Task partitioning upon memory-constrained multiprocessors [*]

Nathan Fisher James H. Anderson Sanjoy Baruah
Department of Computer Science
The University of North Carolina at Chapel Hill
{fishern, anderson, baruah}@cs.unc.edu

Abstract

Most prior theoretical research on partitioning algorithms for real-time multiprocessor platforms has focused on ensuring that the cumulative computing requirements of the tasks assigned to each processor does not exceed the processor's processing power. However, many multiprocessor platforms have only limited amounts of local per-processor memory; if the memory limitation of a processor is not respected, thrashing between "main" memory and the processor's local memory may occur during run-time and may result in performance degradation. We formalize the problem of task partitioning in a manner that is cognizant of both memory and processing capacity constraints as the memory constrained multiprocessor partitioning problem, *prove that this problem is intractable, and present efficient algorithms for solving it under certain – well-defined – conditions.*

Keywords: *Multiprocessor systems; Partitioned scheduling; Memory-constrained systems; Utilization-based schedulability tests.*

1 Introduction

As the functionality demanded of real-time embedded systems has increased, it is becoming unreasonable to expect to implement them upon uniprocessor platforms [19]; hence, multiprocessor platforms are increasingly used for implementing such systems. Efficient system implementation on such multiprocessor platforms may require the careful management of several key resources, such as processor capacity, memory capacity, communication bandwidth, etc. For instance, in assigning tasks to processors care must be taken to ensure that both the limited computing capacity of a processor *and* its limited local memory is taken into consideration.

Consider as an example the IXP 2800, which is the high-end member of the Intel IXP family of programmable *network processors* (NPs). Such NPs offer an alternative to conventional ASIC (application-specific integrated circuit) designs for deploying customized functions, such as firewalls, intrusion detection, load balancing, virtual private networks, protocol conversions, etc., in switches and routers on the Internet. The 2800 contains an XScale (ARM architecture) core processor, sixteen independent RISC CPUs called *microengines* (MEs), interface controls for access to off-chip SRAM and DRAM, and standard interfaces to media or a switch fabric. The 32-bit XScale core processor is intended for use for control functions such as managing routing tables or other state information, or handling exception packets. Each 32-bit ME has access to private storage for 4K instructions and 640 words of local memory: these MEs are intended for creating multiple parallel task pipelines for performing (perhaps different) processing functions on multiple packets concurrently. In mapping tasks to these MEs it is necessary that, in addition to not overloading the ME's computing capacity, the total code-size of all tasks assigned to a particular ME not exceed 4K instructions.

Most prior theoretical research on partitioning algorithms for real-time multiprocessor platforms has focused on ensuring that the cumulative computing requirements of the tasks assigned to each processor does not exceed the processor's computing capacity [15, 12]. Our research can be considered to be a generalization of this earlier work, in the sense that it is aimed at determining strategies for assigning tasks to processors in multiprocessor platforms in which *several resources are only available in limited amounts on each processor*. In this paper, we describe in detail our findings concerning systems in which there are two such constraining resources – *local memory* for storing program code, and *computation capacity*. Given a multiprocessor comprised of several processors, each with its own (limited) processing capacity and local memory, and a collection of tasks, each characterized by its code-size and its computation requirement, the *memory-constrained multiprocessor partitioning problem* attempts to partition the tasks among the processors such that neither the memory capacity, nor the computing capacity on any processor is exceeded.

The remainder of this paper is organized as follows. In Section 2, we formally define the problem that we wish to solve, prove that it is intractable, and briefly list related research. In Section 3 we describe how our problem may be mapped on to an equivalent Integer Linear Programming

[*]Supported in part by the National Science Foundation (Grant Nos. ITR-0082866, CCR-0204312, and CCR-0309825).

(ILP) problem. In Section 4, we briefly review some properties of linear programs. In Section 5, we use these properties to derive an efficient algorithm for obtaining a partial mapping of tasks to processors; in Section 6, we describe how this partial mapping may, under certain (well-defined) circumstances, be extended to obtain a complete mapping. In Section 7, we describe a heuristic algorithm for those cases where system parameters render the approach in Section 6 inapplicable: our heuristic is based upon the idea of hierarchically partitioning the problem into smaller, more tractable, ones. We have experimentally evaluated both of our proposed algorithms by simulations on synthetic workloads; we briefly describe these experiments in Section 8. (The extended version of this paper will include the results and discussion of these experiments.) We conclude in Section 9, with a summary of the results presented here.

2 System Model

In this paper, we consider the problem of mapping a given collection of tasks upon a platform comprised of multiple processors. We will assume that all processors are *identical*, in the sense that they have exactly the same computing capacity and the same amount of local memory available.

A *task* i is characterized by two parameters:

- its *utilization* u_i, denoting the fraction of the computing capacity of a single processor that must be reserved for executing it; and
- its *code-size* s_i, denoting the fraction of the local memory associated with a single processor that must be reserved for storing its program code.

(Note that we make no assumptions about the relationship between u_i and s_i for a task i)

We will represent a system Γ comprised of such tasks, to be scheduled upon a platform comprised of identical processors, by a 3-tuple comprised of two equal-sized vectors of positive real numbers ranging over $(0, 1]$, and an integer. Let $\vec{\mathbf{u}}_n$ and $\vec{\mathbf{s}}_n$ denote vectors of size n, and let m be a positive integer. Then $\Gamma \stackrel{\text{def}}{=} (\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ denotes the memory-constrained system consisting of n tasks, in which the i^{th} task has utilization u_i and code-size s_i, that is to be implemented on a platform comprised of m processors each of unit computing and memory capacity.

Some additional notation:

$$u_{\text{sum}}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m) \stackrel{\text{def}}{=} \sum_{i=1}^{n} u_i \ , \qquad u_{\text{max}}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m) \stackrel{\text{def}}{=} \max \left\{ u_i \right\}_{i=1}^{n}$$

$$s_{\text{sum}}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m) \stackrel{\text{def}}{=} \sum_{i=1}^{n} s_i \ , \qquad s_{\text{max}}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m) \stackrel{\text{def}}{=} \max \left\{ s_i \right\}_{i=1}^{n} .$$

When it is clear from context precisely which system we are referring to, we will sometimes omit the system specification from this notation and use u_{max} to denote $u_{\text{max}}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ (similarly for u_{sum}, s_{max}, and s_{sum}).

We are now ready to define our problem precisely.

Definition 1 *Given a system* $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$*, the* Memory-constrained multiprocessor partitioning problem *is to determine a mapping function* $\chi : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$ *such that the following* $2m$ *conditions are satisfied:*

$$\textit{for all } j, 1 \leq j \leq m, \left(\sum_{\{\textit{all } i \ | \ \chi(i)=j\}} u_i \leq 1, \right)$$

$$\textit{for all } j, 1 \leq j \leq m, \left(\sum_{\{\textit{all } i \ | \ \chi(i)=j\}} s_i \leq 1 \right) \ \blacksquare$$

Here, the first m conditions assert that the utilization bound of each processor is respected, while the second m conditions assert that the memory constraints of each processor is respected.

It is not difficult to see that this problem is, in fact, intractable.

Theorem 1 *The memory-constrained multiprocessor partitioning problem is NP-complete in the strong sense.*

Proof Sketch: Since one can guess a partitioning $\chi : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$ in polynomial time, and verify that this mapping is indeed feasible, it follows that the memory-constrained multiprocessor partitioning problem is in NP.

We can show that it is NP-hard in the strong sense by transforming from bin-packing [8]. Each item to be packed corresponds to a single task, with both parameters, utilization and code-size set, equal to the size of the item. Both the computing capacity and the local memory size of each processor are set equal to the bin size. It is straightforward to observe that a bin packing exists if and only if the transformed memory-constrained multiprocessor system is feasible. ■

Related research. When either the utilization limitation or the memory limitation may be ignored, task partitioning is essentially a bin-packing problem: Each processor is a "bin" of capacity one, and each task assigned to it consumes an amount of this capacity equal to its utilization/code-size. While bin-packing is known to be NP-complete in the strong sense, efficient approximation algorithms and polynomial-time approximation schemes are known [8, 7] that can be used to determine task assignments with behaviour that is bounded in the worst-case. Unfortunately, when both utilization and memory limitations are considered simultaneously, the task assignment problem becomes much more difficult. This problem bears remarkable similarities to the M-Dimensional Vector Packing Problem (M-DVPP) [5] with $M = 2$: tasks can be modeled as two-dimensional vectors (the two dimensions correspond to the code-size and utilization requirements, respectively), and processors as bins that are characterized by two distinct capacities, memory size and computing capacity. The M-DVPP, like bin-packing, is known to be intractable (NP-hard in the strong sense [5]). Unfortunately, unlike bin-packing, for which even simple heuristics (such as best-fit, first-fit, etc. [8, 7]) have good worst-case performance guarantees, good polynomial-time approximation schemes for M-DVPP provably cannot exist [18]. Some theoretical work on obtaining approximation algorithms for vector packing has been done by Chekuri and Khanna [2]; however, it seems unlikely that these results are applicable to our attempts to obtain solutions (exact or approximate) to the memory-constrained multiprocessor partitioning problem. Several greedy heuristic algorithms have also been studied in the literature [4, 5, 14, 20]; in particular, Beck and Siewiorek [1] have experimentally evaluated several vector packing heuristics for task allocation: while some heuristics were able to obtain acceptable allocations for certain specific kinds of task systems, there seems to be no heuristic that consistently (and provably) comes up with near-optimal allocations.

3 An ILP formulation

In a Integer Linear Program (ILP), one is given a set of variables, *some or all of which are restricted to take on integer values only*, and a collection of "constraints" that are expressed as linear inequalities over the variables. The set of all points over which all the constraints hold is called the *feasible region* for the integer linear program. One may also be given an "objective function," also expressed as a linear inequality of these variables, and the goal of finding the extremum (maximum/ minimum) value of the objective function over the feasible region.

Consider any system $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$. For any mapping of the n tasks on the m processors, let us define $(n \times m)$ *indicator variables* $x_{i,j}$, for $i = 1, 2, \ldots, n$; and $j = 1, 2, \ldots, m$. Variable $x_{i,j}$ is set equal to one if the i^{th} task is mapped onto the j^{th} processor, and zero otherwise.

We can represent the memory-constrained multiprocessor partitioning problem as the following **integer programming problem**, with the variables $x_{i,j}$ restricted to non-negative integer values.

ILP$(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$

Minimize $\mathcal{L}$, subject to the following constraints, and the restriction that the variables $x_{i,j}$ $(1 \leq i \leq n;\ 1 \leq j \leq m)$ take on **integer** values only:

$$\begin{array}{lll}
\sum_{j=1}^{m} x_{i,j} = 1 & (i = 1, 2, \ldots, n) & (1a) \\
\sum_{i=1}^{n} (x_{i,j} \cdot u_i) \leq \mathcal{L} & (j = 1, 2, \ldots, m) & (1b) \\
\sum_{i=1}^{n} (x_{i,j} \cdot s_i) \leq \mathcal{L} & (j = 1, 2, \ldots, m). & (1c)
\end{array} \quad (1)$$

Informally, $\mathcal{L}$ represents the maximum fraction of both the computing capacity and the local memory of any processor that is used, and is set to be the objective function (i.e., the quantity to be minimized) of the ILP problem. The n constraints corresponding to (1a) above assert that each task be assigned some processor; the m constraints corresponding to (1b), that no processor's computing capacity is exceeded; and the m constraints corresponding to (1c), that no processor's memory is exceeded. It is not hard to see that an assignment of non-negative integer values to the variables $x_{i,j}$ satisfying these constraints, for which $\mathcal{L} \leq 1$, is equivalent to a feasible partitioning of the n tasks upon the m processors. Thus, obtaining a solution to the ILP (1) above is equivalent to determining whether a given memory-constrained multiprocessor task system is feasible. This is formally stated by the following theorem:

Theorem 2 *The Integer Linear Programming problem (1) has a solution with $\mathcal{L} \leq 1$ if and only if the memory-constrained multiprocessor system is feasible.* ■

Theorem 2 above allows us to transform the problem of determining whether a memory-constrained multiprocessor system is feasible to an ILP problem. At first sight, this may seem to be of limited significance, since ILP is also known to be intractable (NP-complete in the strong sense [16]). However, some recently-devised approximation techniques for solving ILP problems, based upon the idea of *LP relaxations* to ILP problems, may prove useful in obtaining solutions to the memory-constrained multiprocessor partitioning problem under certain circumstances – this is the issue that we we explore in the remainder of this paper.

4 A review of some results on linear programming

In this section, we briefly review some facts concerning linear programming (LP) that will be used in later sections. In a Linear Program (LP) over a given set of n variables, as with ILPs, one is given a collection of constraints that are expressed as linear inequalities over these n variables, and perhaps an objective function, also expressed as a linear inequality of these variables. The region in n-dimensional space over which all the constraints hold is again called the *feasible region* for the linear program, and the goal is to find the extremal value of the objective function over the feasible region. A region is said to be *convex* if, for any two points p_1 and p_2 in the region and any scalar $\lambda, 0 \leq \lambda \leq 1$, the point $(\lambda \cdot p_1 + (1 - \lambda) \cdot p_2)$ is also in the region. A *vertex* of a convex region is a point p in the region such that there are no distinct points p_1 and p_2 in the region, and a scalar $\lambda, 0 < \lambda < 1$, such that $[p \equiv \lambda \cdot p_1 + (1 - \lambda) \cdot p_2]$.

It is known that an LP can be solved in polynomial time by the ellipsoid algorithm [10] or the interior point algorithm [9]. (In addition, the exponential-time simplex algorithm [3] has been shown to perform extremely well "in practice," and is often the algorithm of choice despite its exponential worst-case behaviour.) We do not need to understand the details of these algorithms: for our purposes, it suffices to know that LP problems can be efficiently solved (in polynomial time).

We now state without proof some basic facts concerning such linear programming optimization problems.

Fact 1 *The feasible region for a LP problem is convex, and the objective function reaches its optimal value at a vertex point of the feasible region.* ■

An optimal solution to an LP problem that is a vertex point of the feasible region is called a *basic solution* to the LP problem.

Fact 2 *Consider a linear program on n variables $x_1, x_2, \ldots, x_n$, in which each variable is subject to the constraint that it be at least 0 (these constraints are called* non-negativity constraints*). Suppose that there are a further m linear constraints. If $m < n$, then* at most m of the variables have non-zero values *at each vertex of the feasible region*[1] *(including the basic solution).* ■

We will assume that the LP-solver returns a vertex-optimal solution. See [17] for polynomial-time techniques on obtaining a vertex-optimal solution from LP-solvers that can return non-vertex solutions.

5 Obtaining a partial partitioning using linear programming

By relaxing the requirement that the $x_{i,j}$ variables in the ILP (1) (Section 3) be integers only, we obtain the following LP, which is referred to as the *LP-relaxation* [17] of ILP (1):

LPR($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$)

Minimize $\mathcal{L}$, subject to the following constraints:

$$\begin{array}{lll} \sum_{j=1}^{m} x_{i,j} = 1 & (i = 1, 2, \ldots, n) & (2a) \\ \sum_{i=1}^{n} (x_{i,j} \cdot u_i) \leq \mathcal{L} & (j = 1, 2, \ldots, m) & (2b) \\ \sum_{i=1}^{n} (x_{i,j} \cdot s_i) \leq \mathcal{L} & (j = 1, 2, \ldots, m) & (2c) \end{array} \quad (2)$$

Observe that the LP (2) is a linear program on $(nm + 1)$ variables (the nm $x_{i,j}$'s and $\mathcal{L}$), with only $(n + 2m)$ constraints other than non-negativity constraints. By Fact 2 above, therefore, at most $(n + 2m)$ of these variables have non-zero values at any basic solution to this LP.

The crucial observation is that each of the n constraints (2a) is on a *different* set of $x_{i,j}$ variables — the first such constraint has only the variables $x_{1,1}, x_{1,2}, \ldots, x_{1,m}$, the second has only the variables $x_{2,1}, x_{2,2}, \ldots, x_{2,m}$, and so on. Since there are at most $(n + 2m)$ non-zero variables in the basic solution and $\mathcal{L}$ takes on a non-zero value (for non-trivial systems), it follows from the pigeon-hole principle that at most $2m - 1$ of these constraints (2a) will have more than one non-zero value in the basic solution. For each of the remaining (at least) $(n - 2m + 1)$ constraints, the sole non-zero $x_{i,j}$ variable must equal exactly 1, in order that the constraint be satisfied. Fact 3 follows.

[1]The feasible region in n-dimensional space for this linear program is the region over which all the $n + m$ constraints (the non-negativity constraints, plus the m additional ones) hold.

Fact 3 *For at least $(n - 2m + 1)$ of the integers i in $\{1, 2, \ldots, n\}$,* exactly *one of the variables $\{x_{i,1}, x_{i,2}, \ldots, x_{i,m}\}$ is equal to 1, and the remaining are equal to zero, in any basic solution to LPR($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$).* ■

As a consequence of Fact 3 and the polynomial-time solvability of Linear Programming, it follows that the solution to LPR($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$) immediately yields a partial mapping of tasks to processors, in which all but at most $2m - 1$ tasks get mapped.

6 Completing the partial partitioning

In Section 5, we observed that a *partial* mapping of the tasks in any system $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ to the processors in the system could be efficiently determined in time polynomial in the representation of the system; however, this partial mapping may fail to map up to $2m - 1$ tasks. In this section, we modify the approach of Section 5 so that these unmapped tasks can be easily mapped as well. For ease of exposition, in Section 6.1 below we first present (and prove properties of) a somewhat simplified version of our algorithm; in an extended version of this paper, we describe how the algorithm presented here can be further generalized.

6.1 Algorithm Partition: theoretical description

The modification to the linear program LPR($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$) that we propose is to tighten the constraints somewhat, thereby obtaining the linear program LPR2($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$). Observe that LPR2($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$) is essentially the LP relaxation of the ILP corresponding to mapping the tasks in $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ upon m processors of computing capacity and local memory $(1 - 2u_{\max})$ and $(1 - 2s_{\max})$ respectively. We will describe how the partial mapping obtained from a basic solution to LPR2($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$) may be extended to include the unmapped tasks as well. Our algorithm is formalized as Algorithm Partition (Figure 1).

LPR2($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$)

Minimize $\mathcal{L}$, subject to the following constraints:

$$\begin{array}{lll} \sum_{j=1}^{m} x_{i,j} = 1 & (i = 1, 2, \ldots, n) & (3a) \\ \sum_{i=1}^{n} (x_{i,j} \cdot u_i) \leq (1 - 2u_{\max})\mathcal{L} & (j = 1, 2, \ldots, m) & (3b) \\ \sum_{i=1}^{n} (x_{i,j} \cdot s_i) \leq (1 - 2s_{\max})\mathcal{L} & (j = 1, 2, \ldots, m) & (3c) \end{array} \quad (3)$$

From Figure 1 we see that whenever LPR2($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$) has a solution with $\mathcal{L} \leq 1$, Algorithm Partition successfully determines a partitioning of the tasks in $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ among the m processors. The question we now address is this: under what conditions is LPR2($\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m$) guaranteed to have a solution with $\mathcal{L} \leq 1$? We show in Theorem 3 below that there is a simple sufficient test, similar to the *utilization bounds* for single-criterion schedulability analysis [11, 15, 12, 6], for determining whether

```
Algorithm Partition.

Given: memory constrained task system $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$
Step 0: Construct LPR2$(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$.
Step 1: Obtain a basic solution to LPR2$(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$. If the
    value of $\mathcal{L}$ in this solution is greater than 1, then declare
    failure and return. Else, proceed to Step 2.
Step 2: Observe that, if we have not declared failure in step
    1 above, then
        1. We have obtained a mapping for all but at most
           $(2m-1)$ tasks; and
        2. there is enough remaining capacity available on
           each processor to accommodate an additional
           two tasks.
    Hence, the remaining at most $(2m-1)$ tasks can be
    distributed evenly among the processors, with at most
    two tasks to a processor.
```

Figure 1. Algorithm Partition

system $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ is successfully scheduled by Algorithm Partition.

Theorem 3 *If memory constrained multiprocessor system* $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ *satisfies the following two conditions:*

$$\begin{aligned} & u_{sum}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m) && \leq && m - 2m \cdot u_{max}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m) \\ \textbf{and} \quad & s_{sum}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m) && \leq && m - 2m \cdot s_{max}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m) \end{aligned}$$

then it is successfully partitioned by Algorithm Partition.

The proof of Theorem 3 will be included in an extended version of this paper.

7 A Hybrid Approach

The Partition algorithm, described in Section 6 above is applicable only to systems $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ for which both $u_{\max}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ and $s_{\max}(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$ are each less than one-half. However, if a feasible task system has at least one task with either codesize or utilization at least one-half, LPR2 becomes infeasible and Partition will return failure.

In this section, we propose an algorithm HybridPartition for partitioning tasks in systems where tasks can have a utilization or code size greater than one-half. This algorithm is based upon the notion of *hierarchically decomposing* the input system into smaller systems. Section 7.1 presents the notation needed for describing the algorithm HybridPartition presented in Section 7.2.

7.1 Notation

The general approach we propose is to divide the system into smaller systems and use ILP and the Partition algorithm as subroutines. However, before we can can describe our approach, we need to formalize what it means to divide the system into smaller systems.

Informally, our approach will map the processors in the system to a smaller set of "virtual" processors. The capacity (utilization or memory) of each virtual processor is the sum of the capacities of all processors that map to that virtual processor. More formally, let X be the set of all processors, and Y be the set of b virtual processors. Then, a mapping of processors to virtual processors is any surjective function $v : X \rightarrow Y$. For any $i \in X$, let c_i be the utilization capacity, and d_i be the memory capacity of processor i. Define the utilization capacity of each virtual processor, $j \in Y$ to be $C_j = \sum_{i \in \{l : v(l) = j\}} c_i$. The memory capacity of j, D_j is defined similarly.

In addition to aggregating the processors into virtual processors, we will divide the set of tasks into *light* and *heavy* task sets. A task is considered to be *heavy* if either its utilization or code-size exceeds or equals half available respective capacities of a processor or memory. More formally stated, a task t is heavy and an element of the set T_{heavy} if

$$u_t \geq \frac{1}{2} \min_{i \in X}\{c_i\} \ \textbf{or} \ s_t \geq \frac{1}{2} \min_{i \in X}\{d_i\}.$$

Otherwise, a task is considered to be an element of T_{light}. Let $\hat{n}$ denote the number of heavy tasks in a system, and let $\vec{\mathbf{u}}_{\hat{n}}$ and $\vec{\mathbf{s}}_{\hat{n}}$ denote the utilization and code-size vectors for the set T_{heavy}. Also, let $\vec{\mathbf{C}}_b$ and $\vec{\mathbf{D}}_b$ denote the utilization and memory capacity vectors for virtual processors, Y. Then, we let $(\vec{\mathbf{u}}_{\hat{n}}, \vec{\mathbf{s}}_{\hat{n}}, \vec{\mathbf{C}}_b, \vec{\mathbf{D}}_b)$ denote the system defined by considering only heavy tasks and b virtual processors. The ILP for this system is given below.

ILP$(\vec{\mathbf{u}}_{\hat{n}}, \vec{\mathbf{s}}_{\hat{n}}, \vec{\mathbf{C}}_b, \vec{\mathbf{D}}_b)$

Minimize $\mathcal{L}$, subject to the following constraints, and the restriction that the variables $x_{i,j}$ $(1 \leq i \leq \hat{n}; 1 \leq j \leq b)$ take on **integer** values only:

$$\begin{array}{lll} \sum_{j=1}^{b} x_{i,j} = 1 & (i = 1, 2, \ldots, \hat{n}) & (4a) \\ \sum_{i=1}^{\hat{n}} (x_{i,j} \cdot u_i) \leq C_j \cdot \mathcal{L} & (j = 1, 2, \ldots, b) & (4b) \\ \sum_{i=1}^{\hat{n}} (x_{i,j} \cdot s_i) \leq D_j \cdot \mathcal{L} & (j = 1, 2, \ldots, b) & (4c) \\ & & (4) \end{array}$$

7.2 Algorithm HybridPartition: theoretical description

Given a system $(\vec{\mathbf{u}}_n, \vec{\mathbf{s}}_n, m)$, our algorithm works by evenly grouping the processors into b virtual processors. The parameter b is referred to as the *branching factor* of our algorithm. Assume for simplicity that b, m, and n are powers of 2, and $m > b$. The processor mapping function is $v(i) = \lceil \frac{b \cdot i}{m} \rceil$ (meaning the i^{th} processor is mapped to the $v(i)^{\text{th}}$ virtual processor). The set of tasks are divided into T_{heavy} and T_{light}. We solve the ILP defined by

$(\vec{\mathbf{u}}_{\hat{n}}, \vec{\mathbf{s}}_{\hat{n}}, \vec{\mathbf{C}}_b, \vec{\mathbf{D}}_b)$ using well-known integer programming methods (see, e.g., [17]). Note, this ILP contains only $2b+\hat{n}$ constraints, while ILP (1) contains $2m+n$ constraints. After solving the ILP, we have defined a mapping of T_{heavy} to Y. Let T_j ($\subseteq T_{\text{heavy}}$) be the set of heavy tasks that are mapped to the j^{th} virtual processor ($1 \leq j \leq b$). For each T_j, we define a new system corresponding to the tasks of T_j and set of processors $\{i : v(i) = j\}$ and recursively repeat the process from the beginning.

The recursion terminates when $b \geq m$. At which point, Integer is called and the heavy tasks are bound to actual processors, and removed from the set of available tasks. If during any of the recursive calls an ILP is declared infeasible, our algorithm returns failure. The capacities (utilization and memory) of each processor are updated by subtracting the utilization and code-size of the tasks bound to them. Note that by recalculating the capacities of the processors, some tasks may be reclassified as *heavy* tasks. While T_{heavy} is non-empty, we repeat the entire recursive process again. Finally, when all the heavy tasks have been bound to processors and T_{heavy} is empty, we define a system using T_{light} and all m processors with their remaining capacities and call Partition.

8 Experimental Results

The algorithms Partition, HybridPartition, and the integer program solver (referred to as Integer) were implemented for comparison. However, due to space limitations, we are unable to include the results of these experiments. The results of the experiments can be found in an extended version of this paper.

9 Summary and Conclusions

There are generally multiple resources, such as the computing capacities of the processors, the amount of local memory available at each processor, the available communication bandwidth, etc., that are available in limited quantities in each processor in a multiprocessor platform. However, much prior theoretical research on task allocation and scheduling algorithms for multiprocessor platforms has focused primarily on one resource: the computing capacity. Our major contribution in this paper is to devise techniques for simultaneously considering constraints due to *several* resources; in particular, we have considered systems in which both the computing capacity at each processor, and the amount of local memory, are available in limited amounts. We have formalized this task-partitioning problem; proved that it is intractable (NP-hard in the strong sense); and obtained an efficient polynomial-time partitioning algorithm that often works, as well as a simple utilization-based test for determining whether a system is guaranteed to be successfully scheduled by this polynomial-time algorithm. To address systems where our polynomial-time algorithm is incapable of finding a partition, we have designed a different algorithm which hierarchically decomposes the system, and partitions the smaller systems. We have experimentally compared both our algorithms with the integer programming methods producing exact solutions.

References

[1] Beck, J., and Siewiorek, D. Modeling multicomputer task allocation as a vector packing problem. In *Proceedings of the 9th International Symposium on System Synthesis* (San Deigo, CA, Nov. 1996), pp. 115–121.

[2] Chekuri, C., and Khanna, S. On multi-dimensional packing problems. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* (January 1999), pp. 185–194.

[3] Dantzig, G. B. *Linear Programming and Extensions.* Princeton University Press, 1963.

[4] de la Vega, W. F., and Lueker, G. S. Bin packing can be solved within $1 + \epsilon$ in linear time. *Combinatorica 1*, 4 (1981), 349–355.

[5] Garey, M. R., Graham, R. L., Johnson, D. S., and Yao, A. C. C. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory (Series A) 21* (1976), 257–298.

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

[7] Johnson, D. Fast algorithms for bin packing. *Journal of Computer and Systems Science 8*, 3 (1974), 272–314.

[8] Johnson, D. S. *Near-optimal Bin Packing Algorithms.* PhD thesis, Department of Mathematics, Massachusetts Institute of Technology, 1973.

[9] Karmakar, N. A new polynomial-time algorithm for linear programming. *Combinatorica 4* (1984), 373–395.

[10] Khachiyan, L. A polynomial algorithm in linear programming. *Dokklady Akademiia Nauk SSSR 244* (1979), 1093–1096.

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

[12] Lopez, J. M., Garcia, M., Diaz, J. L., and Garcia, D. F. Worst-case utilization bound for EDF scheduling in real-time multiprocessor systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems* (Stockholm, Sweden, June 2000), IEEE Computer Society Press, pp. 25–34.

[13] Makhorin, A. *GNU Linear Programming Kit Reference Manual (Version 4.1).* Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA, August 2003.

[14] Maruyama, K., Tang, D. T., and Chang, S. K. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer and Information Sciences 6*, 2 (June 1977), 131–149.

[15] Oh, D.-I., and Baker, T. P. Utilization bounds for N-processor rate monotone scheduling with static processor assignment. *Real-Time Systems: The International Journal of Time-Critical Computing 15* (1998), 183–192.

[16] Papadimitriou, C. H. On the complexity of integer programming. *Journal of the ACM 28*, 4 (1981), 765–768.

[17] Schrijver, A. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1986.

[18] Woeginger, G. J. There is no asymptotic PTAS for two-dimensional vector packing. *Information Processing Letters 64* (1997), 293–297.

[19] Wolfe, W. *Computers as Components: Principles of Embedded Computing Systems Design.* Morgan Kaufmann Publishers, 2000.

[20] Yao, A. C.-C. New algorithms for bin packing. *Journal of the ACM 27*, 2 (Apr. 1980), 207–227.