Code-size Minimization in Multiprocessor Real-Time Systems

Sanjoy Baruah and Nathan Fisher
Department of Computer Science
The University of North Carolina at Chapel Hill

***Abstract*— Program code size is a critical factor in determining the manufacturing cost of many embedded systems, particularly those aimed at the extremely cost-conscious consumer market. However, most prior theoretical research on partitioning algorithms for real-time multiprocessor platforms has only focused on ensuring that the cumulative computing requirements of the tasks assigned to each processor does not exceed the processor's computing capacity. We consider the problem of task partitioning in multiprocessor platforms in order to minimize the total code size, in application systems in which there may be several different implementations of each task available, with each implementation having different code sizes and different computing requirements. We prove that the general problem is intractable, and present polynomial-time algorithms for solving (well-defined) special cases of the general problem.**

***Index Terms*— Multiprocessor systems; Partitioned scheduling; Minimal-memory partitioning; Multiple task implementations.**

I. Introduction

As the functionality demanded of real-time embedded systems has increased, it is becoming unreasonable to expect to implement them upon uniprocessor platforms [22]; hence, multiprocessor platforms are increasingly used for implementing such systems. This fact is particularly true for systems that are aimed at the consumer market, where cost considerations rule out the use of the most powerful (and expensive) processors. 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.

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

For many embedded applications, a major determinant of system cost is the total amount of memory needed. For such systems the *program code size* is a critical factor in determining the manufacturing cost of the system [5], [17], since reducing code-size results in an implementation with less memory. One promising code size reduction technique that has recently been much explored is to use processor architectures that support multiple instruction sets. Examples include the ARM Thumb [3] and MIPS16 [19], each of which has *two* instruction sets: a normal 32-bit instruction set and a smaller 16-bit instruction set with a smaller set of opcodes and access to fewer registers. During run-time, 16-bit instructions may be dynamically decompressed by hardware into 32-bit equivalent ones before execution: this approach reduces the program code size at the cost of increased computation during run-time. Processors supporting dual instruction sets typically allow programs to contain a mix of normal mode and reduced-width mode instructions, by providing a single instruction that toggles between the two modes. This feature affords the system designer the capability of considering a range of different implementations of any particular process or task, each of which may choose a different tradeoff between code size and execution time by having a different fraction of its code compressed.

In this paper, we address the following question: *Given a multiprocessor platform comprised of m such processors, and a collection of n tasks each with up to t different implementations, determine a partitioning of the tasks among the processors such that the memory required for storing the program code is minimized.* We focus upon shared-memory multiprocessors (SMP's), in which all the code is stored in the shared memory; however, our techniques are easily adapted to handle distributed memory multiprocessors, in which each proces-

sor has local memory and the code for a task assigned to a processor is resident on the processor's local memory, as well.

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 do not exceed the processor's computing capacity [13], [11]. Our research can be considered to be a generalization of this earlier work, in the sense that there is an additional criterion to be optimized – namely, the total amount of memory used to store the program code.

The remainder of this paper is organized as follows. In Section II, we formally define the problem that we wish to solve, prove that it is intractable, and briefly list related research. In Section III, we describe how our problem may be mapped on to an equivalent Integer Linear Programming (ILP) problem. In Section IV, we briefly review some properties of linear programs. In Section V, we use these properties to derive an efficient approximate algorithm for obtaining a mapping of tasks to processors. We conclude in Section VI, with a summary of the results presented here. and a brief discussion on ongoing research into extensions to these results.

II. 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 in the distributed memory model, the same amount of local memory) available.

Each *task* may have up to t distinct implementations, where t is some known constant. Each *implementation* of a task is characterized by two parameters:

- its *utilization*, denoting the computing capacity that is needed for executing it; and
- its *code-size*, denoting the amount of memory that is needed for storing its program code.

We now describe how systems comprised of such tasks, to be scheduled upon multiprocessor platforms comprised of identical processors, may be formally denoted.

Definition 1 (System Specification): Let m be a positive integer and c a positive real number, and let $\mathbf{u}_{n\times t}$ and $\mathbf{s}_{n\times t}$ denote two $(n \times t)$ matrices of non-negative real numbers. Then $\Gamma \stackrel{\text{def}}{=} (\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle)$ denotes the system consisting of n tasks that is to be implemented on a platform comprised of identical m processors each of computing capacity c. The j'th implementation of the i'th task has utilization $u_{i,j}$ and code-size $s_{i,j}$ respectively. Without loss of generality, we will assume that the implementations of each task are indexed according to decreasing code-size (and correspondingly, increasing utilization[1]); if there are $k < t$ implementations for a particular task i, we will set $s_{i,k+1}, s_{i,k+2}, \ldots, s_{i,t}$ each equal to zero, and $u_{i,k+1}, u_{i,k+2}, \ldots, u_{i,t}$ each equal to a number greater than c (thereby implying that the corresponding implementation will not "fit" on a processor). We therefore have $u_{i,1} \leq u_{i,2} \leq \cdots \leq u_{i,t}$, and $s_{i,1} \geq s_{i,2} \geq \cdots \geq s_{i,t}$, for all i, $1 \leq i \leq n$. ■

We illustrate our specification by an example.

Example 1: Consider $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$ with $n = 5$, $t = 3$, $m = 2$, and $c = 1$, and let $\mathbf{u}_{n\times t}$ and $\mathbf{s}_{n\times t}$ be as presented in Figure 1.

Each row of the two matrices taken together specifies one task. Let us, for instance, consider the fourth row. There are two possible implementations of the corresponding task: one that has code-size equal to 0.3 and processor utilization equal to 0.05, and a second in which the code-size is halved but the utilization is doubled. (Note that it is a coincidence here that halving the code-size doubles the utilization — the model does not require that the code-size be linearly related to utilization. Thus for example the third implementation of the first task has code-size one-half that of the first implementation, while its utilization is only $1\frac{2}{3}$ times as much.) ■

One further definition.

Definition 2: For any $\mathcal{U} \geq 0$, let $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle^{\mathcal{U}}$ denote the system obtained from the system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$ by deleting all those task implementations for which the utilization is $> \mathcal{U}$. That is, $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle^{\mathcal{U}}$ is obtained from $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$ by setting $u_{i,j}$ to be greater than c and $s_{i,j}$ equal to zero, for every (i, j) such that $u_{i,j} > \mathcal{U}$ in $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$, and keeping the remaining parameters unchanged.

■

Example 2: Consider the system $\langle \mathbf{u}_{5\times 3}, \mathbf{s}_{5\times 3}, 2, 1\rangle$ specified in Example 1. The system $\langle \mathbf{u}_{5\times 3}, \mathbf{s}_{5\times 3}, 2, 1\rangle^{0.5}$

[1]Note that it makes no sense to consider two implementations such that *both* the code-size and the utilization of one are smaller than those of the other — from the perspective of our problem, the implementation with the smaller parameters is superior to the other implementation, which therefore needs no further consideration.

$$\mathbf{u}_{n\times t} = \begin{bmatrix} 0.3 & 0.4 & 0.5 \\ 0.1 & 0.4 & 0.6 \\ 0.15 & 0.25 & 0.3 \\ 0.05 & 0.1 & 1.1 \\ 0.3 & 1.1 & 1.1 \end{bmatrix} \quad \textbf{and} \quad \mathbf{s}_{n\times t} = \begin{bmatrix} 0.4 & 0.25 & 0.2 \\ 0.5 & 0.4 & 0.3 \\ 0.4 & 0.25 & 0.2 \\ 0.3 & 0.15 & 0.0 \\ 0.6 & 0.0 & 0.0 \end{bmatrix}$$

Fig. 1. Task specifications for Example 1.

is obtained from the original system by "eliminating" all task implementations in which the utilization exceeds 0.5; this results in

$$\mathbf{u}_{5\times 3} = \begin{bmatrix} 0.3 & 0.4 & 0.5 \\ 0.1 & 0.4 & \boxed{\mathbf{1.1}} \\ 0.15 & 0.25 & 0.3 \\ 0.05 & 0.1 & 1.1 \\ 0.3 & 1.1 & 1.1 \end{bmatrix}$$

and

$$\mathbf{s}_{5\times 3} = \begin{bmatrix} 0.4 & 0.25 & 0.2 \\ 0.5 & 0.4 & \boxed{\mathbf{0}} \\ 0.4 & 0.25 & 0.2 \\ 0.3 & 0.15 & 0.0 \\ 0.6 & 0.0 & 0.0 \end{bmatrix}$$

■

We are now ready to define our problem precisely.

Definition 3 (Code-size minimal task assignment.): ***Given*** a system $(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle)$, ***determine*** a *choice function* $\theta : \{1,\ldots,n\} \rightarrow \{1,\ldots,t\}$ and a *processor mapping function* $\chi : \{1,\ldots,n\} \rightarrow \{1,\ldots,m\}$ such that the following m conditions are satisfied:

$$\text{for all } k, 1 \leq k \leq m, \left(\sum_{\{\text{all } i \mid \chi(i)=k\}} u_{i,\theta(i)} \leq c \right), \quad (1)$$

and the following quantity is minimized:

$$\sum_{i=1}^{n} s_{i,\theta(i)} \quad (2)$$

■

Intuitively, the choice function $\theta(i)$ designates which of the available alternative implementations of task i is chosen, and the processor mapping function $\chi(i)$ designates which processor this implementation goes on. The m conditions (1) assert that the utilization bound of each processor is respected, while Expression 2 represents the total amount of memory that is needed to store the chosen program code

Example 3: Let us return to the system considered in Example 1. Observe that the numbers in the first column of the utilization matrix (Figure 1) sum to 0.9; hence, the collection of task implementations obtained by taking the first (minimum-utilization) implementation of each task is in fact *uni*processor feasible on a unit-capacity processor. The total code-size for these five implementations is the first column-sum of the $\mathbf{s}$ matrix, and equals 2.2.

It can be shown through exhaustive enumeration that the total code-size is minimized on the two unit-capacity processors that are available for the following choice function:

i	1	2	3	4	5
$\theta(i)$	3	3	3	2	1

and the following processor mapping function:

i	1	2	3	4	5
$\chi(i)$	1	2	2	2	1

In this partitioning, the total utilization on processor 1 is $0.5 + 0.3 = 0.8$, while on processor 2 it is $0.6 + 0.3 + 0.1 = 1.0$. The total code-size is $0.2 + 0.3 + 0.2 + 0.15 + 0.6 = 1.45$. ■

It is not difficult to see that the code-size minimal task assignment problem is intractable; indeed, even severely restricted versions are provably intractable:

Theorem 1: The code-size minimal task assignment problem is intractable, even under either of the following two restrictions: **(i)** there is only one processor ($m = 1$); or **(ii)** each task has only one possible implementation ($t = 1$).

Proof Sketch: On uniprocessors, the problem can be transformed to the *multiple-choice knapsack* problem [18], [10], [1], which is known to be NP-hard.

On multiprocessors but with each task having only one implementation, the problem can be transformed to the *bin-packing* problem [7], [6], which is known to be

NP-hard in the strong sense. ■

Our results. In this paper, we derive a polynomial time algorithm for obtaining code-size minimal task assignments, that makes the following performance guarantee.

> **If** there exists an implementation of $(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle)$ of cost $\mathcal{C}$ and some constant $\mathcal{U}$ ($\mathcal{U} < c$) such that **(i)** at most an amount $(c - \mathcal{U})$ of each processor is used, and **(ii)** no *individual* task occupies more than $\mathcal{U}$ of the capacity of any processor (i.e., if the j'th implementation of task i is the one selected, then $u_{i,j} \leq \mathcal{U}$)
> **then** our algorithm will produce an implementation of $(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle)$ of cost at most $\mathcal{C}$.

However, it is possible that there is an implementation of the system of cost $\mathcal{C}$ which does not satisfy the conditions listed above; in that case, our algorithm may fail to find an implementation of cost $\mathcal{C}$ — given that the problem is NP-hard (Theorem 1), this is only to be expected.

Related research. When the code-size minimization objective may be ignored, task partitioning on multiprocessors is essentially a bin-packing [7], [6] 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. This relationship between bin-packing and task partitioning is explored in, e.g, [13], [11]. In addition to bin-packing based research, there is much prior work on multiprocessor task scheduling and allocation problems based upon heuristic approaches. Algorithms have been proposed based on genetic algorithms [12], constraint logic programming [20], [21], integer programming [15], and other heuristic approaches [4]; while it may be of some interest to determine whether such approaches are applicable for the code-size minimal task assignment problem, this is not within the scope of the current paper.

The issue of obtaining code-size minimal implementations of real-time systems upon *uni*processor platforms has been studied by Shin et al. [17], for a more general periodic task model – one in which tasks may have arbitrary initial arrival-times, and deadlines distinct from their periods. Since the uniprocessor feasibility-analysis problem for collections of such periodic tasks is known to be NP-hard in the strong sense, several heuristics are proposed in [17], and are evaluated via simulations.

ILP$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle)$

Minimize

$$\sum_{i=1}^{n} (x_{i,j,k} \times s_{i,j}) \tag{3}$$

subject to the following constraints, and the restriction that the $x_{i,j,k}$ variables take on **integer** values only:

$$\sum_{\textbf{all } j,k} x_{i,j,k} = 1 \qquad (i = 1, 2, \ldots, n) \tag{4a}$$

$$\sum_{\textbf{all } i,j} (x_{i,j,k} \cdot u_{i,j}) \leq c \qquad (k = 1, 2, \ldots, m) \tag{4b}$$

(4)

Fig. 2. ILP representation of the code-size minimal task-assignment problem.

III. An ILP formulation

In an 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 or minimum) value of the objective function over the feasible region.

Consider any system $(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle)$. For any mapping of the n tasks on the m processors, let us define $(n \times t \times m)$ *indicator variables* $x_{i,j,k}$, for $i = 1, 2, \ldots, n$; $j = 1, 2, \ldots, t$; and and $k = 1, 2, \ldots, m$. Variable $x_{i,j,k}$ is set equal to one if

$$\theta(i) = j \textbf{ and } \chi(i) = k \ ;$$

i.e., if the j'th implementation of the i'th task is mapped onto the k'th processor, and zero otherwise.

We can represent the code-size minimal task assignment problem as the **integer programming problem** of Figure 2, with the variables $x_{i,j,k}$ restricted to non-negative integer values.

The n constraints corresponding to (4a) above assert that each task be assigned some processor, while the m constraints corresponding to (4b), that no processor's computing capacity is exceeded. It is not hard to see that an assignment of non-negative integer values to the

variables $x_{i,j,k}$ satisfying these constraints is equivalent to a feasible partitioning of the n tasks upon the m processors. Thus, obtaining a solution to the ILP (4) above is equivalent to determining whether a given system is feasible. This is formally stated by the following theorem:

Theorem 2: The Integer Linear Programming problem (4) has a solution if and only if the multiprocessor system is feasible. ■

Theorem 2 above allows us to transform the code-size minimal task assignment problem 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 [14]). 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 code-size minimal task assignment problem – we explore these approximation techniques in the remainder of this paper.

IV. 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 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 [9] or the interior point algorithm [8]. (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 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[2] (including the basic solution). ■

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 basic solution. For LP problems that have several solutions, however, interior-point or ellipsoid algorithms do not guarantee to find a vertex solution (although the simplex algorithm does). There are efficient polynomial-time algorithms (see, e.g, [16]) for obtaining a basic solution given any non-vertex optimal solution to a LP problem – if the LP-solver being used does not guarantee to return a basic solution, then one of these algorithms may be used to obtain a basic solution from the optimal solution returned by the LP-solver.

V. An approximate algorithm

In this section, we derive a polynomial-time algorithm that obtains a code-size minimal task assignment of a given system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c \rangle$, provided $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c \rangle$ satisfies certain conditions. Let $\mathcal{U}$ denote a positive real-number no larger than c. Recall (from Definition 2) that $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c \rangle^{\mathcal{U}}$ denotes the restricted system obtained from $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c \rangle$ by

[2]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.

LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-\mathcal{U}\rangle^{\mathcal{U}})$

Minimize

$$\sum_{i=1}^{n}(x_{i,j,k} \times s_{i,j}) \tag{5}$$

subject to the following constraints, and the restriction that the $x_{i,j,k}$ variables take on **non-negative** values only:

$$\sum_{\textbf{all } j,k} x_{i,j,k} = 1 \qquad (i = 1, 2, \ldots, n) \tag{6a}$$

$$\sum_{\textbf{all } i,j} (x_{i,j,k} \cdot u_{i,j}) \leq c - \mathcal{U} \qquad (k = 1, 2, \ldots, m) \tag{6b}$$

(6)

Fig. 3. The linear program LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-\mathcal{U}\rangle^{\mathcal{U}})$. The values of the $s_{i,j}$ and $u_{i,j}$ parameters are those of system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-\mathcal{U}\rangle$.

only considering those task implementations that have utilization at most $\mathcal{U}$. We will now consider the scheduling of this restricted system upon m processors, each of computing capacity $c-\mathcal{U}$. That is, we will consider the scheduling of the system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-\mathcal{U}\rangle^{\mathcal{U}}$. Observe that the ILP formulation of this problem is virtually identical to the one presented in Section III, the only differences being that **(i)** the values of the $s_{i,j}$ and $u_{i,j}$ parameters in Equation 3 and the Equations 4b respectively are taken from the specification of system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-\mathcal{U}\rangle^{\mathcal{U}}$, rather than the specification of $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$; and **(ii)** the right hand sides of each of the Equations 4b is set equal to $c-\mathcal{U}$, rather than c.

By relaxing the requirement that the $x_{i,j,k}$ variables in this ILP (4) be integers only, we obtain the linear programming *relaxation* [16] of ILP$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-\mathcal{U}\rangle^{\mathcal{U}})$, which is presented in Figure 3. Since this is a linear (as opposed to *integer* linear) program, it can be solved in polynomial time [9], [8]; furthermore, basic solutions can be determined in polynomial time [16].

Observe that LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-\mathcal{U}\rangle^{\mathcal{U}})$ is a linear program on $(n\times t\times m)$ variables with only $(n+m)$ constraints other than non-negativity constraints. By Fact 2 above, therefore, at most $(n+m)$ of these variables have non-zero values at any basic solution.

The crucial observation is that each of the n constraints (6a) is on a *different* set of $x_{i,j,k}$ variables — the first such constraint has only the variables $x_{1,j,k}$, the second has only the variables $x_{2,j,k}$, and so on. Since there are at most $(n+m)$ non-zero variables in the basic solution, it follows from the pigeon-hole principle that at most m of these constraints (6a) will have more than one non-zero value in the basic solution. For each of the remaining (at least) $(n-m)$ constraints, the sole non-zero $x_{i,j,k}$ variable must equal exactly 1, in order that the constraint be satisfied. Fact 3 follows.

Fact 3: For at least $(n-m)$ of the integers i in $\{1, 2, \ldots, n\}$, *exactly* one of the variables $\{x_{i,j,k}\}$ is equal to 1, and the remaining are equal to zero, in any basic solution to LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, \mathcal{U}\rangle^{\mathcal{U}})$. ■

Thus, a basic solution to this linear program LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-\mathcal{U}\rangle^{\mathcal{U}})$ gives us a mapping for all but (at most) m tasks, upon m processors each of computing capacity $(c-\mathcal{U})$. Let

$$u_{\max} \stackrel{\text{def}}{=} \max_{\text{all } i,j}\{u_{i,j}\}.$$

Observe that

1) the characteristics of all the tasks in the system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-u_{\max}\rangle^{u_{\max}}$ are exactly the same as the characteristics of all the tasks in the system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-u_{\max}\rangle$, and
2) since the partial mapping obtained by solving the linear program LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-u_{\max}\rangle^{u_{\max}})$ leaves spare computing capacity of an amount equal to $u_{\max}$ on each processor, the m tasks potentially left unmapped by the solution to LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-u_{\max}\rangle^{u_{\max}})$ can be mapped on to the processors by *adding one additional task per processor*; furthermore, the minimum code-size implementation of each such task can be chosen for implementation.

This immediately suggests the following mapping algorithm for systems for which LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-u_{\max}\rangle^{u_{\max}})$ is feasible:

1) Obtain a basic solution to linear program LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c-u_{\max}\rangle^{u_{\max}})$; for each 3-tuple (i,j,k) such that $x_{i,j,k} \equiv 1$ in this basic solution, assign the j'th implementation of task i to processor k.
2) Map the remaining (at most m) tasks one to a processor; for each, choose the minimum code-size implementation.

Lemma 1 formally states the performance guarantee that

is made by this mapping algorithm:

Lemma 1: If there exists an implementation of the n tasks in task system $(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle)$ upon m processors consuming at most computing capacity $(c - u_{\max})$ on each processor and with total memory requirement $\mathcal{C}$, then the method described above — extending the mapping obtained by solving LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - u_{\max}\rangle^{u_{\max}}$ by distributing the minimimum code-size implementations of the unmapped tasks one to a processor — will obtain an implementation of task system $(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle)$ with total memory requirement $\mathcal{C}$. ■

(That is, our algorithm requires each processor to have computing capacity $u_{\max}$ greater than the optimal in order to obtain a code-size optimal implementation.)

What happens, however, if LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - u_{\max}\rangle^{u_{\max}})$ is not feasible? Let us suppose that we are able to determine an x, $0 < x < c$, such that LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x\rangle^{x})$ is feasible, and that the corresponding minimum value of the objective function, as determined by solving LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x\rangle^{x})$, is $\mathcal{C}$. Observe that the m tasks potentially left unmapped by the solution to LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x\rangle^{x})$ can be mapped on to the processors by adding one additional task per processor; furthermore, the smallest code-size implementation of each such task for which the corresponding utilization is $\leq x$ can be chosen for implementation. That is, for any x, *any basic feasible solution to LPR*$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x\rangle^{x})$ *yields a complete mapping of tasks to processors, with total memory required for program code no more than the value of the objective function in the feasible solution.*

Observe that there may be at most $(n \times t)$ distinct values of the utilization parameter $u_{i,j}$ in system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$. Consider now any x_1 and x_2, such that $x_1 < x_2$ and none of the distinct values of the utilization parameter fall in the interval $[x_1, x_2)$. Since LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x_2\rangle^{x_2})$ is a more constrained linear program that LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x_1\rangle^{x_1})$ (the task implementations available in $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x_1\rangle^{x_1}$ and $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x_2\rangle^{x_2}$ are the same, and each processor in the problem modelled by LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x_2\rangle^{x_2})$ has smaller computing capacity than in the problem modelled by LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x_1\rangle^{x_1})$), it follows that LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x_2\rangle^{x_2})$ has a feasible solution with optimal value of objective function $\mathcal{C}$ only if LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x_1\rangle^{x_1})$ has a feasible solution with optimal value of objective function $\mathcal{C}$. In other words, we need only check the feasiblity of LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - x\rangle^{x})$ for x taking on the (at most) $(n \times t)$ distinct values of the utilization parameter, to determine whether the method of the previous paragraph will yield us a feasible mapping, and the corresponding minimal memory required. The pseudocode implementing this algorithm is presented in Figure 4; Theorem 3 follows directly.

1) Let $u_1, u_2, \ldots,$ denote the (at most $n\times t$) distinct values of the utilization parameters $u_{i,j}$ in $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$ sorted in decreasing order.
2) Let ℓ denote the smallest i such that LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - u_\ell\rangle^{u_\ell})$ is feasible. (If no such i exists, declare failure. This entire step takes at most $(n \times t)$ calls to an LP solver.)
   a) Obtain a basic solution to LPR$(\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c - u_\ell\rangle^{u_\ell})$. For each 3-tuple (i, j, k) such that $x_{i,j,k} \equiv 1$ in this basic solution, assign the j'th implementation of task i to processor k.
   b) Map the remaining (at most m) tasks one to a processor; for each, use the minimum code-size implementation.

Fig. 4. Mapping system $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$.

Theorem 3: **If** there exists an implementation of $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$ of cost $\mathcal{C}$ and some constant $\mathcal{U}$ ($\mathcal{U} < c$) such that **(i)** at most $(c - \mathcal{U})$ computing capacity is used on each processor, and **(ii)** no *individual* task occupies more than $\mathcal{U}$ of the capacity of any processor, **then** our approximation algorithm will produce an implementation of $\langle \mathbf{u}_{n\times t}, \mathbf{s}_{n\times t}, m, c\rangle$ of cost at most $\mathcal{C}$. ■

VI. Summary and Conclusions

For many embedded applications, a key determinant in manufacturing cost is the amount of memory needed for storing program code. One promising code size reduction technique that has recently been much explored is to use processor architectures that support multiple instruction sets, and trade off some computing capacity for smaller code size.

In this paper, we have considered the problem of task partitioning in multiprocessor platforms in order to minimize the total code size, for application systems in which

there may be several different implementations of each task available, with each implementation having different code sizes and different computing requirements. We have formalized this problem as the *code-size minimal task assignment* problem, have shown that this problem is intractable even under severe simplifying assumptions, and have derived efficient approximate algorithms for solving it.

The results presented in this paper can be extended to task assignment algorithms for *memory-constrained* multiprocessor systems with multiple task implementations. In a memory-constrained systems, the amount of available memory capacity (either distributed or shared) for program code is known *a priori*. The task assignment algorithm presented in Figure 4 can easily be extended to a memory-constrained system with shared memory by running the algorithm and checking that memory requirement, $\mathcal{C}$, does not exceed the memory capacity. However, the algorithm will require modification for task assignment in distributed memory systems with memory constraints. For the time being, we leave this problem open.

REFERENCES

[1] ARMSTRONG, R. D., KUNG, D. S., SINHA, P., AND ZOLTNERS, A. A. A computational study of a multiple-choice knapsack algorithm. *ACM Trans. Math. Softw. 9*, 2 (1983), 184–198.

[2] DANTZIG, G. B. *Linear Programming and Extensions*. Princeton University Press, 1963.

[3] GOUDGE, L., AND SEGARS, S. THUMB: Reducing the cost of 32-bit RISC performance in portable and consumer applications. In *Proceedings of COMPCON* (1996).

[4] GRANDPIERRE, T., LAVARENNE, C., AND SOREL, Y. Rapid prototyping for real-time embedded heterogeneous multiprocessors. In *International Workshop on Hardware/Software Co-Design (CODES)* (Rome, Italy, 1999), ACM Press.

[5] HALAMBI, A., SHRIVASTAVA, A., BISWAS, P., DUTT, N., AND NICOLAU, A. An efficient compiler technique for code size reduction using reduced bit-width ISAs. In *Proceedings of DATE: Design, Automation and Test in Europe* (2002), pp. 402–408.

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

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

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

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

[10] KUNG, D. S. *The Multiple Choice Knapsack Problem: Algorithms and Applications*. PhD thesis, The University of Texas at Austin, 1982.

[11] 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.

[12] MADSEN, J., AND BJORN-JORGENSEN, P. Embedded system synthesis under memory constraints. In *International Workshop on Hardware/Software Co-Design (CODES)* (Rome, Italy, 1999), ACM Press.

[13] 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.

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

[15] PRAKASH, S., AND PARKER, A. C. Synthesis of application-specific multiprocessor systems including memory components. *Journal of VLSI Signal Processing 8* (1994), 97–116.

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

[17] SHIN, I., LEE, I., AND MIN, S. L. Embedded system design framework for minimizing code size and guaranteeing real-time requirements. In *Proceedings of the IEEE Real-Time Systems Symposium* (Austin, TX, December 2002), IEEE Computer Society Press, pp. 201–211.

[18] SINHA, P., AND ZOLTNERS, A. A. The multiple choice knapsack problem. *Operations Research 27* (1979), 503–515.

[19] SWEETMAN, D. *See MIPS Run*. Morgan Kaufman, San Francisco, CA, 1999.

[20] SZYMANEK, R. W., AND KUCHCINSKI, K. A constructive algorithm for memory-aware task assignment and scheduling. In *International Workshop on Hardware/Software Co-Design (CODES)* (Copenhagen, Denmark, 2001), ACM Press.

[21] SZYMANEK, R. W., AND KUCHCINSKI, K. Partial task assignment of task graphs under heterogeneous resource constraints. In *International ACM/ IEEE Design Automation Conference (DAC)* (Anaheim, CA, 2003), ACM Press, pp. 244–249.

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