Techniques for Automating Distributed Real-Time Applications Design

Dong-In Kang, Richard Gerber, Leana Golubchik, Jeffrey K. Hollingsworth
Department of Computer Science & UMIACS
University of Maryland, College Park MD 20742
{dikang,rich,leana,hollings}@cs.umd.edu

Abstract

We present a performance-based methodology for designing a high-bandwidth radar application on commodity platforms. Unlike many real-time systems, our approach works for commodity processors running commodity operating systems. Our technique is innovative because it uses stochastic models of the processing time at each step in the process to allow for the variabilities of running on a non-realtime operating system. We show how our system synthesizes the runtime parameters for a synthetic aperture radar application under a variety of loading conditions.

1 Introduction

Designing a contemporary high-resolution radar processor is a complex art. These applications are extremely compute-intensive, due to their high pulse rates, the huge amount of data sampled at each pulse, and the complexity of the operations carried out on the data. Given these factors, high-end radars often demand CPU loads of 10^8 to 10^{13} FLOPS. This throughput can only be achieved by relying on parallel processing. Fortunately, most radar processors do contain a large amount of data parallelism. However, raw computing power is only part of the problem. Radars generally must satisfy latency (or deadline) requirements. In fact, meeting the end-to-end latency constraints is as important as throughput, since overly-late images may be worthless to the rest of the system. When meeting these latency requirements, we must also account for cache and pipeline effects, interrupts, queuing delays, context-switches, and the resolution of the real-time clock.

In addition, there is the problem of fault-tolerance. Traditionally, hardware-level redundancy has been the rule. If the primary system is not overly utilized, single-node failures could theoretically be repaired via repartitioning extra computing power, and multi-threading some of the functions from the failed node. In practice, this is not done due to a lack of tools to allow developers to characterize the interaction effects between the software, the hardware, and the runtime environment. Without good tools, designing a totally static software system is sufficiently difficult that software fault tolerance is not even considered.

In this paper, we focus on solving the scheduling synthesis problem which is an important subproblem of the overall process of parallelization and task placement. The scheduling synthesis problem is to find the proportion of load to allocate each task, and to derive an optimal service interval over which all load proportions should be guaranteed. To solve this problem, our system requires the following inputs: (a) the system topology, including the thread-to-CPU mapping; (b) the per-task load models; and (c) the required input rate and latency constraints. Internally, the design algorithms use analytic approximations to quickly estimate output rates and propagation delays for candidate solutions.

When all parameters are synthesized, the estimated end-to-end performance metrics are re-checked by simulation. The per-component load reservations can then be increased and the synthesis algorithms re-run to improve performance.

Our results rely on time-division multiplexing (TDM) to ensure that a task is guaranteed a fixed number of "time-slots" over pre-defined periodic intervals. This approach allows running different tasks on a single node. Since CPU workloads in real-time systems cannot simply be "reshaped," and since end-to-end latency guarantees still must be meet, we have found that TDM ensures a reasonable level of fairness between different tasks on a resource. Also TDM-based schedulers and drivers are fairly easy to implement.

A consequence of our multi-threading model is that it can also be used for fault-tolerance, since unused processing resources (slack) can be redistributed to off-load processes from failed nodes. In this approach, backup configurations are stored on every node. When a fault is detected, the system switches its mode to one of these backups. While this may seem like a fairly weak form of fault tolerance, recall that in many radars, an entirely unloaded system, worth perhaps millions of dollars, is used to handle the sort of faults we describe here. Hence, we believe our solution is quite a bit cheaper - though perhaps not achieving the same level of redundancy.

Throughout this paper, we use the RASSP Synthetic Aperture Radar benchmark as a running example for our design scheme (from here on referred to as the SAR), and we show how it operates on two different layouts of the radar system. We also show how it reconfigures under single-node failures, and compare the estimated performance to a simulation model. The SAR was posed as a "challenge" signal-processing problem for COTS-based development. In the realm of advanced radars, this SAR's throughput is quite modest - 1.1 GFLOPS for processing three polarizations, at the highest input pulse frequency. However, the point of the SAR exercise was not to build the most advanced radar. Rather, it was to find scalable, methods to perform pulse-compression and range-compression on commodity systems.

2 Related Work

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. Several papers document experiments with radar processors in these sorts of systems, and many of these are based on the SAR benchmark [14]. Researchers at the Mitre Corporation, used an Intel Paragon to implement one SAR channel entirely in software [1]. Their design phase was guided by coarse, deterministic load models based on simple Flop-counts. However, to our knowledge, no work has been done on using stochastic performance models for the purpose of system synthesis.

On the other hand, much has been published in the area of real-time systems synthesis for other domains. In our previous work [4] we relaxed the "classic" real-time systems constraint that period and deadline parameters are known ahead of time. Rather, we used the system's end-to-end delay and jitter requirements to automatically *derive* each task's constraints and ensure that the end-to-end requirements will be met. A similar approach was explored in [2], where imprecise computations were used as a metric to gauge the "goodness" of candidate solutions. These concepts were applied to other applications including discrete and continuous control problems [8, 11], scheduling real-time traffic over fieldbus networks [3], etc. A modification of our theory was even used to help solve some basic parameters of the SAR problem [6] - however, load requirements were not taken into account; rather the method was used to derive per-component frequencies and deadlines. Hong et. al [10] addressed distributed real-time synthesis in a deterministic context by extending our work to statically partition the end-to-end delays via heuristic metrics.

To accommodate bursty arrivals in network traffic, many service disciplines re-distribute work over longer intervals by occasionally postponing the projected completion times of some tasks, e.g., as in [9]. These models have also been used to derive statistical delay guarantees; e.g., [13]. Also, many of these disciplines have analogues in CPU scheduling, e.g., [12].

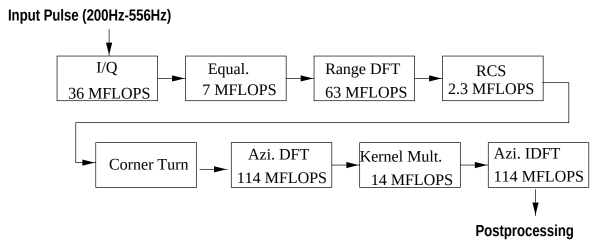


Figure 1. SAR Channel Flowgraph

3 The Synthetic Aperture Radar

Figure 1 shows the dataflow for one SAR channel, with some sample computation requirements for each component[1]. The key parameters that affect the resources required for the SAR are:

Number of channels. The SAR is composed of three parallel channels whose outputs are composed to build the full image. In this paper, we present our results for synthesizing a single channel. Our algorithms easily synthesized the full 3-channel system; however, one channel illustrates our approach and makes the figures and graphs significantly more readable.

Pulse rate. The SAR requirements stipulate a pulse rate between 200Hz and 556Hz, with higher rates preferred because they lead to better temporal resolution. We chose 556Hz as our target input rate.

Ranges per pulse, and their precision. In the SAR, 2048 ranges are sampled per pulse, and each is represented as a single-precision floating point number.

Number of pulses per image. A single channel's image is formed by concatenating two *frames*, each of which corresponds to 1/2 of the image's temporal resolution. In the SAR, one frame is formed out of 512 consecutive pulses; hence 1024 pulses are required to produce a full image. However, every frame is used in two images - first as the "new temporal part," and then shifted into the next image's "old part." Hence, images are produced at the frame processing rate, or $556/512 = 1.09$ Hz.

Required latency. The end-to-end latency is bounded by three seconds, where latency is measured from the arrival of the last pulse in a frame, to the time the frame is produced.

Functional Units. Major phases of the SAR:

- IQ stage: A pulse's samples are converted and filtered from video format to in-phase and quadrature 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.

[1]The FLOP counts were taken from a Mitre Corporation report [1].

- Corner Turn (CT): The corner turn 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, two 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 by ADFT); (2) a convolution (denoted by KM, for "Kernel Multiplication"); and an inverse Fourier transform (denoted by AIDFT, for Inverse Discrete Fourier Transform).

In the preceding section, we outlined some of the challenges involved in building this sort of a system. However, perhaps the largest challenge lies in sorting out the incredible number of design choices available. Complex radars possess almost infinite degrees of freedom in decomposing the problem along the spatial/temporal domains.

4 Model and Solution Overview

We model the system as a flowgraph, where tasks are mapped to some CPU or network resource. Formally, a system possesses the following structure and constraints:

Bounded-Capacity Resources: There is a set of resources $R_1, R_2, \ldots, R_m$, where a given resource R_i corresponds to one of the system's CPUs or network links. Associated with R_i is a maximum allowable capacity, or ρ_i^{Max}, which is the maximum load the resource can effectively handle. The parameter ρ_i^{Max} will typically be a function of its scheduling policy (for a workstation), or its switching and arbitration policies (for a LAN). In the examples in this paper, we set ρ_i^{Max} for all resources to 0.95.

Acyclic Flow-graph. A system is represented as an acyclic flow-graph, where vertices represent tasks (denoted by the letter τ), and edges represent a producer/consumer relationship between a pair of tasks. We assume unlimited buffer space available between each such pair, before the system is designed. As we will see, the dynamics of the application do serve to put an upper bound on the queue-depth, and thus in practise buffer space is finite.

Channels: When a flow-graph includes disjoint subgraphs, we say it has multiple *channels*. Since there are no explicit data or control dependencies between channels, we treat their latency analysis independently. They may be indirectly dependent, via resource sharing, and input sharing. For example, the three channels in the SAR may share resources or process different parts of a single image.

Task Chains: A task chain is a feed-forward pipeline of tasks, where each task has only one predecessor, and one successor. We denote chains with the Greek letters $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$, where the j^{th} task in a chain Γ_i is denoted $\tau_{i,j}$. Each computation on Γ_i carries out a transformation from an input (or a split) to an output (or a join point).

Stochastic Processing Costs: A task's cost is modeled via a discrete probability distribution function, whose random variable characterizes the time it needs for one execution instance on its resource.

Latency Bound ($\mathtt{ML_l}$*):* The delay constraint, $\mathtt{ML_l}$, is an upper bound on the *average time* it should take a computation to flow through a channel ($\mathtt{l}$) measured from the arrival time of the last pulse in a frame to when the entire frame is produced.

4.1 Run-Time Model

Within the system model, all tasks are considered to be scheduled in a quasi-cyclic fashion, using time-division multiplexing for resource sharing, over $\mathtt{I_l}$-sized intervals.

In particular, per-task load-shares are guaranteed for $\mathtt{I_l}$-sized intervals on all constituent resources. Hence, the synthesis algorithm's job is to (1) assign each task $\tau_{i,j}$ a proportion of its resource's capacity (which we denote as $u_{i,j}$) and (2) assign an $\mathtt{I_l}$-sized interval on a per channel-basis. Given this, $\tau_{i,j}$'s runtime behavior can be described as follows:

(1) Within every $\mathtt{I_l}$-sized interval, $\tau_{i,j}$ can use up to $u_{i,j}$ of its resource's capacity. This is policed by assigning $\tau_{i,j}$ an execution-time budget $E_{i,j} = \lfloor u_{i,j} \times \mathtt{I_l} \rfloor$; that is, $E_{i,j}$ 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.) In other words, $\tau_{i,j}$ is actually given as $E_{i,j}/\mathtt{I_l}$ proportion of a resource, which we call *effective load* of $\tau_{i,j}$ at the service interval $\mathtt{I_l}$.

(2) A particular execution instance of $\tau_{i,j}$ may require multiple intervals to complete, with $E_{i,j}$ of its running time expended in each interval.

(3) A new instance of $\tau_{i,j}$ will be started within an interval if no previous instance of $\tau_{i,j}$ is still running, and there is a fresh input.

Examples. Figures 2 & 3 show two layouts for a polarization of the SAR. Rectangles denote CPUs and arrows denote network connections. CPU names start with "`r`" and network link names start with "`n`". Each task represents a processing unit, and is connected to a channel.

The layouts also convey the degrees of parallelism used. For example, in Figure 3 IQ_1 has two copies - $\text{IQ}_{1(1)}$ and $\text{IQ}_{1(2)}$. In the flowgraph, resource-sharing is denoted by units mapped to the same resource names. For example, resource "r2" in Figure 3 has 4 tasks on it - $\text{EQ}_{1(1)}$, $\text{EQ}_{1(2)}$, $\text{KM}_{1(1)}$, $\text{KM}_{1(2)}$. Additionally, we note that the RDFT and RCS phases have been coalesced into single units, simply

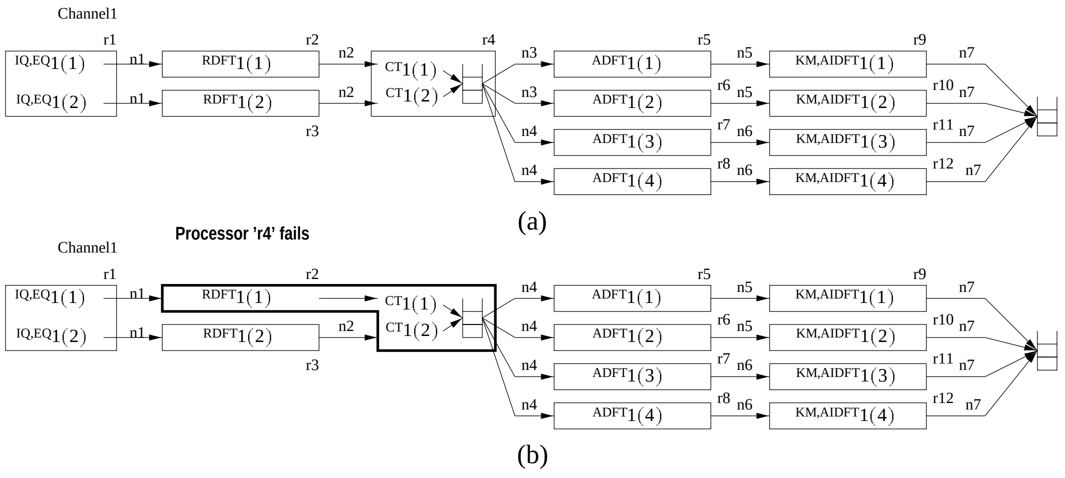


Figure 2. SAR channel Design I (a), and reconfiguration for “r4” failure (b).

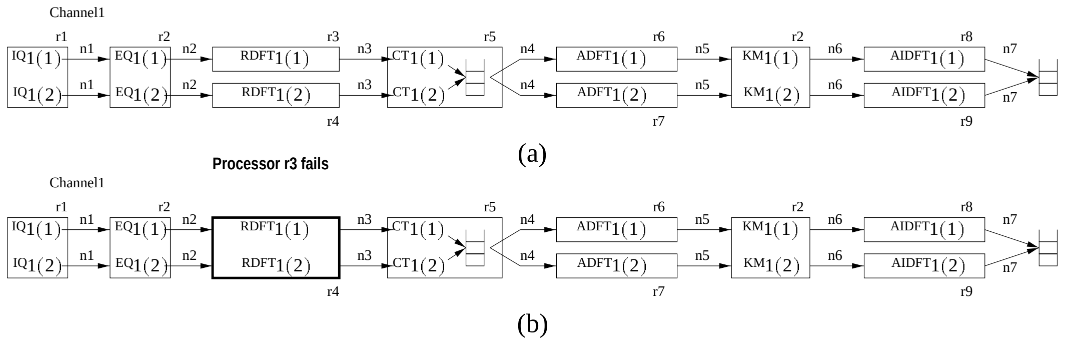


Figure 3. SAR channel Design II (a), and reconfiguration for “r3” failure (b).

denoted “RDFT.” The two layouts in Figures 2 & 3 differ in the degree of parallelism and the placement of tasks.

Data synchronization occurs at the input and output of pulse compression, which is shown pictorially as a stack in Figures 2 & 3. Note that in Figures 2 & 3 two chains join before passing data to pulse compression.

In any system, a task’s load demand varies stochastically, due to second-order effects like cache memory behavior, DMA interference, pipeline stalls, bus arbitration delays, transient head-of-line blocking, etc. By using one random variable to model a task’s load, we collapse all these residual effects into a single Probability Distribution Function (PDF). To obtain a load model, one could profile each task in isolation, and then post-process the resulting histogram into a service-time distribution. (We have experienced good results via this method in our work on digital video playout systems [5]). Another technique can be used at a more preliminary stage: the designer can coarsely estimate a task’s average load, and use it to create a synthetic distribution (e.g., normal). We take the latter approach in our running example: we discretize two different continuous distributions: normal and exponential for the SAR tasks.

We assumed CPUs capable of handling 70 MFLOPS on average, and network links of 120 MBytes/sec on average. We then synthesized execution times by using operation counts from the Mitre study as the mean of the PDFs. The stochastic variation in the PDFs accounts for response-times that deviate from the average. We believe this treatment of stochastic effects is a crucial element that previous efforts have overlooked.

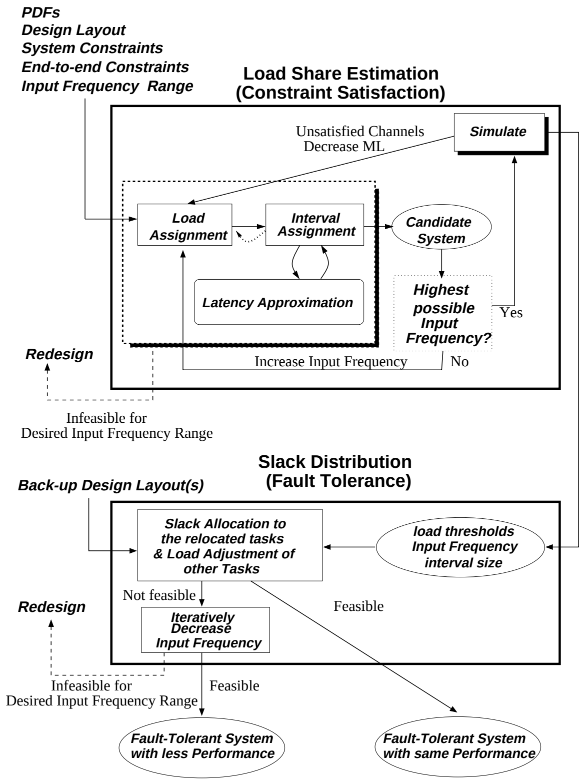
Fault Tolerance. As described above, we use multi-threading to permit on-the-fly reconfiguration. Here we present a static reconfiguration plan, with backup configurations pre-loaded. Our design method predicts the performance of the alternative reconfigurations to help determine which would be best in different scenarios. As we show later, a reconfigured system may or may not satisfy the end-to-end constraints, due to the lack of available resources at the original input frequency. If not, our design method estimates the input frequency range where the latency constraints can be satisfied.

Figure 2 shows a backup configuration of Design I, for the case where resource “r4” fails. Two corner turn tasks, $CT_{1(1)}$, and $CT_{1(2)}$ are moved to processor “r2,” which is shared with “$RDFT_{1(1)}$”. Likewise, Figure 3 shows a backup configuration of Design II, for failures of “r3.” The task “$RDFT_{1(1)}$” is migrated to processor “r4,” and it is then multi-threaded with $\text{RDFT}_{1(2)}$.

Solution Overview. A schematic of the design process is illustrated in Figure 4, where the algorithms carry out the following functions:

Load Share Estimation. The local schedulers are calibrated to satisfy the end-to-end constraints, via (1) partitioning the CPU and network capacity between the tasks; (2) selecting the service intervals to minimize latency; and (3) validating the solution via simulation, to verify the integrity of the approximations used, and to ensure that each channel's profile is sufficiently smooth (i.e., not bursty).

Slack Distribution. Slack is used to either enhance output quality, or to produce alternative layouts for fault-tolerance. It includes (1) calibrating loads for migrated tasks, and adjusting the load of the other tasks; and (2) estimating the performance of the reconfiguration.



PDFs
Design Layout
System Constraints
End-to-end Constraints
Input Frequency Range
Load Share Estimation (Constraint Satisfaction)
Unsatisfied Channels
Decrease ML
Simulate
Load Assignment
Interval Assignment
Candidate System
Latency Approximation
Highest possible Input Frequency?
Yes
No
Increase Input Frequency
Redesign
Infeasible for Desired Input Frequency Range
Slack Distribution (Fault Tolerance)
Back-up Design Layout(s)
Slack Allocation to the relocated tasks & Load Adjustment of other Tasks
load thresholds Input Frequency interval size
Not feasible
Feasible
Redesign
Iteratively Decrease Input Frequency
Infeasible for Desired Input Frequency Range
Feasible
Fault-Tolerant System with less Performance
Fault-Tolerant System with same Performance

Figure 4. Design Overview

The main partitioning algorithm processes each channel, and finds a candidate load-assignment vector for it. Given a load assignment for each channel, the synthesis algorithm attempts to find a service interval at which the channel achieves its nominal latency constraint. This computation is done approximately. For a given service interval, a latency distribution of the channel is derived. If the end-to-end latency exceeds the requirement, then the load assignment vector is increased. Finally, if sufficient load is found for all the system tasks, the candidate system is simulated to ensure that the approximations were sound – after which excess capacity can be allocated for the sake of fault-tolerance.

4.2 Latency Estimation

We now briefly describe how the system's latency is approximated, given candidate load and service-interval parameters. A detailed description of our latency estimation methodology is given in [7]. We go about constructing a model in a compositional (albeit approximate) manner, using the following techniques.

Decomposition into Chains: We decompose a channel into its constituent chains, by traversing the flow-graph between all fork/join points. In analyzing each chain, we abstract it as being independent of all others[2]. For instance, the graph in Figure 3 can be decomposed into 4 chains.

Per-Chain Analysis: For each chain Γ_i, we generate an approximate latency distribution in a compositional manner, by processing each task locally, and using the results for its successors. If multiple subframes in a frame go through a chain, the chain's latency is considered to be that of the last subframe exiting.

Synchronization: At a synchronization point, a frame is composed from the subframes of joining chains. 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.

5 System Design

We now revisit the "high-level" problem of determining the system's parameters, with the objective of satisfying each chain's performance requirements. (The pseudo-code for this synthesis process is given in [7].) As stated in the introduction, the design problem may be viewed as inter-related sub-problems:

Input Frequency Setting. If the system fails to find a feasible solution at the lowest input frequency (LF_l), the design is infeasible. When a feasible load allocation is found, the frequency is increased. At the new input frequency, the same load assignment procedure is repeated. This procedure ends either when a feasible load allocation is found at the highest input frequency (UF_l) or when no more improvement in the load allocation can be done. As for SAR, $\text{LF}_l = 200\text{Hz}$, and $\text{UF}_l = 556\text{Hz}$. The largest input frequency with a feasible load allocation is returned, which is the best achievable performance that our approach can find.

Load Assignment. Load-assignment works by iteratively refining the load vectors (the $\mathbf{u}_i$'s), until a feasible solution is found. The entire algorithm terminates when the latency for all channels meet their performance requirements or when

[2]Although this may not be true for the real system, our use of the TDM abstraction and stochastic models of the task duration make this approximation acceptable.

no solution is possible. We do not employ backtracking, and a task's load is never reduced. This means that the full solution space is not searched and in some tightly constrained systems, potential feasible solutions may not be found.

Interval Assignment. A feasible service interval is derived (if one exists), while ensuring the following requirements: (1) the *true, usable* load for a task $\tau_{i,j}$ in Γ_i in C_l is given by $\lfloor u_{i,j} \times \mathtt{I}_l \rfloor / \mathtt{I}_l$, due to the fact that the system cannot multiplex load at arbitrarily fine granularities of time; (2) tasks only finish at the end of the service interval; (3) the utilization factor of task $\tau_{i,j}$ may vary with the service interval.

Slack Distribution. Slack can be used either for fault-tolerance, or for increasing performance. If the latter is desired, one need only re-run the "Constraint Satisfaction" algorithm, with a higher target performance. Here we focus on slack distribution for fault-tolerance. Given a remapping of the tasks, our system uses the algorithm in Figure 5 to allocate slack.

```
Slack Distribution():
(1)    must_reduce_frequency = false
(2)    foreach resource $R_i$ with remapped tasks from failed node
(3)       $S^o$ = set of original tasks on $R_i$ in the affected channel(s)
(4)       $S^r$ = set of tasks remapped onto $R_i$ due to failure
(5)       $LS_j$ = load share of task $j$
(6)       if slack $R_i \geq \sum_{j \in S^r} LS_j$
(7)          Allocate slack to remapped tasks.
(8)       else
(9)          AvailableLS = $\sum_{j \in S^o} LS_j +$ slack $R_i$
(10)         TotalLS = $\sum_{j \in \{S^o \cup S^r\}} LS_j$
(11)         $LS'_j$ = AvailableLS $\times \frac{LS_j}{\text{TotalLS}}, \forall j \in \{S^o \cup S^r\}$
(12)         must_reduce_frequency = true
(13)   if must_reduce_frequency
(14)      Re-run "Constraints Satisfaction" with
             initial interval size = prior to failure interval size
             and initial load assignment
                of each task $j \in \{S^o \cup S^r\}$, on affected $R_i, = LS'_j$
                and of each task $k$, on unaffected $R_i, = LS_k$
```

Figure 5. Slack Distribution Algorithm.

Figure 6 shows the load allocations for tasks in Design II that were produced by the synthesis algorithm. We now consider how the reconfiguration technique would work for each example design.

A. Synthesized Load-shares for CPU Tasks.

Task	IQ	EQ	RDFT	CT	ADFT	KM	AIDFT
Load-share	0.372	0.159	0.591	0.163	0.924	0.201	0.924

B. CPU Utilization.

CPU	$r1$	$r2$	$r3$	$r4$	$r5$	$r6$	$r7$	$r8$	$r9$
Utilization	0.743	0.720	0.591	0.591	0.326	0.924	0.924	0.924	0.924

Figure 6. Solution of Design II.

Design I: Resource r2 could accommodate the load shares of the relocated tasks $\text{CT}_{1(1)}$, and $\text{CT}_{1(2)}$. Slack of resource r2 is allocated to the relocated tasks as shown on lines (6)-(7) in Figure 5.

Design II: Resource "r4" does not have sufficient slack (0.359) to accommodate the load share (0.591) of the relocated task $\text{RDFT}_{1(1)}$. Using the algorithm in Figure 5, $S^o = \{\tau_{RDFT1(1)}\}$ and $S^r = \{\tau_{RDFT1(2)}\}$. We first "steal" load shares of tasks in S^o, as is shown on line (9) in Figure 5, and make the available load share (AvailableLS) of "r4" 0.95 - the peak allowed capacity. Now each task on "r4" is given a load share proportional to its original load share prior to failure - half of the available load share, as is shown on lines (10)-(11). However, no service interval can be found that satisfies latency constraints at 556 Hz input frequency. Now we iteratively reduce the input frequency until the reconfigured system satisfies design requirements by re-running "Constraints Satisfaction" as is shown on lines (13)-(14). The result is a sampling frequency of 506 Hz - not the peak frequency, but still falling within the SAR guidelines. When the reconfigured system cannot satisfy design requirements at even the lowest input frequency, "Constraints Satisfaction" increases the loads of other tasks of the affected channel(s) to help compensate for delays at the bottleneck.

6 Simulation

Since the latency analysis uses some key simplifying approximations, we validate the resulting solution via simulation. The main sources of approximations are: (1) we assume periodic arrivals of inputs to all tasks; (2) we use an approximate joint probability calculation to determine latency at synchronization points; (3) statistical modeling of periodic input arrivals is highly inaccurate, due to quantization by the service interval; (4) our analysis assumes that a task's state-changes always occur at its interval boundaries; hence, even intermediate output times are assumed to take place at the interval's end. However, the simulation model discards these approximations by keeping track of all subframes and frames throughout the channel, as well as the "true" states they induce in their participating tasks. Also, the clock progresses along the real-time domain; hence, if a task ends in the middle of an interval, it gets placed in the successor's input buffer at that time.

On the other hand, the simulator does inherit some other simplifications used in our analysis model. For example, inputs are assumed to be read at the start of an interval. As in the analysis, context switch overheads are not considered; rather, they are implicitly modeled through PDFs of tasks service time.

Lastly, the analysis assumes infinite buffer space. Our simulator records the maximum buffer space occupancy for a candidate system to verify reasonable buffer space require-

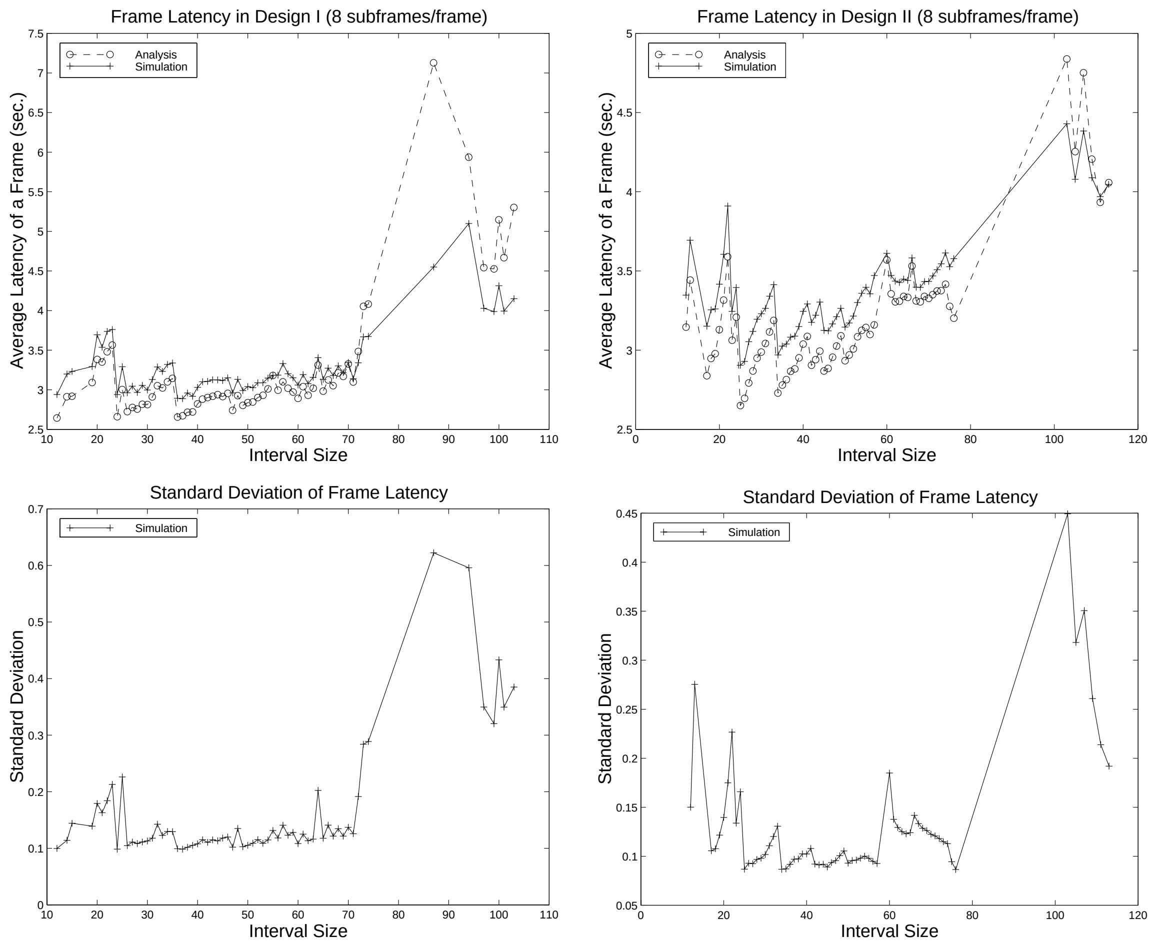


Figure 7. Latency for Designs I & II at different service interval sizes and input frequency of 556 Hz.

ments of a design. Given a typical 3 second end-to-end delay, there is little chance that the resulting design would have large buffer space requirements - and indeed, it did not in the simulations.

Each simulation trial runs for 20,000 frame inputs, which corresponds to approximately 20,000 seconds of radar processing. The obtained confidence intervals for these simulation are 99% $\pm$ 1%.

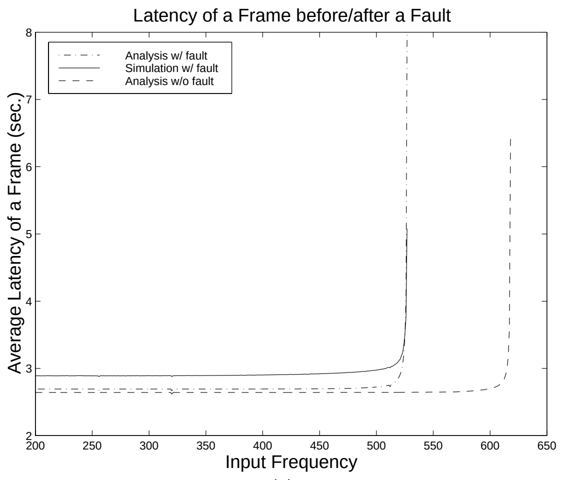
Figure 7 shows the latencies of Designs I & II estimated by analysis, and computed by simulation at different service interval sizes. Figure 8 shows the latencies of Design II at different input frequencies before and after reconfiguration at a fault. Design II is feasible at the highest input frequency, 556Hz, before a fault occurs. After the reconfiguration, it is feasible at 506Hz.

From Figure 7, we note that analysis crosses between conservative and optimistic estimation. We conjecture that the conservatism is largely due to the high resource utilization. Consider Design I and $\text{RDFT}_{1(1)}$. At service intervals of 12ms and 84ms, the utilizations of $\text{RDFT}_{1(1)}$ are 0.9026 and 0.9946, respectively. At such high utilizations, the system comes closer to its instability region, and the approximations for inter-output times start deviating from the true times. We conjecture that this is why service-interval graphs possess some spikes. In our experiments, however, more than 85% of the analytical estimations were within 10% of the simulated results.

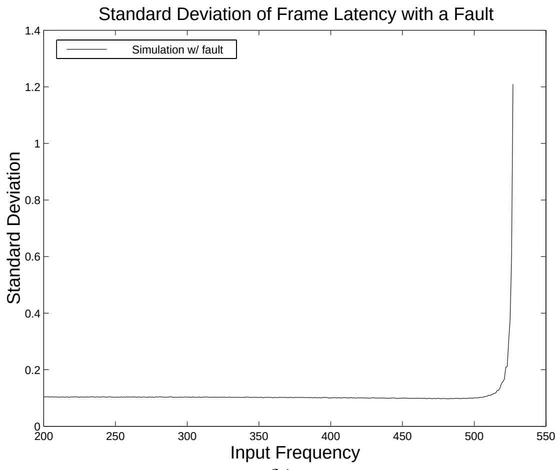
7 Conclusion

We presented a semi-automated design synthesis technique for calibrating resources in a signal processing application. We also showed how stochastic models can be harnessed to produce more efficient, flexible, and scalable systems than are currently deployed via deterministic models. We showed how a large SAR could exploit a simple software fault-tolerance scheme - while still giving designers a priori confidence in the performance and the quality of their system. Our synthesis approach uses a variety of simple analytic techniques to estimate latency, in tandem with heuristic search algorithms to find a feasible load partitioning.

Though approximations are used, the results are promising. Our two example layouts (and two fail-over configurations) consist of more than 25 tasks, and 15 shared resources.



(a)



(b)

Figure 8. (a) Design II Latency vs. Input Frequency, original & reconfigured systems. (b) Design II, Standard Deviation of Latency vs. Input Frequency, reconfigured system. In all cases $I_2 = 25ms$.

Nonetheless, our methods found results which achieved the SAR requirements, and which also could be validated via an independent simulation model.

However, simulation is not the end of the story. We are currently implementing a full-scale version of the SAR benchmark on a network of workstations; specifically we are using off-the-shelf PentiumII processors, connected via a gigabit-ethernet, and the Linux Operating System. In designing a signal processing application on this sort of a system, with all the stochastic effects they contain, we believe that a statistical technique like ours is not just one option - it could be the only option.

Acknowledgments
This research is supported in part by NSF Grants: CCR-9804679, CCR-9896232, ACI-9711364, and ASC-9703212.

References

[1] 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.

[2] W.-C. Feng and J. W.-S. Liu. Algorithms for scheduling real-time tasks with input error and end-to-end deadlines. *IEEE TSE*, 23(2):93–106, Feb. 1997.

[3] L. Franco. Communication configurator for fieldbus: An algorithm to schedule transmission of data and messages. In *Proceedings of IFAC/IFIP Workshop on Real Time Programming*, November 1996.

[4] 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.

[5] L. Gharai and R. Gerber. Multi-platform simulation of video playout performance. In *Proceedings of SPIE/IS&T Multimedia Computing and Networking*, Jan. 1998.

[6] S. Goddard and K. Jeffay. Analyzing the real-time properties of a dataflow execution paradigm us ing a synthetic aperture radar application. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, June 1997.

[7] Dong In Kang, R. Gerber, and L. Golubchik. Automated techniques for designing embedded signal processors on distributed platforms. Technical Report CS-TR-3944, Dept. of Computer Science, University of Maryland, Oct. 1998.

[8] N. Kim, M. Ryu, S. Hong, M. Saksena, C.-H. Choi, and H. Shin. Visual assessment of a real-time system design : A case study on a cnc controller. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 300–310. IEEE Computer Society Press, Dec. 1996.

[9] 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, March 1992.

[10] 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, Oct. 1996.

[11] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On Task Schedulability in Real-Time Control System. In *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1996.

[12] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Management. In *OSDI '94*, Nov. 1994.

[13] Z.-L. Zhang, D. Towsley, and J. Kurose. Statistical Analysis of Generalized Processor Sharing Scheduling Discipline. In *Proceedings of ACM SIGCOMM*, pages 68–77. ACM Press, Aug. 1994.

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