HAL open science

Pfair scheduling improvement to reduce interprocessor migrations

Dalia Aoun, Anne-Marie Déplanche, Yvon Trinquet

To cite this version:

Dalia Aoun, Anne-Marie Déplanche, Yvon Trinquet. Pfair scheduling improvement to reduce interprocessor migrations. 16th International Conference on Real-Time and Network Systems (RTNS 2008), Isabelle Puaut, Oct 2008, Rennes, France. inria-00336513

HAL Id: inria-00336513

https://inria.hal.science/inria-00336513v1

Submitted on 4 Nov 2008

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.

Pfair scheduling improvement to reduce interprocessor migrations

Dalia Aoun, Anne-Marie Déplanche, Yvon Trinquet
Université de Nantes / IRCCyN
Nantes, France
{Dalia.Aoun, Anne-Marie.Deplanche, Yvon.Trinquet}@irccyn.ec-nantes.fr

Abstract

Proportionate-fair (Pfair) scheduling is a particulary promising global scheduling technique for multiprocessor systems. Actually three Pfair scheduling algorithms have been proved optimal for scheduling periodic, sporadic and rate-based tasks on a real-time multiprocessor. However, task migration is unrestricted under Pfair scheduling. In the worst case, a task may be migrated each time it is scheduled. To correct this problem, we propose to complement initial Pfair scheduling algorithm by some heuristics that strive to minimize the total number of migrations. Experimental simulations are presented to evaluate and compare the proposed heuristics and we show that the number of migrations can be substantially reduced by adding these simple rules. [1]

1. Introduction

Multiprocessor scheduling techniques in real-time systems fall into two general categories: partitioning and global scheduling. Under partitioning, each task is assigned to a particular processor and is only scheduled on that processor. Moreover, each processor schedules its assigned tasks independently from a local ready queue. In contrast, under global scheduling, all ready tasks are stored in a single queue with a single system-wide priority space: the highest priority task is selected to execute whenever the scheduler is invoked, regardless of which processor is being scheduled. In the global scheme, migration is allowed.

Partitioning. Presently, "partitioning" is the favored approach. This is mainly due to the fact that it has proved to be efficient and reasonably effective when using well-understood uniprocessor scheduling algorithms, such as the earliest-deadline-first (EDF) and the rate-monotonic (RM) algorithms [16]. However, partitioning has some drawbacks. Regardless of the scheduling algorithm used, partitioning is sub-optimal when scheduling periodic tasks. A well-known example of this is a system with two processors and three synchronous tasks, each with an execution time of 2 units, a period of 3 units and an implicit deadline. Completing each task before its next release is impossible without migration. Hence, this system is not schedulable under the partitioning approach. But partitioning is problematic too in dynamic systems in which tasks may join and leave: re-partitioning of the entire system will likely result in unacceptable overheads. Moreover, since the assignment of tasks to processors is a bin-packing problem proved NP-hard in the strong sense, online task assignments are done using heuristics, which may be unable to schedule offline-schedulable task systems.

Global scheduling. Since the alternative "global scheduling" does not reduce the multiprocessor scheduling problem to a uniprocessor scheduling problem as partitioning does, and since tasks in the global scheme are allowed to migrate, this gives rise to disadvantages that complicated its study and design. Moreover it is well known that the Dhall and Liu examples [12] went against it: a task set may have utilization arbitrarily close to 1 and still not be schedulable on m processors using RM or EDF scheduling. Thus, this research field is quite new. Only recently, progress has been made in understanding global multiprocessor scheduling: it is the mixing of two extreme kinds of tasks, "heavy" ones (with a high ratio of computation time to deadline) and "light" ones (with a low ratio), that causes a problem. This observation has been generalized. Hybrid scheduling algorithms that give higher priority to heavy tasks have been proposed, such as the most famous EDF-US [18], RM-US [7], EDF(k), or PriD [13]. At the same time, a large number of interesting and important results have been obtained on schedulability analysis that make global scheduling quite competitive [8].

Pfair scheduling. A major step forward in the evolution of processor scheduling techniques was achieved in the work of Baruah et al. on fairness [9, 10]. *Proportionate-fair* scheduling (Pfair) is presently the only known optimal method for scheduling recurrent real-time tasks on a multi-

[1] Work supported by ANR grants ANR-06-ARFU-003.

processor system in polynomial time. Under Pfair scheduling, tasks are explicitly required to make progress at steady rates. Currently, three optimal Pfair scheduling algorithms are known: PF [10], PD [11], and the most recently developed and most efficient PD^2 [2]. Besides its ability to schedule any feasible periodic, sporadic or rate-based task system [4, 5] and to handle dynamic systems, Pfair scheduling is often dismissed as an impractical approach due to preemptions and migrations which both tend to occur frequently. This tendency limits the effectiveness of the first-level caches and can lead to increased execution times due to cache misses. Such resulting overheads may be particularly present in loosely-coupled processor architectures but should be significantly smaller in tightly-coupled multicore ones where one or more levels of caches are shared by the different cores. However, in [19], Srinavasan et al. have investigated how preemption and migration overheads affect schedulability under the PD^2 algorithm, using the EDF-FF partitioning scheme as a basis for comparison. Their experimental results show that PD^2 performs competitively

Pfair related work. In [14], Holman and Anderson showed how Pfair scheduling actually promotes bus contention resulting from the simultaneous scheduling of all processors in symmetric multiprocessors, and then proposed an alternative scheduling model, the *staggered* model, that strives to avoid it by more evenly distributing bus traffic over time. Furthermore, they developed and experimentally evaluated an efficient scheduling algorithm to support this model that also produces less scheduling overhead than current Pfair algoritms. In another respect, since the migration assumption underlying Pfair scheduling may be problematic for "fixed" tasks that need to execute on specific processors, Moir and Ramamurthy proposed the use of *supertasks* in [17]. Each supertask represents a set of *component* tasks, which are scheduled as a single entity. Whenever a supertask is Pfair-scheduled, one of its component tasks is selected to execute according to an internal scheduling algorithm. However, as stated in [1], there is no known feasibility condition for systems in which periodic tasks are scheduled within Pfair-scheduled supertasks.

To the best of our knowledge, in term of improvement of Pfair scheduling for real-time periodic tasks, there is no recent work that studies the reduction of the number of migrations. Therefore, we have studied how to adapt the Pfair strategy with this goal. So we have proposed heuristics to assign tasks to processors and have developed a Pfair simulator to evaluate their performances.

The remainder of this paper is organized as follows. In section 2, we define the Pfair scheduling and its extensions. Then in section 3 we present our processor allocation heuristics and then our experimentation and results in section 4. We conclude in section 5.

2. Pfair scheduling

2.1. Definitions

Consider a collection of synchronous and periodic tasks to be executed on a system of multiple processors. Each task τ_i is characterized by a period T_i , and an execution cost C_i. Implicit-deadline systems are considered here, i.e. at every T_i time units, a new invocation or *job* of τ_i is released and must complete before the beginning of the next job of τ_i.

Under Pfair scheduling, processor time is allocated in discrete time units, or *quanta*, and *slot* t refers to the time interval $[t, t+1)$ (where t is a nonnegative integer). A task may be allocated over time on different processors, but not within the same slot, i.e. migration is allowed but parallelism is not. The sequence of allocation decisions over time defines a *schedule*. It can be formally defined as a mapping $S : \tau \times N \rightarrow \{0, 1\}$, where τ is the set of n tasks to be scheduled and N is the set of nonnegative integers. $S(\tau_i, t) = 1$ iff τ_i is scheduled in slot t. In any m-processor schedule, $\sum_{\tau_i \in \tau} S(\tau_i, t) \leq m$ for all t.

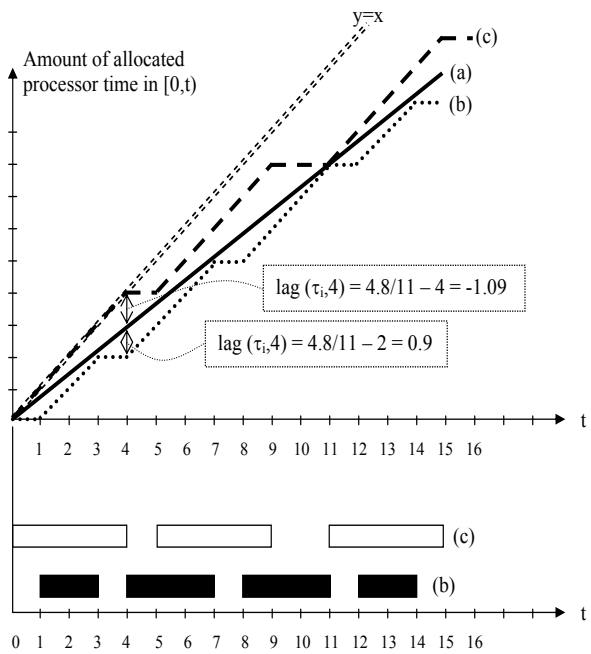

Figure 1. (a) the ideal fluid schedule of a task τ_i with weight $8/11$; it corresponds to the function $wt(\tau_i) \cdot t$ - (b) a Pfair schedule of τ_i - (c) a non-Pfair schedule of τ_i.

Following Baruah et al. [3], we refer to the ratio of C_i/T_i as the weight of task τ_i , denoted by $wt(\tau_i)$. Each task's weight is assumed strictly less than one (a task with

weight one would require a dedicated processor, and thus is quite easy scheduled). In an ideal fluid schedule, $wt(\tau_i)$ processor time would be allocated to each task τ_i in each slot. However, such idealized sharing is not possible in a quantum-based schedule. Deviation from this fluid schedule is captured by the concept of *lag*. It measures the difference between the number of resource allocations that a task "should" have received in $[0, t)$ and the number that it actually received. Formally, the lag of τ_i at time t $(t > 0)$ is given by $lag(\tau_i, t) = wt(\tau_i) \cdot t - \sum_{u=0}^{t-1} S(\tau_i, u)$. A schedule is Pfair iff:

$$-1 < lag(\tau_i, t) < 1 \; \forall \tau_i \in \tau, \; t \in N^* \tag{1}$$

Informally, the allocation error associated with each task must always be less than one quantum.

We propose to illustrate these notions by drawing the amount of processor time allocated to τ_i in $[0, t)$ with the execution sequences resulting from a Pfair scheduling and a non-Pfair one on a single processor, as shown by figure 1.

2.2. Windows

The Pfair lag bounds given in (1) have the effect of breaking each task τ_i into an infinite sequence of quantum-length subtasks. The kth subtask $(k \geq 1)$ is denoted τ_i^k. Moreover, (1) constrains each subtask τ_i^k to execute in an associated window denoted $w(\tau_i^k)$. It extends from τ_i^k's pseudo-release, denoted $r(\tau_i^k)$, to its pseudo-deadline, denoted $d(\tau_i^k)$ (the prefix "pseudo" will be omitted for conciseness). These dates are formally defined as follows:

$$r(\tau_i^k) = \left\lfloor \frac{k-1}{wt(\tau_i)} \right\rfloor \tag{2}$$

$$d(\tau_i^k) = \left\lceil \frac{k}{wt(\tau_i)} \right\rceil \tag{3}$$

As an example, figure 2 depicts the first windows of the task used in figure 1. Since its weight is $8/11$, each job of this task contains eight subtasks. Each subtask must be scheduled within its window in order to satisfy the Pfair property. In [6], several properties about subtask windows are stated. The main ones are that consecutive windows of a task are either disjoint or overlap by one slot; and either all windows of a task are of the same length, or they are of two different lengths.

2.3. Feasibility

By means of a network-flow construction, Baruah et al. [11] showed that a Pfair schedule on m processors exists for τ iff:

$$\sum_{\tau_i \in \tau} C_i / T_i \leq m \tag{4}$$

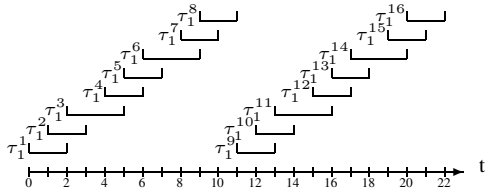

Figure 2. The Pfair windows of the first two jobs of τ_1 with $wt(\tau_1) = 8/11$

2.4. Pfair algorithms

At present, three Pfair scheduling algorithms have been proved optimal for scheduling synchronous, periodic, and deadline-constrained tasks on an arbitrary number of processors: PF [10], PD [11], and PD^2 [2]. All of them sort subtasks on an earliest-pseudo-deadline first basis and then use the *successor bit* parameter for breaking ties. The successor bit $b(\tau_i^k)$ of the subtask τ_i^k is defined as follows:

$$b(\tau_i^k) = d(\tau_i^k) - r(\tau_i^{k+1}) = \left\lceil \frac{k}{wt(\tau_i)} \right\rceil - \left\lfloor \frac{k}{wt(\tau_i)} \right\rfloor \tag{5}$$

Informally, it is equal to the number of slots by which τ_i^k's window overlaps τ_i^{k+1}'s window. For example, in figure 2, $b(\tau_1^k) = 1$ for $1 \leq k \leq 7$ and $b(\tau_1^8) = 0$.

Thus, the two following rules are successively used by all these Pfair scheduling algorithms for prioritizing subtasks. At time t, if subtasks τ_i^k and τ_p^q are both ready to execute, then τ_i^k has a higher priority than τ_p^q (denoted $\tau_i^k \succ \tau_p^q$) if:

(i) $d(\tau_i^k) < d(\tau_p^q)$

(ii) $d(\tau_i^k) = d(\tau_p^q)$ *and* $b(\tau_i^k) > b(\tau_p^q)$

The intuition behind this second rule is to favor a subtask with $b(\tau_i^k) = 1$ and thus, executing τ_i^k early prevents it from being scheduled in its last slot and leaves more slots available for τ_i^{k+1}.

The Pfair algorithms differ in their third tie-breaking rule that we briefly describe hereafter.

- Under the PF algorithm, it is given by:

  (iii) $d(\tau_i^k) = d(\tau_p^q)$, $b(\tau_i^k) = b(\tau_p^q) = 1$ et $\tau_i^{k+1} \succ \tau_p^{q+1}$

- In PD algorithm, the third rule is in fact composed of three elementary ones which involve simple calculations but have been shown useless regarding PD^2.

- The rule (iii) of the PD^2 priority definition is based on comparing *group deadlines* of competing subtasks. It

is needed in systems containing tasks with windows of length two, i.e. tasks τ_i such that $1/2 \leq wt(\tau_i) < 1$. The group deadline of a subtask is the earliest time by which the cascade that forces the other subtasks in this sequence to be scheduled in their last slot must end. Formally for τ_i^k, it is the earliest time t, where $t \geq d(\tau_i^k)$, such that either $(t = d(\tau_i^q) \wedge b(\tau_i^q) = 0)$ or $(t + 1 = d(\tau_i^q) \wedge |w(\tau_i^q)| = 3)$ for some subtask τ_i^q. PD^2 favors subtasks with later group deadlines because not scheduling them can lead to longer cascades. Up to now, it is the most efficient of the three Pfair scheduling algorithms.

For all these algorithms, if neither subtask has priority over the other, then the tie is broken arbitrarily. At each decision instant, once the ready subtask queue is sorted, the Pfair scheduler assigns the m (at most) highest priority subtasks to the processors for the next slot, in an arbitrary way.

The pseudo-code of the Pfair scheduler is given in algorithm 1. The list of ready subtasks at time t is designated by L, i.e $L = \{\tau_i^k / r(\tau_i^k) \leq t \; and \; \tau_i^k \; not \; executed \; yet\}$. L^+ represents the ordered list of the m (at most) highest priority subtasks. The elements of L^+ are subtasks that must be executed at time t or, in other words, subtasks that must be assigned to the m processors.

1 sort L following the Pfair tie-break rules;
2 build L^+ by extracting the m (at most) first elements of L;
3 assign the subtasks of L^+ to the m processors;

Algorithm 1: The decision algorithm of a Pfair scheduler

2.5. Pfair extensions

Pfair scheduling algorithms are not necessarily work-conserving, i.e. processors may idle unnecessarily. In [2, 4], Anderson and Srinavasan proposed a work-conserving variant called *early-release fair* (ERfair) scheduling. Under ERfair scheduling, if two subtasks are part of the same job, then the second subtask becomes eligible for execution as soon as the first completes. In other words, a subtask may be released "early", i.e. before the beginning of its Pfair window. If such early releases are allowed, then it is not required that the negative lag constraint of (1) holds. ER-PD^2 algorithm is obtained from the related PD^2 algorithm for Pfair systems and it can be used to efficiently schedule any task system whose total utilization is at most the number of available processors. ERfair is the preferred choice for applications with average job response time requirements.

In other work [3], the same authors extended the model to also allow subtask to be released "late", i.e. there may be separation between consecutive subtasks of the same task. The resulting model, called the *intra-sporadic* (IS) model, generalizes the well-known sporadic model (which allows separation between consecutive jobs of the same task). In [3], an IS task system τ is proved to be feasible on m processors if and only if equation (4) holds.

Finally, *generalised intra-sporadic* fairness (GISfairness) extends ISfairness by allowing subtasks to be omitted. In [3], PD^2 was shown optimal for scheduling generalised intra-sporadic tasks on m processors.

2.6. Motivation of this work

The Pfair scheduling algorithms are a way of optimally and efficiently schedule periodic tasks on multiprocessor system in polynomial time. Since tasks are scheduled at a constant rate, Pfair's optimality is achieved at the cost of more frequent context switching. Notice that the tie-break rules of a Pfair schedule determine at time t the tasks to schedule and do not indicate how to assign these tasks on the different processors. This imperfection may increase the degradation of Pfair algorithms performance and can lead to a degradation of the *processor affinity* (i.e. the tendency of a task to execute faster when scheduling several times on the same processor).

Thus, Pfair scheduling can produce an important number of preemptions and migrations that increases scheduling overheads and negatively impact cache performance. To the best of our knowledge, very few (or even none) work has been concerned with decreasing those migrations that result from these scheduling techniques and with similar assumptions on the task model. For example, actually in [15], the goal is to find a schedule that minimizes job migrations between processors while guaranteeing a fair schedule. But this work addresses a quite different model composed of persistent tasks all having the same resource requirement, and relies on a generalized notion of fairness. Thus we propose to improve the Pfair scheduling techniques by adding a helpful procedure that organizes the allocation of subtasks to different processors and that consequently minimizes the total number of context-switching overheads (including or not task migration). We present in what follows heuristics for this by simply specifying judicious rules for the allocation function of a Pfair scheduler (line 3 in algorithm 1).

3. Processor allocation heuristics

In this section, we propose heuristics for minimizing the total number of migrations while referring to the function that allocates the m highest priority subtasks to the processors given in algorithm 1 line 3. We give in this section a detailed description of each proposed heuristic.

3.1. Heuristic 1

The first heuristic is a simplistic one; it assigns the tasks to the processors according to their order in L^+. We present its pseudo-code in heuristic 1.

```
for q ← 1 to |L+| do
    assign L+[q] to P_q;
    /* we consider L+[q] as the q^th
       element of L+ and P_q the q^th
       processor                      */
end
```

Heuristic 1: H_1.

3.2. Heuristic 2

The intuition behind this heuristic is to keep the scheduling of a task on the same processor as much as possible, without preemption. In other words, we avoid interleaved executions of different tasks on the same processor to reduce context switching. At time t, L^+ is seeked from head to the last element. For each task, H_2 finds the slot t^* and the processor P_j to which this task was previously assigned. If all the slots from t^* till t were free on P_j, then this subtask is assigned to the jth processor. Finally, the first heuristic is called to assign non-assigned subtasks to available processors.

```
for q ← 1 to |L+| do
    /* L+[q] = τ_i^k, thus the values of i
       and k are known                   */
    if k > 1 then
        find P_j to which τ_i^(k-1) was assigned;
        /* t* = slot of τ_i^(k-1)          */
        if P_j was free from t* till t then
            assign τ_i^k to P_j;
            remove τ_i^k from L+;
        end
    end
end
H_1(L+);
```

Heuristic 2: H_2.

3.3. Heuristic 3

This heuristic is well-suited for systems in which context-switching overheads on a processor are small. Therefore, this heuristic tries to reduce the migrations of the subtasks of a same job. We call *starting substask* the first subtask of a job and *ending subtask* the last subtask of a job. This heuristic seeks L^+ and tries at first to assign starting subtasks to processors that have just finished scheduling an ending subtask. The remaining subtasks of L^+ are assigned, if possible, to the same processor as their predecessors (without the idle constraint of heuristic 2). And finally, heuristic 1 is called. Heuristic 3 represents the pseudo-code of this heuristic.

```
for q ← 1 to |L+| do
    if τ_i^k is a starting subtask then
        if any processor has finished scheduling an
        ending subtask then
            assign τ_i^k to this processor;
            remove τ_i^k from L+;
        end
    end
end
for q ← 1 to |L+| do
    if k > 1 then
        find P_j to which τ_i^(k-1) was assigned;
        if P_j is free at t then
            assign τ_i^k to P_j;
            remove τ_i^k from L+;
        end
    end
end
H_1(L+);
```

Heuristic 3: H_3.

A judicious implementation for these three heuristics based on appropriate data structures yields to an $O(m)$ time complexity where m is the number of processors.

3.4. Variants of H_2 and H_3: H_2^+ and H_3^+

We try with these heuristics to give heavy tasks a higher priority since the number of their subtasks is higher than the one of light tasks. So the L^+ list is sorted by decreasing task's weight (L^{++} designates the resulting list); the first element of this list will be the heaviest. Then heuristic 2 or heuristic 3 respectively apply to L^{++}; this gives rise to H_2^+ and H_3^+ respectively.

The complexity of the heuristics is then raised to $O(mlog(m))$ due to the sort operation.

4. Experimentation

In order to estimate the profit in term of migrations produced by these different heuristics, we carried out a statistical study. So, we developed a program that simulates the scheduler behaviour on an hyperperiod. Because of the assumption of synchronous, periodic and independent tasks,

the same release scenario of the tasks occurs every hyper-period. Thus it is sufficient to perform the schedulability analysis in the interval $[0, lcm)$ where lcm is the least common multiple of $\{T_i | \tau_i \in \tau\}$. At each slot, this program applies the tie-break rules given above and the output is a scheduled sequence for each processor. The heuristics described above were implemented in this simulator and they were all associated to the PF scheduler for establishing the sorted list of ready subtasks L^+. Even though it is not the most efficient of the Pfair algorithms, PF has been chosen as a starting point for our work because of its simplicity.

4.1. Context

These experiments used a task generator developed by our team. Actually, the values of lcm, the number of tasks nb_task and the total load of the system $U = \sum_i u_i = \sum_i C_i/T_i$ should be given as inputs. It generates a set of tasks with randomly computed periods and execution times.

To show and to characterise the behaviour of our generator, we calculated the average of the task loads (u_i) and its standard deviation for some sets of 30 configurations obtained with $lcm = 150$, $nb_task = 8$ and different values of U. The result gives us an idea about the weight variations for a set of tasks. Figure 3 shows the average of u_i and its standard deviation per configuration, for $U = 4.5$, $U = 4.8$, $U = 5.1$, and $U = 5.4$ (from left to right, and from top to bottom). Based on standard deviation shape, we can see that task weights vary enough; the total load is not divided and spread equally to all the tasks. So it guarantees a list of light and heavy tasks.

4.2. Results

Our experiment consisted in scheduling tasks generated by the task generator. For a given lcm and nb_task, the value of U is 2 at the beginning and then incremented by a step of 0.2. The number of processors is calculated as follows: $nb_proc = \lceil U \rceil$. For each U value, 30 configurations of tasks are generated and treated. Then $Total_migration = \sum_{i=1}^{nb_task} nb_mig(\tau_i)$ is computed for each configuration, where $nb_mig(\tau_i)$ is the number of migrations of τ_i within a job for all its jobs over the hyper-period. A job migrates if two of its consecutive executions (which may or may not be in consecutive time slots) are not on the same processor. $Average_migration$, the average of $Total_migration$ is computed for the 30 configurations. The result of H_1 is considered as the reference, since it is the algorithm that we want to improve. The improvement of a heuristic H_i is evaluated at last by the following metric:

$$improvement(H_i) = \frac{average_migration(H_i)}{average_migration(H_1)} \times 100 \quad (6)$$

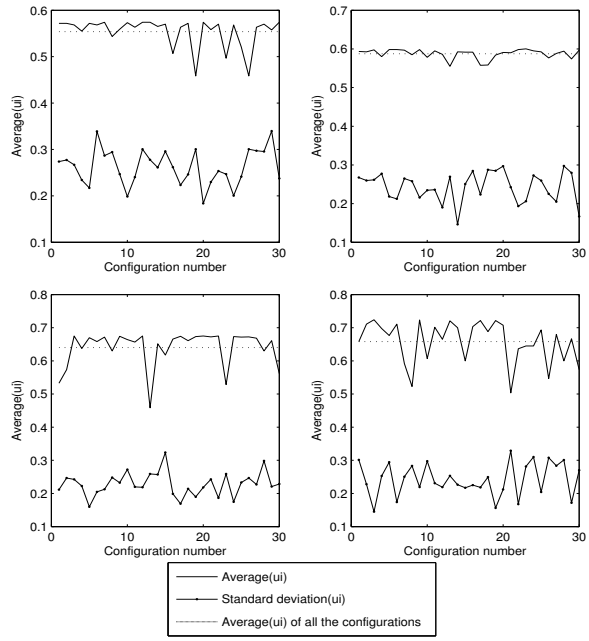

Figure 3. Average value and standard deviation of u_i for $U = 4.5$, $U = 4.8$, $U = 5.1$, and $U = 5.4$ (from left to right, and from top to bottom).

Figure 4 shows this improvement for all the proposed heuristics (the position of H_1 is obviously at 100%).

This experiment was repeated several times, producing almost identical results.

H_2 reduces the total number of migrations by 40% when using 2 or 3 processors and by 60% for a number of processors exceeding 3. By adding a simple rule that doesn't increase significantly the time complexity of the scheduling algorithm, the total number of migrations can be reduced by 50% on average.

Under H_3, we notice an important reduction of the number of migrations, it's about 55% for a 3-processor system and 75% for more than 3 processors.

H_2^+ gives almost the same result as H_2. Similarly, H_3^+ and H_3 produce a close result. We can conclude that sorting L^+ by task's weight does not affect the number of migrations.

The results presented in figures 5 and 6 confirm the performances of H_2 and H_3 algorithms for some other combinations of the parameters.

5. Conclusion

Under partitioning, migration is restricted: a task is scheduled on only one processor. On the other hand, recent

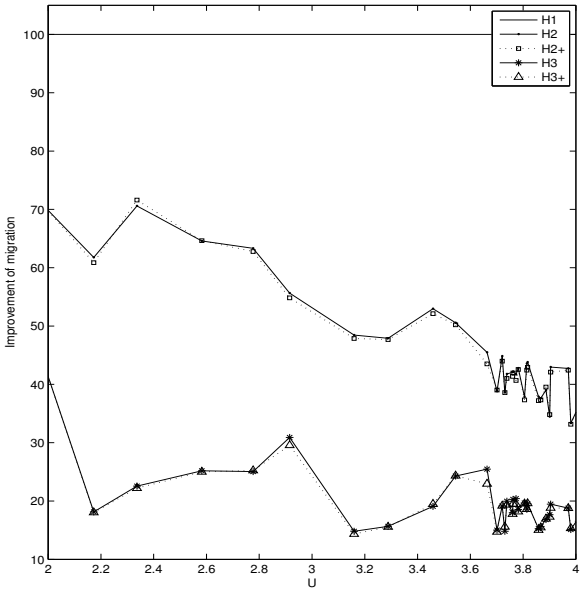

Figure 4. $Improvement(H_i)$ for $lcm = 150$, $nb_task = 6$ and $2 \leq U \leq 4$.

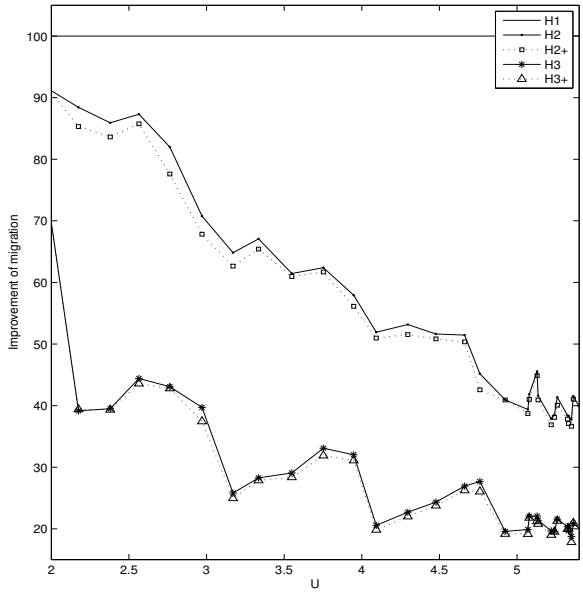

Figure 5. $Improvement(H_i)$ for $lcm = 150$, $nb_task = 8$ and $2 \leq U \leq 5.4$.

study shows many advantages of global scheduling over partitioning approaches. Pfair scheduling and many of the extensions to it that have been proposed earlier are promising global scheduling approaches. Under Pfair scheduling, interprocessor migration is permitted. In the worst case, a task may be migrated each time it is scheduled producing a large number of migrations.

We were interested by investigating and proposing heuristics which minimize the number of migrations with low cost overheads. The most efficient proposed heuristic is H_3. It decreases the number of migrations by more than 50% and practically reduces their negative effect on cache performance.

Presently we are investigating an indeep evaluation of the overheads (preemption overhead and migration one) in relation with hardware architecture characteristics. Moreover our work will be extended by simulating PD^2 algorithm; the order of the subtasks in L^+ built by a PD^2 scheduler may influence the proposed heuristics. ERfair scheduling algorithms may lead to lower average job response times since they prevent the spread of subtasks over the time. Applying these heuristics to an ERfair scheduling system and studying their results will be an interesting future work.

References

[1] J. Anderson, P. Holman, and A. Srinivasan. *Fair scheduling of real-time tasks on multiprocessors. Chapter 31 of Handbook of Scheduling, Algorithms, Models and Performance Ananlysis*. CHAPMAN & HALL/CRC, 2004.

[2] J. Anderson and A. Srinivasan. Early-release fair scheduling. *12th Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

[3] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. *7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, December 2000.

[4] J. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *13th Euromicro Conference on Real-Time Systems*, pages 76–85, June 2001.

[5] J. Anderson and A. Srinivasan. Optimal rate-based scheduling on multiprocessors. *34th Annual ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[6] J. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 1(68):157–204, February 2004.

[7] J. Anderson, A. Srinivasan, and J. Jonsson. Static-priority scheduling on multiprocessors. *22nd IEEE Real-Time Systems Symposium*, December 2001.

[8] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic systems. *Research Report TR-060601*, 2006.

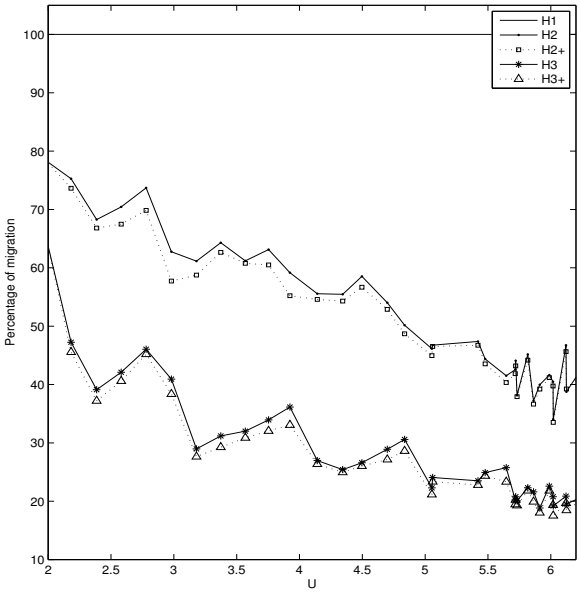

Figure 6. $Improvement(H_i)$ for $lcm = 200$, $nb_task = 9$ and $2 \leq U \leq 6.2$.

[9] S. Baruah. Fairness in periodic real-time scheduling. *Proceedings of the 16th IEEE Real-Time Symposium*, pages 200–209, december 1995.

[10] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[11] S. Baruah, J. Gehrke, and C. Plaxton. Fast scheduling of periodic tasks on multiple resources. *In proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, april 1995.

[12] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 1(26), 1978.

[13] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems Journal*, 2/3(25):187–205, 2003.

[14] P. Holman and J. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564, December 2005.

[15] T. Kimbrel, S. Baruch, and M. Sviridenko. Minimizing migrations in fair multiprocessor scheduling of persistent tasks. *Journal of Scheduling*, 9(4), August 2006.

[16] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 1(20):46–61, september 1973.

[17] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. *Proc. of 20th IEEE Real-Time Systems Symposium*, pages 294–303, December 1999.

[18] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, (84):93–98, 2002.

[19] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. *International Parallel and Distributed Processing Symposium*, April 2003.