RAGS: Real-analysis, ALAP Guided Synthesis

David L. Rhodes and Wayne Wolf

Abstract—**A new technique for single-bus heterogeneous system scheduling and hardware/software co-design is presented. This technique addresses challenging real-time problem domains (*e.g.* multi-rate periodic, dependent tasks) for preemptive target systems including real-time operating system overheads and communication contention along with proper protocol handling. Such realistic system attributes are often times ignored in scheduling and co-design efforts, but are addressed here using a novel 'real-analysis' approach. This technique foregoes the usual list- or cluster-oriented scheduling techniques for an as-late-as-possible (ALAP) guided iterative improvement procedure. A detailed system simulation is used at the scheduling level, which in turn is used as the core of the overall co-synthesis. Since an accurate simulation is the basis for the schedule feasibility check, separate verification steps become unnecessary. In addition to using a real-analysis as part of the scheduling, several other unique or unusual features are employed including the use of ALAP in heterogeneous scheduling, recursive search level at a time, allowing backtracking while maintaining polynomial execution time, as well as others. This framework appears to be unique in its ability to address the allocation/scheduling and co-design of heterogeneous systems which, in particular, employ arbitrated busses for inter-task communication.**

Keywords—**heterogeneous processor allocation and scheduling, communication scheduling, hardware/software co-design, system modeling, busses, arbitration, optimization, hill-climbing**

I. Introduction

The design and scheduling of systems with real-time constraints has certainly received a good deal of (warranted) attention. Such systems may be employed in a variety of application domains, including real-time control, databases, transactional servers, streaming video/audio/-telephonic applications, etc. Here, we are interested in target systems that are composed as a communicating set of processing elements (PEs) each running a real-time operating system (RTOS). Such systems are somewhere in the 'middle-ground' of parallel computing as scaled according to the ratio of communication overhead to average task computation time. For example, multi-core ASIC-based target systems which utilize on-chip interconnects provide very fast interconnects while loosely coupled, network connected computers provide communication with much greater delay. This 'middle range' is a very interesting and challenging region for scheduling and co-synthesis efforts. In contrast, as the average communication time to computation time ratio decreases, the common assumption of ideal (no overhead) communication becomes valid while at the other extreme as the communication time to computation time ratio increases a parallel approach becomes detrimental (for an unbounded number of processors) [1], [2].

In addition to addressing the scheduling and co-synthesis of systems with significant communication utilization, we simultaneously address many other realistic system characteristics which are often times ignored. This is accomplished via careful construction of a system model which includes appropriate RTOS overheads, proper bus contention and protocol handling, inter-PE and intra-PE data transfer and its implementation in a contemporary RTOS, etc. While such characteristics reside within the solution (*i.e.* target system) domain, a different set of characteristics apply to the problem domain. These aspects include task dependency, periodic or aperiodic execution, use of hard or soft deadlines, etc. To illustrate these concepts, many significant characterizations for both the target system and input problem domain are captured in the taxonomy shown in Figure 1. Characteristics listed along the top edge of the figure represent those that are more general while the structure shown below illustrates various specializations—for example, homogeneous processors are a special case of heterogeneous processors; a single processor case is a special case of multiple processors. The figure divides characteristics in terms of whether they are inherent to the problem itself ('Input Space') or are properties of possible solutions ('Target System Space').

The new scheduling and co-design system, called real-analysis, ALAP guided synthesis (**RAGS**), developed here permits the characteristics shown in the grey boxes of Figure 1. RAGS provides the designer a very general model for behavior and performance. Although the RAGS co-design framework should allow additional situations, an area of particular interest is in communication busses which are *arbitrated*. The arbitration protocol, along with other communication resource protocols, fully dictate bus access yet are rarely considered in embedded system scheduling or co-design. Arbitrated busses, in particular, are used in PCI [3] and Small-PCI/Compact-PCI [4] embedded systems.

Communication scheduling is sometimes considered as a topic completely independent from other system considerations (*e.g.* [5], [6], [7]). While it may make sense in some circumstances to treat communication separately (*e.g.* when a good deal of control over communication scheduling is possible or when communication time is insignificant when compared to computation times), assuming that communication doesn't affect the rest of the design is not generally valid. Arbitrated bus based systems are particularly difficult to design because the scheduling of communication tasks is a direct function of task allocation which, in turn, must be decided by the scheduling algorithm. These difficulties are even further pronounced in the heterogeneous processor case where task run-times are now processor dependent as well. By tracking the desired versus actually occurring communication task scheduling, it will be shown that the samples used to demonstrate the new method exhibit significant arbitration effects. Interestingly, the po-

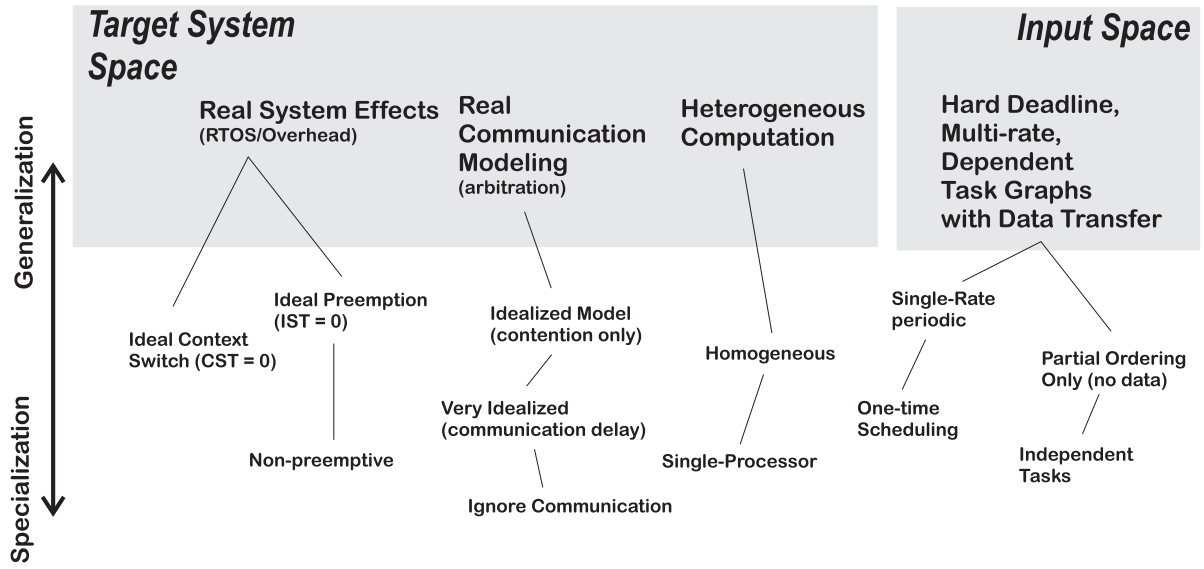

Fig. 1. A taxonomy of input and target system characteristics

tentially adverse impact of bus arbitration on scheduling has been studied before, but only in a statistical sense assuming bus requests occur as Markovian arrival processes [8].

Even without considering communication, heterogeneous system scheduling of dependent tasks represents a significant challenge (even the homogeneous form is a well known NP-complete problem [9], [10]). Indeed, some prior efforts deal with the heterogeneous allocation problem but ignore communication [11], or treat communication in a simplified manner—*e.g.* without true resource contention [12], [13]; consider only bus bandwidth effects and not true contention [14]; are oriented to the non hard real-time arena [15]; or include realistic communication models, but are not fully automated [16], etc.

In addition to communication arbitration and heterogeneous processor scheduling, the final major target system considerations are related to task initiation and context switching, as well as preemption and interrupts, which cause additional non-ideal behavior. Once again, such characteristics are often idealized in scheduling or co-design efforts where, for example, ideal preemption and context switching is often assumed [17], [18], [19]. Thus, as readily visible in Figure 1, the areas of proper RTOS overheads, communication bus modeling and heterogeneous computing are key target system features to be addressed by the scheduling and co-design tools presented here.

To our knowledge, no prior scheduling or co-synthesis effort has dealt with this set of characteristics; but related efforts will be discussed in the ensuing sections. The need to address this range of non-ideal effects has resulted in a new 'real-analysis' approach which takes advantage of the fact that realistic system effects such as RTOS task management overheads and communication protocols can be readily modeled and simulated. The underpinnings of this system model, presented in the next section (Section II), is derived from execution traces from a modern RTOS. The co-design approach, described and illustrated in Section V, divides the overall problem into one of generating possible hardware configurations each of which is checked for feasibility. This feasibility check is actually a complete allocation and scheduling attempt of the proposed system configuration and is a useful tool in its own right. This 'inner' scheduling routine is described in Section IV. It makes use of the as-late-as-possible (ALAP) schedule as compared to the actual schedule analysis (hence the monikers 'ALAP-guided' and 'real-analysis') to perform the scheduling. Note that another distinct advantage of this approach is that subsequent verification steps are not needed, all target system effects are accounted for in the check for schedule feasibility.

II. An Accurate System Model

Several aspects of the target system must be examined in order to develop a complete, and valid, system model. Our categorization classifies effects as belonging to one of the following topics: RTOS task execution and preemption; bus protocols; and intra- and inter-processor data transfer. However, these aspects are *interacting* in that each cannot be considered entirely independently (*e.g.* inter-processor data transfer relies on transport over the bus and the associated RTOS actions taken by the sending and receiving processor). RTOS topics are discussed first, followed by communication and bus issues. The section concludes with a summary of the findings.

Since the PEs are each executing a RTOS, we need to (i) select the appropriate OS mechanisms for use in the system; and (ii) model these mechanisms to a level of accuracy reasonable for the given problem. By examining execution traces (*e.g.* Figure 2) from a representative RTOS systems, namely VxWorks© RTOS from WindRiver Systems, Inc. [20] the best selections for task management and inter-task communication were made. Complete details can be found elsewhere [21], but the experiments reveal that the

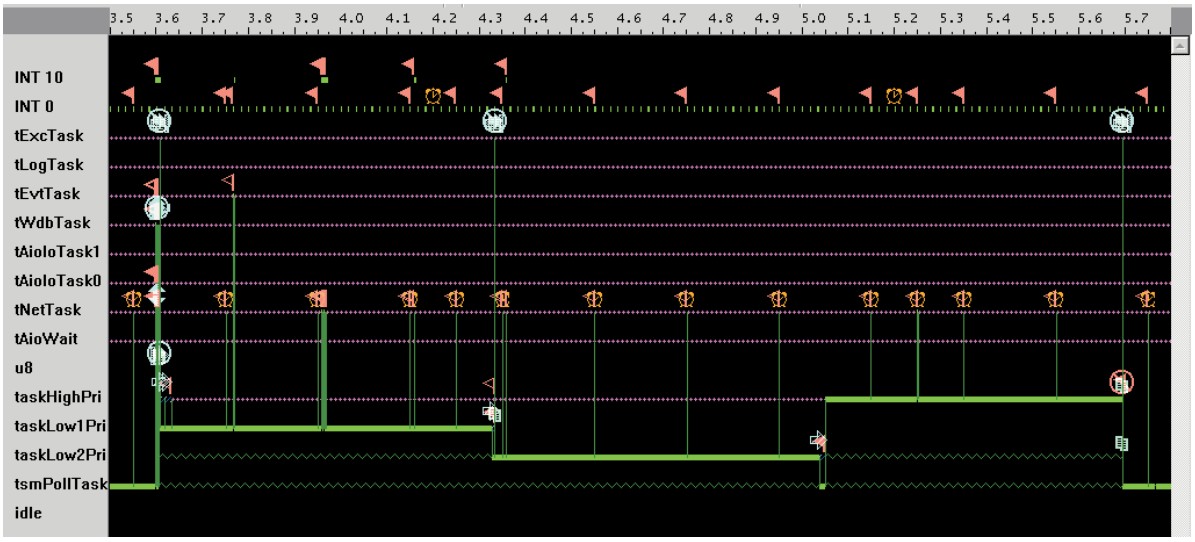

Fig. 2. VxWorks scheduling trace for three user tasks, taskHighPri, taskLow1Pri and taskLow2Pri, using an 90 MHz Intel Pentium© processor target system. *License courtesy of WindRiver Systems, Inc.*

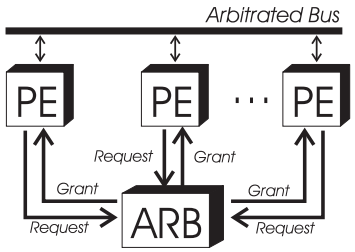

Fig. 3. Target system model

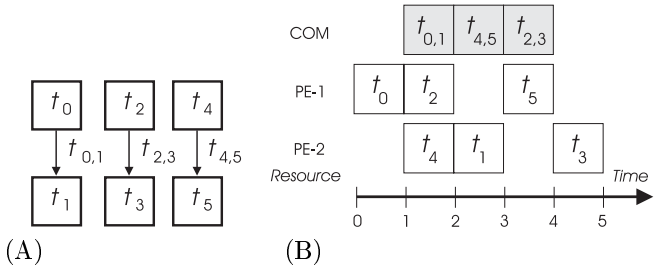

Fig. 4. A task graph (A) and possible no-overhead schedule (B)

most important preemptive task-scheduling overheads can be captured using two parameters: interrupt service time (IST) and context-switching time (CST).

Moving on to consider communication effects, even among those embedded system design efforts which include communication contention, rarely are bus protocols properly considered. A particular focus here is on single-bus systems, as depicted in Figure 3, which employ an *arbitrated* communication protocol. Note that in the heterogeneous case, each processing element (PE) may be of a different type and relative speed. As is apparent, communication is controlled via the process of arbitration.

Multi-processor PCI-based embedded systems (*e.g.* Small-PCI/Compact-PCI [4]), which use an arbitrated bus protocol, are the primary target architecture for the RAGS scheduling and co-design effort. Interestingly, the PCI standard [3] does not specify a particular arbitration policy but only says that it should be 'fair.' Typical implementations use a *round-robin* protocol, where the bus grant is given to the requester with the device index following the current user (modulo the number of bus devices). The model developed here follows a round-robin arbitration protocol for which results are presented. A *fixed* arbitration protocol model, where preference is given to the bus device with the lowest index, and a *fully schedulable bus* model which is unrealistic but useful for comparison purposes, have also been developed. Each PE is able to act as a PCI bus master, and note that PCI's 'hidden arbitration' feature implies that almost no overhead (other than several bus clock cycles) is required between bus master switching.

To illustrate arbitration, consider the task graph (dag) depicted in Figure 4.(A) where all task run-times are 1 on any PE, that communication time is also 1 unit, and that the deadline for task T_3 is 4. Figure 4.(B) depicts the situation where $T_0, T_2, T_5 \mapsto pe_1$ and $T_1, T_3, T_4 \mapsto pe_2$ ($\mapsto$ is the symbol used to represent allocation). At $time = 2$, both $T_{4,5}$ and $T_{2,3}$ are ready for communication over the COM resource (the bus). In a system (or design tool) which allowed arbitrary communication bus scheduling, one would be free to select the ordering of these to meet goals. However, this is not the case for bus systems which dictate protocol, in particular for this case, round-robin arbitration goes to pe_2 for communication task $T_{4,5}$ (representing $a_{4,5}$) before $T_{2,3}$ since pe_1 was the last to use the bus. In this situation, the deadline is missed. Alternatively, if fixed arbitration were used, or if $T_0 \mapsto pe_2$ then $T_{2,3}$ would be selected over $T_{4,5}$ and the deadline would be met.

The system model is summarized below:

- Communication can overlap computation, this implies that a reasonably capable communication controller is employed by each PE.

- Communication tasks are scheduled in accordance with the bus protocol, which is nominally round-robin arbitrated. Local ordering of communication tasks (those stemming from a particular PE) is possible.
- Communication tasks, once begun, are not preempted. This implies that the various PCI latency timers are set to sufficiently large values to accommodate data transfers. Note that the size of every data transmittal is known in advance.
- A context switch time (CST) is incurred prior to every task initiation or preemption.
- An interrupt service time (IST) is incurred on receiving PEs when data is to be transferred.
- As was verified experimentally, no additional overhead is incurred for inter-task data transfer for co-allocated (same PE) tasks. Once tasks are allocated, shared memory mechanisms can be used with fast local semaphores to manage this data [21].

The major extension of this model over that more typically used in real-time scheduling or co-design [12], [13], [15] is the inclusion of communication protocol and RTOS task scheduling and preemption overheads.

III. Problem Definition

Using the "*.name*" syntax to represent object attributes, the input, I, consists of: (i) a task graph set, $\mathbf{G} = (\mathbf{T}, \mathbf{a})$, where each task, $T_i \in \mathbf{T}$, has a run-time as a function of the PE type to which it is allocated (T_i.RUNTIME(PE_j)) and optionally a deadline, T_i.DEADLINE and each arc $a_{j,k} \in \mathbf{a}$ (labeled for convenience using the task from/to index), conveys an amount of data $a_{j,k}$.DATASIZE; (ii) a set of PE types, $\mathbf{PE} = \{PE_1, PE_2, \ldots\}$, each member of which has a COST attribute and an execution time for each of the tasks in $\mathbf{G}$; and (iii) a communication resource, *com*, which has a COST and DATARATE attribute. A *hardware configuration*, *config* $= \{\mathbf{pe}, com\}$, composed as a set PE instances $\mathbf{pe} = \{pe_1, pe_2, \ldots\}$ where each pe_i is an instance of a PE type from $\mathbf{PE}$ and one instance of *com* (unless a single PE solution exists). *Allocation* of a task, T_i, to a processor, pe_j, is denoted $T_i \mapsto pe_j$ for an individual task and $\mathbf{T} \Rightarrow$ *config* for a total mapping; in general, this notation is meant to include both the static assignment of the task to a PE and its priority setting (local scheduling).

From $\mathbf{G}$ an augmented task graph, $\mathbf{G}'$, is derived which includes tasks representing bus transfers which take place on behalf of inter-PE communication. Whenever the sender and receiver are allocated to the same PE (*e.g.* $T_i \mapsto pe_k$ and $T_j \mapsto pe_k$) then $T_{i,j}$.RUNTIME $= 0$, otherwise $T_{i,j}$.RUNTIME $= a_{i,j}$.DATASIZE/*com*.DATARATE, where *com*.DATARATE is the data-rate for the COM resource. An example of this augmentation is shown in Figure 5; Figure 4 also labels communication tasks in this way.

With this input specification, the co-design problem is to find a feasible configuration, $config_\star$, such that the sum of the constituent costs is minimized. This single-goal co-design optimization problem is readily recognized as being NP-hard, since the underlying inner-level allocation/scheduling problem is NP-complete [22], [12], and hence heuristic approaches are expected. RAGS and its inner scheduler make use of least common multiple (LCM) expansion for treating multiple input task graph sets, where each has its own period. After the input task graphs have been expanded in accordance with their LCM, both the inner-level scheduler and overall co-design method maintain a low-order polynomial complexity as long as their respective loop iteration counts are polynomial. LCM expansion—included in the construction of $\mathbf{G}'$ from $\mathbf{G}$—is not polynomially bounded in the worst case, however in practice may not be too problemmatic as task graphs with highly co-prime periods may not be encountered. Nonetheless, once this LCM expansion has occured, it has been shown [21] that the complexity of the overall co-design method RAGS is:

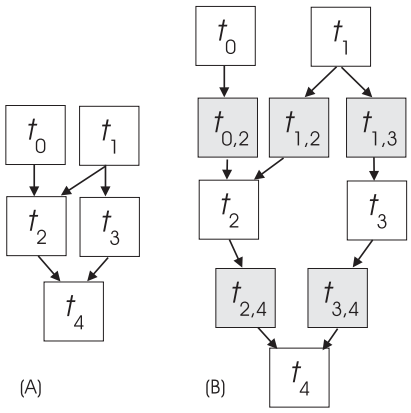

Fig. 5. (A) A task graph; (B) augmented with communication tasks

$$O\left(|\mathbf{PE}| + |\mathbf{V}|^3 + |\mathbf{E}|^2\right) \quad (1)$$

where $|\mathbf{PE}|$ is the number of PE types available for selection, $|\mathbf{V}|$ is the sum of the number of tasks in the (LCM expanded) task graphs and, $|\mathbf{E}|$ is sum of the number of arcs (inter task communication) in the expanded graph. Since the co-design technique makes direct use of a scheduler, this topic will be presented in the next section, including heterogeneous ALAP computation and use.

IV. Scheduling

This section presents the details of the allocation and scheduling routine used in RAGS. Like many NP-complete or NP-hard solvers, the scheduler's feasibility check entails finding a solution for the target system. That is, the result is not only 'pass' or 'fail' but, in the case of success, a complete system task allocation and schedule including protocol consistent inter-PE communication and OS overheads. While the next section will show the complete co-design algorithm, it is essentially an optimization technique that makes direct use of this scheduler.

```
InnerLoop(G', config, LoopCount)
    //
    // note that config = (pe, com)
    //
    Randomly initialize G' ⇒ config
    Compute ALAP for all tasks, T ∈ G'
    foreach T_i ∈ G'     // simple feasibility check
        if (T_i.ALAPEND − T_i.BEST-RUNTIME) < 0 then
            return false
    dowhile LoopCount not exceeded
        if UtilizationFailure(G', config) and Uallowed
            then     // Utilization Update
                r ← most overutil. resource (pe or com)
                if UpdateUtil(r, config) ¬ OK
                    then make random re-allocation
            else     // Speed Update
                Analyze schedule for the current config.,
                  finding start/end-times ∀T ∈ G'
                if all deadlines met then return true
                l ← {T_f} where T_f first ALAP violation
                if Speed(l, G', config)     // is successful
                    then make random re-allocation
    return false
```

Fig. 6. InnerLoop of RAGS

While analyzing a schedule (for a given allocation and priority setting) on a target system which includes various detailed RTOS and communication effects is rather straightforward, deciding upon a feasible schedule or co-designing both hardware and software (the scheduling) is very challenging. Arbitrated communication, in particular, greatly complicates the problem since communication task ordering is a function of computational task allocation. This issue is further exacerbated in the heterogeneous case where task run-times are also dependent on allocation. As can be seen, arbitration (as well as some other communication protocols) couples the separately hard problems of processor allocation/scheduling and communication scheduling.

The classic approaches of list-based or cluster-based scheduling, a summary of which can be found in [23], will not fare well in this arena. For example, in list-based scheduling a *priority-list* is maintained where each task at the head of the list is next selected for scheduling until all tasks are scheduled. The variants of list scheduling differ in the formation of the priority list (its ordering) and PE allocation for the selected task [23]. Even if the communication bus were included as a resource to schedule, the technique essentially assumes that each resource can be freely scheduled—but this is not at all the case in arbitrated bus systems. A similar argument can be made for cluster-based approaches. A further limitation in schedulers derived from these classes is that, for the most part, they are aimed at non-preemptive systems.

A new approach, shown as InnerLoop in Figure 6 attempts to make *incremental improvement* to the actual schedule analysis using the ALAP schedule as a guide. A task's ALAP end-time is [24]:

$$T_i.\text{ALAPEND} = \min \left(\begin{array}{l} T_i.\text{DEADLINE}, \\ \\ \min\limits_{T_j} (T_j.\text{ALAPEND} - T_j.\text{RUNTIME}) \end{array} \right) \quad (2)$$

where T_j are successor tasks to T_i. However, since the problem is heterogeneous the question of which run-time to use for ALAP computation arises. By using the *smallest possible* run-time for each task in $\mathbf{G}'$, including $T_{j,k} = 0$ for communication tasks, ALAP end-times become *true* ALAP times which *must* be met, fully independent of allocation.

Following the computation of the ALAP schedule, the allocation (and associated local scheduling, determined by relative priority settings) is first randomly initialized. A simple check, pertinent to heterogeneous systems, is next made to quickly assess feasibility; since the ALAP end-time is set in such a way that meeting it becomes a necessary condition, if any task's ALAP end-time less its best possible (heterogeneous) run-time is negative, then this hardware configuration can be quickly dismissed as infeasible. Following these initialization steps, the iterative improvement steps occur.

A maximum of LoopCount iterative improvement steps are then allowed, where each step falls into one of two types (as expressed in the **then** and **else** clauses): a 'utilization' update mostly performed by UpdateUtil, or a Speed update. Each of these steps are discussed in detail in the following subsections, but a quick, basic overview of the iterations is presented here. Since the 'real schedule analysis' is a computationally expensive operation, a simple utilization check, along with simple remedies, is made when the system is *over-utilized*. The value of these 'utilization steps' is limited however by the inherently NP-hard nature of even just allocating (without local scheduling nor communication effect and overhead considerations) tasks onto the heterogeneous processors. Hence the method limits their use appropriately. The alternative action in the iteration, which is the heart of the scheduler, is a complete schedule analysis followed by a check for schedule feasibility, in turn followed by corrective actions (a Speed update) should deadlines fail to be met.

A. Utilization-based Updates

A *utilization failure* occurs whenever the current allocation causes over-utilization of any resource, either a *pe* or *com*. This situation is easily detected (in the UtilizationFailure function here) by summing the run-time to period ratio for each PE and *com* for the given allocation—it is a *necessary condition* that utilization be less than or equal to 1. Since the schedule analysis is a rather 'expensive' operation, the utilization failure check and subsequent 'utilization update' are used as a quickly executing alternative. When the most over-utilized resource, r in algorithm, is a PE, say pe_k, the UpdateUtil routine tries to re-allocate the task with the largest run-time allocated to pe_k to another PE—these 'other PEs' are selected in order of least

to most utilized. Note that throughout the method, care is used to *avoid repeated configurations*, here at the scheduling and allocation level and later in RAGS at the hardware configuration level. Thus UpdateUtil may fail, and a random re-allocation performed, whenever the technique fails to produce an allocation which is distinct from those tried earlier (a list of prior allocations in canonical form is kept but not depicted in the algorithm for simplicity).

The other situation handled by UpdateUtil is when the *com* resource is the most over-utilized. Obviously, re-allocation of communication tasks is not an alternative in single-bus target systems. So here, the method performs a clustering operation. The communication task with the longest run-time is identified, $T_{j,k}$ along with its associated sending and receiving tasks (which are designated T_j and T_k, respectively, in accordance with the nomenclature defined earlier). Based on which will provide the lower total system utilization, which is the weighted sum of the utilizations of each resource, either T_j or T_k is allocated to the other's PE (thereby eliminating the communication task). Again this is done only if a repeated configuration will not result.

A final consideration within the regime of utilization updates is the term Uallowed appearing in InnerLoop, Figure 6. Achieving a utilization which is not over-utilized can be seen to be a difficult problem in itself [21] by showing equivalence with the NP-complete bin-packing problem [10]. We would not necessarily expect the rather simplistic possibilities for utilization updates, *e.g.* single task re-allocation or clustering, to be very effective for this problem. Hence the method provides a mechanism for *limiting* the utilization update steps via the Uallowed condition—which is true if the number of utilization updates performed is below the limit. Additional discussion concerning the setting of this parameter appears during the presentation of results.

B. Speed-*based Updates*

If the system is not over-utilized nor prevented from using utilization updates (via Uallowed), then a Speed-based update is performed. A complete **schedule analysis**, based on the model developed earlier including for example arbitrated communication, is performed. If all deadlines are met, then the scheduler (InnerLoop) is successful and a return value of true indicates this. If one or more deadlines have not been met then, following the schedule analysis, the earliest starting task which misses its deadline, T_f in InnerLoop, is identified and passed on a list, l, as an argument to Speed. The recursive routine Speed is, in general, trying to identify a useful allocation and local scheduling change to improve the schedule. This is done with respect to T_f, to begin with, using the current schedule analysis and the ALAP schedule as a source of valuable guidance.

The reason for selecting the earliest starting task which misses its ALAP schedule is twofold: first, since the ALAP end-times were set using the best possible run-times for all tasks, it is a *necessary condition* that this task be sped up, and secondly, T_f may be the cause of subsequent ALAP misses. In the case where an (potential) improvement cannot be identified, a random change is made to continue the iterations. In our experience with the method, this random step is typically used for about 1% of the iterations.

```
Speed(l, G', config)
    T_p ← task on list l with latest end-time
    if SpeedTask(T_p, l, config)
        then return true
        else
            m ← unique predecessors of each task ∈ l
            if m = ∅
                then return false     // nothing left
                else return Speed(m, G', config)
```

Fig. 7. Speed routine is the recursive entry point for the scheduling algorithm.

Figure 7 provides the algorithm for the Speed routine. It operates by first selecting the task on list l, T_p, which has the latest end-time (in the actual schedule analysis). T_p, along with l, becomes an argument to SpeedTask which assesses the possibility of improving the ending time of T_p without worsening the end-time of the entire list l. If such a (possible) improvement is found, the change is made and a positive return value is given to InnerLoop (indicating a viable change was made). Otherwise, Speed recurses to the *predecessor tasks* of the tasks on l, if there are any, and recursively applies itself to this predecessor list.

This recursive action can be viewed in the context of Figure 5. If T_4 was the first task to miss its deadline, then, barring a successful intervening SpeedTask call, the recursion would be to the list $\{T_{2,4}, T_{3,4}\}$ and then to the list $\{T_2, T_3\}$ and so forth. As can be seen, due to the construction of $\mathbf{G}'$ from $\mathbf{G}$, alternating task lists of PE-allocated and communication tasks are examined. The net result is that Speed is providing lists of tasks, including communication tasks, for examination by SpeedTask in the recursive fan-in of T_f. Since it is a necessary condition that T_f be sped up, this recursion is appropriate in that it aims to improve the maximum of the task end-times for each recursive level leading up to T_f.

Continuing in the top-down description of the routines comprising the overall method, SpeedTask is given in Figure 8. This routine is trying to identify possible scheduling changes which will improve the end-time of the list l given T_p (which is particularly important in that it is the task on list l with the latest current end-time). It is important to consider the end-time for the list l since, by construction via Speed, these are siblings in the task graph—improving the end-time of one at the equivalent expense of delaying another is not a fruitful improvement.

The detailed action of SpeedTask proceeds in accordance with a 'principle of least disturbance,' reserving more drastic changes for last. Algorithmic definitions for the routines used in SpeedTask are given in [21], while a description is given here. The first change considered is merely one of altering the priority of T_p while keeping all allocations the same. This is done in the RePrior routine—

```
SpeedTask(T_p, l, config)
    if RePrior(T_p, l, config) then return true
    if T_p is communication task then return false
    if MoveBlocking(T_p, config) then return true
        else
            endtime ← T_p.ENDTIME
            foreach pe_i ∈ R  such that T_p runs on pe_i
                pe_i.EST_END_TIME ←
                        EstEndTime(l, Allocate(T_p, pe_i), config)
            Sort PEs according to EST_END_TIME
            foreach pe_i   (such that T_p runs on pe_i)
                if pe_i.EST_END_TIME < endtime
                    AND MoveOK(T_p ↦ pe_i)
                then
                    T_p ↦ pe_i      // re-allocate
                    return true
    return false
```

Fig. 8. SpeedTask accesses the possibility of improving the maximum end-time of all tasks on the list, l, with respect to changes in relation to T_p. When a new mapping is found (third to last line), T_p.PRIORITY is set such that it will be the highest among all co-allocated tasks which execute within the time range $[T_p.\text{READY}, T_p.\text{START}]$.

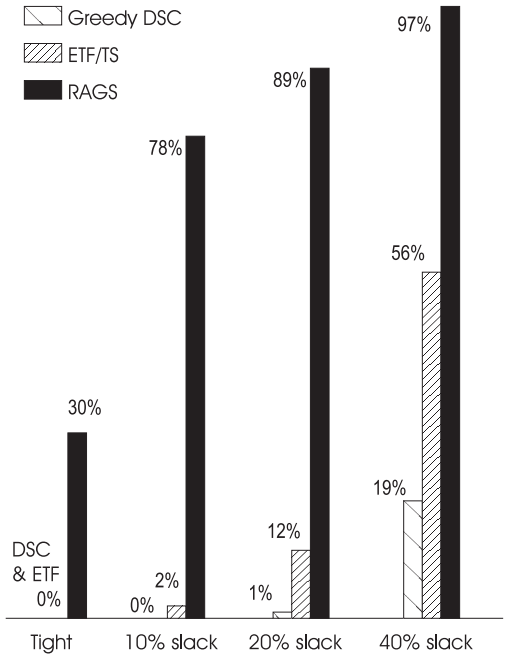

Fig. 9. Scheduling success rate (%) comparison of *non-preemptive*, no overhead scheduling of different methods. CPU times are: 53.3, 6.1 and 7450.0 seconds for all 100 cases for Greedy DSC, ETF/TS and RAGS, respectively.

note that if l (and therefore T_p) is composed of communication tasks, then re-priorization: (i) affects only local outgoing communication scheduling and is subject to bus arbitration (see Section II); (ii) is the only possibility since single-bus systems are considered. Thus only RePrior applies for communication tasks, while the remaining steps apply to PE-allocated tasks. The next consideration is via the MoveBlocking routine. Here, consideration is given to re-allocating tasks which may be *blocking* T_p. The final consideration is a re-allocation and re-prioritization of T_p. An examination of the algorithm will reveal that a re-allocation to each PE that supports T_p in order of best to worst end-time (estimated with EstEndTime) for the list l is done. If the end-time improves, and a duplicate configuration is not presented (checked with MoveOK), then it is accepted.

While the InnerLoop and Speed framework provides the general search strategy in the quest for schedule improvement, the low-level routines in SpeedTask, namely RePrior, MoveBlocking and EstEndTime, provide the 'knowledge' of the system model. These routines perform their respective functions in the context of the current system model, for example, when assessing a task re-allocation, EstEndTime incorporates factors concerning the impact arbitration will have. An actual (accurate) schedule analysis occurs after each change at the InnerLoop level and is always the sole basis for feasibility checking, but these low-level routines provide quickly evaluated *schedule estimates* for the purposes of design space exploration. Even so, the estimates may be quite accurate in that they are done in the context of the current schedule analysis.

C. Comparison with List- and Cluster-based Schedulers

Even though the scheduler in RAGS is aimed at systems which exhibit non-ideal effects, a comparison to traditional list- and cluster-based approaches can be made by applying RAGS to an idealized situation. This is done by removing all communication overhead/contention, RTOS effects and using a *non-preemptive* schedule for 100 single-rate samples on homogeneous PEs. In accordance with this simplified problem form, several modifications to the RAGS method were made. Note that only changes to the schedule analyzer itself, as well as the routines aimed at estimating end times (EstEndTime), re-prioritization analysis (RePrior), are necessary.

The particular list-based method used for comparison is the *Earliest Task First (ETF)* with topological sort (ETF/TS) [23], [25] while *DSC* [2], coupled with the allocation and scheduling method in [26] modified using the longest sequence (a critical path) clustering [12] technique, is used as the representative clustering method. Figure 9 shows the success rates (ratio of the number of cases meeting deadlines to total number of cases) for each of these methods as the deadline slack is adjusted (see [21] for additional details). The superiority of the RAGS scheduler is clear in this figure. Although we might expect various improvements and special modifications to list- and cluster-based scheduling to provide somewhat better results, it is not clear that such an approach will reach the results obtainable by RAGS. Of course, the adaptability of such methods to the real-life situations capable of being addressed by RAGS is also unclear.

```
RAGS(I, DesignSteps)
    // Note that:  I = (G, PE, COM)
    // COM is not a set but a single bus type

    C ← {}    // C is the list of prior configs.
    config* ← ∅
    create G' from G
    dowhile DesignSteps not exceeded
        config_c = copy of last config. on C
        if config* ≠ ∅ then
            if single-PE elimination on config_c then
                goto TRY-CONFIG
            if single-PE substitution on config_c then
                goto TRY-CONFIG

        // no reduced cost config.--hill climb
        dowhile config_c not duplicated on C
            Add a PE instance to config_c
                  // may add randomly or balanced

        TRY-CONFIG:    // Got a config to try
        Append config_c to C
        if InnerLoop(G', config_c, LoopCount) successful then
            if config_c.COST < config*.COST then
                config* ← config_c

    report config*
```

Fig. 10. The RAGS routine. In the 'Add PE' step two distinct possibilities where examined: either adding completely randomly or in a 'balanced' manner (see text).

V. Co-Design

The full co-design method can now be described in detail. While the 'inner loop' level (the heterogeneous scheduler just described) is much different than schedulers derived from list- or cluster-oriented approaches, the 'outer loop' is based on a *hill-climbing* approach (the bimodal technique reported in [21] does not have hill-climbing). The possibility of changing prior decisions during a search or optimization is called *backtracking*. Since the allocation/priority of any task may change many times during the iterations in InnerLoop the scheduler is backtracking—this is quite unlike traditional list- or cluster-based scheduling which typically make one-time fixed decisions in the scheduling process. At the outer-level, backtracking is common in co-design efforts [27], [28], [29], [30], [31], although hill-climbing techniques to be described here are tailored to the problem at hand.

The complete co-synthesis method, RAGS, is listed in Figure 10. The initial steps are as follows: set C, the list of previously explored configurations, to the empty list; set $config_\star$, the best solution found, to nil; create the communication task augmented graph, $\mathbf{G}'$, from $\mathbf{G}$. The iterative method then loops for a count of DesignSteps, a user provided constant, followed by reporting of the best solution.

The steps within the loop provide the tailored hill-climbing operations. A new configuration, $config_c$, is initialized as a copy of the last configuration on the list C. If a solution has never been found, the possible PE elimination or substitutions steps are skipped and only a PE addition step(s) is performed. In relation to hill-climbing the PE elimination step is a descending cost search while the PE addition step is ascending cost search (the PE substitution step may be either). Skipping the descending steps prior to finding the first solution aids in avoiding the initial configurations which do not have sufficient system processing capabilities. However, once a solution has been found the technique favors looking at reduced cost configurations in preference to increased cost solutions. Using the last configuration as a starting point, the method first tries PE elimination which alters the configuration by eliminating each PE instance in turn. If none of these alterations result in a non-repeated configuration (using list C), PE substitution, where an instance of each PE type is substituted for each existing PE instance, is tried. If all of these possibilities also result in duplicated configurations then a hill-climbing step is used. Here, PE instances are *added* to the configuration, $config_c$, until a unique, unrepeated configuration is found.

Once a new configuration, $config_c$, is found to try, it is added to C and then checked for feasibility using the scheduler (InnerLoop). In addition to avoiding resource configurations that are easily decided to be insufficient during the initial search, another unique feature is the method for adding PEs during the hill-climbing. Rather than add them randomly, they are added in such a way to maintain balance among PE types. For example, if there are 4 PE types ($|\mathbf{PE}| = 4$) and the current configuration has 2 PE instances say one of type PE_1 and one of PE_4, then the next one to be added will be of type either PE_2 or PE_3 (with equal probability). There may be some concern that this unwisely *biases* the general hill-climbing approach, but subsequent comparisons show improved performance. The avoidance of recently repeated configurations and careful selection of a new design point following local exploration are the hallmarks of the Tabu Search method [32], [33]. RAGS implements these principles by *fully* avoiding repeated configurations (these checks do not alter the computational complexity of the method as their contribution is dominated by the other functions) and exploring the design space in a balanced way looking at possible solutions from lower to higher cost.

VI. Examples and Results

As a first example, a comparison against two other co-design tools is presented. As was the case at the scheduling level, due to the uniqueness of the target system, which includes RTOS effects and communication protocols, we must simplify (idealize) the model to enable the comparison. Table I compares SOS [22], MOGAC [27], [28] and RAGS using the single-bus examples of SOS. Note that MOGAC finds the same solutions as SOS and hence only it's CPU time is shown in the Table. Although these examples are small, none having more than 9 tasks, the MILP solutions given by SOS are known to be optimal. As can be seen, RAGS compares favorably with MOGAC, even

TABLE I

SINGLE-BUS EXAMPLE COMPARISON WITH SOS AND MOGAC. CPU TIMES PRESENTED ARE *not* ON THE SAME COMPUTER TYPES (SEE TEXT).

	SOS			MOGAC	RAGS		
Example	COST	Sol'n	CPU time	CPU time	COST	Sol'n	CPU time
SOS II-4	7	pe_1, pe_3	28s	3.3s	7	*same*	<1s
SOS II-7	5	pe_2	37s	2.1s	6	pe_1, pe_3	<1s
SOS V-6	10	$2\times pe_1$, pe_3	107.3m	*n/a*	10	*same*	682s
SOS V-7	6	pe_1, pe_3	89.5m	*n/a*	6	*same*	<1s
SOS V-15	5	pe_2	61.5m	*n/a*	6	pe_1, pe_3	<1s

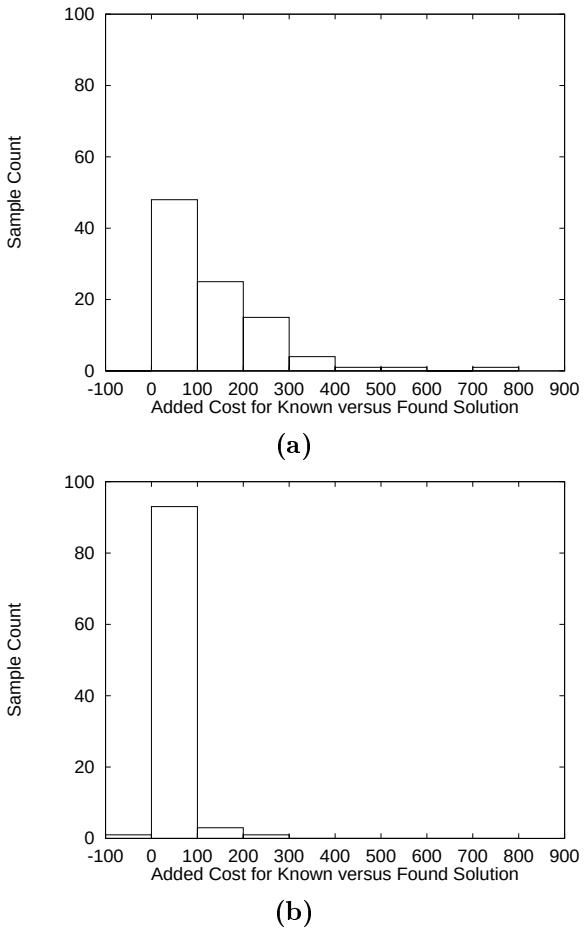

Fig. 11. Comparison of results obtained by varying DesignSteps, (a) 20 and (b) 100. Bar height is the number of samples within the x-axis range and 0 on the x-axis is the cost for the known solution.

though RAGS is using the real-analysis technique aimed at target systems with non-ideal behavior.

For the next experiment, a set of 100 random input samples sets were generated such that: there are 32 tasks total organized into 2 task-graphs; 3 PE types are available; CST and IST average 2% and 1.5% of average task runtimes; a heterogeneous solution for 4 PE instances is known such that PE utilization is over 83% and bus utilization is over 31% in the known solution. The solution is only known to exist when communication can be scheduled arbitrarily, however the target system is defined to use round-robin arbitration (see [34], [21] for details concerning case generation). Table II summarizes the results as parameterized by the user provided constants DesignSteps and LoopCount. As can be seen, increasing values for the iteration counts provides improvement in the co-design goal, cost, while of course requiring additional CPU time. The CPU times given are the average per case seconds for a (modest) Sun SparcStation-20 system. The average cost for the known solution is 493.6 (with 4 PEs). As another point of consideration, the final column in Table II shows the results when DesignSteps and LoopCount are set to 50 and 10,000, respectively. While this improves the results over the 50/5,000 settings, it is inferior to the 100/5,000 setting in both system cost/number of PE metrics and in CPU time.

The improvement garnered by increasing DesignSteps can also be visualized. For example, Figure 11 shows sample density (count) for the per-case difference of found solution cost versus that of the known solution. In both cases, LoopCount is fixed at 5,000—these graphs correspond to columns 2 and 4 of Table II. Figure 11.(a), depicts the situation when DesignSteps is set to 20 while Figure 11.(b) shows the situation for a setting of 100 DesignSteps. As is expected, increasing DesignSteps results in finding lower cost solutions, in one case better than the known solution (which was previously described as not necessarily being the optimal solution).

As a point of further comparison, two examples from Li and Wolf [7] were used. The first is a real-world derived MPEG-1 example requiring two data-dependent task graphs; this example contains a video processing task graph comprised of 11 tasks and an audio processing task graph composed of 5 tasks. Each of these runs at different periodic rates and the problem is presented as a four PE homogeneous scheduling problem including bus contention. Task execution and communication is non-preemptive. Since the periods of the two task graphs are rather co-prime, RAGS addresses this by slightly lowering the period of the audio task-graph to lessen the LCM effects. RAGS successfully schedules this on four PEs as well, finding a solution that requires approximately 26.5% communication bus utilization and PE utilizations ranging from 74.3% to 93.7%. MPEG-1's moderately high communication utilization illustrates the importance of accurately including communication effects during synthesis.

TABLE II

RESULTS FROM 100 CASE STUDY. UTILIZATION STEPS LIMITED TO ONE-HALF LoopCount.

	DesignSteps/LoopCount				
	10/5000	20/5000	50/5000	100/5000	50/10000
No. Sol'n Found	95	98	98	98	98
Avg. Cost	606	597	523	506	520
Avg. Number PEs	5.11	5.07	4.41	4.23	4.32
CPU time (s)	897.3	1389.1	3216.7	6526.2	7659.2

The other example presented in this reference ([7] as Fig. 9) is a co-design problem. This example uses non-preemptive, homogeneous processors communicating over a single bus. As before, the bus includes contention and is non-preemptive. RAGS finds the same two PE solution as Li and Wolf did (note that the original source of the example only found a three PE solution). Although each of these examples do not include the realistic effects of overhead nor bus arbitration both of which are a particular focus of the method, these serve to further validate the RAGS approach.

Experiments were also carried out to assess the potential bias introduced by adding PEs in a balanced way, rather than totally *randomly* (as would be more typical in hill-climbing approaches). These experiments revealed, for the settings of DesignSteps and LoopCount in Table II, slightly inferior results (as judged by average cost and number of PEs) as compared to the balanced PE addition technique presently used—*e.g.* for the 20/5000 DesignSteps / LoopCount setting a cost of 605 is obtained (rather than 597). Thus, the balanced add-in method was selected.

VII. CONCLUDING REMARKS

A new scheduling and co-design technique, RAGS, which exploits an actual schedule analysis has been described. Use of an actual schedule analyzer was motivated by the need to include significant non-ideal behaviors stemming from both RTOS managed task execution as well as protocol-based communication, specifically arbitrated communication. In addition to the incorporation of an accurate schedule analyzer, several other unique concepts have been developed and demonstrated. These primarily include the use of ALAP as a guide in heterogeneous system allocation and scheduling and the employment of iterative improvement methods that include backtracking at both the scheduling and co-design levels.

This technique was used for preemptively scheduled heterogeneous multiprocessor systems with RTOS-derived overheads using an arbitrated bus for communication, as well as compared with scheduling and co-design tools aimed at more idyllic circumstances. Generally, the current approaches of list- and cluster-based scheduling are ill-suited to this domain which exhibits tight coupling of the difficult problems of task allocation and communication scheduling. RAGS was also shown to provide superior results against these methods in the idealized system regime, although the improvement comes along with an increase in run-time. This is to be somewhat expected as RAGS is not at all targeted to idealized systems. With the attendant modifications to the scheduler and the RePrior and EstEndTime routines, the approach should be well-suited to addressing other non-idealized domains.

REFERENCES

[1] A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 686–701, June 1993.

[2] T. Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 951–67, Sept. 1994.

[3] Tom Shanley and Don Anderson, *PCI System Architecture*, Addison-Wesley, Reading, MA, 3rd edition, 1995.

[4] PCI Industrial Computers Manufacturing Group (PICMG), *CompactPCI Specification – Short Form*, r2.1 edition, Sept. 1997.

[5] John P. Lehoczky and Lui Sha, "Performance of real-time bus scheduling algorithms," *ACM Performance Evaluation Review, Special Issue*, vol. 14, no. 1, pp. 44–53, May 1986.

[6] Ti-Yen Yen and Wayne Wolf, "Communication synthesis for distributed embedded systems," in *1995 IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD'95)*, 5–9 November 1995, pp. 288–94, San Jose, CA.

[7] Yanbing Li and Wayne Wolf, "Hierarchical scheduling and allocation of multirate systems on heterogeneous multiprocessors," in *Proc. of IEEE European Design and Test Conference (ED&TC'97)*, 17–20 March 1997, pp. 134–9, Paris, France.

[8] Chang Yeol Choi and Heonshik Shin, "Impact of bus arbitration on the schedulability of real-time shared-bus multiprocessors," in *Proc. IEEE Region 10's Ninth Annual Conference*, 22–26 August 1994, pp. 602–6, Singapore.

[9] J. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, pp. 384–93, Oct. 1975.

[10] Michael R. Garry and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, NY, 1979.

[11] Ellis Horowitz and Sartaj Sahni, "Exact and approximate algorithms for scheduling nonidentical processors," *Journal of the ACM*, vol. 23, no. 2, pp. 317–27, Apr. 1976.

[12] Stefan Rönngren and Behrooz A. Shirazi, "Static multiprocessor scheduling of periodic real-time tasks with precedence constraints communication costs," in *Proc. 28th Annual Hawaii IEEE Int. Conf. on System Sciences*, 3–6 January 1995, pp. 143–52.

[13] Peter Bjørn-Jørgensen and Jan Madsen, "Critical path driven cosynthesis for heterogenious target architectures," in *Proc. 5th IEEE/ACM Int. Workshop on Hardware/Software Co-Design (CODES/CASHE'97)*, 24–27 March 1997, pp. 15–9, Braunscheig, Germany.

[14] Jean-Marc Daveau, Tarek Ben Ismail, and Ahmed A. Jerraya, "Synthesis of system-level communication by an allocation approach," in *Proc. of the 8th ACM/IEEE Int. Symp. on System Synthesis*, 13–15 September 1995, pp. 150–5, Cannes, France.

[15] Gilbert C. Sih and Edward A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. on Parallel and Distributed Computing*, vol. 4, no. 2, pp. 175–87, Feb. 1993.

[16] Tarek Ben Ismail, Mohammed Abid, and Ahmed Jerraya, "COS-

MOS: A codesign approach for communicating systems," in *Proc. 3rd IEEE Int. Workshop on Hardware/Software Co-Design (CODES/CASHE'94)*, 22–24 September 1994, pp. 17–24, Grenoble, France.

[17] Michael Gonzalez Härbour, Mark H. Klein, and John P. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Trans. on Software Engineering*, vol. 20, no. 1, pp. 13–28, Jan. 1994.

[18] John P. Lehoczky and Sandra Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *Proc. of IEEE Real-Time Systems Symposium*, 2–4 December 1992, pp. 110–23, Phoenix, AZ.

[19] Yanbing Li, *Hardware-Software Co-Synthesis of Embedded Real-Time Multiprocessors*, Ph.D. thesis, Princeton University, Mar. 1998.

[20] WindRiver Systems, Alameda, CA, *VxWorks Programmer's Guide (v5.3.1)*, Mar. 1997.

[21] David L. Rhodes, *Real-analysis, ALAP-Guided Synthesis of Real-time Embedded Systems*, Ph.D. thesis, Princeton University, Nov. 1999.

[22] Shiv Prakash and Alice C. Parker, "SOS: Synthesis of application specific heterogenious multiprocessor systems," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 338–51, 1992, Academic Press.

[23] Marios D. Dikaiakos, Kenneth Steiglitz, and Anne Rogers, "A comparison of techniques used for mapping parallel algorithms to message-passing multiprocessors," in *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, 26–29 October 1994, pp. 434–42, Dallas, TX.

[24] Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, 1994.

[25] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal of Computing*, vol. 18, no. 2, pp. 244–57, Apr. 1989.

[26] T. Yang and A. Gerasoulis, "PYRROS: static scheduling and code generation for message passing multiprocessors," in *Proc. of Sixth ACM Int. Conf. on Supercomuting*, July 1992, pp. 428–37, Washington, DC.

[27] Robert P. Dick and Niraj K. Jha, "MOGAC: a multiobjective genetic algorithm for the cosynthesis of hardware-software embedded systems," in *Proc. of IEEE Int. Conference on Computer-Aided Design (ICCAD'97)*, 9–13 Nov. 1997, pp. 522–9, San Jose, CA.

[28] Robert P. Dick and Niraj K. Jha, "MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of hierarchical heterogeneous distributed embedded systems," *IEEE Trans. on Computer-Aided Design*, vol. 17, no. 10, pp. 920–35, Oct. 1998.

[29] Hyunok Oh and Soonhoi Ha, "A hardware-software cosynthesis technique based on heterogeneous multiprocessor scheduling," in *Proc. of IEEE/ACM International Workshop on Hardware/-Software Co-Design (CODES'99)*, 3–5 May 1999, pp. 183–7, Rome, Italy.

[30] Hyunok Oh and Soonhoi Ha, "A static scheduling heuristic for heterogeneous processors," in *Proc. of 2nd Int. Euro-Par Conference*, Aug. 1996, vol. 2, Lyon, France.

[31] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha, "COSYN: hardware-software co-synthesis of heterogeneous distributed embedded systems," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 92–104, Mar. 1999.

[32] Fred Glover, "Tabu search - part I," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[33] Fred Glover, "Tabu search - part II," *ORSA Journal on Computing*, vol. 2, no. 1, pp. 4–32, 1990.

[34] David L. Rhodes and Wayne Wolf, "Co-synthesis of heterogeneous multiprocessor systems using arbitrated communication," in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD'99)*, 11–17 Nov. 1999, San Jose, CA.