# A Software Synthesis Tool for Distributed Embedded System Design

D.-I. Kang  R. Gerber  L. Golubchik  J. K. Hollingsworth
Department of Computer Science and UMIACS, University of Maryland
{dikang,rich,leana,hollings}@cs.umd.edu

M. Saksena
Department of Computer Science, Concordia University
manas@cs.concordia.ca

## Abstract

We present a design tool for automated synthesis of embedded systems on distributed COTS-based platforms. Our synthesis tool consists of (1) a graphical user interface for input of software layouts, which maps tasks to resources and (2) a constraints solving engine, which allocates local resources to tasks, all with the goal of meeting specified performance criteria. Our tool differs from previous work in that it allows (a) use of stochastic (rather than worst-case) models of resource usage and (b) resource sharing among components. Our approach uses analytical approximate solutions for quick estimates of the desired performance measures. These estimates permit an efficient search of the possible design space. Once candidate designs are determined, they are validated through a simulation model. We demonstrate the efficiency and robustness of this tool on a synthetic aperture radar benchmark.

## 1 Introduction

Real-time embedded systems are characterized by their end-to-end real-time constraints, throughput requirements, as well as the characteristics of external inputs and functional components. In this paper, we focus on design of embedded systems that are dominated by throughput and latency requirements, e.g., digital signal processing systems, rather than control-type embedded systems where reactive behavior is dominant. Within the class of such embedded systems, we assume the following in our design synthesis. The system's processing requirements are specified as data-flow graphs with producer-consumer relationships. Furthermore, high throughput requirements, typical of such systems, (e.g., on the order of $10^9$ FLOPS to $10^{12}$ FLOPS, or more), necessitate the use of multiple processing units. Consequently, we consider *distributed* designs, where the use of COTS (Commercial Off-The Shelf) processors and software tasks results in cost-effective systems. Resource sharing through multi-tasking is desirable not only due to cost considerations but also due to the resulting flexibility that allows system reconfiguration, for instance, after hardware failures. Thus, low-cost NOWs (networks of workstations) are good candidates for building such real-time embedded systems. Lastly, we note that satisfaction of end-to-end latency constraints is of great importance, for at least a sub-class of these systems.

Traditional real-time system design approaches are rigid and tend to under-utilize system resources while often resulting in costly systems. They assume no data loss is allowed, i.e., the design must ensure that input samples are processed at least at the specified input sample rate. Consequently, traditional designs are built using dedicated and often special hardware and software components under worst-case execution time assumptions. Hence a resource is fully dedicated to one specific function.

In such a traditional approach, the design problem can be summarized as follows: (1) *placement of tasks on resources*, i.e., aggregation and decomposition of functional components to fit resource capacities; (2) *estimation of worst-case execution times*, where achievement of tight execution time bounds is virtually impossible due to hardware effects such as caching; and (3) scheduling of tasks on each processor to maximize throughput, where the performance of the resulting design is evaluated through simulation or through measurements of the actual system. The custom task placement approach results in inflexible design procedures, where a slight change in requirements might invalidate the entire placement. The absence of analytical performance evaluation techniques results in extremely long and inefficient design cycles.

In contrast, our approach supports a more flexible and cost-effective design via automatic synthesis. The hardware/software interaction is captured through design parameters. As a result, our design methodology is minimally platform-dependent and can be used to design a COTS-based system. The main differences between our approach and traditional design methods are as follows (these are summarized in Figure 1).

First, stochastic (rather than worst-case) resource usage is considered, where each task's resource requirements are expressed statistically, in terms of a discrete Probability Distribution Function (PDF) to characterize the resource requirements for one execution instance of that task. Second, resource sharing is allowed among tasks via multi-threading using proportional resource schedulers. Third, our approach supports two distinct design requirements: (1) deterministic throughput guarantees with soft latency guarantees, where our analytical models are used to estimate average end-to-end latency; and (2) stochastic throughput guarantee with firm latency guarantees, where our analytical models are used to estimate average end-to-end throughput. Fourth, the choice of candidate task placements is left to the designer, and thus the scheduling problem is reduced to finding load assignments for tasks.

The design process includes evaluation of and comparison among several software layouts, or threads-to-resource mappings, on several hardware platforms. The end-to-end constraint satisfaction of the resulting designs is automatically verified through the use of stochastic analysis techniques which estimate the feasibility of a potential design. Candidate feasible solutions are then validated through detailed simulations. Backup designs are included as a provision for hardware failures.

This paper presents a general outline of our embedded distributed systems design methodology, and focuses on the CAD tool which implements this methodology. A good design methodology and the corresponding synthesis tool should be: (1) *scalable* to large systems; (2) *flexible*, such that small changes in design requirements do not necessitate complete re-design of the entire system; (3) *platform-independent*, where the properties of the underlying hardware are parameterized in the design; and (4) *fast* and reasonably *accurate* when evaluating feasibility of candidate designs — this suggests the use of analytical techniques to help prune possible designs to a few "good" candidates. We believe our tool exhibits all these features, and in the rest of the paper we explain how our tool works via a running example of designing a synthetic aperture radar system.

## 2 Related Work

In our previous work, we proposed a semi-automatic end-to-end design scheme for hard real-time uniprocessor systems [7, 8]. We relaxed the precondition that service intervals and deadline parameters are always known before design time. Rather we derived each task's constraints — service interval, offset, and deadline — automatically from the system's end-to-end delay requirements. Our scheme was extended in the design of hard real-time distributed systems [18], where the authors partitioned end-to-end delay requirements statically via heuristics, to compute the local delay bounds.

Although our run-time system model assumes a soft real-time model from the viewpoint of end-to-end behavior, we rely on traditional hard real-time uniprocessor scheduling theory at each resource [3, 9, 15, 16]. Much of such scheduling theory work predicts load thresholds under which tasks, running on the same resource, are guaranteed to meet their deadlines. These load thresholds are used in our tool as upper limits on the amount of resource sharing allowed among tasks on a particular resource (see Section 3 for details).

For the analytic latency estimations, used by the constraints solving engine (see Figure 3), we rely on a form of proportional resource sharing. Time Division Multiplexing (TDM) may be the simplest form of proportional resource sharing. Even though TDM is not as fair as other recently proposed proportional resource share service disciplines, such as the "Virtual Clock Method" [21], "Fair-Share Queuing" [4], "Generalized Processor Sharing(GPS)" [1, 17], and "Rate Controlled Static Priority Queuing" (or RCSP) [20], it guarantees a minimum amount of a resource to a task, within a service interval. Also TDM-based schedulers and drivers are fairly easy to implement, using credit/debit token-bucket schemes. Using the more recent proportional resource share techniques may reduce end-to-end latency, but at the cost of additional complexity of the analysis (and possibly implementation).

Our work on distributed real-time system design [12, 13, 14] provides the theoretical basis for our design tool. In [14, 13] we present a technique for designing distributed systems with statistical real-time performance constraints, under different software layouts as well as different performance metrics.

Throughout this paper, we use the RASSP SAR benchmark as a running example for our design scheme. (Due to lack of space, we omit full details of case study of the SAR benchmark using our design approach; they are given in [12].) The RASSP SAR (Synthetic Aperture Radar) was posed as a "challenge" signal-processing problem for COTS-based development. In the realm of advanced radars, the SAR's throughput is quite modest — 1.1 GFLOPS for processing three polarizations, at the highest input pulse frequency. However, SAR is a good candidate to illustrate the features of our tool since its computation flow is well defined, and it has a real-time end-to-end latency limit of 3 seconds. Moreover, its computational needs can be scaled by changing resolution and throughput parameters. The major phases of the RASP SAR are:

- Video to Baseband I/Q Conversion stage: A pulse's samples are converted and filtered from video format to in-phase and quadrature(**IQ**) bands, represented internally by complex numbers.
- Range Compression: Range compression consists of three steps. First, an equalization filter (**EQ**) normalizes the data for range processing. Then a discrete Fourier transform (the **RDFT** phase) converts the data to the frequency scale. The result is run through another filter (the **RCS** phase), to compensate for cross-section variations produced by the DFT.
- Corner Turn: The corner turn(**CT**) is an all-to-all communication step, and thus a bottleneck in most adaptive radars. The RCS phase produces 2048 range coefficients for each of the 512 pulses in a frame, but before the pulse compression stage can start it requires all 512 readings for that range. Hence, the corner-turn's job is to accumulate the 512x2048 matrix, and then send the columns to the pulse compression stage.
- Pulse Compression: In this radar, 2 sequential frames form a processing array of size $2048 \times 1024$, where columns correspond to pulses, and rows correspond to range gates. The actual pulse compression phase consists of three steps: (1) a discrete Fourier transform (denoted **ADFT**); (2) a convolution (denoted **KM**, for "Kernel Multiplication"); and an inverse Fourier transform (denoted **AIDFT**, for Inverse Discrete Fourier Transform).

Although SAR is composed of multiple channels, we present our examples using a single channel, for ease of illustration. (An example of a one channel design is depicted in Figure 2(a)).

Hundreds of books have been written on radar systems; however relatively little has been written about deploying high-performance radars on clusters of general-purpose computers [22, 2, 5, 11]. To our knowledge, no work prior to ours has been done on using stochastic performance models for the purpose of system synthesis.[1] Finally, a variety of real-time systems tools are surveyed in [10]; however, most tools

[1]Note that some results in this area are classified as military secrets - and hence not published.

| Aspect | Traditional Approach | Our Approach |
|---|---|---|
| Multi-Threading | Not Used | Available |
| Resource Sharing | Not Allowed | Allowed |
| Modeling of | Deterministic | Stochastic |
| Resource Usage | Worst-Case Exec. Time | Discrete PDF |
| Performance Estimation | Simulation or Build and Measure | Analytical Approximations validated via Simulation |
| Local Resource Tuning | Trial and Error Using Measurement or Simulation | Automatic using Analytical Techniques validated via Simulation |
| Hardware Assumed | Special Purpose | COTS |
| Fault-Tolerance | | |
| Method | Hardware Redundancy | Software Redundancy using Excess Capacity |
| Recovery | Hardware Replacement | Task Relocation |
| Performance Model | Live or Die | Gradual Degradation |

Figure 1: Differences between Traditional and Our Approach

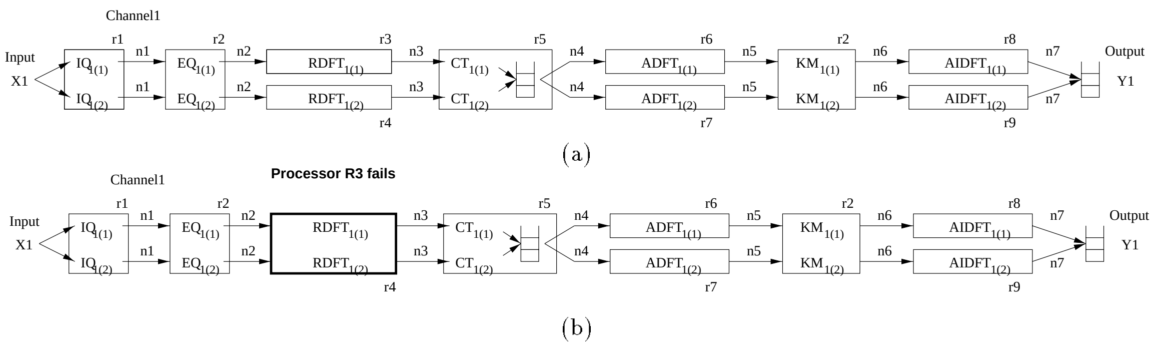


Figure 2: A layout of one channel of the SAR benchnark (a), and reconfiguration for failure of "R3" (b).

tend to be for verification and analysis rather than design synthesis. Existing software synthesis tools [19] tend to be for control-type reactive systems. A prototyping and simulation environment for real-time systems is described in [6].

## 3 Tool Description

The architecture of our tool, illustrated in Figure 3, consists of the following components: "Interactive Graph Editor", "Constraints Parser", "Load Threshold Estimator", "Simulator", and "Slack Distributor". The "Interactive Graph Editor" is used by the designer to specify (a) software layouts, i.e., to input thread-to-resource mapping information, and (b) PDFs of task execution times. The "Constraints Parser" interprets the constraints file which specifies design and system constraints.

The "Load Threshold Estimator" analyzes designs to estimate satisfaction of end-to-end design constraints and tunes local resources until these constraints are satisfied (or it is determined that this is infeasible). The theories behind the "Load Threshold Estimator" is presented in detail in [13]. The "Simulator" is used to validate candidate designs produced by the "Load Threshold Estimator".

The "Slack Distributor" distributes the resource slack available in the resulting design, either to increase throughput or to provide backup designs in the event of hardware failure. More specifically, when a resource fails, we achieve fault-tolerance by using the "Slack Distributor" to redistribute tasks assigned to the faulty resource to other resources in the system. Given the new mapping, if sufficient resource slack is available to "absorb" the tasks from the failed component, then the system still satisfies the original design constraints. Otherwise, the reconfigured thread-to-resource mappings are passed to the "Load Threshold Estimator" which determines a new, possibly degraded design.

Lastly, the arrows in Figure 3 depict flow of control and data. The solid lines denote forward flow while the dashed lines denote back-tracking, which occurs when a candidate solution is determined not to satisfy the design constraints.

### 3.1 Tool Inputs/Outputs

The tool inputs include "Software Layout", "Thread to Resource Mapping Information", "Design Constraints", "System Constraints", "Design Factors", "Per-Component Parameters" (such as PDFs of execution times). As outputs,

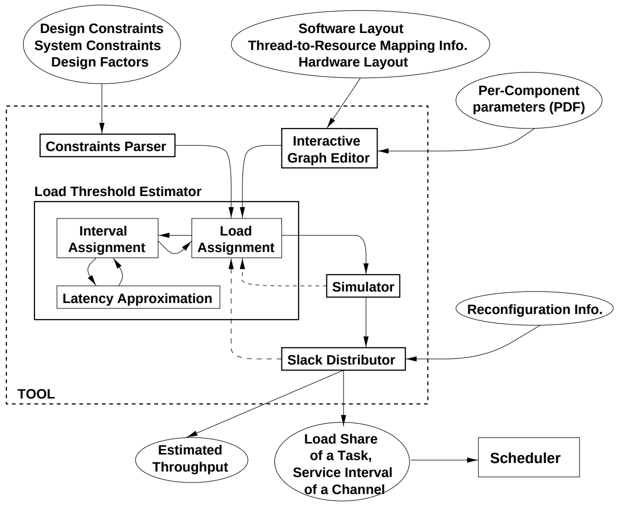
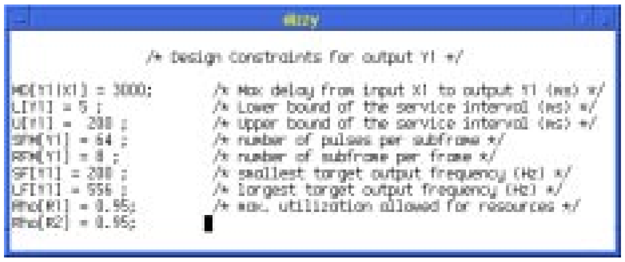


Figure 3: The structure of the Tool

the tool produces "Estimated Throughput", "Load Share" of each task, "Service Interval" of each channel. Since all inputs are parameterized, from the user's point of view, design changes are as easy as editing a text file or modifying/redrawing flow graphs using the graphical user interface.

**Design constraints** include real-time and performance constraints. Real-time constraints define an upper bound on the amount of time it should take a computation to flow through a channel which we term delay constraints of a channel[2]. For instance refer to `MD[Y1|X1]` in Figure 4. Performance constraints also include throughput or output rate.



```
              /* Design constraints for output Y1 */

MD[Y1|X1] = 3000;         /* Max delay from input X1 to output Y1 (ms) */
L[Y1] = 5 ;               /* Lower bound of the service interval (ms) */
U[Y1] =  200 ;            /* upper bound of the service interval (ms) */
SPM[Y1] = 64 ;            /* number of pulses per subframe */
RFM[Y1] = 8 ;             /* number of subframe per frame */
SF[Y1] = 200 ;            /* smallest target output frequency (Hz) */
LF[Y1] = 556 ;            /* largest target output frequency (Hz) */
Rho[R1] = 0.95;           /* max. utilization allowed for resources */
Rho[R2] = 0.95;
```

Figure 4: A constraints file for the input to the tool

**System constraints** include a set of resources, where a given resource corresponds to either a CPU or a network link. Associated with each resource $i$ is a maximum allowable capacity, $\rho_i^{Max}$, which corresponds to the maximum load that resource can multiplex effectively[3]. The parameter $\rho_i^{Max}$ is typically a function of the resource's scheduling policy (as in the case of a workstation) or its switching and arbitration policies (as in the case of a LAN). In the examples in this paper, we set $\rho_i^{Max}$ of all resources to 0.95 for ease of illustration.

[2]Recall that we allow specification of either (a) soft real-time constraints, in which case this corresponds to no dropping of data or (b) hard real-time constraints, in which case this corresponds to allowing dropping of data (see Section 1.

[3]In the constraints file depicted in Figure 4, $\rho_i^{Max}$ is replaced with Rho[i] due to the limitations of the text editor.

**Design factors** are parameters that a system designer can specify for performance or for ease of design and implementation considerations. For instance, inputs can be aggregated or partitioned for better performance. In the case of the SAR benchmark, data is divided into frames, which consist of 512 pulses. To exploit spatial and temporal parallelism, a SAR frame is usually divided into smaller subframes for processing. The number of sub-frames per frame affects the possible degree of spatial parallelism. Thus, a designer can specify input characteristics in terms of subframes.

### 3.1.1 Resource Mapping Information

The assignment of tasks to resources, and the flow of information between resources is expressed using flow graphs. Figure 5(a) shows the overall view of a layout drawn by our tool. Rectangles denote CPUs and network connections, and arrows denote data flow between CPUs and network connections. Figure 5(b) shows the view when we magnify the CPU denoted by "R2" in Figure 5(a). In this view, tasks are shown as circles, and small ellipses are used to represent connections to other resources. There are 4 tasks in "R2". For example, task `Teq1_1` has input from resource "N1", and sends its output to resource "N2".

### 3.1.2 Per-Component parameters

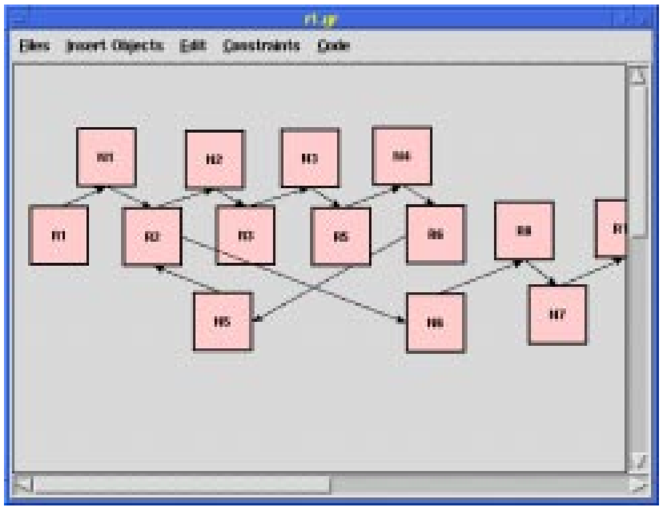
To synthesize the overall design, the tool requires information about the runtime behavior of each task. This information is supplied to the tool via a discrete PDF, which characterizes the time needed for one execution instance of a task on a resource. These PDFs can either be supplied by the designer based on the expected performance, or can be measured from actual executions of tasks. By using profile-based measurements, our tool captures the stochastic variations in task execution times due to hardware features such as caches and operating system features such as device interrupts.

Figure 6(a) shows the tool window through which a designer can bind a PDF to a task. The PDF can be synthesized from normal, exponential, or Erlang distributions with minimum and maximum range values, or for profile based data, a text file describing an arbitrary discrete PDF can be used. With the "NumStep" parameter shown in Figure 6(a), a designer can set the granularity of a discrete time PDF. For example, in Figure 6 the PDF of task $T_{km1(1)}$ is synthesized from the exponential distribution with a mean of 23 ms, which ranges from 20 ms to 35 ms. The PDF is discretized into 15 discrete intervals. Figure 6(b) shows the PDF of task $T_{km1(1)}$ synthesized from user-given parameters for the task in the Figure 6(a).

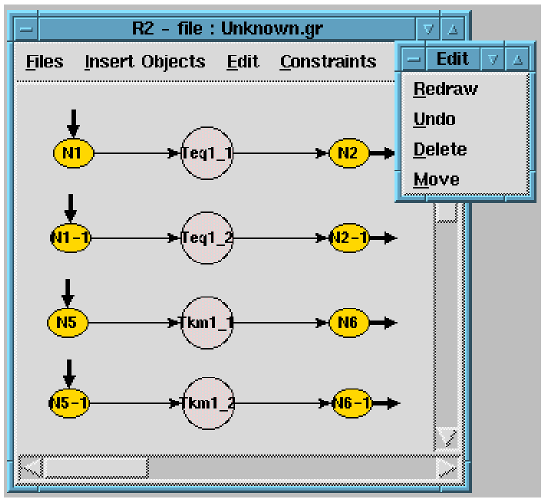
## 3.2 Run-Time System Model

In order to solve for the desired output parameters, the tool requires a few simple properties from the runtime system that will execute the target application.

First, disjoint subgraphs of the task graph, called *channels*, are treated independently while designing the whole system. We can make this assumption since there are no explicit data or control dependencies between channels. Each channel is composed of one or more task chains. A task chain is a feed-forward pipeline of tasks, where each task has only one predecessor, and one successor. In Figure 2(a), a single channel's flow graph with 4 chains is shown.



(a) Top level view of the example design



(b) Detailed view of 'R2'

Figure 5: A layout using Graphical User Interface of the tool

Second, we assume either unlimited buffer space or one slot of buffer space available between a pair of communicating tasks. The buffering policy is specified as an input to the tool. Unlimited buffer models imply that there should be no data loss in the middle of the channel, while one slot buffer space models imply that only the most recent input is used for processing.

Third, all tasks in a channel ($l$) are considered to be scheduled in a quasi-cyclic fashion, using time-division multiplexing for resource sharing, over $\mathtt{I}_l$-sized intervals. That is, load-shares of all tasks in the channel are guaranteed for $\mathtt{I}_l$-sized interval on all constituent resources. A task $\tau$'s runtime behavior can be described as follows:

(1) Within every $\mathtt{I}_l$-sized interval, a task $\tau$ can use up to $u$ of its resource's capacity. This is policed by assigning $\tau$ an execution-time budget $E = \lfloor u \times \mathtt{I}_l \rfloor$; that is, $E$ is an upper bound on the amount of resource time provided within each $\mathtt{I}_l$-sized interval, truncated to discrete units. (We assume that the system cannot keep track of arbitrarily fine granularities of time.) $\tau$ is actually given as $\frac{E}{\mathtt{I}_l}$ proportion of a resource, which we call *effective load* of $\tau$ at the service interval $\mathtt{I}_l$.

(2) A particular execution instance of $\tau$ may require multiple intervals to complete, with $E$ of its running time expended in each interval.

(3) A new instance of $\tau$ will be started within an interval if no previous instance of $\tau$ is still running, and there is a fresh input at the start of the interval.

A proportional resource scheduler guarantees a minimum load share of a task, and helps isolate one task's demand for the resource from another. Therefore, even when multiple tasks share a resource, their execution time distribution can be independently adjusted with added context switching overhead as the only shared effect.

### 3.3 Load Threshold Estimator

The heart of the automated synthesis tool presented in this paper is the *Load Threshold Estimator*. This component of the tool assigns scheduler parameters to each task to satisfy the end-to-end constraints supplied by the user. This process needs to solve three inter-related sub-problems:

1. **Target Throughput Settings.** Given a set of channels and load allocations, determine the minimum target throughput for the channel.

2. **Load Assignments.** Given a set of channels, and target throughput, assign a share of each CPU and network segment for each tasks so that all constraints are met.

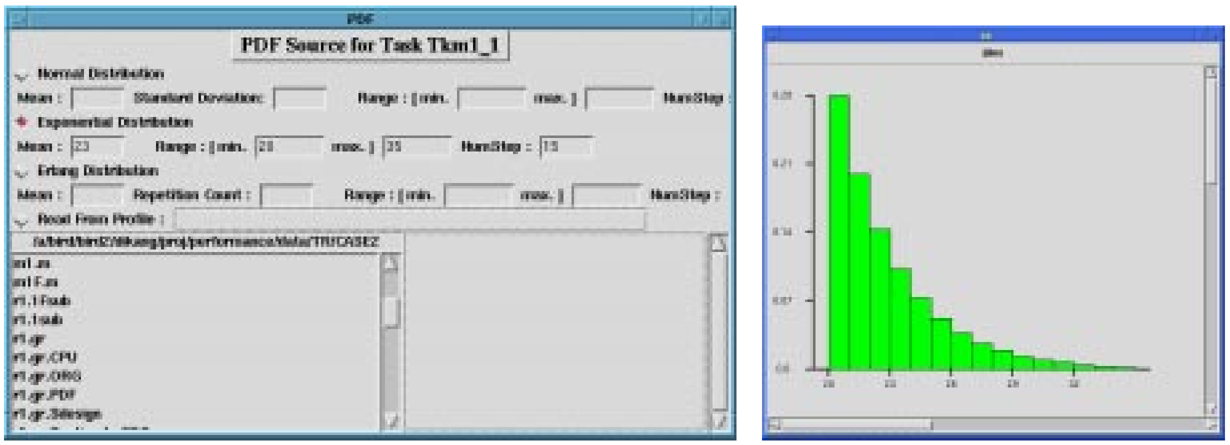
3. **Interval Assignments.** Given a load-assignment to the tasks in the channel, and the target throughput, compute an optimal interval for the channel, such that the effective latency is minimized.

Since each channel may be serviced at different interval sizes, interval-assignment is strictly an "intra-channel" issue. However, tasks in different channels but on the same resource compete for shared resources, and thus load share is an "inter-channel" problem. We now describe each of these sub-problems in more detail.

**Target Throughput Settings.** To find a load assignment for a target throughput range, our tool starts at the lowest allowable throughput. If a feasible solution is not found at the lowest target throughput, the design is deemed infeasible[4]. When a feasible load allocation is found, the target throughput is increased. At the new target throughput, the same load assignment procedure is repeated. This procedure repeats until either a feasible load allocation is found at the highest target throughput or when no feasible solution is found at a particular throughput level.

**Load Assignments.** Load-assignment works by iteratively refining the load vectors of channels, until a feasible solution is found. The entire algorithm terminates when the latency for all channels meet their performance requirements — or when it is discovered that no solution is possible.

[4]Since our search procedure does no backtrack, some heavily constrained problems may be declared infeasible when there might exist a feasible solution.



PDF

PDF Source for Task Tkm1_1

Normal Distribution

Mean : Standard Deviation: Range : [min. max. ] NumStep :

Exponential Distribution

Mean : 23 Range : [min. 20 max. ] 35 NumStep : 15

Erlang Distribution

Mean : Repetition Count : Range : [min. max. ] NumStep :

Read From Profile :

/a/bird/bird2/dikang/proj/performance/data/TR/CASE2

m1.m
m1F.m
r1.1Fsub
r1.1sub
r1.gr
r1.gr.CPU
r1.gr.ORG
r1.gr.PDF
r1.gr.Sdesign

(a) (b)

Figure 6: (a) PDF binding window (b) Synthesized discrete time PDF.

Load-assignment is task-based, i.e., it is driven by assigning additional load to the task estimated to need it the most using heuristics. After additional load is given to the selected task, the channel's new interval-size and latency are determined; if it meets its maximum latency requirements at the highest target throughput, it can be removed from further consideration.

**Interval Assignment.** Several non-linear constraints complicate the problem of interval-assignment: First, the *true, usable* load for a task $\tau$ with service interval $\mathrm{I}_l$ is given by $\lfloor u \times \mathrm{I}_l \rfloor / \mathrm{I}_l$, due to the fact that the system cannot multiplex load at arbitrarily fine granularities of time. Second, in our latency analysis, we assume that a task finishes only at the end of the service interval, which errs on the side of conservatism. Third, the utilization factor of task $\tau$, may vary with the service interval.

**End-to-End Latency Estimation.** An integral part of our "Load Threshold Estimator" is approximation of end-to-end latency at a given candidate load and service interval parameters. Since the search space of load threshold estimation is huge, fast analytical approximations are necessary. Here we outline the basic technique. Although the details of the analysis technique vary as system models change (e.g., infinite vs. single slot buffer models), the role of analytic solutions remains the same. A detailed description of our latency analysis estimation method for both infinite buffer and a single slot buffer can be found in [12, 14]. We go about constructing the analytical solution in a compositional (albeit approximate) manner, using the following techniques:

**Decomposition into Chains:** We first decompose a channel into its constituent chains, by simply traversing the flow-graph between all fork/join points. In analyzing each chain, we abstract it as being independent of all others. For instance, the graph in Figure 2(a) can be decomposed into 4 chains.

**Per-Chain Analysis:** For each chain, we generate an approximate latency distribution in a compositional manner, by processing each task locally, and using the results for its successors. A simple embedded Markov chain analysis is used to estimate the waiting time of data in the buffer. In this fashion, we generate an approximate latency distribution for each chain from the head task to the tail task of the chain.

**Synchronization Analysis:** At a synchronization point, partial results from separate chains are combined. For example, in our SAR example, a frame is composed from subframes. The latency of a whole frame is estimated from those of joining chains while setting the per-frame latency distribution to reflect that of the largest chain feeding into the synchronization point.

The end-to-end latency of a frame is estimated from the head chain(s) of a channel to the tail chain(s) of the channel. The latency distribution of an input frame to a chain is approximated from the latency distribution(s) of an output frame of its predecessor chain(s) using Synchronization Analysis. The latency distribution of a frame at the tail chain(s) is(are) the end-to-end latency distribution(s).

## 3.4 Validation through Simulation.

Since our latency analysis uses some key simplifying approximations, we validate the resulting solution via a simulation system that is an integral part of our tool. In particular, there may be many data-dependent correlations between the response-times, and they are ignored when computing the approximate solution. However, the simulation model keeps track of all data flowing throughout the channels, as well as the "true" states they induce on their participating tasks. Also, the simulation clock advances at every operation (rather than at the end of intervals); hence, if a task ends in the middle of an interval, it gets placed in the successor's input buffer at that time. Finally, the simulation model schedules resources using a modified deadline-monotonic dispatcher (where a deadline is considered to be the end of an interval), so more urgent tasks will get to run earlier than the analytical method assumes.

On the other hand, the simulator does inherit some other simplifications used in our analytical model. For example, inputs are assumed to be read at the start of an interval. As in the analysis, OS overhead is not considered, but is instead captured by the PDF's of each task.

## 3.5 Slack Distribution

Slack can be used either for fault-tolerance or to increase the performance of existing tasks. In this section, we focus on slack distribution for fault-tolerance.

Fault-tolerance is achieved by (1) distributing resource slack to the tasks which are relocated due to resource failures (2) adjusting load allocation of the tasks which are in the same channel as the relocated tasks. When a resource has slack larger than the sum of the load thresholds of the tasks relocated to it (or when the system would not be overloaded even after activating those relocated tasks), all tasks are given their load thresholds. If this occurs on all system resources, the system's performance is not affected by the failure, however the system-wide slack is reduced. Otherwise, slack distribution may not be sufficient to sustain the pre-failure level of performance. In this case, increasing the load of other tasks in the same channel may improve the performance of the channel. To contain the effect of a fault to affected channels only, we define the following rules for adjusting the load of other tasks:

**rule 1:** To isolate the fault's effect to the channels affected, loads associated with channels unaffected by the fault are not changed. This is necessary to prevent the effect of the fault from propagating to other channels.

**rule 2:** In a resource overloaded due to relocated tasks, the loads of those tasks which are in the channel(s) affected by the fault, are reduced. This spreads the effect of the fault evenly throughout the affected channels. In the implementation, we first increase the slack of the overloaded resource by taking away the load share allocated to the tasks that are not relocated but are in the channel(s) affected by the fault. Next, we distribute the slack proportionally to each task's load threshold such that the resource is not overloaded.

**rule 3:** When there is slack in the resources where tasks in an affected channel reside but they are not involved in the rule 2, the slack is distributed to the tasks to improve the overall throughput and latency of the affected channels. This process is the same as the "Constraint Satisfaction" process.

## 4 A Case Study : SAR

In this section, we present more complete details about our running example of the SAR benchmark. In particular, we present the specific constraints used, and report on the resulting design for both normal operation and operation under hardware failure. For the input of SAR benchmark design illustrated in Figure 2(a), we used synthesized PDFs for tasks, and defined 95% as the maximum resource share available. Figure 7 shows the synthesized PDFs used for the SAR benchmark example. The average execution time is computed assuming we have CPUs of 70 MFLOPS on the average, and network links of 120MB/sec on the average. For the floating point operation count of each task, we relied on the calculation presented in [2]. Figure 8 shows the resulting load allocations produced by our tool for each task in our example design. We found a set of feasible load allocations at 556 Hz input frequency. A service interval of 25 ms is suggested by the tool for these load allocations. It took 45 minutes to find this solution on 350 MHz Pentium II PC running Linux. To validate the design result, we ran the corresponding simulation with the load allocations and the service interval supplied by the analytic model. In Figure 8, $\mathtt{L}_i(A)$ denotes the latency estimated analytically, and $\mathtt{L}_i(S)$ denotes the latency measured in the simulation model.

A. Synthesized Solutions for Channels.

| $\mathrm{I}_i$ | $\mathrm{L}_i(A)$ | $\mathrm{L}_i(S)$ | $u_{\mathrm{iq}}$ | $u_{N1}$ | $u_{\mathrm{eq}}$ | $u_{N2}$ | $u_{\mathrm{rfft}}$ | $u_{N3}$ | $u_{\mathrm{ct}}$ |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 2.65 | 2.9047 | 0.372 | 0.090 | 0.159 | 0.090 | 0.591 | 0.090 | 0.163 |
| | | | $u_{N4}$ | $u_{\mathrm{afft}}$ | $u_{N5}$ | $u_{\mathrm{km}}$ | $u_{N6}$ | $u_{\mathrm{aidft}}$ | $u_{N7}$ |
| | | | 0.121 | 0.924 | 0.121 | 0.201 | 0.121 | 0.924 | 0.171 |

B. Resource Capacity Used by System.

| $\rho_{R1}$ | $\rho_{R2}$ | $\rho_{R3}$ | $\rho_{R4}$ | $\rho_{R5}$ | $\rho_{R6}$ | $\rho_{R7}$ | $\rho_{R8}$ | $\rho_{R9}$ |
|---|---|---|---|---|---|---|---|---|
| 0.743 | 0.720 | 0.591 | 0.591 | 0.326 | 0.924 | 0.924 | 0.924 | 0.924 |
| $\rho_{N1}$ | $\rho_{N2}$ | $\rho_{N3}$ | $\rho_{N4}$ | $\rho_{N5}$ | $\rho_{N6}$ | $\rho_{N7}$ | | |
| 0.181 | 0.181 | 0.181 | 0.243 | 0.243 | 0.243 | 0.342 | | |

Figure 8: Synthesized Solution of the design.

For the example design of Figure 2(a) and the corresponding load allocations of Figure 8, Figure 9 depicts the latencies estimated through analysis as well as measured by simulation at different service intervals. From the graphs in Figure 9, we note that analysis crosses between pessimistic estimation to optimistic estimation. The main source of the over estimation comes from the high utilization factors of the tasks in the chain. The utilization of a task may vary with its service interval size due to the error caused by discretization of time. We conjecture that this is why service-interval graphs possess some spikes. In the experiments we ran, however, more than 85% of the analytical estimations were within 10% of the simulated results.

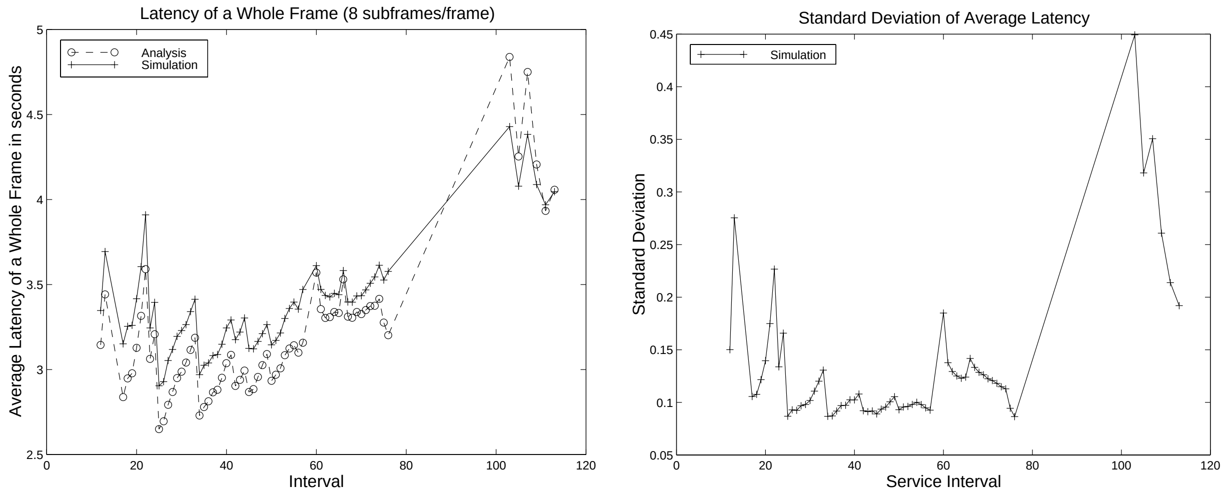
Lastly, we illustrate how our tool addresses component failure. Suppose that resource "R3" fails in the design of Figure 2(a), and furthermore that according to the "Reconfiguration Information" the tasks from "R3" are migrated to "R4". While reconfiguring the design due to this failure, it is determined that resource "R4" does not have sufficient slack (0.359) to accommodate the load threshold (0.591) of the relocated task $\mathrm{RDFT}_{1(1)}$. Hence, by load adjustment rule 2, we "steal" some load from $\mathrm{RDFT}_{1(2)}$, and thus reach our threshold limit of 0.95, i.e., the peak allowed capacity. At this point both tasks are given half of the load — however, under these conditions no service interval can be found that satisfies latency constraints at the 556 Hz input frequency. We then use rule 3 and increase the loads of other tasks to help compensate for delays created due to failure at "R4". The result is a design with a somewhat degraded performance, using a sampling frequency of 506 Hz — no longer the peak frequency, but still falling within the SAR guidelines. Figure 2(b) depicts the results of this reconfiguration, while Figure 10 depicts the corresponding estimated latency as a function of interval size.

## 5 Conclusions

We presented a semi-automated design synthesis tool that calibrates local resources and service intervals to achieve specified performance requirements as well as fault-tolerance. We illustrated our methodology on a design example of a SAR benchmark. In summary, our tool can be used to design embedded systems with: (1) end-to-end real-time delay constraints; (2) throughput constraints expressed either as input or output rates; and (3) task specifications expressed as flow graphs consisting of a set of pipelines connected with synchronization points. The target platform for designs generated by our tool is COTS-based systems running conven-

| CPU Tasks | Derived From | E[$t$] (ms) | Var[$t$] | [Min,Max] (ms) | NumSteps |
|---|---|---|---|---|---|
| I/Q | Normal | 59 | 100 | [50,83] | 33 |
| EQ | Normal | 11 | 16 | [9,24] | 15 |
| RDFT | Exponential | 103 | | [87 ,140] | 53 |
| CT | Exponential | 14 | | [12,30] | 18 |
| ADFT | Exponential | 187 | | [160,210] | 50 |
| KM | Exponential | 23 | | [20,35] | 15 |
| AIDFT | Exponential | 187 | | [160,210] | 50 |
| Network Tasks | Derived From | E[$t$] (ms) | Var[$t$] | [Min,Max] (ms) | NumSteps |
| Net[I/Q $\rightarrow$ EQ] | Normal | 6 | 25 | [4,20] | 16 |
| Net[EQ $\rightarrow$ A DFT] | Normal | 6 | 25 | [4,20] | 16 |
| Net[RDFT $\rightarrow$ CT] | Normal | 6 | 25 | [4,20] | 16 |
| Net[CT $\rightarrow$ ADFT] | Normal | 13 | 49 | [8,40] | 32 |
| Net[ADFT $\rightarrow$ KM] | Normal | 13 | 49 | [8,40] | 32 |
| Net[KM $\rightarrow$ AIDFT] | Normal | 13 | 49 | [8,40] | 32 |
| Net[AIDFT $\rightarrow$ Y1] | Normal | 13 | 49 | [8,40] | 32 |

Figure 7: Synthesized PDF of each task for processing a subframe



(a) (b)

Figure 9: Average latency (a) and its standard deviation (b) at different service intervals.

tional operating systems. Our hardware/software co-design approach parameterizes the behavior of software and hardware combinations, and makes the design procedure platform independent. We are currently implementing a scaled version of the SAR benchmark on a network of PCs running stock Linux.

## Acknowledgements

This research is supported in part by the following NSF Grants: CCR-9804679, CCR-98-96232, ACI-9711364, and ASC-9703212.

## References

[1] Jon C.R. Bennett and Hui Zhang. WF2Q : Worst-case Fair Weighted Fair Queueing. In *Proceedings of IEEE INFOCOM*. IEEE Computer Society Press, March 1996.

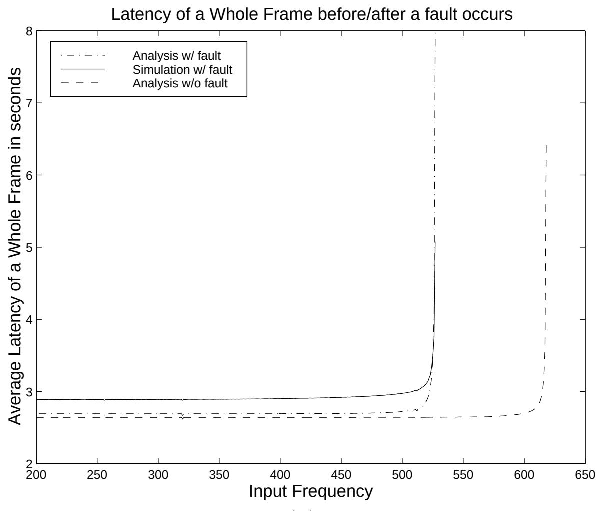
[2] C.P. Brown, R. A. Games, and J.J. Vaccaro. Real-Time Parallel Software Design Case Study: Implementation of the RASSP SAR Benchmark on the Intel Paragon. Technical Report MTR 95BTBD, The MITRE Corporation, Bedford, MA, 1995.

[3] Alan Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. In Sang Son, editor, *Principles of Real-Time Systems*. Prentice Hall, 1994.

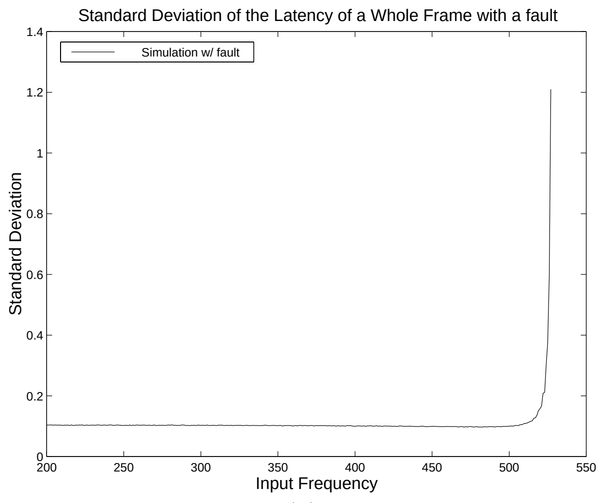
[4] Alan Demers. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12. ACM Press, September 1989.

[5] Peter Dinda, Thomas Gross, David O'Hallaron, Edward Segall, James Stichnoth, Jaspal Subhlok, Jon Webb, and Bwolen Yang. The CMU Task Parallel Program Suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March 1994.

[6] J. Liu et al. Perts: A prototype environment for real-time systems. Technical Report UIUCDCS-R-93-1802, Department of Computer Science, University of Illinois, 1993.



(a)



(b)

Figure 10: (a) Latency vs. Pulse Rate, base & reconfigured systems. (b) Standard deviation of the Latency vs. Pulse Rate, reconfigured system. In all cases $I_l = 25ms$.

[7] R. Gerber, S. Hong, and M. Saksena. Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes. *IEEE Transactions on Software Engineering*, 21, July 1995.

[8] R. Gerber, Dong-In Kang, Seongsoo Hong, and Manas Saksena. *End-to-End Design of Real-Time Systems*, chapter 10, pages 237–265. Wiley, 1996. In *Formal Methods for Real-Time Computing*, edited by Constance Heitmeyer and Dino Mandrioli.

[9] M. Harbour, M. Klein, and J. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 116–128, December 1991.

[10] Constance Heitmeyer and editors Dino Mandrioli. *Formal Methods for Real-Time Computing*. Wiley, 1996.

[11] Wagner Meira Jr. Understanding Parallel Program Performance Using Cause-Effect Analysis. Technical Report Ph.D. Thesis, University of Rochester, 1997.

[12] Dong-In Kang, Richard Gerber, and Leana Golubchik. Automated techniques for designing embedded signal processors on distributed platforms. Technical Report CS-TR-3944, UMIACS-TR-98-57, Department of Computer Science, University of Maryland, October 1998.

[13] Dong-In Kang, Richard Gerber, Leana Golubchik, and Jeffrey K. Hollingsworth. Techniques for automating distributed real-time applications design". In *IEEE International Symposium on High Performance Distributed Computing.* IEEE Computer Society Press, August 1999.

[14] Dong-In Kang, Richard Gerber, and Manas Saksena. Performance-based design of distributed real-time systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium.* IEEE Computer Society Press, June 1997.

[15] J. Leung and M. Merill. A Note on the Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[16] C. Liu and J. Layland. Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[17] A. K. Parekh and G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks - The Single Node Case. In *Proceedings of IEEE INFOCOM*, pages 915–924. IEEE Computer Society Press, March 1992.

[18] M. Saksena and S. Hong. Resource Conscious Design of Real-Time Systems: An End-to-End Approach. In *IEEE International Conference of Engineering Complex Computer Systems.* IEEE Computer Society Press, October 1996.

[19] F. Thoen, M. Cornero, G. Goossens, and H. De Man. Software synthesis for real-time information processing systems. In *Proceedings of Workshop on Languages Compilers and Tools for Real-Time Systems*, pages 60–69. ACM SIGPLAN, June 1995. Appears as ACM SIGPLAN Notices, 30(11).

[20] Hui Zhang and D. Ferrari. Rate-controlled static-priority queueing. In *Proceedings of IEEE INFOCOM*, pages 227–236. IEEE Computer Society Press, September 1993.

[21] Lixia Zhang. VirtualClock : A New Traffic control Algorithm for Packet Switching Networks. In *Proceedings of ACM SIGCOMM*, pages 19–29. ACM Press, September 1990.

[22] B. Zuerndorfer and G.A. Shaw. SAR Processing for RASSP Application. In *Proceedings of the First Annual RASSP Conference*, August 1994.