Partitioned real-time scheduling on heterogeneous shared-memory multiprocessors

Martin Niemeier
École Polytechnique Fédérale de Lausanne
Discrete Optimization Group
Lausanne, Switzerland
martin.niemeier@epfl.ch

Andreas Wiese
Technische Universität Berlin
Institut für Mathematik
Berlin, Germany
wiese@math.tu-berlin.de

Sanjoy Baruah
University of North Carolina
Department of Computer Science
Chapel Hill, US
baruah@cs.unc.edu

***Abstract*—We consider several real-time scheduling problems on heterogeneous multiprocessor platforms, in which the different processors share a common memory pool. These include (i) scheduling a collection of implicit-deadline sporadic tasks with the objective of meeting all deadlines; and (ii) scheduling a collection of independent jobs with the objective of minimizing the makespan of the schedule. Both these problems are intractable (NP-hard). For each, we derive polynomial-time algorithms for solving them approximately, and show that these algorithms have bounded deviation from optimal behavior. We also consider the problem of determining how much common memory a platform needs in order to be able to accommodate a specified real-time workload.**

***Index Terms*—Unrelated multiprocessors; memory-constrained scheduling; partitioned scheduling; approximation algorithms; bicriteria approximation.**

I. Introduction

Embedded real-time systems are often implemented upon heterogeneous hardware, consisting of several distinct processors each with its own instruction set. This trend towards platform heterogeneity is becoming more pronounced as real-time systems increasingly come to be implemented upon multicore platforms containing specialized hardware components that accelerate certain computations — examples include multicore CPUs with specialized graphics processing cores, signal-processing cores, floating-point units, customizable FPGAs etc. Such heterogeneous multiprocessor platforms may be modeled by the scheduling model called the *unrelated* multiprocessor machine model. In the unrelated multiprocessor machine model, each schedulable entity (job or task) is characterized by a different execution rate for each of the processors that comprise the platform.

The heterogeneous multiprocessor platforms upon which embedded real-time systems are implemented often possess a *shared pool of memory* in addition to the computing cores [2]. For instance, a modern multicore CPU typically contains some on-chip cache memory that is shared among the different processing cores of the CPU. In order to make best use of the capabilities of the multicore CPU, this common memory must be taken into consideration, along with the processing cores, during scheduling and resource allocation. In this paper, we consider the problem of scheduling real-time workloads upon unrelated multiprocessor platforms that contain several processing cores of different kinds as well as a shared pool of on-board memory, which is to be shared amongst all the processing cores. On such unrelated multiprocessors, we consider the following two problems:

1) Partitioning a given collection of *preemptive implicit-deadline sporadic tasks*, when each task needs exclusive access to a specified amount of the shared memory. The amount of memory needed by a task depends upon the processor to which the task gets assigned.
2) Partitioning a given collection of *independent preemptive jobs* that share a common release time and a common deadline, when each job only needs access to a share of the common memory pool while it is executing on its assigned processor. (An alternative model, which we also consider, has each job retaining the memory throughout the interval between the instants it begins and completes execution.)

We formally define each of these problems, demonstrate that they are intractable, and provide polynomial-time algorithms for solving them approximately with provably bounded deviation from optimality. To our knowledge, our efforts represent the first rigorous look at this problem of real-time scheduling on unrelated multiprocessors with memory issues also taken into consideration. Hence, in addition to the results we have obtained here, we consider the techniques we have developed to be important and interesting; we believe that they may be applicable to other scheduling problems on unrelated multiprocessors.

The remainder of this paper is organized as follows. In Section II, we formally define the two problems, provide some justification for why we believe they are worth studying, and summarize our results. We also describe some prior work in the scheduling literature, that we will be building upon. In Sections III and IV, we describe our findings with respect to the first and the second problem, respectively. In Section V we consider memory-aware scheduling problems from a different perspective: how much common memory must be made available in order to be able to guarantee the timeliness of the real-time workload that is to be implemented upon the platform? In Section VI we discuss the applicability of our techniques to some other real-time scheduling problems upon heterogeneous multiprocessors.

II. Models

Throughout this paper, let m denote the number of processors in the unrelated multiprocessor platform, and M the total size of the common pool of memory to be shared amongst these processors. Let n denote the number of jobs/tasks to be partitioned among these m processors.

A. **Problem 1: Memory-constrained recurrent tasks**

In this problem, we are given the specifications of n preemptable implicit-deadline tasks, and seek to determine a partitioning of these tasks among the m processors. The i'th task ($1 \leq i \leq n$) is characterized by a period T_i; m worst-case execution time (WCET) parameters $C_{i,1}, C_{i,2}, \ldots, C_{i,m}$; and m *memory requirement* parameters $m_{i,1}, m_{i,2}, \ldots, m_{i,m}$. Such a task generates an unbounded number of *jobs* during run-time, with successive jobs being released at least T_i time-units apart and with each having a deadline T_i time units after its release time. The interpretation of the WCET and memory requirement parameters is that if the i'th task is assigned to the j'th processor ($1 \leq j \leq m$), then each of its jobs may need to execute for up to $C_{i,j}$ time units, *and* an amount $m_{i,j}$ of the common memory pool must be reserved for the use of this task. Let $u_{i,j}$ denote the ratio $C_{i,j}/T_i$; we will refer to this ratio as the *utilization* of the i'th task on the j'th processor.

For this problem, a partitioning of the tasks to the processors is a function $\sigma : \{1, 2, \ldots, n\} \longrightarrow \{1, 2, \ldots, m\}$ with $\sigma(i) = j$ meaning that the i'th task is assigned to the j'th processor. We will assume that each individual processor is scheduled during run-time using the preemptive uniprocessor Earliest Deadline First (EDF) scheduling algorithm [1], [6]. It hence follows from results in [6] that a necessary and sufficient condition for an assignment of the tasks to the processors to be correct is that

$$(i) \quad \forall j : 1 \leq j \leq m : \quad \sum_{(i::\sigma(i)=j)} u_{i,j} \leq 1;$$

$$\text{and} \quad (ii) \quad \sum_{i=1}^{n} m_{i,\sigma(i)} \leq M.$$

B. **Problem 2: Memory-constrained jobs**

In this problem, we are given the specifications of n preemptable jobs and and seek to determine a schedule for these jobs on the m processors — as in the problem above, these jobs have both execution and memory requirements. The difference between this problem and the one above is that the memory requirements are not persistent; instead, *a job only needs to be assigned the memory while it is executing*. The i'th job ($1 \leq i \leq n$) is characterized by m WCET parameters $C_{i,1}, C_{i,2}, \ldots, C_{i,m}$; and m memory requirement parameters $m_{i,1}, m_{i,2}, \ldots, m_{i,m}$. All m jobs are assumed to be released at time-instant zero. The interpretation of the WCET parameters is that if the i'th job is assigned to the j'th processor ($1 \leq j \leq m$), then it may need to execute for up to $C_{i,j}$ time units. The interpretation of the memory requirement parameters is that if the i'th job is assigned to the j'th processor ($1 \leq j \leq m$), then an amount $m_{i,j}$ of the common memory pool is used by the i'th job *while it is executing*. Hence in a correct schedule at all time-instants t, the sum of the memory requirements of jobs that are executing at time t must not exceed M. The objective is to minimize the makespan of the schedule, where the *makespan* is defined to be the time-instant at which the last job completes its execution. (In another version of the problem, there is also a common deadline D specified for all the jobs, and the makespan of the schedule is required to be $\leq D$.)

Motivation for this problem. Our eventual interest is in the scheduling of collections of recurrent tasks of the kind addressed in Problem 1 (Section II-A), each of which can generate an infinite number of jobs with each job only needing memory during the times that it is executing. (An alternative possibility would have each job of each task require the memory between the instants it begins and completes execution, with the task releasing the memory between the completion of a job and the beginning of the execution of the subsequent job of the same task.) However, scheduling such recurrent task systems turns out to be an extremely difficult problem; hence, we have focused here on the simpler problem of scheduling non-recurring jobs — as will be seen in Section IV, we needed to develop fairly sophisticated analysis to analyze even this simpler problem. Our results may be considered as a first step towards a more comprehensive analysis of systems of recurrent tasks. In addition, these results have immediate applicability for the scheduling of certain kinds of *frame-based* recurrent real-time systems, in which the recurrent nature of the behavior is expressed as the infinite repetition of a finite collection of jobs of the kind considered here.

C. *Some prior results*

Let us give the name ***basic scheduling problem*** to the following classical problem for scheduling on unrelated multiprocessors. Given n jobs $\{1, \ldots, n\}$ and m machines $\{1, \ldots, m\}$, assign the jobs to the machines. When a job i is assigned to a machine j, it requires a processing time of $p_{i,j}$ — these $p_{i,j}$ parameters are provided as part of the specification of the problem instance. The objective is to assign the jobs in such a way that the makespan is minimized.

Several results are known if the $p_{i,j}$ values have some structure. If for each job i all values $p_{i,j}$ are equal (*identical* machines) the problem is already NP-hard, see Hochbaum and Shmoys [3]. These authors also considered the *approximation* version of this problem: an algorithm for solving this problem is said to be a k-approximation algorithm ($k \geq 1$) if it guarantees to find a schedule with makespan no more than k times the optimum and runs in polynomial time. In particular, they present a polynomial-time $(1 + \epsilon)$-approximation algorithm, for each $\epsilon > 0$. This result was later generalized to the setting of *related* ("uniform") machines [4], where each job i has a processing demand p_i and each machine j has a speed s_j. The processing time of a job i on a machine j is then given by p_i/s_j.

For the general problem (without assuming any specific structure on the $p_{i,j}$ values) Lenstra, Shmoys, and Tardos [5] have provided a 2-approximation algorithm, and have shown that it is NP-hard to obtain a k-approximation algorithm for any $k < 3/2$.

Shmoys and Tardos [7] subsequently studied a generalization of the basic scheduling problem, which we call the ***scheduling problem with costs***. Suppose that there is a specified *cost* $c_{i,j}$ associated with assigning the i'th job on the j'th processor, and the objective is to obtain a schedule of minimum makespan that has the minimum total cost. Since the basic scheduling problem is NP-hard, so is this generalization. However, Shmoys and Tardos were able to obtain an algorithm with polynomial running time that makes the following performance guarantee:

Theorem II.1 ([7]). *Consider an instance of the scheduling problem with costs for which there is a machine assignment with makespan at most T and total cost C. Let*

$$\tau := \max_{i,j\ :\ p_{i,j} \le T} \{p_{i,j}\}$$

denote the largest processing time of any job on any processor. Given the instance and the value of T as input, the polynomial time algorithm in [7] computes a machine assignment of makespan at most $T + \tau$ and cost at most C.

We now introduce some notation. A call to the algorithm of Theorem II.1 will be denoted $\mathbf{ST}(\{\mathbf{p_{i,j}}\}_{(\mathbf{n}\times\mathbf{m})}, \{\mathbf{c_{i,j}}\}_{(\mathbf{n}\times\mathbf{m})}, \mathbf{T})$. That is, the algoritm accepts as input the specification of an instance of the scheduling problem with costs as the $p_{i,j}$ and $c_{i,j}$ values, and an upper bound T on the makespan of a schedule for this instance. If there is indeed a schedule for this instance with makespan $\le T$, then $ST(\{p_{i,j}\}_{(n\times m)}, \{c_{i,j}\}_{(n\times m)}, T)$ returns an assignment of jobs to processors such that scheduling according to this assignment results in a makespan at most $(T + \tau)$ and cost equal to the cost of the minimum-cost optimal schedule with makespan T.

D. Our results

We obtain a polynomial time algorithm for the memory-constrained recurrent tasks problem (Section II-A) that makes the following guarantee. If there is a partitioning of the task system upon the given platform in which no task has a utilization more than U on the processor to which it was assigned, then our algorithm can partition the system upon a platform in which the capacity of each processor is inflated by a factor $(1 + U)$. (Alternatively, if our algorithm fails to partition the system upon the given platform, then there is no partitioning of the system upon a platform in which each processor has $(1 - U)$ times as much capacity, and in which no task has a utilization greater than U upon the processor to which it has been assigned.) This implies the approximation result that *our algorithm is able to schedule any system for which there exists an optimal schedule in which no processor is utilized for more than half its capacity.*

With regards to the memory-constrained jobs problem (Section II-B), our main result is a polynomial time algorithm that makes the following guarantee. If an optimal schedule of makespan T exists for the system, then our algorithm can generate a schedule of makespan $\rho{\cdot}T$, for ρ defined as follows:

$$\rho := \frac{1}{1 - \min(\frac{\mu}{M}, \frac{1}{2})} + \left(1 + \frac{\tau}{T}\right),$$

where $\mu := \max_{i,j}\{m_{ij}\}$ is the largest memory requirement, and $\tau := \max_{i,j}\{C_{ij}\}$ is the largest WCET. This implies the approximation result that *our algorithm is able to schedule any system with a makespan no more than four times the optimal* (a 4-approximation algorithm).

We also study *bicriteria* approximations for the memory-constrained jobs problem. Suppose that there is an optimal schedule with a makespan T when the platform has M units a memory. What if we could make more memory available? We present a polynomial-time algorithm which, if given access to αM units of memory for $\alpha \ge 1$, determines a schedule with a smaller makespan than ρT — Corollary IV.9 quantifies the tradeoff between the obtained makespan and the value of α. This tradeoff motivates our study of Pareto optimal pairs. A pair (T, M) is called *Pareto optimal* for a given instance of the scheduling problem if there is a schedule of makespan T on M units of memory, but decreasing either the makespan or the amount of available memory requires that the other parameter increases. Ideally, one would like to present the system designer the set of Pareto optimal pairs among which they can choose the pair that suits them best. Unfortunately, we show in Section V there are instances where the number of such pairs is exponential in the size of the instance. Hence it is computationally very costly to compute them all.

III. The memory-constrained recurrent tasks problem

The relationship between this problem and the scheduling problem with costs as studied by Shmoys and Tardos [7] is based on the following two observations. First, assigning jobs to processors to minimize makespan is equivalent to assigning implicit-deadline tasks to processors to minimize the cumulative utilization of the tasks that are assigned to each processor. Second, if we consider the amount of memory needed by a task when assigned to a processor to be the "cost" of that assignment, then the problem of determining whether the task system can be partitioned is equivalent to determining whether there is a partitioning in which the cumulative utilization assigned to each processor does not exceed one (since each processor is scheduled during runtime using preemptive EDF), and the minimum cost of the assignment is at most the amount of available memory.

We now describe the equivalence between these two problems more precisely. Recall that an instance of the problem of partitioning implicit-deadline tasks is given by the tasks i with their respective periods T_i, WCET values $C_{i,j}$, and memory requirements $m_{i,j}$, and the total amount of memory M.

Let $\ell_1, \ell_2, \ldots$ denote the distinct $u_{i,j}$ values which are $\leq \frac{1}{2}$ in decreasing order (i.e., $\ell_k > \ell_{k+1}$ for each k). For notational convenience, let us set $\ell_o \leftarrow 0.5$. Recall that $ST(\{p_{i,j}\}_{(n\times m)}, \{c_{i,j}\}_{(n\times m)}, T)$ denotes a call to an implementation of the procedure of Theorem II.1. Algorithm 1 describes an algorithm for attempting, in polynomial time, to partition the n tasks upon the m processors. The algorithm makes repeated calls to $ST(\{p_{i,j}\}_{(n\times m)}, \{m_{i,j}\}_{(n\times m)}, T)$, the implementation of the procedure of Theorem II.1.

Algorithm 1 Scheduling memory-constrained recurrent task systems

```
$k \leftarrow 0$
loop
    $\forall i \forall j \Big($ if $(u_{i,j} \leq \ell_k)$  $p_{ij} \leftarrow u_{ij}$; else $p_{ij} \leftarrow \infty \Big)$
    $\sigma \leftarrow ST(\{p_{i,j}\}_{(n\times m)}, \{m_{i,j}\}_{(n\times m)}, 1 - \ell_k)$
    if $\sigma$ is an assignment of cost $\leq M$ return $\sigma$
    if $\ell_k$ is the smallest utilization return failure
    $k \leftarrow k + 1$
end loop
```

We will first show that Algorithm 1 *runs in polynomial time*. As stated in Theorem II.1, each call to $ST(\{p_{i,j}\}_{(n\times m)}, \{m_{i,j}\}_{(n\times m)}, T)$ takes polynomial time. Since this call may be made no more than once for each distinct utilization value in the specification of the implicit-deadline partitioning problem instance, it is called no more than $n \cdot m$ times, thereby yielding an overall polynomial time bound.

First we argue that Algorithm 1 is *correct*: if it returns an assignment σ, then assigning the tasks to the processors according to this assignment yields a feasible solution to the implicit-deadline task partitioning problem. This is because the makespan of the partitioning computed by the Shmoys-Tardos algorithm is bounded by $T + \tau$, as stated in Theorem II.1. Here by construction we have $T = 1 - \ell_k$ and $\tau \leq \ell_k$ for some ℓ_k, which implies that the cumulative utilization of the tasks assigned to each processor is bounded by 1.

Next, we show that Algorithm 1 is *effective*, in the sense that it makes the following guarantee:

Theorem III.1. *If Algorithm 1 fails to partition an instance of the memory-constrained recurrent tasks problem, then for all $U \geq 0$ there is no partitioning of the tasks upon a platform in which each processor has $(1 - U)$ times as much capacity, and in which no task has a utilization greater than U upon the processor to which it has been assigned.*

Proof: Let us suppose that the theorem is false. That is, Algorithm 1 fails to partition some instance but there is a U' such that (i) the system can be partitioned using only a fraction $(1 - U')$ of each processor's capacity, and (ii) in this partitioning, no task has a utilization greater than U' upon the processor to which it has been assigned. Without loss of generality, we can assume that $U' = \ell_{k'}$ for some k', as otherwise we can round down U' to the next ℓ_k value.

Consider the iteration of the loop in Algorithm 1 when k has the value k' such that $\ell_{k'} = U'$ (since we assume that Algorithm 1 fails, it will have iterated through every distinct utilization value for ℓ_k prior to declaring failure, and hence this value as well). We have assumed that the system can be partitioned using only a fraction $(1 - U')$ of each processor's capacity, with no task having a utilization on its assigned processor greater than U'. Therefore, setting $p_{ij} \leftarrow u_{ij}$ for all $u_{ij} \leq U'$ and $c_{ij} \leftarrow m_{ij}$ yields an instance of the scheduling problem with costs which can be partitioned on capacity $(1 - U')$-processors with total cost $\leq M$. According to Theorem II.1, $ST(\{p_{i,j}\}_{(n\times m)}, \{c_{i,j}\}_{(n\times m)}, (1-\ell_k))$ with these p_{ij} and c_{ij} values yields a partitioning of cost $\leq M$, in which no processor is used for more than a fraction $(1 - U') + \max_{i,j}\{p_{ij}\}$ which is ≤ 1. ■

Now suppose that a task system can be partitioned upon a platform with total memory M, in which each processor has capacity equal to $1/2$. According to Theorem II.1, the first iteration of the loop in Algorithm 1 would yield a valid partitioning. We therefore conclude the following corollary.

Corollary III.2. *If Algorithm 1 fails to partition a given instance upon a particular platform, then the instance cannot be partitioned upon a platform in which the capacity of each processor is halved.*

Assume now that there exists a feasible solution for the given instance. If we divide each utilization $u_{i,j}$ of each job i on each processor j by two before executing the algorithm, the first loop will return a feasible schedule on a platform in which each processor has twice the capacity. This implies the following corollary.

Corollary III.3. *Consider an instance of the memory-constrained recurrent tasks problem can be partitioned upon a particular heterogeneous multiprocessor platform. Then with Algorithm 1 we can compute in polynomial time a valid partitioning for a platform with the same amount of common memory, in which each processor has twice the capacity.*

IV. The memory-constrained jobs problem

Recall that in an instance of this problem the i'th job ($1 \leq i \leq n$) is characterized by m WCET parameters $C_{i,1}, C_{i,2}, \ldots, C_{i,m}$, and m memory requirement parameters $m_{i,1}, m_{i,2}, \ldots, m_{i,m}$. In addition, a deadline D for all the jobs may be specified. Since all the jobs will have the same deadline D if this is specified, scheduling to meet this deadline is equivalent to obtaining a schedule that has makespan $\leq D$. Therefore for ease of presentation, henceforth for the most part *we will ignore the deadline D even if specified, and instead consider the problem as one of makespan minimization.*

For the problem considered in Section III above, it is sufficient to define a solution to the problem as the assignment σ of tasks to processors. Since the tasks require that memory is allocated for their exclusive use throughout the duration of the system execution, the precise manner in which the different processors are scheduled over time is not relevant, and run-time dispatching decisions on each processor can be made independently.

For the problem we consider in this section, this is different. Recall that jobs only require memory while they are executing; hence the platform-wide requirement is that at any time t during run-time, the sum of memory requirements of the jobs executing at time t may not exceed M. We therefore need a more detailed notion of a schedule, for which we introduce the following additional notation. A schedule S is a mapping from jobs to processor-time ordered pairs

$$S : \{1, 2, \dots, n\} \longrightarrow \{1, 2, \dots, m\} \times \mathbb{R}^{\geq 0}$$

The interpretation is that if $S(i) = (j, t)$, then the i'th job will execute on the j'th processor, starting at time t and occupying the processor for the time interval $[t, t + C_{ij})$.[1] We say that the *finishing time* of job i is $t + C_{ij}$. We call a job *active* at time t, if it is currently processed at that time, i.e. it was started at time t' and $t \in [t', t' + C_{ij})$.

We now present an approximation algorithm for computing a schedule for an input instance of the memory-constrained jobs scheduling problem. The algorithm operates in two phases. In the first phase, an assignment σ of jobs to processors is determined; in the second, the actual schedule S is constructed. For simplicity we assume that the processing times are integer, i.e. $C_{i,j} \in \mathbb{N}$ for all $1 \leq i \leq n, 1 \leq j \leq m$.

A. Phase 1

To compute an assignment of jobs to processors, we will once again use the result of Shmoys and Tardos [7] presented in Theorem II.1 above, by mapping our problem to the problem of scheduling with costs. Recall that in the problem of scheduling with costs, there is a processing time $p_{i,j}$ and a cost $c_{i,j}$ specified for each job-processor pair i, j. In mapping an instance of our problem of scheduling memory-constrained jobs to an instance of the problem of scheduling with costs, the parameters for the problem of scheduling with costs are obtained as follows: For each i, j,

- $p_{i,j} \leftarrow C_{i,j}$ (i.e., the *processing times* are the specified WCETs), and
- $c_{i,j} \leftarrow m_{i,j} \cdot C_{i,j}$ (i.e., the *costs* are the product of the corresponding memory requirement and the WCET).

Informally speaking, the rationale for defining the costs in this manner is as follows. If the i'th job is assigned to the j'th processor, then it needs $m_{i,j}$ amount of memory for a total duration of $C_{i,j}$ time units: the memory-time product, or *memory consumption*, for this particular assignment decision is equal to $m_{i,j} \cdot C_{i,j}$. Now since the total amount of memory available is M, a lower bound on the makespan T of the schedule is given by C/M, where C denotes the sum of the memory consumptions of all the job-processor pairs i, j such that the i'th job is assigned to the j'th processor. In order to obtain a small makespan we would therefore like to minimize the sum of the memory consumptions of all the job-processor pairs that comprise the partitioning. This, in essence, is why we have defined the *cost* parameters $c_{i,j}$ to equal the memory consumptions — we seek to limit the total memory consumption in order to be able to compute a good schedule in the second phase. We can also reverse the argument from above: If there is a schedule for the job partitioning problem of makespan T, then its total memory consumption is at most $M \cdot T$. We conclude:

Lemma IV.1. *If there is a schedule of makespan T for the memory-constrained job partitioning problem, then there is a solution to the scheduling problem with costs of makespan T and cost $\leq M \cdot T$.*

Combining this lemma with Theorem II.1, we get:

Corollary IV.2. *If there is a schedule of makespan T for the memory-constrained job partitioning problem, the Shmoys-Tardos algorithm determines a partitioning of the jobs on the processors with makespan at most $(T + \max_{i,j}\{C_{i,j}\})$ and total cost at most $M \cdot T$.*

Let us suppose for now that we know an upper bound, denoted T, on the makespan of a schedule for the jobs in the memory-constrained jobs scheduling problem. Let $\sigma \leftarrow ST(\{p_{i,j}\}_{(n\times m)}, \{c_{i,j}\}_{(n\times m)}, T)$ denote the job to processor assignment obtained according to Corollary IV.2, of total cost $\leq MT$ and makespan at most $(T + \max_{i,j}\{C_{i,j}\})$; here $\sigma(i)$ denotes the processor upon which the i'th job is assigned for all i, $1 \leq i \leq n$.

Binary search to determine makespan. Above, we had assumed that the upper bound T on the makespan is known. We now describe how this is determined. We use a binary search to determine the smallest value T such that $ST(\{p_{i,j}\}_{(n\times m)}, \{c_{i,j}\}_{(n\times m)}, T)$ finds an assignment. If the common deadline D is specified then since a makespan greater than D would not yield a feasible schedule, D is an obvious upper bound on the value of the T parameter we are interested in. If D is not specified, then an upper bound on the makespan is given by $\sum_{i=1}^{n} \min_j\{C_{i,j}\}$ (this being the makespan if only one job is executed at a time, upon the processor on which it executes the fastest). By standard binary search we can therefore obtain, in a polynomial number of calls, a bound on the makespan that is exponentially close to the optimal value. (We can in fact terminate the binary search procedure once the difference between the values of T used in successive calls is $\leq \min_{i,j} C_{i,j}$; under reasonable assumptions on the values of the $C_{i,j}$'s, it can be shown that this will happen within polynomially many calls.)

B. Phase 2

We now describe how to generate a schedule from the job to processor assignment σ. To simplify notation, we define processing times and memory requirements for each job as implied by the job to processor assignment σ determined in

[1]Note that we are thus restricting ourselves to *non*preemptive scheduling only, even if the model allows for preemption. However, we will show that we do not pay too significant a price for this restriction, by comparing our non-preemptive algorithm to an optimal one that may be preemptive.

Phase 1:

$$p_i := C_{i,\sigma(i)} \qquad m_i := m_{i,\sigma(i)} \quad \forall i, 1 \leq i \leq n.$$

The schedule is generated using a greedy list scheduling approach. We sort the jobs non-increasing by their memory requirement m_i and create an ordered list

$$\mathcal{L} = (j_1, j_2, \dots, j_n) \tag{1}$$

of all the jobs with the property that for any two jobs j_a, j_b with $a < b$ we have $m_{j_a} \geq m_{j_b}$. The list defines job priorities for our phase-two algorithm, with the leftmost entry (i.e. the one with the highest memory requirement) having greatest priority and the rightmost entry having least priority. The list scheduler always assigns the job i with the greatest priority that is ***available*** at given time t, where availability is determined by the following three conditions:

1) The processor $\sigma(i)$ that the job is assigned to is not processing any other job at time t.
2) Its memory requirement m_i does not exceed the free memory, i.e. the memory unoccupied by other active jobs.
3) The job i has to be ***ready***. We call a job ready at time t, if every job i' with $\sigma(i) = \sigma(i')$ and $m_{i'} > m_i$ has finished at time t.

The full algorithm is presented in Listing 2.

Algorithm 2 Scheduling memory-constrained jobs

```
{Phase 1: partition the jobs, assuming minimum makespan $T$ is known}
$\sigma \leftarrow ST(\{C_{i,j}\}_{(n\times m)}, \{m_{i,j} \times C_{i,j}\}_{(n\times m)}, T)$
{Phase 2: generate the schedule}
$\mathcal{L} \leftarrow (j_1, j_2, \dots, j_n)$ {as defined in (1)}
$t \leftarrow 0$
while $\mathcal{L}$ has unscheduled jobs do
  for $\ell = 1$ to $n$ do
    if $j_\ell$ is unscheduled and available at time $t$ then
      Schedule job $j_\ell$ on processor $\sigma(j_\ell)$ at time $t$: $S(j_\ell) \leftarrow (\sigma(j_\ell), t)$
    end if
  end for
  $t \leftarrow$ next finishing time of a job.
end while
```

C. Algorithm 2: properties

We first show that Algorithm 2 *runs in polynomial time*. As listed, Phase 1 assumes that the value of T is known and makes only one call to the procedure of Theorem II.1. But as we had shown above, binary search for the appropriate value may require polynomially many calls to this polynomial-time procedure of Theorem II.1. For the second phase, the **while** loop is executed at most once for each job completion, and hence n times; during each iteration, the inner **for** loop can clearly be implemented to execute in polynomial time.

It is obvious that Algorithm 2 is *correct*: since a job is deemed available only if there is sufficient memory for it to execute, the schedule generated by Algorithm 2 respects memory constraints and is therefore a correct schedule.

Finally, we will show that Algorithm 1 is *effective*. Specifically we will prove that the schedule S generated by Algorithm 2 is a 4-approximation with regards to makespan: the makespan of S is no more than 4 times the minimum makespan of any feasible schedule for the problem instance. We start with a basic observation regarding the structure of this schedule:

Lemma IV.3. *If at any time t a job i has not yet completed execution in the schedule S generated by Algorithm 2, then some job of memory requirement at least m_i is active.*

Proof: For simplicity of presentation we assume that all processing times p_i are integer, which can be achieved by scaling all processing times with a suitable common factor (the least common denominator). Then it suffices to show the statement for integer t and we prove it by induction on $t \in \mathbb{N}_0$.

For the base case of $t = 0$, observe that our scheduling algorithm at time $t = 0$ always picks the first item from the list $\mathcal{L}$ and schedules it, i.e. this job becomes active at time 0. As the first item is the job with the biggest memory requirement, the statement of the lemma is therefore true for $t = 0$.

Now assume that the statement holds for time t, and we want to prove it for time $t + 1$. Consider a job i that is unfinished at time $t + 1$. We can assume that job i is ready at time $t + 1$, as otherwise there would be a job i' assigned to the same machine with $m_{i'} \geq m_i$ that is ready at time $t + 1$. Observe that it is then sufficient to show that the statement holds for i'.

Hence i is ready at time $t + 1$. We distinguish two cases. The first case is that no job of memory requirement at least m_i finishes at time $t + 1$. Then the statement is trivially true for $t + 1$ as all jobs with bigger memory requirement that where active at time t are also active at time $t + 1$. The second case is that a job i' with $m_{i'} \geq m_i$ finishes at time $t + 1$. Because i is ready, the scheduler will pick i or a job with higher memory requirement to start processing at time $t + 1$, which will become active. Hence the statement is true. ∎

Recall that $\mu = \max_{i,j}\{m_{ij}\}$ and $\tau = \max_{i,j}\{C_{ij}\}$ denote the largest memory requirement and the largest WCET respectively. Let $\bar{\mu}$ denote the largest memory requirement, upper bounded by $\frac{M}{2}$, i.e. $\bar{\mu} := \min\{\mu, \frac{M}{2}\}$. The following lemma reveals an important structural property of the generated schedule. Once the total memory requirement of the schedule drops below $M - \bar{\mu}$, it will never rise above this threshold again.

Lemma IV.4. *Consider the schedule generated by the algorithm. Let*

$$\hat{t} := \min \left\{ t \ : \ \sum_{i \in \mathcal{J}(t)} m_i < M - \bar{\mu} \right\}. \tag{2}$$

Then $\sum_{i\in\mathcal{J}(t)} m_i < M - \bar{\mu}$ for all $t \geq t'$.

Proof: For each time t, define $\mathcal{J}(t)$ as the set of jobs that are active at time t and $\mathcal{P}(t)$ as the set of processors that are executing some job at time t. Let t' be a time with $t' > \hat{t}$. We show that the memory requirement at that time is less than $M - \bar{\mu}$.

First consider the case that $\mathcal{P}(t') \subseteq \mathcal{P}(\hat{t})$. In this case for each job $i' \in \mathcal{J}(t')$ there is a distinct job $i \in \mathcal{J}(\hat{t})$ satisfying $\sigma(i) = \sigma(i')$. Moreover due to the precedence constraints we enforced among jobs assigned to the same machine, for two jobs processed on the same machine the one with the higher memory requirement is always scheduled before the other. Hence we have $m_{i'} \leq m_i$. This implies that

$$\sum_{i\in\mathcal{J}(t')} m_i \leq \sum_{i\in\mathcal{J}(\hat{t})} m_i < M - \bar{\mu}.$$

Now assume that $\mathcal{P}(t') \not\subseteq \mathcal{P}(\hat{t})$. Hence there is a processor $j \in \mathcal{P}(t')$ that is idle at time $\hat{t}$ but processing a job at time t'. This implies that a job i with $\sigma(i) = j$ was ready at time $\hat{t}$ but has not been scheduled due to memory restrictions. As at least a $\bar{\mu}$ portion of the memory was free at time $\hat{t}$, this implies $m_i \geq \bar{\mu}$. By definition of $\bar{\mu}$ this implies $m_i \geq \frac{M}{2}$ and $\bar{\mu} = \frac{M}{2}$. On the other hand, as job i is is not finished at time $\hat{t}$, Lemma IV.3 states that there is a job i' active at time $\hat{t}$ with $m_{i'} \geq m_i \geq \frac{M}{2}$. Hence the memory load at time $\hat{t}$ is at least $m_{i'} \geq \frac{M}{2} = M - \bar{\mu}$ in contradiction to the definition of $\hat{t}$. This finishes the proof. ∎

Lemma IV.5. *Let $\hat{t}$ be as defined by Equation 2 of Lemma IV.4. Then for any $t \geq \hat{t}$ a processor is idle only if all the jobs that were assigned to it have completed execution.*

Proof: We first prove that all jobs with memory requirement of more than $\frac{M}{2}$ are finished at time $\hat{t}$. This is due to the fact that if there is a job i with $m_i > \frac{M}{2}$, then $\bar{\mu} = \frac{1}{2}$. Thus whenever i is active, the memory load is at least $\frac{M}{2} = M - \bar{\mu}$. Thus the claim follows with Lemma IV.4.

Now let $t \geq \hat{t}$ and consider a processor j that is idle at time t but has a ready job i. As seen above, we have $m_i \leq \frac{M}{2}$. Hence $m_i \leq \bar{\mu}$. Lemma IV.4 states that at least $\bar{\mu}$ memory is free at time t. Hence the algorithm would have scheduled j. A contradiction. ∎

We are now ready to prove the main theorem.

Theorem IV.6. *Algorithm 2 is a ρ-approximation with regards to makespan, where*

$$\rho := \frac{1}{1 - \min(\frac{\mu}{M}, \frac{1}{2})} + \left(1 + \frac{\tau}{T}\right)$$

Proof: As described above, assume T to be the minimum value such that $ST(\{C_{i,j}\}_{(n\times m)}, \{m_{i,j} \times C_{i,j}\}_{(n\times m)}, T)$ returns a partitioning (with total cost $\leq MT$). Let $\hat{t}$ be as in Lemma IV.4. The lemma states that for each $t = 0, \ldots, \hat{t} - 1$, the memory consumption is at least $M - \bar{\mu}$. On the other hand, we have seen in Lemma IV.1 that the total memory consumption of all tasks is at most $M \cdot T$. Hence $\hat{t} \cdot (M - \bar{\mu}) \leq M \cdot T$ holds which implies

$$\hat{t} \leq \frac{1}{1 - \frac{\bar{\mu}}{M}} T.$$

After time $\hat{t}$, Lemma IV.5 shows that the memory constraints are not a limitation anymore. Hence, an upper bound for the makespan of the schedule starting at time $\hat{t}$ is given by the makespan of the Shmoys-Tardos schedule which is $T + \tau$. Therefore, the total makespan of our schedule is no more than $(\hat{t} + T + \tau)$, or

$$\left(\frac{1}{1 - \min(\frac{\mu}{M}, \frac{1}{2})} + \left(1 + \frac{\tau}{T}\right)\right) \cdot T$$

which shows the claim. ∎

As trivially we have $\tau \leq T$ and $\min(\frac{\mu}{M}, \frac{1}{2}) \geq \frac{1}{2}$, we get the following corollary.

Corollary IV.7. *The algorithm is a 4-approximation with regards to makespan.*

Under the (admittedly not always valid) "speedup assumption" that the WCET of any job is scaled down by a factor s on processors that are faster by a factor s, the implication of Theorem IV.6 and Corollary IV.7 is that any instance that is feasible on a given unrelated multiprocessor platform is successfully scheduled in polynomial time by our Algorithm 2 on a platform with the same amount of memory and in which each processor is ρ times as fast; furthermore, $\rho \leq 4$.

Preemption: The schedule generated by Algorithm 2 does not have any preemptions even if the workload model allows for preemption. However, observe that in an optimal partitioned schedule of independent jobs (ignoring memory constraints), no preemption is needed to minimize makespan — simply execute the jobs assigned to each processor in a work-conserving fashion on that processor. As a consequence, preemptive optimal schedules will not have a smaller makespan than the non-preemptive optimal schedules assumed in the derivation of Theorem IV.6 and Corollary IV.7. We conclude that the approximation factors of Theorem IV.6 and Corollary IV.7 are valid with respect to optimal preemptive schedules as well.

A different memory model. Thus far, we have assumed that each job needs memory only when it is executing. As stated in Sections I and II, an alternative model would have each job retain access to the memory throughout the interval between the instants it begins and completes execution. (This latter model may be the more realistic one if, for instance, the shared memory is a cache and we seek to minimize cache overhead by letting a job hold on to its share of the cache even upon being preempted, until it completes execution.) Although these two models are different, we note that this difference is not relevant to non-preemptive schedules since a job executes continually between the instants it begins and completes execution in such a schedule. Hence, the results we have obtained above are equally valid for this memory model as well – in particular, the approximation factors of Theorem IV.6 and Corollary IV.7 remain valid.

D. Bicriteria approximation

The analysis in Section IV-C above had obtained an upper bound on the multiplicative factor by which the makespan of the schedule generated by Algorithm 2 on a given platform may exceed the makespan of the schedule generated by an optimal algorithm on the same platform, for any problem instance.

In a ***bicriteria*** generalization of this optimization problem, we may seek to determine what the smallest makespan is that can be generated in polynomial time, if the polynomial-time algorithm were to have access to *more memory* than is available to the optimal algorithm. That is, let us consider an instance of the memory-constrained jobs problem for which the optimal makespan is equal to T, using the available amount of memory M. Suppose that the amount of memory available to the polynomial-time algorithm is αM, for some constant $\alpha \geq 1$. We will modify Algorithm 2 to obtain an algorithm that uses this available memory and obtains a schedule of makespan $\leq \beta T$, where β is defined by

$$\beta = \begin{cases} 1 + \frac{\tau}{T} + \frac{2}{\alpha}, & \text{if } 1 \leq \alpha < 2 \\ 1 + \frac{\tau}{T} + \frac{1}{\alpha - 1}, & \text{if } \alpha \geq 2. \end{cases}$$

Using binary search, we first determine the smallest value T such that $ST(\{C_{i,j}\}_{(n\times m)}, \{m_{i,j} \times C_{i,j}\}_{(n\times m)}, T)$ returns a solution. As mentioned above, the computed assignment of jobs to processors has a makespan of at most $T + \max_{i,j}\{C_{i,j}\}$. Then we run Phase 2 as described in Section IV-B but with an available memory of αM rather than M. Having more memory available results in a different schedule than with only a memory of size M. In the following theorem we bound the makespan of the schedule computed by this algorithm which we denote by Algorithm 3.

Theorem IV.8. *If for an instance there is a schedule with makespan T using an available amount of memory M then Algorithm 3 computes a schedule with a makespan of at most*

$$\left(\frac{1}{\alpha - \min\left\{\frac{\mu}{M}, \frac{\alpha}{2}\right\}} + \left(1 + \frac{\tau}{T}\right)\right) \cdot T$$

which uses αM units of memory.

Proof: In the analysis of Algorithm 2 the value $\bar{\mu}$ was defined by $\bar{\mu} := \min\{\mu, \frac{M}{2}\}$. For our purposes here we define $\bar{\mu} := \min\{\mu, \frac{\alpha M}{2}\}$. Analogous to the proof of Lemma IV.4 we can show for $\hat{t} := \min\left\{t \ : \ \sum_{i\in\mathcal{J}(t)} m_i < \alpha M - \bar{\mu}\right\}$ that $\sum_{i\in\mathcal{J}(t)} m_i < \alpha M - \bar{\mu}$ for all $t \geq t'$. Since $\hat{t} \cdot (\alpha M - \bar{\mu}) \leq M \cdot T$ we reason that $\hat{t} \leq \frac{M\cdot T}{\alpha M - \bar{\mu}}$. Like in Theorem IV.6, after time $\hat{t}$ the memory constraint is not a limitation anymore. Hence, the makespan of the overall schedule is bounded by $(\hat{t}+T+\tau)$, or equivalently the expression in the theorem statement. By construction, at each point in time the constructed schedule uses at most αM units of memory. ■

Using the fact that $\mu \leq M$ we can simplify the bound stated in Theorem IV.8.

Corollary IV.9. *If for some instance there is a schedule with makespan T using an available amount of memory M then Algorithm 3 computes a solution with a makespan of $\leq \beta T$ with*

$$\beta = \begin{cases} 1 + \frac{\tau}{T} + \frac{2}{\alpha}, & \textit{if } 1 \leq \alpha < 2 \\ 1 + \frac{\tau}{T} + \frac{1}{\alpha - 1}, & \textit{if } \alpha \geq 2, \end{cases}$$

using at most αM units of memory.

Proof: Using that $\mu \leq M$ we obtain a bound of

$$\left(\frac{1}{\alpha - \min\left\{1, \frac{\alpha}{2}\right\}} + \left(1 + \frac{\tau}{T}\right)\right) \cdot T$$

which yields the bounds stated in the corollary for the two relevant cases for α. ■

V. Provisioning shared memory

In both of Sections III and IV, we have considered *scheduling* problems: given the specifications of an instance to be scheduled upon a particular unrelated multiprocessor platform with a fixed amount of shared memory M, the goal was to determine a correct schedule.

In designing embedded devices, however, it is often the case that the hardware platform is not a priori fixed; rather, the design process for the device involves exploring the space of possible hardware platforms upon which a desired set of functionalities can be implemented in a cost-effective manner. This is essentially the converse of the question we have been asking in Sections III and IV: rather than being given the platform and seeking a correct schedule for a given problem instance, we now seek to know what the parameters of the platform need to be, in order to be able to achieve a correct schedule. If there are multiple such platforms possible (and in general, there will be), we would like to be able to present these alternatives to the system designers and give them the freedom to choose from among these different viable alternatives.

We formalize this somewhat fuzzy notion of *different viable alternatives* for the memory-constrained jobs problem in the concept of ***Pareto-optimal memory-makespan pairs***; a similar formalization can be defined for the memory-constrained tasks problem.

Definition V.1. *An ordered pair (T, M) of positive integers is said to be a Pareto-optimal memory-makespan pair for a given instance of the memory-constrained jobs problem if there is a feasible schedule for this instance with makespan at most T and memory requirement at most M, and there are no feasible schedules with a makespan $\leq T$ and memory requirement $< M$ or with a makespan $< T$ and memory requirement $\leq M$.*

Informally speaking, Pareto-optimal memory-makespan pairs represent "best" design choices with regards to the platform for implementing the instance. If there are multiple Pareto-optimal memory-makespan pairs for a given instance, we would like to be able to present all these choices to the system designers, letting them base their final choice on other

		processors					
		0	1	2	$\cdots$	$n-1$	n
	0	$(M, 2^0)$			$\cdots$		$(2^0, 0)$
	1		$(M, 2^1)$		$\cdots$		$(2^1, 0)$
jobs	2			$(M, 2^2)$	$\cdots$		$(2^2, 0)$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		$\vdots$
	$n-1$					$(M, 2^{n-1})$	$(2^{n-1}, 0)$
	n						$(M, 0)$

TABLE I

THE PAIRS (p, m) INDICATE THE PROCESSING TIME AND THE MEMORY REQUIREMENT OF THE RESPECTIVE JOB ON THE GIVEN PROCESSOR. TO SIMPLIFY THE TABLE, WE LEFT ALL ENTRIES BLANK WHERE THE RESPECTIVE JOB HAS INFINITE PROCESSING TIME ON THE GIVEN PROCESSOR.

factors. Unfortunately, it turns out that this is not really a reasonable option: there are, in general, too many Pareto-optimal memory-makespan pairs for a given problem instance. This is formalized in the following theorem.

Theorem V.2. *For each $n \in \mathbb{N}$, there is an instance with $(n+1)$ jobs and processors but 2^n Pareto optimal points. Hence, there is a family of instances whose number of Pareto optimal points is exponential in the size of the instances.*

Proof: We define our family of instances with exponentially many Pareto optimal points. Let $n \in \mathbb{N}$. Our instance has $n+1$ jobs and $n+1$ processors, the jobs and processors being numbered from 0 to n. Let $M \in \mathbb{N}$ be a sufficiently large integer with $M > 2^n$. For $i = 0, \ldots, n-1$, job i has an execution time of M on processor i, execution time 2^i on processor n and execution time ∞ on all other processors. Moreover, job i has memory requirement 2^i on processor i and 0 on all other processors. Finally, job n can only be assigned to processor n with execution time M and memory requirement 0. See Table I for an overview of the pairs of processing time and memory requirement.

Let $k \in \{0, 1, 2, \ldots, 2^n - 1\}$. Let $(k)_2$ be the binary representation of k. Let $(k)_2^j$ be the j-th bit in $(k)_2$ (counted from least value to highest value, the smallest index being 0). Let $S_k := \{j : (k)_2^j = 1\}$, i.e. S_k is the set of bit-indexes which are set to 1 in k. Note that $S_k \subseteq \{0, \ldots, n-1\}$. In the sequel, we will identify S_k with a subset of the jobs.

The idea behind this construction is as follows: We want to create a schedule whose makespan is determined by the execution time of the jobs on processor n. If we decide *not* to put job i on processor n, the only other feasible choice is to put it to processor i. Since job n can be scheduled only on processor n, the makespan of n is $M + \sum_{i \in J(n)} 2^i$ where $J(n)$ denotes the jobs which are assigned to processor n. If M is large enough, in every schedule with that makespan the jobs assigned to processors other than n will always overlap. Therefore, the total memory requirement will be $\sum_{i \in \{0,1,\ldots,n\} \setminus J(n)} 2^i$. Hence, the amount of time we save for the makespan when putting a subset of jobs to processors other than n *equals* the memory requirement. This gives us a different Pareto optimal pair for each subset of jobs we decide not to put to processor n.

Now we prove the above statements formally. As mentioned above, the binary representation of every integer $k \in \{0, ..., 2^n - 1\}$ encodes a feasible schedule using k units of memory: For each $i \in S_k$, schedule job i on processor i at time 0. All other jobs are scheduled on processor n in arbitrary order. This results in a schedule with makespan $M + 2^n - 1 - k < 2M$ since

$$\sum_{i \notin S_k} p_{n,i} = M + \sum_{\substack{i \notin S_k \\ 0 \le i < n}} 2^i = M + 2^n - 1 - k.$$

The memory requirement equals k since

$$\sum_{i \in S_k} m_{i,i} = \sum_{i \in S_k} 2^i = k.$$

Conversely, given a schedule with a makespan less than $2M$ which needs a memory of size exactly $k \in \{0, ..., 2^n - 1\}$. We choose M large enough such that this implies that at some point in the schedule every processor $i \in S_k$ executes the respective job i. Note that by construction there is no other way of obtaining a memory usage of exactly k. Since the makespan is less than $2M$ all jobs $i \notin S_k$ are scheduled on processor n. Hence, the makespan of the schedule equals $\sum_{i \notin S_k} p_{n,i} = M + 2^n - 1 - k$. This implies that there can be no schedule using k units of memory with a smaller makespan than $M + 2^n - 1 - k$. Hence, the pair $(M + 2^n - 1 - k, k)$ is Pareto optimal. For the described instance, there are 2^n values k. Therefore, the described family of instances has an exponential number of Pareto optimal points. ■

VI. RECURRENT TASKS; ADDITIONAL OPTIMIZATION CRITERIA

As we had stated in Section II-B, our ultimate interest is in scheduling recurrent task systems, rather than task systems consisting of a finite number of jobs with a common deadline. We are working on extending the techniques we had introduced in Section IV from the analysis of jobs to the analysis of systems of recurrent tasks.

Although we do not yet have complete results for scheduling such recurrent tasks under the memory model assumed in the memory-constrained jobs problem, we would like to point out that the techniques we had introduced in Sections IV and V can be applied to some *other* bicriteria optimization problems concerning preemptive recurrent task systems. Consider some system of implicit-deadline sporadic tasks to be partitioned among the processors of an unrelated multiprocessor platform that also includes some additional resource that gets consumed at a constant rate $r_{i,j}$ whenever a job of task i is executing on the processor j to which it has been assigned, and that is being replenished platform-wide at a constant rate as well. (Under some grossly simplifying assumptions, for example, this resource could be *energy*: while executing, different tasks consume energy at different rates on different processors.) This problem could be dealt with in a manner similar to the way we have handled our problem of scheduling memory-constrained jobs: we could map this problem on to the scheduling problem with costs by setting $p_{i,j} := u_{i,j}$ and $c_{i,j} := u_{i,j} \times r_{i,j}$ (here u_{ij} denotes the utilization of the i'th task if implemented on the j'th processor, as stated in Section II-A), and then applying the approximation algorithm of [7] to obtain a 2-approximation.

VII. Conclusion

The accelerating trend in embedded systems design towards the use of heterogeneous multicore CPU's calls for corresponding advances in multiprocessor real-time scheduling theory. In this work, we have begun studying the problem of partitioning real-time workloads on heterogeneous multiprocessors that share a pool of common memory. We have formalized several interesting aspects of such problems, and have obtained some novel results concerning approximate scheduling in such systems. As important, we have identified some powerful techniques from the general scheduling literature, and have introduced some new techniques of our own, for dealing with the multi-criteria optimization problems that arise when doing real-time scheduling on heterogeneous multiprocessor platforms.

References

[1] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[2] F. T. Hady, M. B. Cabot, J. Beck, and M. Rosenbluth. Heterogeneous processors sharing a common cache. United States Patent 7,577,792. Assignee: Intel Corporation, 2009.

[3] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[4] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing*, 17:539–551, 1988.

[5] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *Mathematical Programming 46*, pages 259–271, 1990.

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

[7] D. B. Shmoys and É. Tardos. Scheduling unrelated machines with costs. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1993)*, pages 448–454. ACM, 1993.