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

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 approximate algorithms for solving it.

Keywords: Multiprocessor systems; Partitioned scheduling; Memory-constrained systems; Approximation algorithms.

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

As the functionality demanded of real-time embedded systems has increased, it is becoming unreason-able to expect to implement them upon uniprocessor platforms [22]; 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). 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 [16, 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

1Such NPs offer an alternative to conventional ASIC (application-specific integrated circuit) designs for deploy-ing customized functions, such as firewalls, intrusion detection, load balancing, virtual private networks, protocol conversions, etc., in switches and routers on the Internet. Among the benefits of programmable NPs over ASIC designs is that NPs allow such custom functions to be deployed in a rapid and cost-effective manner; as the Internet continues to evolve at a rapid rate, such NPs are expected to become important building blocks for network elements in all parts of the network, from the edge to the core. Technical specifications of the IXP 2800 are available at http://www.intel.com/design/network/products/npfamily/ixp2800.htm.
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 (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 this approximation algorithm 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 describe these experiments, and our findings, in Section 8. 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 $\Gamma$ 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 $\mathbf{u}_n$ and $\mathbf{s}_n$ denote vectors of size $n$, and let $m$ be a positive integer. Then $\Gamma \stackrel{\text{def}}{=} (\mathbf{u}_n, \mathbf{s}_n, m)$ denotes the memory-constrained system consisting of $n$ tasks, in which the $i^{\text{th}}$ 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:


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

def\ s_{\text{sum}}(\mathbf{u}_n, \mathbf{s}_n, m) \stackrel{\text{def}}{=} \sum_{i=1}^n s_i, \quad s_{\text{max}}(\mathbf{u}_n, \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 $u_{\text{max}}$ to denote $u_{\text{max}}(\mathbf{u}_n, \mathbf{s}_n, m)$ (similarly for $u_{\text{sum}}, s_{\text{max}},$ and $s_{\text{sum}}$).

We are now ready to define our problem precisely.

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

\[
\text{for all } j, 1 \leq j \leq m, \left( \sum_{\{i \mid \chi(i) = j\}} u_i \leq 1, \right)
\]

\[
\text{for all } j, 1 \leq j \leq m, \left( \sum_{\{i \mid \chi(i) = j\}} s_i \leq 1 \right) \quad \blacksquare
\]

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) [4] with \( M = 2 \). In particular, the tasks can be modeled as two-dimensional vectors (the two dimensions correspond to the code-size and utilization requirements, respectively), and the processors as bins that are characterized by two distinct capacities, memory size and computing capacity.\(^2\)

The M-DVPP, like bin-packing, is known to be intractable (NP-hard in the strong sense [4]). 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 algorithms for M-DVPP provably cannot exist [21], although several greedy heuristic algorithms have been studied in the literature [3, 4, 15, 23]. 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.

In addition to the research described above, there is much prior work on multiprocessor task scheduling and allocation problems based upon heuristic approaches. Algorithms have been proposed based on genetic algorithms [13], constraint logic programming [19, 20], and other heuristic

---

\(^2\) The M-DVPP differs from the well-known M-dimensional bin-packing problem (M-DBPP) in that the M different dimensions in M-DVPP are independent of one another while the M-DBPP concerns the packing of hyper-rectangles into hyper-cubes.
approaches [6]; while it may be of some interest to determine whether such approaches are applicable for the memory-constrained multiprocessor partitioning problem, this is not within the scope of the current paper.

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 \((\bar{u}_n, s_n, m)\). 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.

\[
\text{ILP}(\bar{u}_n, s_n, m)
\]

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:

\[
\sum_{j=1}^{m} x_{i,j} = 1 \quad (i = 1, 2, \ldots, n) \quad (1a)
\]

\[
\sum_{i=1}^{n} (x_{ij} \cdot u_i) \leq \mathcal{L} \quad (j = 1, 2, \ldots, m) \quad (1b)
\]

\[
\sum_{i=1}^{n} (x_{ij} \cdot s_i) \leq \mathcal{L} \quad (j = 1, 2, \ldots, m) \quad (1c)
\]

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 [17]). 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 approximate solutions to the memory-constrained multiprocessor partitioning problem – we explore these approximation techniques 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 [2] 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
Figure 1: Example illustrating Fact 2 in 2-dimensions. Notice the vertex points have at most one non-zero component.

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\(^3\) (including the basic solution). Figure 1 illustrates this for two-dimensional space. ■

Note that Fact 1 above does not claim that all points in the feasible region that correspond to optimal solutions to an LP are vertex points; rather, the claim is that *some* vertex point is guaranteed to be in the set of optimal solutions. For LP problems with a unique optimal solution, it is guaranteed that this unique solution is a vertex and hence all (correct) LP solvers will return a vertex solution. For LP problems that have several solutions, however, interior-point or ellipsoid algorithms do not guarantee to find a vertex point (although the simplex algorithm does). There are efficient polynomial-time algorithms (see, e.g, [18]) for obtaining a vertex optimal solution.

\(^3\)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.
given any non-vertex optimal solution to a LP problem – if the LP-solver being used does not guarantee to return a vertex-optimal solution, then one of these algorithms may be used to obtain a vertex-optimal solution from the optimal solution returned by the LP-solver.

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 [18] of ILP (1):

\[
\text{LPR}(\mathbf{u}_n, \mathbf{s}_n, m)
\]

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

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

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.

**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($\mathbf{u}_n, \mathbf{s}_n, m$).

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

\[
\begin{align*}
\sum_{j=1}^{m} x_{i,j} &= 1 \\
\sum_{j=1}^{m} (x_{i,j} \cdot u_i) &\leq (1 - 2u_{\text{max}})\mathcal{L} \\
\sum_{i=1}^{n} (x_{i,j} \cdot s_i) &\leq (1 - 2s_{\text{max}})\mathcal{L}
\end{align*}
\]

(3a) \hspace{1cm} (3b) \hspace{1cm} (3c)

Figure 2: The linear program $\text{LPR}(\tilde{u}_n, \tilde{s}_n, m)$

6 Completing the partial partitioning

In Section 5, we observed that a partial mapping of the tasks in any system $(\tilde{u}_n, \tilde{s}_n, m)$ 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 approximation algorithm; later in Section 6.2, we describe how the algorithm presented here can be further generalized.

6.1 Algorithm Partition: theoretical description

The modification to the linear program $\text{LPR}(\tilde{u}_n, \tilde{s}_n, m)$ that we propose is to tighten the constraints somewhat, thereby obtaining the linear program $\text{LPR2}(\tilde{u}_n, \tilde{s}_n, m)$ (Figure 2). Observe that $\text{LPR2}(\tilde{u}_n, \tilde{s}_n, m)$ is essentially the LP relaxation of the ILP corresponding to mapping the tasks in $(\tilde{u}_n, \tilde{s}_n, m)$ upon $m$ processors of computing capacity and local memory $(1 - 2u_{\text{max}})$ and $(1 - 2s_{\text{max}})$ respectively. We will describe how the partial mapping obtained from a basic solution to $\text{LPR2}(\tilde{u}_n, \tilde{s}_n, m)$ may be extended to include the unmapped tasks as well. Our approximation algorithm is formalized as Algorithm Partition (Figure 3).

From Figure 3 we see that whenever $\text{LPR2}(\tilde{u}_n, \tilde{s}_n, m)$ has a solution with $\mathcal{L} \leq 1$, Algorithm Partition successfully determines a partitioning of the tasks in $(\tilde{u}_n, \tilde{s}_n, m)$ among the $m$ processors. The question we now address is this: under what conditions is $\text{LPR2}(\tilde{u}_n, \tilde{s}_n, m)$ guaranteed to have a solution with $\mathcal{L} \leq 1$? We show in Theorem 3 below that there is a simple sufficient
Given: memory constrained task system \((\bar{u}_n, \bar{s}_n, m)\)

**Step 0:** Construct LPR2\((\bar{u}_n, \bar{s}_n, m)\), which is the linear program (3) in Figure 2.

**Step 1:** Obtain a basic solution to LPR2\((\bar{u}_n, \bar{s}_n, m)\). 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 3: Algorithm Partition

test, similar to the utilization bounds for single-criterion schedulability analysis [11, 16, 12, 5], for determining whether system \((\bar{u}_n, \bar{s}_n, m)\) is successfully scheduled by Algorithm Partition.

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

\[
\begin{align*}
\text{utotal}(\bar{u}_n, \bar{s}_n, m) & \leq m - 2m \cdot \text{utotal}(\bar{u}_n, \bar{s}_n, m) \\
\text{and} \quad \text{stotal}(\bar{u}_n, \bar{s}_n, m) & \leq m - 2m \cdot \text{stotal}(\bar{u}_n, \bar{s}_n, m)
\end{align*}
\]

then it is successfully partitioned by Algorithm Partition.

The proof of Theorem 3 is in an appendix.

**6.2 Algorithm Partition: pragmatic improvements**

Algorithm Partition, as presented above, could be improved in several ways to improve both its average and its worst-case behavior:

1. Observe that it not necessary to stop and declare failure in Step 1 of Algorithm Partition if \(\mathcal{L}\) has a value greater than one in the optimal solution. Since it is quite likely that not
Minimize $\mathcal{L}$, subject to the following constraints:

$$
\sum_{j=1}^{m} x_{i,j} = 1 \quad (i = 1, 2, \ldots, n) \quad (6a)
$$

$$
\sum_{i=1}^{n} (x_{i,1} \cdot u_i) \leq (1 - u_{\text{max}})\mathcal{L} \quad (6b.1)
$$

$$
\sum_{i=1}^{n} (x_{i,j} \cdot u_i) \leq (1 - 2u_{\text{max}})\mathcal{L} \quad (j = 2, 3, \ldots, m) \quad (6b.2)
$$

$$
\sum_{i=1}^{n} (x_{i,1} \cdot s_i) \leq (1 - s_{\text{max}})\mathcal{L} \quad (6c.1)
$$

$$
\sum_{i=1}^{n} (x_{i,j} \cdot s_i) \leq (1 - 2s_{\text{max}})\mathcal{L} \quad (j = 2, 3, \ldots, m) \quad (6c.2)
$$

Figure 4: The linear program LPR3($\vec{u}_n, \vec{s}_n, m$)

all ($2m - 1$) tasks that remained unmapped at the end of Step 1 have their utilization and
code-size parameters exactly equal to $u_{\text{max}}(\vec{u}_n, \vec{s}_n, m)$ and $s_{\text{max}}(\vec{u}_n, \vec{s}_n, m)$ respectively; it may
be possible to fit these remaining tasks on the processors even when $\mathcal{L} > 1$ in the optimal
solution to LPR2($\vec{u}_n, \vec{s}_n, m$).

2. Step 2 of Algorithm Partition needs to place ($2m - 1$) tasks, rather than $2m$; however, solving
LPR2($\vec{u}_n, \vec{s}_n, m$) in Step 1 leaves place to accommodate $2m$ tasks. We could instead set up
LPR2($\vec{u}_n, \vec{s}_n, m$) to leave place for ($2m - 1$) tasks; to do so, we would treat one processor
(without loss of generality, the first one) differently from the others, and leave space on it
for just one task. The linear program modified in this manner is given in Figure 4. It
can be proved that if this linear program, rather than LPR2($\vec{u}_n, \vec{s}_n, m$), is used in Step 1
of Algorithm Partition, then Algorithm Partition is able to successfully partition any system
($\vec{u}_n, \vec{s}_n, m$) satisfying the 2 conditions

$$
\begin{align*}
\sum_{i=1}^{n} (x_{i,1} \cdot u_i) & \leq m - (2m - 1) \cdot u_{\text{max}}(\vec{u}_n, \vec{s}_n, m) \\
\text{and} \quad s_{\sum}(\vec{u}_n, \vec{s}_n, m) & \leq m - (2m - 1) \cdot s_{\text{max}}(\vec{u}_n, \vec{s}_n, m)
\end{align*}
$$

3. Although we have focused upon two constrained resources – the execution requirement and the
code-size – only, it is relatively straightforward to generalize to a larger number of resources
(such as communication bandwidth, energy, etc.) that are available in limited quantities on
each processor. Suppose that there are $k$ such constrained resources per processor, and that
each task is characterized by its need of each of these $k$ resources.
• Each additional resource would give rise to \( m \) more constraints in the LP (3); as a consequence, with \( k \) kinds of constrained resources there would be \((mk - 1)\) tasks left unassigned after Step 1 of Algorithm Partition.

• To accommodate all these tasks in Step 2 of Algorithm Partition, it suffices to have ensured that Step 1 left enough resources unused per processor to be able to accommodate \( k \) additional tasks.

• This can be achieved by setting up the LP appropriately, ensuring that the capacities consumed during the solution to the linear program leave enough capacity to accommodate \( k \) additional tasks. This is done by replacing the terms \( "(1 - 2u_{\text{max}})" \), \( "(1 - 2s_{\text{max}})" \), etc., on the right hand sides of the constraints by \( "(1 - ku_{\text{max}})" \), \( "(1 - ks_{\text{max}})" \), etc.

7 A Hybrid Approach

The Partition algorithm, described in Section 6 above is applicable only to systems \((\vec{u}_n,\vec{s}_n, m)\) for which both \( u_{\text{max}}(\vec{u}_n,\vec{s}_n, m) \) and \( s_{\text{max}}(\vec{u}_n,\vec{s}_n, m) \) are each less than one-half. However, if a feasible task system has at least one task with either codesize or utilization at least \( \frac{1}{2} \), LPR2 becomes infeasible and Partition will return failure.

In this section, we propose an approximation algorithm HybridPartition for partitioning tasks in systems where tasks can have a utilization or code size greater than \( \frac{1}{2} \). This approximation 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 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 \( v : X \to Y \). 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 \( C_j = \sum_{i \in \{ v(t)=j \}} c_i \). 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 \( \text{light} \) and \( \text{heavy} \) task sets. A task is considered to be \( \text{heavy} \) if either its utilization or code-size exceed or equal half available respective capacities of a processor or memory. More formally stated, a task \( t \) is heavy and an element of the set \( T_{\text{heavy}} \) if

\[
u_t \geq \frac{1}{2} \min_{i \in X} \{ c_i \} \quad \text{or} \quad s_t \geq \frac{1}{2} \min_{i \in X} \{ d_i \}.
\]

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

\[
\text{ILP}(\vec{u}_n, \vec{s}_n, \vec{C}_b, \vec{D}_b)
\]

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

\[
\begin{align*}
\sum_{j=1}^{b} x_{i,j} &= 1 \quad (i = 1, 2, \ldots, \hat{n}) \quad (7a) \\
\sum_{i=1}^{\hat{n}} (x_{i,j} \cdot u_i) &\leq C_j \cdot \mathcal{L} \quad (j = 1, 2, \ldots, b) \quad (7b) \\
\sum_{i=1}^{\hat{n}} (x_{i,j} \cdot s_i) &\leq D_j \cdot \mathcal{L} \quad (j = 1, 2, \ldots, b) \quad (7c)
\end{align*}
\]

### 7.2 Algorithm HybridPartition: theoretical description

Given a system \((\vec{u}_n, \vec{s}_n, m)\), our algorithm works by evenly grouping the processors into \( b \) virtual processors. The parameter \( b \) is referred to as the \textit{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 \( v(i) = \left\lfloor \frac{2i}{m} \right\rfloor \) (meaning the \( i \)th processor is mapped to the \( v(i) \)th virtual processor). The set of tasks are divided into \( T_{\text{heavy}} \) and \( T_{\text{light}} \). We solve the ILP defined by \((\vec{u}_n, \vec{s}_n, \vec{C}_b, \vec{D}_b)\) using well-known integer programming methods (see, e.g., [18]). 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 \( T_{\text{heavy}} \) to \( Y \). Let \( T_j \) (\( \subseteq T_{\text{heavy}} \)) be the set of heavy tasks that are mapped to the \( j \)th virtual processor.
Figure 5: Example of one pass of HybridPartition (b=2) demonstrating how heavy tasks are recursively partitioned by solving ILPs of smaller problems using Integer. After all heavy tasks have been assigned to processors, Partition is used to assign the remaining light tasks.

(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 \( T_{\text{heavy}} \) is non-empty, we repeat the entire recursive process again. Finally, when all the heavy tasks have been bound to processors and \( T_{\text{heavy}} \) is empty, we define a system using \( T_{\text{light}} \) 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. Each of the algorithms was implemented in C++ using the GNU Linear Programming Kit (GLPK)\(^4\). In all implementations, the simplex algorithm is used as the linear program relaxation solver. The implementations were tested on an Intel Pentium 4 2.66GHz machine running Windows XP with 256MB of RAM.

Figure 6: Percentage of systems in which Integer exceeded the ten minute time limit as a function of the number of variables in the system.

Section 8.1 compares the experimental results from Partition and Integer. Section 8.2 compares the implementation of HybridPartition with Integer. Both sections include a discussion of testing methodology and an interpretation of the results.

8.1 Comparison of Partition and Integer

As a basis for comparison, we generated a test suite of random task sets that are able to be partitioned by both Partition and Integer (i.e., Inequalities 4 and 5 are satisfied). Systems with \( m = 2, 4, \) and 8 processors were generated. For each value of \( m, \) 140 task sets were generated with the number of tasks \( n \) in the range \([4, 17]\) (10 task sets for each value in the range). We are constrained to pick small values of \( n \) and \( m \) because of the intractability of running Integer over larger problem sizes. The random tasks were generated by randomly selecting utilization and code-size parameters from the available capacity of processors, and then testing whether the Inequalities 4 and 5 are violated by adding the task. If the bounds are violated, the task is rejected, otherwise it is added to the task set.

Both Integer and Partition were run using the generated test suite. A time limit of ten minutes for each system in the test suite was imposed. If an algorithm could not find a partition within ten minutes, failure was declared. Figure 6 shows the percentage of failures by Integer due to the ten minute time limit being exceeded. The percentage is plotted against the number of \( x_{i,j} \) variables in the ILP defined by \((\bar{u}_n, \bar{s}_n, m)\) which is \( n \times m \). Note as the number of variables in the system exceeds 100, Integer is unable to partition a majority of these systems within the time limit. In contrast, Partition is able to partition each of these system well within the time limit and no failures
Figure 7: (a) Average runtimes of both Integer and Partition with respect to the number of variables in the system. Note the range of the $y$-axis is from 0 to 0.01 seconds. (b) Same data as (a), but the range of the $y$-axis has been increased so that the plot of Integer’s average runtimes will be fully displayed. 95% confidence intervals are also shown for each point on the line.

are declared (in fact, the time to partition each of the systems is under 10 milliseconds).

Figure 7 shows the average running time of Partition and Integer on our test suite. Only systems that were successfully partitioned within the time limit are included in the averages. The running time of Integer dominates the Partition’s average running time. Notice that for Integer the confidence intervals grow extremely large as the number of variables increase. Partition on the other hand, has very small confidence intervals. Further testing is needed to reduce the confidence interval range, but these preliminary results suggest that Partition algorithm is a much more predictable and efficient algorithm when the Inequalities 4 and 5 are satisfied.

8.2 Comparison of Hybrid Partition and Integer

Another test suite of random tasks sets was generated to compare the running times and efficacy of Hybrid Partition and Integer. In this test suite, at least one heavy task was included in each system generated. Note the existence of at least one heavy task in the system precludes the use of Partition. We generated a partition for each system considered. By computing the partition first, we guarantee that given enough time, Integer will produce a valid partition. The number of processors $m$ in this test suite ranged over $\{4, 8, 16, 32, 64, 128\}$. For each value of $m$, 50 task sets were generated with the number of tasks $n$ ranging over $[4, 128]$. Note, unlike the test suite for
Figure 8: Percentage of systems in which Integer exceeded the ten minute time limit as a function of the number of variables in the system.

Section 8.1, the number of samples for each value of $n$ is not uniform. This is due to the difficulty in generating random partitions with the exact parameters of $n$ and $m$.

As in the previous section, a time limit of ten minutes is imposed for each system in the test suite. We used a value of $b = 2$ in all the tests of HybridPartition. Figure 8 shows the percentage of failures by Integer due to the ten minute time limit being exceeded, in systems where we have greater than five test systems with the specified number of variables. Note that HybridPartition did not ever fail to partition due to running out of time.

Figure 9 shows the average running time of HybridPartition on our test suite plotted against the number of variables in the system. Notice that for Integer the confidence intervals grow extremely large as the number of variables grow. HybridPartition on the other hand, has very small confidence intervals. The Integer running time clearly dominates HybridPartition running time.

The correlation between schedulability and average utilization (code-size) per processor has a coefficient of approximately $-.6$. This suggests that schedulability is negatively correlated with these two metrics. Therefore, we would expect HybridPartition to have difficulty packing systems that are more fully utilized with respect to memory and processor utilization. Figure 10 plots all the sample systems in the test suite according to their average code-size and utilization per processor. Observe the large cluster of boxes in the lower-left hand corner, indicating that HybridPartition was very successful at partitioning these system with lower utilization. (Notice there are only three systems with both average code-size and utilization per processor less than $0.20$ that are not
Figure 9: (a) Average runtimes of both Integer and HybridPartition with respect to the number of variables in the system. Note the range of the y-axis is from 0 to 3 seconds. (b) Same data as (a), but the range of the y-axis has been increased so that the plot of Integer's average runtimes will be fully displayed. 95% confidence intervals are also shown for each point on the line.

partitionable by HybridPartition.)

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 available, are available in limited amounts. We have formalized this task-partitioning problem; proved that it is intractable (NP-hard in the strong sense); designed a polynomial-time approximation algorithm for solving it; derived a simple utilization-based test for determining whether a system is guaranteed to be successfully scheduled by our approximation algorithm. To address systems where our polynomial-time algorithm is incapable of finding a partition, we have designed another approximation algorithm which hierarchically de-
Figure 10: (a) Scatter-plot of all the systems HybridPartition is able to partition with respect to the average memory and processing utilization per processor. (b) Scatter-plot of all systems HybridPartition fails to partition.

composes the system, and partitions the smaller systems. Finally, we experimentally compare both our algorithms with the integer programming methods producing exact solutions.

References


Appendix: Proof of Theorem 3

First, we need an auxiliary lemma:

**Lemma 1** If memory constrained multiprocessor system \((\bar{u}_n, \bar{s}_n, m)\) satisfies the following two conditions:

\[
\begin{align*}
  u_{\text{sum}}(\bar{u}_n, \bar{s}_n, m) & \leq m - 2m \cdot u_{\text{max}}(\bar{u}_n, \bar{s}_n, m) \\
  s_{\text{sum}}(\bar{u}_n, \bar{s}_n, m) & \leq m - 2m \cdot s_{\text{max}}(\bar{u}_n, \bar{s}_n, m)
\end{align*}
\]

then the optimal solution to LPR2(\(\bar{u}_n, \bar{s}_n, m\)) has \(\mathcal{L} \leq 1\).

**Proof:** Suppose that system \((\bar{u}_n, \bar{s}_n, m)\) satisfies the conditions of the lemma (i.e., Conditions 4 and 5 of Theorem 3). We will prove the lemma by describing how one could determine a point that lies within the feasible region defined by the Constraints (3a)–(3c), for which \(\mathcal{L} = 1\) – i.e., we will determine values for all the \(x_{i,j}\)'s (possibly non-integral) such that Constraints (3a)–(3c) are satisfied when \(\mathcal{L}\) is assigned the value 1.

Suppose that system \((\bar{u}_n, \bar{s}_n, m)\) satisfies Conditions 4 and 5 above. Consider a hypothetical assignment of tasks to processors according to the utilization parameters only (i.e., ignoring the code-size parameters entirely), in which a task may be split across processors. It is easy to come up with such a task assignment in which at most \((1 - 2u_{\text{max}})\) of the computing capacity of each processor is used — simply pack the tasks into the processors, and split a task among two processors if necessary (For example, if \(u_{\text{max}} = 0.2\) and the first four tasks have utilizations \(u_1 = 0.2, u_2 = 0.15, u_3 = 0.2,\) and \(u_4 = 0.15\), then this initial assignment could set \(x_{1,1}, x_{2,1}, \) and \(x_{3,1}\) to 1 each; \(x_{4,1} \leftarrow \frac{1}{3}\) and \(x_{4,2} \leftarrow \frac{2}{3}\), and all other \(x_{i,j}\)'s to zero.)

From this initial assignment, we can obtain one in which tasks may continue to be split across processors (i.e., \(x_{i,j}\)'s are fractional with \(\sum_{j=1}^{m} x_{i,j} = 1\)), but in which it is also the case that at most \((1 - 2s_{\text{max}})\) of the local memory of each processor is used, as follows:

1. Consider two processors \(p\) and \(q\), such that the total code-size allocated on \(p\) is larger than, and that on \(q\) is smaller than, \((1 - 2s_{\text{max}})\) in the current assignment.

2. There must be tasks \(k, \ell\) such that \(x_{k,p} > 0, x_{\ell,q} > 0,\) and \(\frac{B}{u_k} > \frac{B}{u_\ell}\). (That is, there are tasks \(k\) and \(\ell\) that have been (partially) assigned to processors \(p\) and \(q\) respectively, such that for the same amount of computing capacity consumed, task \(k\) consumes more code-size memory than task \(\ell\).)
3. We will "swap" an arbitrarily small amount $\delta$ of the allocation of computing capacity to task $k$ on processor $p$ with the allocation of computing capacity to task $\ell$ on processor $q$; doing this will leave the total amount of computing capacity allocated on each processor unchanged, while decreasing the total amount of local memory allocated on processor $p$ and increasing the total amount of local memory allocated on processor $q$. This is achieved by changing the $x_{i,j}$ values as follows:

$$
\left( x_{k,p} \rightarrow x_{k,p} - \frac{\delta}{u_k} \right), \left( x_{k,q} \rightarrow x_{k,q} + \frac{\delta}{u_k} \right), \left( x_{\ell,p} \rightarrow x_{\ell,p} + \frac{\delta}{u_\ell} \right), \text{ and } \left( x_{\ell,q} \rightarrow x_{\ell,q} - \frac{\delta}{u_\ell} \right)
$$

4. As a result of the swap, the cumulative computing capacity used on each processor has not changed, while the difference between the cumulative code-sizes on processors $p$ and $q$ has decreased. Hence, it follows by repeated applications of this argument that we can eventually obtain a task assignment in which the cumulative computing capacity and the cumulative code size of all processors is exactly the same.

By repeated swaps in this manner, we will eventually achieve an allocation in which at most $(1 - 2s_{\text{max}})$ of the local memory of each processor is used – this allocation bears witness to the fact that the feasible region of LPR2($\vec{u}_n, \vec{s}_n, m$) is non-empty for $L = 1$.

We now present the proof of Theorem 3.

**Proof of Theorem 3.** Suppose that system $(\vec{u}_n, \vec{s}_n, m)$ satisfies the conditions of the theorem. Then by Lemma 1, the feasible region of LPR2($\vec{u}_n, \vec{s}_n, m$) is non-empty for $L = 1$; consequently, the optimal value of $L$ for LPR2($\vec{u}_n, \vec{s}_n, m$) is $\leq 1$. There is therefore, by Fact 1, a basic solution to LPR2($\vec{u}_n, \vec{s}_n, m$) with $L \leq 1$; this basic solution is used by Algorithm Partition to obtain a feasible partitioning of the tasks in ($\vec{u}_n, \vec{s}_n, m$) among the $m$ processors. \[\square\]