \( R_i \) the response time of a task \( \tau_i \) and \( B_i \) the blocking factor incurred by the task \( \tau_i \). We also denote \( A^c_i \) (resp. \( A^f_i \)) the value of WCET (resp. frequency) margin of \( \tau_i \).

1.3 Organization

The rest of this paper is organized as follows. In Section 2 we present several synchronization protocols for multiprocessor systems. In Section 3, we present the concept of robustness to WCET overruns faults and MIT violations. We describe how to compute the margin of the tasks on their WCET and their period. In Section 4, we propose a partitioning algorithm which maximizes the robustness in terms of WCET overruns faults. In Section 5, we compare by simulation our approach with existing ones. Finally, we conclude and give some perspectives in Section 6.

2. SYNCHRONIZATION PROTOCOLS

In real-time systems, the synchronization protocols are used to avoid the unbounded priority inversions which could occur with a simple lock mechanism. In the multiprocessor context, three main protocols are often cited in the literature. We present them in the following sections.

2.1 MPCP

Multiprocessor Priority Ceiling Protocol (MPCP) is an extension to the multiprocessor case of the Priority Ceiling Protocol (PCP) proposed by Sha, Rajkumar and Lehoczky in [16]. MPCP has been proposed by Rajkumar in [15].

With PCP, a priority ceiling \( \pi(R_k) \) is associated with each shared resource \( R_k \). \( \pi(R_k) \) is defined as the priority of the highest priority job which can lock \( R_k \). At time \( t \), the system priority ceiling \( \pi(t) \) is given by the highest priority ceiling of the locked resources at \( t \). A job \( J_i \) can lock a resource \( R_k \) at time \( t \) only if the priority of \( J_i \) is strictly greater than the system priority ceiling. Otherwise, the job which blocks \( J_i \) inherits the priority of \( J_i \).

With MPCP, two types of resources are distinguished: local and global resources. The local resources are only shared by jobs on the same processor whilst global ones are shared by jobs which can be assigned on several processors. When a job tries to lock a local resource, the behavior of MPCP is the same as PCP. For the case of global resources, a set of priorities higher than the highest priority ceiling is used. Then, accesses to global resources are made uppermost to limit indirect blocking.

2.2 MSRP

Multiprocessor Stack Resource Protocol (MSRP) is an extension to the multiprocessor case of the Stack Resource Protocol (SRP) proposed by Baker in [1]. MSRP has been proposed by Gai et al. in [11].

With SRP, each job \( J_i \) is characterized by a preemption level \( \rho(J_i) \). \( J_i \) is allowed to preempt \( J_j \) only if \( \rho(J_i) > \rho(J_j) \). Each resource \( R_k \) has a preemption ceiling \( \rho(R_k) \) defined as the highest preemption level of the jobs which can lock \( R_k \). At time \( t \), the system preemption ceiling is given by the highest preemption ceiling among resources currently locked. The active jobs are stored in a stack in decreasing order of their priority. \( J_i \) is allowed to preempt \( J_j \) if \( J_i \) is the highest active job and if its preemption level is strictly greater than the system preemption ceiling.

With MSRP at time \( t \), a preemption ceiling \( \rho(t, \tau_j) \) is fixed for each processor \( \tau_j \). For each global resource \( R_{Gk} \), each processor \( \tau_j \) defines a preemption ceiling \( \rho(R_{Gk}) \) higher than the highest preemption ceiling on \( \tau_j \). For local resources, the MSRP behavior is the same as SRP. When a job attempts to lock a global resource \( R_{Gk} \) on \( \tau_j \), \( \rho(t, \tau_j) \) is raised to \( \rho(R_{Gk}) \) making \( J_i \) non-preemptive.

2.3 FMLP

Flexible Multiprocessor Locking Protocol (FMLP) is a synchronization protocol exclusively developed for the multiprocessor systems. It has been proposed by Block et al. in [3]. Initially, FMLP has been proposed to synchronize shared resources in both partitioned and global EDF scheduling. It has been extended to deal with the partitioned Fixed-Task-Priority scheduling approach of Brandenburg and Anderson in [5].

FMLP takes advantage of both busy-wait and by suspension approaches. To do that, two types of resource are considered: short (\( R_{s_k} \)) and long (\( R_{l_k} \)) resources. When a job \( J_i \) attempts to lock an already locked short resource \( R_{s_k} \), it busy waits non-preemptively for \( R_{s_k} \). This behavior tends to reduce response-time of jobs when they use shared resources for short time.

3. TASK MARGIN

We focus on temporal robustness which consists in tolerance to temporal faults in systems. The schedulability analysis of fault-tolerant systems has been studied by Burns, Davis and Punnekkat [6]. More recently, Bougueroua, George and Midonnet has proposed an approach to compute a margin of tolerance on the WCET [4]. We present it in Section 3.1 and we propose its extension to the domain of periods in the Section 3.2.

In this section, we use the notation \( C^*_i \) which is defined by \( C^*_i = C_i + B_i \).

3.1 WCET margin
The constraints in a hard real-time system are defined such that no deadlines of any task are missed. Moreover, the WCET of a task is estimated or computed in order to ensure that the task never runs for a duration longer than its WCET. If a task commits a WCET overrun fault, the system may fail unless the task has enough WCET margin.

**Definition 1** (WCET margin). The WCET margin $A_i^C$ of a task $\tau_i$ is the additional execution time which can be added to the WCET $C_i$ of $\tau_i$ in such a way that no task of $\tau$ misses its deadline.

A value $A_i^C$ is a correct WCET margin value for $\tau_i$ on $\pi_j$ if it verifies the inequalities:

$$U(\pi_j) + \frac{A_i^C}{T_i} \leq 1$$  \hspace{1cm} (1)

$$R_i^{k+1} = C_i^* + \sum_{\tau_h \in hp(\tau_i)} \left[ \frac{R_h^k}{T_h} \right] C_h \leq D_i$$  \hspace{1cm} (2)

$$\forall \tau_i \in lp(\tau_i),$$

$$R_i^{k+1} = C_i^* + \sum_{\tau_h \in hp(\tau_i)} \left[ \frac{R_h^k}{T_h} \right] C_h \leq D_i$$  \hspace{1cm} (3)

The maximum value of WCET margin for $\tau_i$ can be computed by binary search over the interval $[0, \min(D_i - C_i, (1 - U(\pi_j))/T_i)]$.

### 3.2 Frequency margin

The sporadic (and periodic) tasks are characterized by a minimum inter-arrival time. If a job is activated as earlier than planned (e.g., faulty external component), the system may fail unless the task has enough frequency margin.

**Definition 2** (Frequency margin). The frequency margin $A_i^f$ of a task $\tau_i$ is a period of time which can be subtracted to the period $T_i$ of $\tau_i$ in such a way that no task of $\tau$ misses its deadline.

A value $A_i^f$ is a correct frequency margin value for $\tau_i$ on $\pi_j$ if it verifies the inequalities:

$$U(\pi_j) - U_i + \frac{C_i}{T_i} - A_i^f \leq 1$$  \hspace{1cm} (4)

$$R_i^{k+1} = C_i^* + \sum_{\tau_h \in hp(\tau_i)} \left[ \frac{R_h^k}{T_h} \right] C_h \leq T_i - A_i^f$$  \hspace{1cm} (5)

$$\forall \tau_i \in lp(\tau_i),$$

$$R_i^{k+1} = C_i^* + \sum_{\tau_h \in hp(\tau_i)} \left[ \frac{R_h^k}{T_h} \right] C_h \leq D_i$$  \hspace{1cm} (6)

The maximum value of frequency margin for $\tau_i$ can be computed by binary search over the interval $[0, T_i - 1]$.

### 3.3 Margin of dependent tasks

For the case of independent tasks, if a fault occurs during the execution of a job $J_i$, $J_i$ can continue its execution for $A_i^C$. If the tasks share resources, the WCET margin of $J_i$ in a critical section can be less than $A_i^C$. In Figure 1(a), we represent the schedule of 4 jobs characterized by the parameters given in Table 1. The synchronization protocol is FMLP and the resources are considered as long. We consider each job executing for its WCET ($e_i = C_i$). If a fault occurs at the end of the execution of $J_4$, it can continue until time 16 in such a way that no job misses its deadline. Then the value of WCET margin $A_i^C$ is 5 units of time if it is used at the end of the execution of $J_3$.

In Figure 1(b), we represent an execution overrun in the critical section associated to $R_{G1}$. $J_3$ can increase its budget of time by just 1 unit of time. We notice that the WCET margin of a job in a critical section is bounded by the minimum margin of all the jobs which share the resource guarded by this section.

### 4. ROBUST PARTITIONING

Simulated annealing is a generic algorithm which has been firstly proposed by Kirkpatrick, Gelatt and Vecchi [12] for the optimization problems. The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of...
its crystals and to reduce their defects. In this work, we apply the simulated annealing technique to build an algorithm which finds a feasible partition of a set of tasks where the robustness to the WCET overruns faults is maximized. We describe this algorithm, Robust Simulated Annealing (RPSA) in Algorithm 1. The initialization is made as follows. At line 1, the function random_partition() build a partition $P$ by allocating each of the $n$ tasks on one of the $m$ processors randomly. This partition may be unfeasible. At line 2, an initial temperature is computed such as 99% of the partitions are kept even if they do not improve the solution. By cooling the system (decreasing the temperature), the unsatisfying solutions will be eliminated. At line 3, we initialize the max_try value to an integer which depends both of the number of tasks and of the number of processors. The loop at line 4-21 performs max_try iterations of the loop at line 6-19. After each iteration, the system is cooled by dividing the temperature by 2. The main part of the algorithm is the loop at line 6-19. At line 7, the function compute_energy() computes the energy of the partition $P$. This function is more detailed later in Algorithm 2. At line 8, a partition $P_n$ which is the neighbor of the partition $P$ is computed. This partition $P_n$ is obtained either by randomly swapping two tasks of $P$ or by randomly moving a task of $P$ from a processor to another. The energy of this new partition is computed at line 9. If the value $E_n$ of the energy of $P_n$ is less than the value $E_p$ of the energy of $P$ then $P$ is replaced by $P_n$. Otherwise, a random number is drawn between 0 and 1. The more the temperature $temp$ high is, the more the probability that the value $e^x (x = \frac{E_p - E_n}{temp})$ is greater than the random number is. If $e^x > \text{random}(0,1)$ then $P$ is also replaced by $P_n$ else $P_n$ is discarded. This behavior avoids that the energy converge to a local minimum. We of energy is computed as follows. For each processor $\pi$, if the processor is empty then the value of energy is increased by 1. This behavior increase the probability that the tasks are well distributed among the processor and no processors stay empty. If the set of tasks allocated on this processor is unschedulable then the value of energy is also increased by 1 to eliminate the unfeasible partitions. For each processor $\pi$, where the set of tasks is schedulable, the sum of margin on the execution duration values of each task is stored in the array margin at index $j$. At the end of the loop at line 3-10, the value of energy is increased by the sum of all the values stored in the array margin. Consequently the more the value of margin of each task great is, the less the value of energy is.

![Algorithm 1: RPSA](image)

```plaintext
Algorithm 1: RPSA
1. $P = \text{random\_partition}(n, m)$;
2. $temp = \frac{n}{m(0.99^m)}$;
3. $\text{max\_try} = n \cdot m$;
4. while $temp > 10^{-7}$ do
5.     $k = 0$;
6.     while $k \neq \text{max\_try}$ do
7.         $E_p = \text{compute\_energy}(P)$;
8.         $P_n = \text{neighbor}(P)$;
9.         $E_n = \text{compute\_energy}(P_n)$;
10.        if $E_n < E_p$ then
11.            $P = P_n$;
12.        else
13.            $x = \frac{E_p - E_n}{temp}$;
14.            if $e^x \geq \text{random}(0,1)$ then
15.                $P = P_n$;
16.        end
17.    $k = k + 1$;
18. end
19. $temp = \frac{temp}{2}$;
```

now describe the function compute_energy(). The aim of this function is to compute a value of energy for a partition such that the more the minimum value of margin for the system is great, the less the value of energy is. The value

5. SIMULATION

5.1 Methodology

We implemented a simulator of real-time systems which provides several partitioning algorithms. Among others, it implements both RPSA and SPA algorithms.

We also implemented a taskset generator which randomly makes sets of tasks with shared resources. The tasks generation process is based on the UUniFast-Discard algorithm proposed by Davis and Burns [7]. This algorithm is the extension to the multiprocessor case of the UUniFast algorithm proposed by Bini and Buttazzo [2].

For each value of processor utilization in [0.025, 0.05, ..., 0.975], we randomly generated 1,000 tasksets with constrained deadlines. Each simulation consists in the partitioning of a taskset of 16 tasks on 4 processors. Each task has a random number of critical section in {0, 1, 2}. There are half as many resources as critical sections.

SPA has been initially proposed in order to use MPCP as synchronization protocol. In the context of our simulations, we have modified it to be based on FMLP.

5.2 Results

In Figure 2, we represent the comparison between RPSA and SPA in terms of schedulability and minimum margin. In Figures 2(a) and 2(c), RPSA has been implemented to maximize the WCET margin. The compute_energy() function of RPSA uses the Equations (1), (2) and (3) described
We show that RPSA outperforms on average SPA in terms of schedulability. Our algorithm takes advantage of the simulated annealing technique. Then it finds solutions which cannot be found by the BF algorithm (the one on which SPA is based).

In terms of robustness, RPSA produces partitions for which the margin of the tasks is increased compared to SPA.

6. CONCLUSION

We have proposed the partitioning algorithm RPSA based on the simulated annealing technique. We have considered a model of sporadic tasks with shared resources. The synchronization of data is performed by the FMLP protocol. Our algorithm allocates the tasks on the processors in order to maximize the robustness to the WCET overruns faults and MIT violations. We have implemented our algorithm in our simulator of real-time systems and we compare it with the SPA heuristic. We have shown that our solution outperforms the heuristic approach in terms of both schedulability and robustness.

7. REFERENCES


