

Task partitioning with replication upon heterogeneous multiprocessor systems

Sathish Gopalakrishnan

sgopalak@uiuc.edu

Marco Caccamo

mcaccamo@uiuc.edu

Department of Computer Science

University of Illinois at Urbana-Champaign

Abstract

The heterogeneous multiprocessor task partitioning with replication problem involves determining a mapping of recurring tasks upon a set consisting of different processing units in such a way that all tasks meet their timing constraints and no two replicas of the same task are assigned to the same processing unit. The replication requirement improves the resilience of the real-time system to a finite number of processor failures. This problem is NP-hard in the strong sense. We develop a Fully Polynomial-Time Approximation Scheme (FPTAS) for this problem.

Keywords: Multiprocessor scheduling; Recurring tasks; Fault tolerance; Partitioning; Approximation scheme.

1. Introduction

Multiprocessor embedded systems have become a reality for many different applications. MPOC [22] is an example of a multiprocessor platform that was originally designed for high-resolution printers. A motivation for this transformation from uniprocessor platforms has been the growing demand for functionality and the infeasibility of executing all tasks on one processor when timing requirements need to be satisfied. Another reason for employing multiprocessors has been the ability to tolerate processor failures via task replication [17, 11]. Maintaining replicas of a task at different processors ensures that single processor failures will not affect the required system behavior. Real-time systems are employed in many safety-critical applications and reliability is of paramount importance for such applications [21]. An application example is video monitoring for remote activities in space. In August 2005, astronauts had to take a spacewalk to repair heat shields on the Dis-

covery orbiter. Similar missions require careful execution and, in some situations, may call for a robotic arm. The processors on the arm are susceptible to cosmic radiation which may introduce errors; at the same time, heavy lead shielding would make it difficult to construct and employ such robotic arms for delicate operations. Redundancy becomes essential to ensuring the success of critical – and expensive – activities.

Modern electronic systems consist of a fairly heterogeneous collection of components; these components are a mix of analog and digital devices, controlled by several software objects. The hardware is constructed from different microprocessors (general-purpose or specific chips sets for tasks such as DSP), various memory chips and dedicated integrated circuits (such as ASICs and FPGAs) and a set of local connections between these components and interfaces (sensors, actuators) between the computing system and the operating environment. Each hardware component is capable of perform multiple functions; some functions may be better served on dedicated processors and some might require the flexibility offered by general-purpose processing units. Such systems are called *heterogeneous platforms* [7]. This heterogeneity implies that the same task (lines of code) may require a different number of clock cycles when executed on different processing units. At the same time, it is not cost-efficient to use a distinct processor for each software process.

We study the task allocation problem with replication in a setting similar to the one explored by Baruah [5]. The tasks we consider are recurring (or periodic) tasks which are associated with release times and deadlines. Moreover, these tasks need to be replicated on multiple processing units to ensure a degree of fault tolerance. Our main contribution is a Fully Polynomial-Time Approximation Scheme for the task partitioning problem with timing and replication constraints. By doing so, we generalize the work started by Baruah and support fault-tolerance primitives. Prior techniques for task

partitioning run into exponential-time complexity (in the number of tasks) when replication is required and we present an algorithm that runs in polynomial-time. The algorithm we present will enable system architects to explore resource tradeoffs at design time.

Organization of this paper In Section 2, we detail the system model and the assumptions made about the model. Section 3 provides a formal definition of the problem we wish to solve and prove its intractability. Section 4 outlines the approach used by Baruah [5] to solve a special case of the problem we tackle and highlights the difficulties in using the existing technique for the problem at hand. Section 5 describes the dynamic programming methodology we use to obtain a Fully Polynomial-Time Approximation Scheme for the replicated task partitioning problem. Section 6 uses examples to illustrate our methodology. In the last two sections, we discuss related work and the ramifications of and further directions for this work.

2. System model and assumptions

2.1. The heterogeneous system platform

Our system model is identical to the model described by Baruah [5]. A heterogeneous real-time platform is specified by a set of M processors (processing units), $\Pi = \{\pi_1, \dots, \pi_M\}$.

2.2. Task model

The set of activities to be performed by the system is specified by a set of N recurring software tasks, $\Gamma = \{\tau_1, \dots, \tau_N\}$. Task τ_i requires an execution time $e_{i,j}$ when assigned to processor π_j and is executed periodically every p_i units of time. p_i is referred to as the period of τ_i . In the *periodic* task model, each task generates an infinite sequence of jobs that needed to be executed (for $e_{i,j}$ time units depending on the processor it is assigned to) and successive jobs are separated by exactly p_i time units. If successive jobs of τ_i are separated by *at least* p_i time units, the task is referred to as a *sporadic* task. Each job is expected to complete within p_i time units after it has been released (made eligible for execution). Thus, the *deadline* of a job relative to its release time is p_i . The heterogeneity of the computing platform is captured by the different execution times on different processors.

Each software task requires a fraction of the processing capacity of the processor it is allocated to. The fraction of capacity demanded by a task is called its *utilization* and varies depending on the processor the task is allocated to. More precisely, if task τ_i were allocated to processor

π_j , its utilization of π_j is $\frac{e_{i,j}}{p_i} = u_{i,j} \in (0, 1] \cup \{\infty\}$. In other words, $\tau_{i,j}$ requires a fraction $u_{i,j}$ of the capacity of π_j . If τ_i cannot be executed on π_j then $u_{i,j} \leftarrow \infty$. For compactness, the values $u_{i,j}$ can be encoded in a $n \times m$ *utilization matrix*, \mathcal{U} . Table 1 shows a utilization matrix for a system with 5 tasks and 4 processors. Notice that the heterogeneity of the platform results in non-identical (and non-uniform) utilization requirements from processor to processor.

Tasks	Processors			
	π_1	π_2	π_3	π_4
τ_1	0.14	0.24	0.12	0.36
τ_2	0.31	0.11	0.23	0.22
τ_3	0.50	0.21	0.21	0.11
τ_4	0.50	0.12	0.14	0.12
τ_5	0.20	0.35	0.30	0.20

Table 1. Example of a utilization matrix

Independence of jobs Different jobs are always assumed to be independent of each other. By independence we mean that there is neither resource sharing nor data dependency. This requirement may appear rather strict because it applies to jobs that belong to the same task as well as to jobs belonging to different tasks. But, when one considers the fact that typical situations where hard-real time tasks execute are control applications, which are repetitive and only update certain values without being dependent on past state, this imposition is not major.

Partitioned execution Tasks are assigned to particular processors at design time and all jobs of the task execute on that processor only. This approach is known as *partitioned scheduling*. Following processor allocation, the problem reduces to uniprocessor scheduling and efficient algorithms have been well-studied for this case.

Non-partitioned schemes allow task and job migration at run-time. Such schemes *may* lead to better utilization of resources but their implementation is more complex, especially on a heterogeneous platform when the same job might require a different set of instructions and data formats when migrated from one processor to another.

Replication To guarantee resilience to faults, each task is assumed to be replicated on K *distinct* processors. If $K = 2$, each task may be operating in a simple *primary/backup* configuration, and for $K > 2$ the output of each job might be sent to a data consumer who might use any scheme (such as voting or averaging) to select the desired output. Triple modular redundancy (when each component is replicated thrice) has

been a very popular engineering technique for improving system reliability for many years [17].

We will assume that each task needs to be replicated K times for simplicity, but the techniques we describe in the paper extend automatically to the case when different tasks might need a different number of replicas. We will refer to K as the *replication factor*.

We denote the k^{th} replica of a task as $\tau_i^{[k]}$. We use the indicator variable $x_{i,j}^{[k]} \in \{0, 1\}$ to indicate that the k^{th} replica of task τ_i has been assigned to processor π_j . If $x_{i,j}^k = 1$ then $\tau_i^{[k]}$ has been assigned to π_j else it has not been assigned to π_j . The requirement that no two replicas of the same task are allocated to the same processor can formally be written as

$$\forall i, \forall j: \sum_{k=1}^K x_{i,j}^{[k]} \leq 1.$$

The set of all replicas is denoted by the set $\Gamma' = \{\tau_1^{[1]}, \dots, \tau_1^{[K]}, \dots, \tau_N^{[1]}, \dots, \tau_N^{[K]}\}$.

Replication is useful in many (other) contexts. Multiple copies of data files might be saved at different locations to ensure data integrity. It is also beneficial to view the the requirement that two replicas not reside on the same processor as an independence requirement. Two tasks, not necessarily replicas of the same task, might need to be allocated to different processors because of resource conflicts. The ideas we investigate in this paper can be extended to obtain solutions to such problems.

No communication By assuming job independence, we are neglecting the overheads of communication. In some systems, communication bandwidth is high and the communication latencies are low enough to be ignored or absorbed by the computation times.

In a replicated task scenario, certain architectures directly connect processors to sensors and actuators; outputs from task replicas can be processed at the actuator (or sensor), which may perform the error correction or other combinatorial operation (like voting).

In general, one should consider the cost of interprocess communication while performing task assignment. Yet, performing task allocation even without detailed analysis of interprocess communication may provide system architects with sufficient insight into the overall system performance.

2.3. Scheduling model

Earliest deadline first (EDF) scheduling is known to be optimal for scheduling periodic tasks upon uniprocessors. Liu and Layland [16]

showed that a set of tasks can be scheduled on a processor if the total utilization of tasks allocated to the processors does not exceed the capacity of the processor. Alternately, we can say that all tasks allocated to π_j meet their timing requirements if and only if

$$\sum_{i=1}^N \sum_{k=1}^K x_{i,j}^{[k]} u_{i,j} \leq 1.$$

We restrict our focus to EDF scheduling given its optimality and the simplicity of testing schedulability on uniprocessors.

3. Problem statement and hardness

3.1. Definitions

The goal of our work is to find a mapping of tasks to processors such that the capacity of any individual processor is not exceeded and no two replicas of the same task are assigned to the same processor. We refer to this problem as the *heterogeneous multiprocessor replicated task partitioning problem*.

Definition 1 For any set of tasks (with replicas) Γ' , $U_j(\Gamma')$ denotes the cumulative utilization of all tasks allocated to processor π_j .

$$U_j(\Gamma') := \sum_{i=1}^N \sum_{k=1}^K x_{i,j}^{[k]} u_{i,j}.$$

Definition 2 A feasible mapping for a set of replicated tasks upon a heterogeneous multiprocessor system satisfies the following condition:

$$U_j(\Gamma') \leq 1, \forall j.$$

3.2. Intractability

Like many combinatorial optimization problems, the heterogeneous multiprocessor replicated task partitioning problem is intractable. This can be shown by a reduction to the 3-PARTITION problem which is known to be NP-hard in the strong sense [10]. We include the proof here for completeness.

Theorem 1 The heterogeneous multiprocessor replicated task partitioning problem is NP-hard in the strong sense.

Proof.

A special case of the heterogeneous multiprocessor replicated task partitioning problem is obtained by setting $K = 1$. This special case, the heterogeneous multiprocessor task partitioning problem, was studied by Baruah [5]. The version of 3-PARTITION that we transform to an instance of the the heterogeneous multiprocessor

task partitioning problem is as follows: Given an integer A and a collection $L = \langle a_1, a_2, \dots, a_{3p} \rangle$ of integers such that $a_i \geq 2, A/4 < a_i < A/2$ and $\sum_{i=1}^{3p} a_i = p \cdot A$, can L be partitioned into p subsets L_1, \dots, L_p such that the elements in each subset sum to A ?

The instance of the 3-PARTITION problem described above can be transformed into a set of $3p$ recurring tasks that need to be partitioned over p identical processors by the following step:

$$\forall i \in \{1, 3p\}, \forall j \in \{1, p\} : u_{i,j} \leftarrow \frac{a_i}{A}.$$

From this transformation, it is clear that the partitioning of these $3p$ tasks over the p identical processors if and only if L can be partitioned into the p subsets. Thus a special case of the heterogeneous multiprocessor replicated task partitioning problem is NP-hard in the strong sense and the theorem follows. \square

4. Linear programming formulation

We consider a slight variation of the task partitioning problem. The problem, as it is defined, is a satisfiability problem: Is there an allocation that does not violate processor capacities and, if so, what is the mapping of tasks to processors?

Changing the problem slightly, we can obtain an optimization problem, \mathcal{P} , that attempts to minimize the maximum utilization, $U = \max\{U_1(\Gamma'), U_2(\Gamma'), \dots, U_N(\Gamma')\}$, on any processor. If $U \leq 1$ then the heterogeneous replicated task partitioning problem has a feasible solution. Notice that this formulation is similar to another NP-hard problem: the *minimum makespan* problem [20, 14] for scheduling non-preemptible tasks on parallel machines. In the minimum makespan problem, a set of non-repeating, non-preemptible tasks need to be scheduled on a given number of parallel machines and the objective is to minimize the completion time – time to finish the assigned jobs – among all the machines.

4.1. Integer programming formulation

The problem of interest can easily be represented by an integer linear program. Given the set of all replicas Γ' , the set of processors Π , and the utilization matrix \mathcal{U} , we have the following integer program involving the indicator variables $x_{i,j}^{[k]}$:

Mathematical Program 1 ILP-FEASIBILITY

Minimize U subject to the following constraints:

- (C1) $x_{i,j}^{[k]} \in \{0, 1\}, \forall i, \forall j, \forall k$
 - (C2) $\sum_{j=1}^M x_{i,j}^{[k]} = 1, \forall i, \forall k$
 - (C3) $\sum_{k=1}^K x_{i,j}^{[k]} \leq 1, \forall i, \forall j$
 - (C4) $\sum_{i=1}^N \sum_{k=1}^K x_{i,j}^{[k]} u_{i,j} \leq U, \forall j$
-

The objective function for this integer linear program is to **minimize** U where U is the maximum utilization of any processor after tasks have been assigned to processors.

It is easy to see that the optimal value of U is less than or equal to 1 if and only if the heterogeneous multiprocessor replicated task partitioning problem has a feasible solution, and the mapping is given by the values of the indicator variables that achieve the optimal U . (This can be proved easily; see Baruah [5] for more details.)

Integer programming formulations cannot get around the intractability of this problem. For small problem sizes, however, most modern optimization products (such as CPLEX [1]) can return optimal solutions in reasonable time. As problems become large, a common approach to obtaining approximate solutions is to relax the integer program to a linear program (which allows fractional solutions) and then develop rounding techniques that bound the distance between the obtained solution and the optimal solution.

4.2. Linear programming relaxation

By relaxing the restrictions that $x_{i,j}^{[k]}$ be integers, we get the a linear program with the the new requirement that $x_{i,j}^{[k]} \geq 0$. Linear programs can be solved efficiently; the ellipsoid algorithm [13] and the interior-point algorithm [12] run in polynomial time and the popular simplex algorithm [8] typically takes polynomial time [23] even if its worst-case behavior is exponential.

Mathematical Program 2 LP-RELAX-FEASIBILITY

Minimize U subject to the following constraints:

- (C1) $x_{i,j}^{[k]} \geq 0, \forall i, \forall j, \forall k$
 - (C2) $\sum_{j=1}^M x_{i,j}^{[k]} = 1, \forall i, \forall k$
 - (C3) $\sum_{k=1}^K x_{i,j}^{[k]} \leq 1, \forall i, \forall j$
 - (C4) $\sum_{i=1}^N \sum_{k=1}^K x_{i,j}^{[k]} u_{i,j} \leq U, \forall j$
-

It is useful to recall the following facts about

linear programming [19]:

Fact 1 *The feasible region for a linear programming problem is convex and the objective function reaches its optimal point at a vertex of the feasible region.*

Fact 2 *Consider a linear program on n variables x_1, \dots, x_n in which each variable is subject to the non-negativity constraint. Suppose there are m linear constraints. If $m < n$, then at most m of the variables have non-zero values at each vertex of the feasible region.*

An insight in Baruah’s work [5] was that when $K = 1$ and the integer constraints were relaxed to obtain a linear program, at most $M - 1$ tasks might be assigned to more than one processor. This observation follows from the fact that when $K = 1$, there are $NM + 1$ variables and $N + M$ linear constraints. Using Fact 1 and Fact 2, $N + M$ variables are non-zero at an optimal solution. U must be non-zero, and that leaves $N + M - 1$ indicator variables that are non-zero. Using the *pigeonhole principle*, it is possible to conclude that at most $M - 1$ of the constraints that ensure that a task is completely allocated (Constraint (C2) in LP-RELAX-FEASIBILITY) will have more than one non-zero variable. Baruah suggested that for these $M - 1$ tasks, an exponential search with time-complexity $O(M^M)$ can be used to determine the task assignment. If M is assumed to be a constant, then Baruah’s two-step process can be used to obtain an algorithm that is typically polynomial time in N . Baruah showed that the two-step algorithm is guaranteed to find a solution to the partitioning problem if there is a feasible mapping of tasks to processors in which at most half the capacity of each processor in Π is used. It may not find a solution otherwise. Essentially, this scheme is a 2-approximation scheme.

In the case of the problem under study in this paper (task allocation with replication), we note that the relaxation LP-RELAX-FEASIBILITY has $NMK + 1$ variables and $NK + NM + M$ constraints. Following the reasoning described by Baruah, we can conclude that at an optimal vertex point of the linear program, $NK + NM + M - 1$ indicator variables are non-zero and at most $NM + M - 1$ of the constraints described by (C2) in LP-RELAX-FEASIBILITY have more than one non-zero variable. This means that an exhaustive scheme used to map the task replicas associated with these constraints would have a time complexity of $O(M^{NM})$ which is exponential in N . The requirement that two replicas of a task not be allocated to the same processor leads to a rapid growth in problem complexity.

The difficulty of applying Baruah’s technique leads us to consider alternate approaches to solving the replicated task allocation problem.

5. Dynamic programming formulation

The heterogeneous multiprocessor replicated task partitioning problem can be solved by a dynamic programming approach if we quantize the utilizations of the tasks. If δ ($0 < \delta < 1$), is the smallest quantum of utilization, then we replace each $u_{i,j}$ with $u'_{i,j} \leftarrow \lfloor u_{i,j}/\delta \rfloor \cdot \delta$. This quantization transforms the original problem where utilization could be any value in $(0, 1]$ to a problem, \mathcal{P}' , where the utilization is an integer multiple of δ .

To minimize the maximum utilization among the processors, consider the dynamic programming approach specified in Algorithm 1.

The dynamic programming approach presented here determines an optimal solution for the quantized version of the problem. Each step of the dynamic programming recurrence requires $O(\binom{M}{K})$ computations.

The size of the $(M + 1)$ -dimensional array that needs to be computed is $O(N \cdot C^M)$ where C is an upper-bound on the maximum achievable processor utilization in discrete steps of size δ , i.e., $C = \max_j \sum_{i=1}^N u'_{i,j}/\delta$. Therefore, this dynamic programming approach to solving the partitioning problem requires $O(\binom{M}{K} \cdot N \cdot C^M)$ time. For fixed M , if C is part of the problem input, the dynamic programming method leads to a pseudopolynomial-time algorithm.

We allow $C\delta$ to exceed 1 because we are solving the optimization problem of minimizing U' . If U' exceeds 1, there is no feasible assignment of tasks such that the replication requirements and timing constraints are satisfied.

The quantization is necessary to obtain a pseudopolynomial-time algorithm. Quantization leads to an array of finite size; allowing all possible values in $(0, 1]$ for values of $u'_{i,j}$ would require an array of infinite size. Let $u_{max} = \max_i \max_j \{u_{i,j}\}$ from all $u_{i,j} \neq \infty$ (eliminate all obviously infeasible assignments). Trivially, $U \geq u_{max}$. Further, we observe that in the quantized version of the problem, C can be no larger than $N \lfloor u_{max}/\delta \rfloor$.

It is, of course, important to note that the dynamic programming technique solves only the quantized version of the replicated task partitioning problem to optimality. The quantization introduces an error of at most δ per task because $u_{i,j} \leq \delta \lfloor u_{i,j}/\delta \rfloor + \delta$. Since at most N task replicas can be assigned to a particular processor¹, the error in the maximum utilization that is being minimized is at most $N\delta$.

If U' is the solution (the maximum utiliza-

¹No two replicas of a task can be assigned to the same processor.

Algorithm 1 DP-Partitioning

- For $i = 1, \dots, N$, let $X(i, U_1, \dots, U_M)$ be the minimum among the maximum utilizations achieved by optimally allocating tasks $\tau_i, \tau_{i+1}, \dots, \tau_n$ given that processor π_j has already achieved a quantized utilization of U_j after the assignment of the first $i - 1$ tasks. Task τ_i needs to be allocated to K distinct processors.
- Let B_1, \dots, B_L be all the possible combinations of K processors from the set Π of processors. There are $L = \binom{M}{K}$ such combinations. Let $y_{j,l}$ be the indicator variable denoting $\pi_j \in B_l$.
- In fact, this formulation suggests the following simple recurrence:

$$\begin{aligned} i = 1, 2, \dots, N: \quad X(i, U_1, \dots, U_M) &= \min_{l=1, \dots, L} X(i+1, U_1 + y_{1,l}u'_{i,1}, \dots, U_M + y_{M,l}u'_{i,M}) \\ i = N+1: \quad X(i, U_1, \dots, U_M) &= \max_{j=1, \dots, M} U_j \end{aligned}$$

This recurrence relation clarifies the process outlined in the first step.

- To obtain the optimal solution to the problem, we need to identify how to allocate the very first task, therefore, we need to determine $X(1, 0, \dots, 0)$. The procedure to obtain $X(1, 0, \dots, 0)$ can be endowed with some extra τ memory to obtain, efficiently, the sequence of allocations that produces a feasible solution.
-

tion of any processor) to problem \mathcal{P}' , we can see that $U \leq U' + N\delta$. Also, it is straight-forward to see that $U' \geq u'_{max}$ where $u'_{max} = \lfloor u_{max}/\delta \rfloor \delta$.

We now have

$$\frac{U}{U'} \leq 1 + \frac{N\delta}{U'} \leq 1 + \frac{N\delta}{u'_{max}} = 1 + \varepsilon,$$

where $\varepsilon = N\delta/u'_{max}$. Thus, if problem \mathcal{P}' has a feasible solution ($U' \leq 1$), then problem \mathcal{P} has a feasible solution if the speeds of some of the N processors are increased by a factor of at most $1 + \varepsilon$. If \mathcal{P}' does not have a feasible solution, \mathcal{P} has no feasible solution. This implies that \mathcal{P}' is a $(1 + \varepsilon)$ -approximation algorithm for \mathcal{P} .

Moreover, using $\varepsilon = \frac{N\delta}{u_{max}}$, we get $C < N^2/\varepsilon$ from which we note that the approximation algorithm requires $O\left(\binom{M}{K} \cdot \frac{N^{2M+1}}{\varepsilon^M}\right)$ time, which is polynomial in N, K and $1/\varepsilon$ for fixed M .

Theorem 2 *For a fixed error parameter ε , the heterogeneous multiprocessor replicated task partitioning problem admits a fully polynomial-time approximation scheme.*

Proof.

Algorithm 1 is accurate to a factor of $(1 + \varepsilon)$ and has time complexity $O\left(\binom{M}{K} \cdot \frac{N^{2M+1}}{\varepsilon^M}\right)$ which is polynomial in N, K and $1/\varepsilon$. It is a FPTAS for the heterogeneous multiprocessor replicated task partitioning problem. \square

Additional remarks In our algorithm construction, we assumed that each processor may achieve a utilization of up to $C\delta$. Solving the

problem in this manner allows a designer to understand how constrained the platform is for executing a set of tasks. This is not strictly necessary if $C\delta > 1$. If feasibility is the only question of interest, the dynamic programming need not extend to processor utilizations beyond 1. In this case, the size of the dynamic programming array need be $O(N(\lfloor 1/\delta \rfloor)^M)$. From an implementation perspective, this observation leads to reduced memory requirements and running times. It is useful to note that the dynamic programming formulation has polynomial running time but requires significant amounts of memory. All the memory, however, is not needed at the same time, and careful allocation of memory can reduce the instantaneous memory load.

6. Examples

To provide a clear understanding of our work, we use examples to illustrate the possible solutions that can be obtained using our dynamic programming approach. We consider two examples where dynamic programming finds a feasible solution for the quantized problem but the original partitioning problem may or may not have a solution. Before proceeding, we reiterate that *if there is no feasible solution to the quantized problem, the original problem is infeasible.*

6.1. Example 1

We begin with the task set described in Table 1. In this case, $N = 5$ and $M = 4$. We set the replication factor, K to 3. In this task set,

$u_{max} = 0.5$ (as can be seen by an inspection of the utilization matrix). For a desired accuracy of $\varepsilon = 0.5$, we need to use $\delta = (\varepsilon \times u_{max})/N = 0.05$. Using dynamic programming, we obtain the maximum utilization of a processor in the quantized setup as $U' = 0.7$.² The task mapping is shown in Table 2. Each task is allocated to exactly 3 processors.

Table 2 contains the actual utilization (not the quantized utilizations) and we find that π_3 achieves the maximum utilization of $U = 0.77$ which is less than 1. In fact, we had an *a priori* guarantee that $U \leq (1 + \varepsilon)U'$. Since U' was 0.7 and $\varepsilon = 0.5$, we could be almost certain, even without the knowing the exact mapping, that the given problem has a feasible solution.

For this particular example, we could have eliminated all uncertainty if we had chosen $\varepsilon = 0.25$ because we would have obtained $U' = 0.725$ and $(1 + 0.25)(0.725) = 0.90625 < 1$ which ensures that the problem has a feasible mapping without verifying the allocation. The choice of ε is a tradeoff between available memory, run time and desired confidence in the existence of a feasible solution.

6.2. Example 2

Consider a task set (Table 3) that is slightly different from the earlier example.

Using the same parameters as in Example 1 ($K = 3, \varepsilon = 0.5$), we get the solution to the quantized problem as $U' = 0.9$. The task mapping is shown in Table 4.

In this example, although $U' = 0.9 < 1$, the actual allocation results in a capacity violation on π_2 . However, given our choice of $\varepsilon = 0.5$, we know that the utilization of π_2 cannot exceed $(1 + 0.5)(0.9) = 1.35$ and that is indeed correct. The utilization of π_2 is 1.02.

Situations like these may be tided over by employing a slightly faster (but similar) processor in place of π_2 . An alternate approach which will avoid such situations, where the quantized problem is feasible but the original problem is not, completely is to scale all utilizations by $1 + \varepsilon$ before quantization. Scaling will ensure that a feasible solution to the quantized problem necessarily implies a feasible solution to the original partitioning problem. However, scaling might make a normally feasible problem infeasible and that possibility should be accounted for.

7. Related work

The work that is most closely related to our work is Baruah's work on task partition-

²We do not discuss the dynamic programming step-by-step because of space constraints.

ing upon heterogeneous multiprocessor platforms [5]. Baruah does not consider task replication and develops a 2-approximation scheme that has polynomial-time complexity in the number of tasks and exponential-time complexity in the number of processors.

We have generalized the problem to include replication with the constraint that no two replicas are mapped to the same processing unit. This is significant from the fault tolerance perspective. Also, our dynamic programming approach yields a fully polynomial-time approximation. The time complexity of our scheme is polynomial in the number of tasks, and in $1/\varepsilon$ where ε is the error tolerance. To obtain a FPTAS, we have followed work due to Woeginger on obtaining approximation schemes from dynamic programming formulations [24].

There has been extensive work on partitioned scheduling of multiprocessor real-time systems. Most of the work in this area has typically focussed on obtaining utilization bounds that ensure schedulability of a set of tasks [18, 2, 4, 9], especially in the context of uniform microprocessor systems (all processing units are alike – they might differ in speeds by some known factors.) None of these works deal with heterogeneous platforms or provide fault tolerance.

Liberato et al. [15] have proposed a fault-tolerant scheduling algorithm which allows tasks to recover from transient failures. Their approach uses global scheduling and does not recover from permanent failures.

Aydin [3] has extended processor demand analysis to analyze *uniprocessor* schedulability in the presence of at most k transient faults when tasks are scheduled using EDF. Aydin considers that systems that use recovery blocks associated with failures and accounts for the extra overhead that recovery blocks may introduce. In the systems we target, there is full replication of tasks and recovery is not performed only after a fault occurs.

Bertossi et al. [6] have analyzed first-fit assignment for real-time tasks scheduled using rate monotonic priority assignments with passive and active task instances. They assume that a passive instance will be invoked only when the active instance of a task fails and allocate passive replicas of many tasks to the same processor assuming that only a few of the passive replicas will need to be activated at the same time. In comparison, we have devised a FPTAS that makes efficient task allocations and allows more than two replicas of a task to execute at the same time: essentially providing for better fault resilience. The choice of using active/passive replicas or all-active replicas is a design choice that should be guided by the reliability requirements of the real-time system.

Processor	Tasks (Utilization)	Total utilization
π_1	$\tau_1(0.14), \tau_2(0.31), \tau_5(0.20)$	0.65
π_2	$\tau_1(0.24), \tau_2(0.11), \tau_3(0.21), \tau_4(0.12)$	0.68
π_3	$\tau_1(0.12), \tau_3(0.21), \tau_4(0.14), \tau_5(0.30)$	0.77
π_4	$\tau_2(0.22), \tau_3(0.11), \tau_4(0.12), \tau_5(0.20)$	0.65

Table 2. Task-to-processor mapping for Example 1

Tasks	Processors			
	π_1	π_2	π_3	π_4
τ_1	0.14	0.24	0.12	0.36
τ_2	0.31	0.31	0.23	0.22
τ_3	0.50	0.41	0.41	0.11
τ_4	0.50	0.12	0.14	0.12
τ_5	0.20	0.35	0.40	0.40

Table 3. Utilization matrix for Example 2

8. Conclusion

We have studied the problem of mapping recurring tasks to a heterogeneous set of processors in a fashion that guarantees that timing requirements are met and tasks are replicated so that processor failures can be tolerated. In this work, we have generalized work done by Baruah [5]. Our work presents a fully polynomial-time approximation scheme to solve this problem, thereby advancing the state of the art. The scheme we outline has been implemented and in our experiments we found that the runtime overhead is acceptable – for an offline process – and is of the order of some tens of seconds for small input sizes (task sets and number of processors) and a few minutes for larger inputs. A more exhaustive performance characterization is a topic for further work.

To be precise, we must mention that our algorithm is fully-polynomial time in the number of tasks and not in the number of processors, but the number of processors in such applications is much smaller and therefore the complexity is acceptable. Whether it is possible to construct an algorithm that is polynomial in the number of processors also is open.

Partitioned scheduling allows a system designer to reason better about the behavior of the system under processor failures. Global scheduling, which allows tasks to migrate from processor to processor, provides better schedulability and we would like to further understand the tradeoff between partitioned and global scheduling as regards fault tolerance. We hope to leverage some of the work that has been carried out in this direction (e.g., [15]).

Task allocation, in this work, has been carried out with timing and fault-tolerant constraints alone. We would like to explore this problem with added constraints. For instance,

memory is a limited resource and designers would like to ensure that task allocation respects the memory limits of a processor. Power consumption is another area of concern for real-time systems and power-aware task assignment poses another challenge for partitioned execution.

Acknowledgements This work was supported by the National Science Foundation through grants CCR-0237884, CCR-0325716 and CNS-0509268.

References

- [1] High performance software for mathematical programming and optimization. <http://www.cplex.com/>.
- [2] ANDERSSON, B., AND JONSSON, J. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Proceedings of the Euromicro Conference on Real-Time Systems* (July 2003), pp. 33–40.
- [3] AYDIN, H. On fault-sensitive feasibility analysis of real-time task sets. In *Proceedings of the IEEE Real-Time Systems Symposium* (Dec. 2004).
- [4] BAKER, T. P. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the IEEE Real-Time Systems Symposium* (Dec. 2003), pp. 120–129.
- [5] BARUAH, S. Task partitioning upon heterogeneous multiprocessor platforms. In *Proceedings of the IEEE Real-Time Systems and Embedded Technology and Applications Symposium* (2004).
- [6] BERTOSSI, A. A., MANCINI, L. V., AND ROSSINI, F. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Transactions on Parallel and Distributed Systems* 10, 9 (Sept. 1999), 934–945.
- [7] BRAUN, T. D., SIEGEL, H. J., AND MACIEJEWSKI, A. A. Heterogeneous computing: Goals,

Processor	Tasks (Utilization)	Total utilization
π_1	$\tau_1(0.14), \tau_3(0.50), \tau_5(0.20)$	0.84
π_2	$\tau_1(0.24), \tau_2(0.31), \tau_4(0.12), \tau_5(0.35)$	1.02
π_3	$\tau_1(0.12), \tau_2(0.23), \tau_3(0.41), \tau_4(0.14)$	0.90
π_4	$\tau_2(0.22), \tau_3(0.11), \tau_4(0.12), \tau_5(0.40)$	0.85

Table 4. Task-to-processor mapping for Example 2

- methods and open problems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (2001), pp. 1–12.
- [8] DANTZIG, G. B. *Linear programming and extensions*. Princeton University Press, 1963.
- [9] FUNK, S., AND BARUAH, S. Task assignment on uniform heterogeneous multiprocessors. In *Proceedings of the Euromicro Conference on Real-Time Systems* (July 2005), pp. 219–226.
- [10] GAREY, M., AND JOHNSON, D. *Computers and Intractability*. W. H. Freeman, N.Y., 1979.
- [11] GUERRAOU, R., AND SCHIPER, A. Software-based replication for fault tolerance. *Computer* 30, 4 (Apr. 1997), 68–74.
- [12] KARMAKAR, N. A new polynomial-time algorithm for linear programming. *Combinatorica* 4 (1984), 373–395.
- [13] KHACHIYAN, L. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR* 244 (1979), 1093–1096.
- [14] LENSTRA, J. K., SHMOYS, D. B., AND TARDOS, E. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming* 46 (1990), 259–271.
- [15] LIBERATO, F., LAUZAC, S., MELHEM, R., AND MOSSÉ, D. Fault tolerant real-time global scheduling on multiprocessors. In *Proceedings of the Euromicro Conference on Real-Time Systems* (1999), pp. 252–259.
- [16] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 1, 20 (1973), 40–61.
- [17] LYONS, R. E., AND VANDERKULK, W. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research* 6, 2 (1962), 200–209.
- [18] OH, D.-I., AND BAKER, T. P. Utilization bounds for N -processor rate monotone scheduling with static processor assignment. *Real-Time Systems* 15 (1998), 183–192.
- [19] PAPANITRIOU, C. H., AND STEIGLITZ, K. *Combinatorial Optimization : Algorithms and Complexity*. Dover Publications, 1998.
- [20] POTTS, C. N. Analysis of a linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics* 10 (1985), 155–164.
- [21] PRADHAN, D. K. *Fault-Tolerant Computer System Design*. Prentice-Hall, New Jersey, 1996.
- [22] RICHARDSON, S. MPOC: A chip multiprocessor for embedded systems. Tech. Rep. HPL-2002-186, HP Laboratories, July 2002.
- [23] SPIELMAN, D. A., AND TENG, S.-H. Smoothed analysis: why the simplex algorithm usually takes polynomial time. *Journal of the ACM* 51, 3 (2004), 385–463.
- [24] WOEGERING, G. J. When does a dynamic programming formulation guarantee the existence of an FPTAS? In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (1999), pp. 820–829.