Automated Techniques for Designing Embedded Signal Processors on Distributed Platforms*

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

Abstract

In this paper, we present a performance-based technique to help synthesize high-bandwidth radar processors on commodity platforms. This problem is innately complex, for a number of reasons. Contemporary radars are very compute-intensive: they have high pulse rates, and they sample a large amount of range readings at each pulse. Indeed, modern radar processors can require CPU loads of in high-gigaflop to tera-flop ranges, performance which is only achieved by exploiting the radar's inherent data parallelism. Next-generation radars are slated to operate on scalable clusters of commodity systems.

Throughput is only one problem. Since radars are usually embedded within larger real-time applications, they also must adhere to latency (or deadline) constraints. Building an embedded radar processor on a network of workstations (or a NOW) involves partitioning load in a balanced fashion, accounting for stochastic effects injected on all software-based systems, synthesizing runtime parameters for the on-line schedulers and drivers, and meeting the latency and throughput constraints.

In this paper, we show how performance analysis can be used as an effective tool in the design loop; specifically, our method uses analytic approximation techniques to help synthesize efficient designs for radar processing systems. In our method, the signal-processor's topology is represented via a simple flow-graph abstraction, and the per-unit load requirements are modeled stochastically, to account for second-order effects like cache memory behavior, DMA interference, pipeline stalls, etc. Our design algorithm accepts the following inputs: (a) the system topology, including the thread-to-CPU mapping, where multi-threading is assumed to be used; (b) the per-task load models; and (c) the required pulse rate and latency constraints. As output, it produces the proportion of load to allocate to each task, set at manageable time resolutions for the local schedulers; an optimal service interval over which all load proportions should be

*This research is supported in part by National Science Foundation Grants CCR9804679 and CCR9619808, ARL Cooperative Agreement DAAL01-96-2-0002 and National Science Foundation CAREER grant CCR-96-25013.

guaranteed; an optimal sampling frequency; and some reconfiguration schemes to accommodate single-node failures. Internally, the design algorithms use analytic approximations to quickly estimate output rates and propagation delays for candidate solutions. When the system is synthesized, its results are checked via a simulation model, which removes many of the analytic approximations.

We show how our system synthesizes a real-time synthetic aperture radar, 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, and the huge amount of range data sampled at each pulse - not to mention the complexity of the operations carried out on the data, e.g., digital filtering, fast Fourier transforms (FFTs), inverse FFTs, matrix inversions, convolutions, and the like. Also, in some radars (e.g., with phased-array antennas or synthetic apertures), multiple signal-processing engines are run in parallel, and the outputs are composed to form single images. Given these factors, high-end radar processors often demand CPU loads of 10^8 to 10^{13} FLOPS, and next-generation phased-array radars are slated to require up to 10^{16} FLOPS.

This throughput can only be achieved by relying on some degree of parallel processing. Fortunately, most radar processors do contain a large amount of data parallelism within the functional units for both temporal and spatial computations. To some degree, this parallelism is currently exploited in custom-logic layouts; however, as throughput demands rise, designers are slowly replacing VLSI solutions with pipelines of commodity systems, hooked together by fast interconnects. Indeed with a sufficiently large number of workstations, any throughput is theoretically attainable.

But handling throughput is not the only problem. Most radars are usually embedded within larger real-time applications; hence they must adhere to latency (or deadline) requirements. In fact, for a real-time radar processor, meeting the end-to-end latency constraints are as important as processing the throughput, since overly-late images may be worthless to a recognition engine.

Hence, building an embedded radar processor on a network of workstations (or a NOW) involves striking a balance between several key factors: (1) configuring a NOW to handle the throughput, at a reasonable price; (2) partitioning the workload to fit the capacity of the system's nodes; and (3) doing this in a manner which meets the end-to-end latency requirements. In this respect, high-performance radar design is similar to many other problems in computer engineering - it only happens to be significantly larger.

The exercise is additionally complicated by the problem of synchronization. It is true that some radar computation stages can be parallelized; however it is not true that these stages are completely independent. To the contrary, all radars possess two large "macro-stages," which are usually performed in sequence: range compression (i.e., spatial processing) and pulse compression (i.e., temporal processing). Between these two stages lies an all-to-all data exchange - a fork/join synchronization which implements a real-time matrix transpose, where incoming rows arrive at staggered intervals, and where each outgoing column gets broadcast in its entirety. If the system is not tuned properly, then a slice of the image may be unduly delayed - thereby making the entire image exceed its allowed latency. Moreover, since a delayed slice of an old image can induce head-of-line blocking effects on newer images, these can end up being worthless too.

Another issue deals with runtime behavior, e.g, accounting for factors like cache and pipeline effects, interrupts, queuing delays, context-switches - not to mention the resolution of the real-time clock. Without reasonable ways of estimating (and controlling) these factors, the chances of obtaining a well-balanced design are fairly low.

Finally there is the problem of fault-tolerance. Traditionally, hardware-level redundancy has been the rule, and it has often been used at great expense. Specifically, the system is deployed with a "twin" - another supercomputer, which sits idle, waiting to repair single node/board failures. Often the probability of failure is quite low. If the baseline processor is not overly utilized, single-node failures could theoretically be repaired via repartitioning slack, and multi-threading some of the functions from the failed node. In practice, this is not done; in fact, multi-threading is rarely used on any real-time radar processor, in any context. The main reason for this is a lack of adequate performance metrics. That is, radar developers do not have sufficient models to help characterize the interaction effects between the software, the hardware, and the runtimes - and then chart their influence on the end-process statistics. Without this technology, the process of designing a single-threaded system is sufficiently difficult - and multi-threading makes things more complicated. Hence it is avoided.

The role of performance modeling. In this paper, we show how performance analysis can be used as an effective tool in the design loop; specifically, our method uses analytic approximation techniques to help synthesize efficient designs for radar processing systems. Using this method, we show how some of the abovementioned problems can be treated at design time, *before the system is built.* Moreover, we illustrate how we can use some simple statistics to account for many runtime effects, as well as nondeterministic execution times, multi-threading overhead, and the like.

In our method, the signal-processor's topology is represented via a simple flow-graph abstraction, where a vertex represents an activity requiring nonzero load from some CPU or network resource. As noted, these load requirements can vary stochastically, due to second-order effects like cache memory behavior, DMA interference, pipeline stalls, bus arbitration delays, transient head-of-line blocking, etc. Instead of explicitly modeling each of these effects individually, we aggregate them with a task's inherent load demand, and use random variables to represent each task's per-service load demand. These random variables range over arbitrary discrete probability distributions, and can be obtained via profiling tasks in isolation, or simply by using an engineer's hypothesis about the system's projected behavior. Successive instances of a single task are modeled to be independent of each other. Our results show that this kind of a simple model, while admittedly coarse, tends to be highly robust for the purposes of this application.

We also use some simple queueing-theoretic techniques as "subroutines" within our load-allocation heuristic. Using these abstractions, our algorithms can efficiently process millions of candidate designs in a few seconds, toward the goal of finding one which meets the end-to-end requirements.

Aside from the technical contributions in this paper, we also hope to provide a "social" contribution: We illustrate how a straightforward performance technique is quite useful when used as *a tool*, functioning within the context of a larger problem - that of design synthesis. We also show how stochastic models can be harnessed to produce more efficient, scalable systems than are currently deployed via deterministic models. Our aim is to help introduce this area to the community of performance analysts.

Overview of the Design Method. The design algorithm accepts the following inputs: (a) the system topology, including the thread-to-CPU mapping, where multi-threading is assumed to be used; (b) the per-task load models; and (c) the required pulse rate and latency constraints.

Using these constraints, the algorithm tries to synthesize ideal scheduling parameters for each task, via solving two sub-problems in tandem: (i) it finds the proportion of load to allocate each task, set at manageable time resolutions for the local schedulers; and (ii) it derives an optimal service interval over which all load proportions should be guaranteed. Internally, the design algorithms use analytic approximations to quickly estimate output rates and propagation delays for candidate solutions. In addition to the scheduling parameters, the algorithm also produces statistics on the number of projected late images, if any (i.e., images which exceed their latency constraints).

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

A consequence of the multi-threading model is that it can be used for the purposes of fault-tolerance, since unused slack can be redistributed to off-load processes from failed nodes. In practice, this means that backup configurations are pre-stored on every node. When a fault is detected, the system switches its mode to one of these backups. Spare slack is redistributed to the relocated tasks, and other tasks may have their load adjusted accordingly, perhaps retaining the original average performance, albeit with a higher output jitter. Or, if there is not sufficient slack to sustain the original performance, our method degrades to a lower output quality, and calibrates the system as such.

Narrowing the Focus. We do not treat the entire design problem here - i.e., parallelization, task partitioning, graphical CAD tools, etc. In particular, as mentioned above, the design process involves exploiting the degrees of parallelism in a problem, and assigning tasks to processors. While this is an important problem, we do not handle it here. We note that tuning the resource schedulers is, by definition, subservient to the allocation phase – which often involves accounting for devicespecific localities (e.g., I/O ports, DMA channels, etc.), as well as system-level issues (e.g., services provided by each node).

However, we narrow our focus to solving several rather important sub-problems, which have not been treated in the literature on radar processing. Yet while we concentrate on the scheduling synthesis problem at hand, we note that a "holistic" design tool could integrate the two problems, and use our system-tuning algorithms as "subroutines."

Running Example. Throughout this paper, we use the RASSP SAR benchmark as a running example for our design scheme, and we show how it operates on two different layouts of the radar system. We also show their reconfigurations under single-node failures, and compare the estimated performance to a simulation model. 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, using tuned software for the benchmark system [27]. However, the point of the SAR exercise was not to build most advanced radar. Rather, it was to find scalable, methods to perform pulse-compression and range-compression in software, on commodity systems. We note that in terms of general-purpose computers, 1.1 GFLOPS - with bounded latency constraints - is still considered quite high.

Related Work. Hundreds of books have been written on radar systems; however the field of deploying high-performance radars on clusters of general-purpose computers is a relatively new one. Its emergence is due to a rapid decline in price-performance ratios, and to the introduction of cheap, fast interconnects - but also, to a paradigm shift in the area of "supercomputing." Now, a network of workstations is the rule, not the exception.

Several papers document experiments with radar processors on this sort of systems, and many of these are based on the RASSP SAR benchmark. Specifically, the SAR's requirements were presented in [27]. In our opinion, the most successful implementation to date was performed by researchers at the Mitre Corporation, using an Intel Paragon [4]. The system design was not particularly sophisticated, but one SAR channel was implemented entirely in software. As for the design phase, it was guided via coarse, deterministic load models - indeed simple Flop-counts were used to determine where to place the functional units. We discuss some of the problems with this approach in the sequel.

Additional work has been done on streamlining the radar code itself, to run on various types of general-purpose computers. Some of the SAR's units were implemented and optimized by a group at Carnegie Mellon [8]; also, its memory allocation issues were studied in [15]. However, to our knowledge, no work has been done on using stochastic performance models for the purpose of system synthesis.[1]

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

On the other hand, much work has been done in the area of real-time systems synthesis, for other contexts. In all this work, the prevailing idea is to design a real-time system by solving scheduling problems from the "inside out." That is, rather than use a theory to *determine* whether a fully-designed system is schedulable, one uses the theory as a metric - to help instantiate parameters like frequencies, deadlines and the like. Specifically, in our previous work [1, 2] we relaxed the precondition that period and deadline parameters are always known ahead of time. Rather, we used the system's end-to-end delay and jitter requirements to automatically *derive* each task's constraints; these, in turn, ensure that the end-to-end requirements will be met on a uniprocessor system. A similar approach for uniprocessor systems was explored in [5], where execution time budgets were automatically derived from the end-to-end delay requirements; the method used an imprecise computation technique as a metric to help gauge the "goodness" of candidate solutions. These concepts were later modified for use in various application contexts, e.g., for discrete and continuous control problems [16, 19], for scheduling for real-time traffic over fieldbus networks [10, 11]. A modification of our theory was even used to help solve some basic parameters of this SAR problem [12] - however, load requirements were not taken into account; rather the method was used to derive per-component frequencies and deadlines. Other results presented in [18] addressed distributed real-time synthesis in a deterministic context: they extended our original theory by statically partitioning the end-to-end delays, via heuristic metrics [18].

The analytic results in this paper rely on a form of proportional-share scheduling, perhaps the oldest variant of PS, namely time-division multiplexing (TDM). We use a TDM abstraction to ensure that a task is guaranteed a fixed number of "time-slots" over pre-defined periodic intervals. Since CPU workloads in real-time systems cannot simply be "re-shaped," and since end-to-end latency guarantees still must be guaranteed, we have found that TDM ensures a reasonable level of fairness between different tasks on a resource – and between successive instances of the same task. Also TDM-based schedulers and drivers are fairly easy to implement, using credit/debit token-bucket schemes. Perhaps more importantly, the TDM abstraction is understood by radar domain experts: the underlying formalism is not radically different from techniques used in VLSI layouts; it is just a bit more nondeterministic.

Many other service disciplines re-distribute slack over longer intervals – at a cost of occasionally postponing the projected completion times of certain tasks. Most of these techniques were conceived for regulated workload models, e.g., linear bounded arrival processes [6]. Most of the queueing techniques developed help provide proportional-share service in high-speed networks, e.g., the VirtualClock [25], Fair-Share Queueing [7], GPS [17], and RCSP [23]. These models have also been used to derive statistical delay guarantees; in particular, within the framework of RCSP (in [24]) and GPS (in [26]). Related results can be found in [9] (for VitualClock) and in [22] (for FCFS, with a variety of traffic distributions). In [14], statistical service quality objectives are achieved via proportional-share queueing, in conjunction with server-guided backoff, where servers dynamically

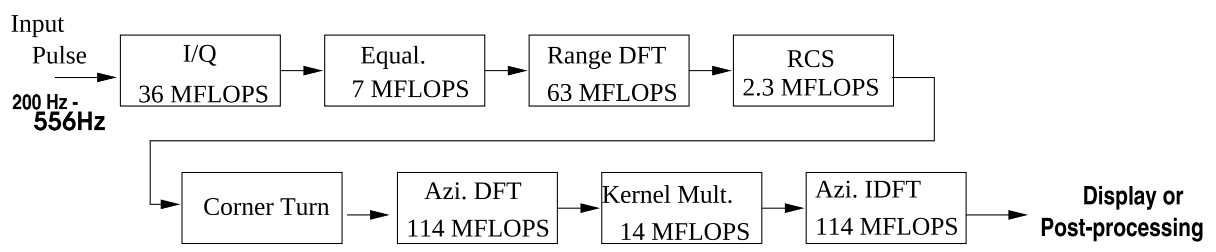


Figure 1: SAR Channel Flowgraph, and Functional Unit Complexity

adjust their rates to help utilize the available bandwidth. Also, many of these disciplines have analogues in CPU scheduling, e.g., Lottery Scheduling [20], Stride Scheduling [21], and hierarchical scheduling [13], among others.

2 The Synthetic Aperture Radar

Figure 1 shows the dataflow for one SAR channel, with some sample computation requirements for each component. These numbers were taken from a Mitre Corporation report [4], documenting their experiences in implementing a single-channel layout on a 16-node Paragon. While the Paragon is about 2 generations behind current supercomputers, the flop-counts of operations like FFTs remain constant on any machine. Perhaps the way these numbers get used is a more crucial issue. In a first pass, they can potentially help a designer compare the relative complexity of the various functional units in the pipeline. However, flop-counts do not convey much information about a function's true response time model. This is particularly true in software layouts, which contain a mixture of floating-point and integer operations. Moreover, in COTS environments, a unit's arithmetic complexity is only one factor contributing to its response time - as noted above, there are many others, which are best quantified in a statistical sense.

Requirements. The SAR's requirements depend on several factors, enumerated as follows:

Number of channels. The RASSP SAR is composed of three channels, all of which are run in parallel as independent radar processing systems. The outputs of these channels are then composed to build the full image; hence the term "synthetic aperture." In this paper, we present our results for synthesizing a single channel. While our algorithms easily synthesized the full 3-channel system (by replicating the performance results for each channel), our reduction here amply illustrates our approach, while making the figures and graphs significantly more readable.

Pulse rate. The SAR requirements stipulate a pulse rate lying between 200Hz and 556Hz, where higher rates lead to better temporal resolution - and hence are preferred. We chose the highest possible resolution as our design goal, i.e., 556Hz was selected as our input arrival rate.

Ranges per pulse, and their precision. In the RASSP 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 this 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 $556/512 = 1.09Hz$.

Required latency. The end-to-end latency is bounded by 3 seconds, where latency is measured from the arrival of the last pulse in a frame, to the time the frame is produced.

Functional Units. In producing images, all radars go through two major stages and some minor stages. The major stages are range compression (where the range samples are processed into frame slices), and pulse compression (where the slices are processed to form a single 2D frame). Minor phases include filter/transform operations, e.g., to normalize the antenna's digital output to some internal form, or to perform basic shifting and transposition functions.

In this SAR, channels are organized into four principal stages - input filtering, range compression, a "corner turn," and pulse compression. The phases are as follows:

- Video to Baseband I/Q Conversion (the 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. Basically, the RCS just normalizes the synthesized frequency coefficients, via applying amplitude weights to the radar's cross-section frequencies.

- Corner Turn (CT): The corner phase is a real-time matrix transpose. As such, it forms the crucial bottleneck in most adaptive radars. The problem is as follows. The RCS phase outputs 2048 range coefficients for each pulse - and 512 pulses are required to form a frame. However, to work on a range gate, the pulse compression engine requires all 512 readings for that range - which were sampled at different pulses. Hence, the corner-turn's job is to accumulate the 512x2048 matrix (with rows corresponding to pulses), and then feed the columns to the pulse-compression engine.

- Pulse Compression: The SAR uses an azimuth processor to handle the pulse data. In this radar, two sequential frames form a processing array of size 2048×1024, where columns correspond to pulses (after the corner turn), and rows correspond to range gates. When a new frame is shifted into the array, the oldest frame is shifted out. 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). Most readers are familiar with Fourier transforms, and the Azimuth processing does not affect the transform's basic arithmetic complexity. Rather, it involves the selection of the matrix coefficients used for this application.

In the preceding sections, we outlined some of the challenges involved in building this sort of a system. However, perhaps largest challenge lies in sorting out the incredible number of design choices available. This radar possesses almost infinite degrees of freedom in decomposing the problem along the spatial/temporal domains. Moreover, software layouts increase the degrees of freedom considerably. VLSI solutions are much more rigid; when an FFT chip for 256-sized vectors is deployed, it imposes a constraint which gets reflected all the way down the radar's datapath. In software, these constraints do not exist, which is both a blessing and a curse. After all, making any layout choice can radically affect the system's latency. Hence, there is a prevailing need for performance models which can help account for system-level effects at the initial design phase. Further, while we do not handle the placement problem in this paper, we do offer methods to model the runtime effects due to interacting units. These methods can potentially help other designers make their placement decisions.

The typical approach to signal-processor design still bears a heavy influence from the field's traditional "computing environment," that is, synchronous logic. For example, the Mitre project - the most successful implementation of the SAR to date - used each Paragon node for a single function, be it an FFT or a simple convolution. In parallelizing the system, a simple worst-case flop-count metric was used to help to slice the datapath. No multi-threading was used, and it is hardly ever used in these systems.

We note that radar designers rarely use queuing-theoretic techniques in carrying out performance estimation. Rather, if a priori estimates are used at all, simulation models are the rule, with actual measurements then carried out on the real system. While neither of these methods can be replaced in the design loop - and testing can never be replaced - we still advocate using analytic techniques at layout time. For obvious reasons, neither simulation nor measurement can be used within the context of automatic synthesis.

As for flop-counts and other deterministic metrics, they obscure the functional units with long-tailed response-time distributions, or those with high variability. They also cannot account for effects due to caching, DMA interference, traps, etc. The result is usually a very imbalanced, underutilized system.

3 Model and Solution Overview

We model a 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 multiplex effectively. The parameter ρ_i^{Max} will typically be a function of its scheduling policy (as in the case of a workstation), or its switching and arbitration policies (in the case of a LAN). In the examples in this paper, we set ρ_i^{Max} of all resources at 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.

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 (or polarizations) in the SAR may share resources, and do process different parts of an image. As for the resource-sharing, the TDM scheduling method allows isolating the analysis (and behavior) of multiplexed tasks. As for image collation, here we consider it external to the design problem, as is done in the requirements specification for this radar. If a designer inserts an explicit join point, correlation can easily be incorporated into our scheme.

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). When multiple chains feed into a synchronization point, we often use the following technique: we abstract a chain as an independent queueing network, and generate its latency accordingly. Then, we use simple probability theory to combine the latencies of all chains flowing into a join point - thereby approximating the latency for join point as a whole.

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 ($\texttt{ML}_1$): The delay constraint, $\texttt{ML}_1$, is an upper bound on the *average time* it should take a computation to flow through a channel. Unlike the original specification of the SAR benchmark, we measure the end-to-end latency from the arrival time of the last pulse in a frame, to the time when the entire frame is produced. Our latency bound is stricter than the original SAR benchmark latency bound, but we believe it is more realistic.

3.1 Run-Time Model

Within the system model, all tasks are considered to be scheduled in a quasi-cyclic fashion, using a time-division multiplexing abstraction for resource sharing, over $\mathtt{I}_1$-sized intervals.

In particular, per-task load-shares are guaranteed for $\mathtt{I}_1$-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 a global $\mathtt{I}_1$-sized interval for the application. Given this, $\tau_{i,j}$'s runtime behavior can be described as follows:

(1) Within every $\mathtt{I}_1$-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 $E_{i,j} = \lfloor u_{i,j} \times \mathtt{I}_1 \rfloor$; that is, $E_{i,j}$ is an upper bound on the amount of resource time provided within each $\mathtt{I}_1$-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 $\frac{E_{i,j}}{\mathtt{I}_1}$ proportion of resource, which we call *effective load* of $\tau_{i,j}$ at the service interval $\mathtt{I}_1$.

(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. Figure 2 and Figure 3 show two layouts for a polarization of the SAR. Rectangles denote CPUs and arrows denote network connections. CPUs have names starting with the letter "`r`," followed by a number; network links have names starting with "`n`," followed by a number. 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 - $\mathrm{IQ}_{1(1)}$ and $\mathrm{IQ}_{1(2)}$. 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 - $\mathrm{EQ}_{1(1)}$, $\mathrm{EQ}_{1(2)}$, $\mathrm{KM}_{1(1)}$, $\mathrm{KM}_{1(2)}$. Additionally, we note that the RDFT and RCS phases have been coalesced into single units, simply denoted "RDFT." The two layouts in Figure 3 and in Figure 2 differ in the degree of parallelism, in placement of tasks in resources, and in task partitioning.

For the sake of efficiency, range processing works on "subframes" - each of which contains a number of pulses. In other words, a subframe is a block of sequential input pulses in a frame, i.e., a block of consecutive range gates in the processing array. The examples in this paper assume a processing array partitioned into 8 subframes.

All subframes in a frame should be collected at the data synchronization points before being passed down the line. Data synchronization occurs at the input and output of pulse compression,

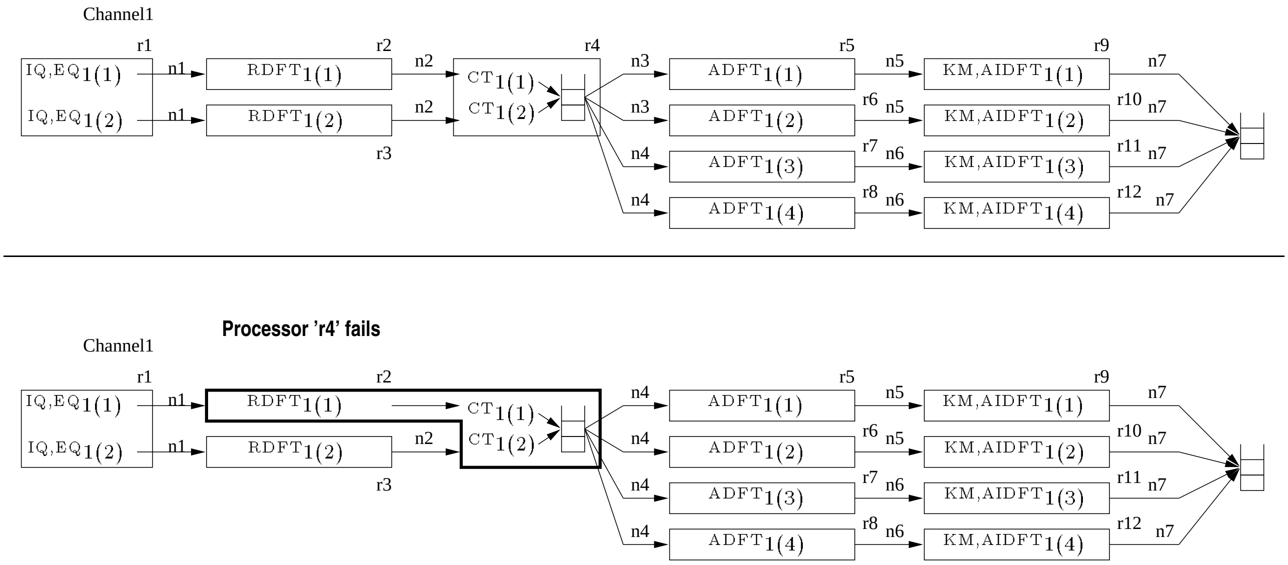


Figure 2: SAR channel Design I (top), and reconfiguration for "r4" failure (bottom).

which is shown pictorially as a stack in Figure 2. Note that in Figure 2, two chains join before passing to pulse compression.

We assume that a subframe is latched into the processor in a synchronous fashion. In some systems, the arrival process for pulses may indeed be staggered; however, in such situations a designer would add a delay pad into the model, to compensate for the asynchronous arrivals. In this paper we merely assume that the arrival time of subframes is synchronized; we discuss correcting the approximation in the sequel.

The per-task load models are shown in Figure 4. 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 essentially collapse all these residual effects into a single PDF, which also accounts for the task's idealized "best-case" execution-time. In our model, any discrete probability distribution can be used for this purpose.

To obtain a load model, one can often appeal to the method outlined above, a method commonly used in Hardware/Software Co-Design: a task is profiled in isolation, and the resulting histogram gets post-processed into a stationary response-time distribution. (We have experienced good results via this method in our multi-platform simulation work on digital video playout systems [3]). A second 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., exponential, normal, chi-square, etc. Such load models would correspond to hypotheses on how a task might behave in the

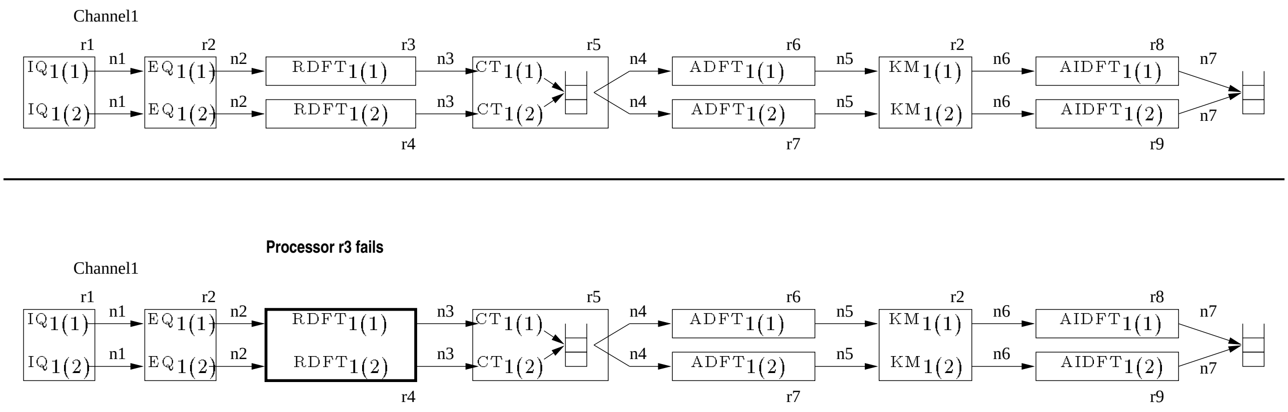


Figure 3: SAR channel Design II (top), and reconfiguration for "r3" failure (bottom).

integrated system. When this is done for all tasks – with the PDFs quantized at some acceptable level – the results can be fed into the synthesis tool. If the system topology can handle a number of hypothetical per-task load profiles, a designer can gain a margin of confidence in the system's robustness to subtle changes in loading conditions.

We take the latter approach in our running example: we discretize two different continuous distributions. In Figure 4, "Derived From" denotes a base continuous distribution generated and quantized using the parameters Min, Max, E[t] (mean), Var[t] (variance) and "NumSteps" (number of intervals). In the case of an exponential distribution, the CDF curve is shifted up to "Min," and the probabilities given to values past "Max" are distributed to each interval proportionally. The granularity of discretization is controlled by "NumSteps," where we assume that execution time associated with an interval is the maximum possible value within that interval.

The execution times were synthesized using the Mitre numbers as a baseline, and assuming we have CPUs capable of handling 70 MFLOPS on average, and network links of 120 MBytes/sec on average. However, the stochastic variation in the PDFs also accounts for response-times that deviate from the average, sometimes to a large extent. The more a designer carries out this sort of exercise, the more confident he/she can be that the synthesized design is robust. We believe this treatment of stochastic effects is a crucial element that previous efforts have overlooked.

Fault Tolerance. While multi-threading achieves a more flexible, efficient design, it also buys the ability to do on-the-fly reconfiguration. Many radar designers want to have this capability for the sake of fault tolerance - though no in-process radar currently uses it, due to the reasons itemized above. In this scheme, each resource has a copy of the software components that will be placed on it when it gets reconfigured. When a resource fails, an executive process activates the redundant software components - whose design was, after all, predetermined before the system was deployed.

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
I/Q,EQ	Normal	70	144	[56,107]	51
KM, AIDFT	Exponential	210		[180 , 245]	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 $\rightarrow$ Display]	Normal	13	49	[8,40]	32

Figure 4: Synthesized PDF of each task for processing a subframe

While this may seem like a fairly weak form of fault tolerance, two notes are in order here. The first is 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. The other observation is this: it is hard to imagine a mission-critical radar ever deploying truly adaptive fault-tolerance, where the system dynamically determines a reconfiguration. Hence, this type of pre-configured scheme may be the best one could expect to achieve.

Here we present a static reconfiguration plan, with backup configurations pre-loaded. Our design method predicts the performance of the alternative reconfigurations; this helps the designer determine which would be best in certain 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 are satisfied - and if down-sampling is an option, then perhaps it can be used.

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 the processor "r2," which is shared with "$RDFT_{1(1)}$". Likewise, Figure 3 shows a backup configuration of design II, for failures in "r3." The task "$RDFT_{1(1)}$" is migrated to the processor "r4," and it is then multi-threaded with $RDFT_{1(2)}$.

Solution Overview. The real-time implementation of the SAR in [4] was a nice experiment, in that a proof of concept was definitely established. Using some simple arithmetic calculations, the Mitre researchers placed the tasks, and measured the latency on the real system. However, the approach had many restrictions. First, resource sharing among independent tasks was not

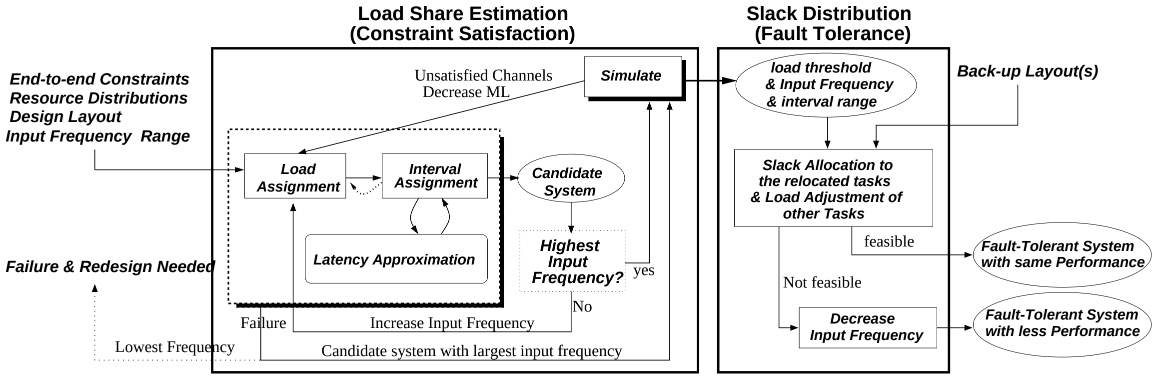


Figure 5: Design Overview

considered; hence layouts like Figures 2 and 3 would not be possible. Moreover, several nodes on the Paragon were left empty - due to no "leftover" functional units - whereas several were fully utilized. Second, the stochastic nature of a task execution time is not exploited. The approach was based on MFLOP requirements for tasks, given certain data paths. This approach may result in a highly underutilized system. However, without using more sophisticated performance models, it would be hard to do much better.

Our method attempts to handle some of these problems automatically.

A schematic of the design process is illustrated in Figure 5, where the algorithms carry out the following functions: (1) partitioning the CPU and network capacity between the tasks; (2) selecting each service interval to minimize latency; and (3) checking 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). The main boxes are outlined as follows:

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.

Slack Distribution. Here 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.

The main partitioning algorithm processes each chain Γ_i, and finds a candidate load-assignment vector for it, denoted $\mathbf{u_i}$. (An element $u_{i,j}$ in $\mathbf{u_i}$ contains the load allocated to $\tau_{i,j}$ on its resource.) Given a load assignment for Γ_i, the synthesis algorithm attempts to find a service interval $\mathtt{I_i}$ at which Γ_i's achieves its nominal latency constraint. This computation is done approximately: For a given $\mathtt{I_i}$, a latency estimate is derived by treating all of Γ_i's outputs uniformly, and deriving an i.i.d. latency distribution for it. If the latency exceeds the requirement, then the load assignment

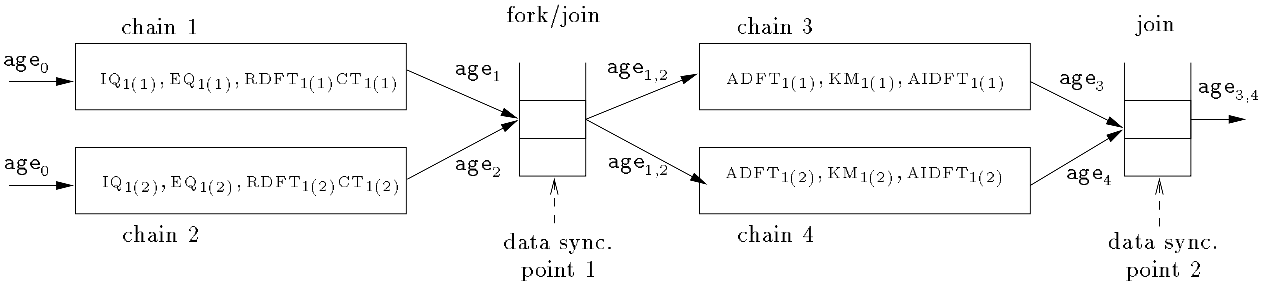


Figure 6: Overview of the channel analysis

vector is increased, and so on. Finally, if sufficient load is found for all the system tasks, the entire setup is simulated to ensure that the approximations were sound – after which excess capacity can be re-allocated for the sake of fault-tolerance.

4 Latency Estimation

In this section we describe how we approximate the system's latency, given candidate load and service-interval parameters. Then in Section 5, we show how we make use of this technique to derive these parameters. In determining latency, we make heavy use of the TDM abstraction. Since each chain is effectively isolated from others over I_1-sized intervals of observation, we can analyze the behavior of each chain independently, without worrying about head-of-line blocking effects from other components.

We use a discrete-time model, in which the time units are in terms of a chain's frame-size; i.e., our discrete domain $\{0, 1, 2, \ldots\}$ corresponds to the real times $\{0, I_1, 2I_1, \ldots\}$. Not only does this reduction make the analysis more tractable, it also corresponds to "worst-case" conditions: Since the underlying system may schedule a task execution at any time within a I_1-sized interval, we assume that input may be read as early as the beginning of an interval, and output may be produced as late as the end of an interval.

We go about constructing a model 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 - while in fact, this may not be the case in the real system. In the case of SAR, joins occur at the Corner Turn task, and at the output of the entire system. As an example, the graph in Figure 3 can be decomposed into 4 chains 'chain 1', 'chain 2', 'chain 3', and 'chain 4', which are shown in Figure 6.

Per-Chain Analysis: Each chain is analyzed in a top-down fashion, with results from producers used to gauge the metrics for consumers. In this fashion, we generate an approximate latency distribution for each chain. Figure 6 shows synthesized random variables $\mathtt{age}_1$ and $\mathtt{age}_2$, which correspond to the latencies of 'chain 1' and 'chain 2', respectively. Likewise, $\mathtt{age}_3$ and $\mathtt{age}_4$ denote the end-to-end latency from external input, to the end of 'chain 3' and 'chain 4', respectively. The arrival time of pulses in a subframe are synchronized - hence, the variable $\mathtt{age}_0$ denotes the latency required to buffer up these pulses. If multiple subframes in a frame go through a chain, the chain's latency is considered to be that of the last subframe exiting. As an example, in Figure 6 our processing granularity is divided into 8 subframes, with 2 parallel pipelines handling range-compression. Hence 4 subframes go through chain 1, and $\mathtt{age}_1$ characterizes the latency of the 4^{th} subframe to pass through chain 1.

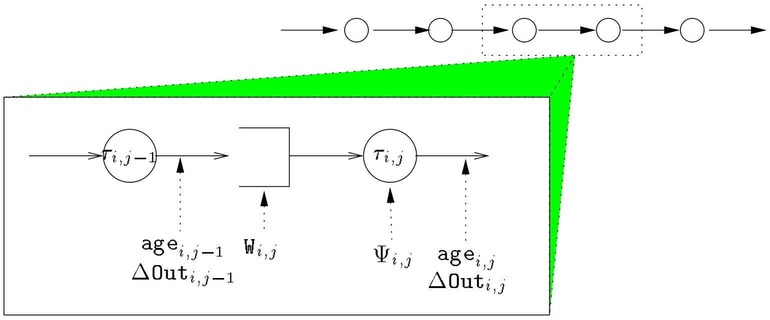
Synchronization: At a synchronization point, a frame is composed from the subframes from joining chains. The latency of a whole frame is estimated from those of joining chains. For example, in Figure 6, the frame latency $\mathtt{age}_{1,2}$ up to the 'data sync. point 1' is estimated from $\mathtt{age}_1$ and $\mathtt{age}_2$ using the following equations.

$$\begin{aligned} Pr[\mathtt{age}_{1,2} = t] &\cong Pr[\mathtt{age}_1 = t] \times Pr[\mathtt{age}_2 < t] + Pr[\mathtt{age}_2 < t] \times Pr[\mathtt{age}_2 = t] \\ &+ Pr[\mathtt{age}_1 = t] \times Pr[\mathtt{age}_2 = t] \qquad \textit{(Eq1)} \end{aligned}$$

This equation sets the per-frame latency distribution to reflect that of the largest chain feeding into the sync point. For three or more chains, we can get the frame latency distribution by applying the above equation repeatedly. Moreover, we use the accumulated latency at the corner-turn to be the base-line latency for successive pulse-compression chains, etc.

4.1 Intra-Chain Analysis

In this section, we describe how to approximate the latency of the output of a chain when the latency of the input to the chain is given. For each chain Γ_i, we go about constructing its latency model in a compositional (albeit inexact) manner, by processing each task locally, and using the results for its successors. Consider the following diagram, which portrays the flow of a computation at a single task:



These random variables are defined as follows:

1. *Data-age* ($\mathtt{age}_{i,j}$): This variable charts a computation's total accumulated time of a subframe in a frame, from entering Γ_i's head, to leaving $\tau_{i,j}$. When the age of the k-th subframe in a frame needs to be presented explicitly, the random variable $\mathtt{age}_{i,j}(k)$ is used.

2. *Waiting time* ($\mathtt{W}_{i,j}$): The duration of time an input is buffered, waiting for $\tau_{i,j}$ to process all the preceding subframes. Like $\mathtt{age}_{i,j}(k)$, random variable $\mathtt{W}_{i,j}(k)$ denotes the waiting time of the k-th subframe in a frame.

3. *Processing time* ($\Psi_{i,j}$): If $\mathtt{t_{i,j}}$ is a random variable ranging over $\tau_{i,j}$'s PDF, then $\Psi_{i,j} \stackrel{\text{def}}{=} \lceil \frac{\mathtt{t_{i,j}}}{E_{i,j}} \rceil$ is the corresponding variable in units of intervals.

4. *Inter-output time* ($\Delta\mathtt{Out}_{i,j}$): An approximation of $\tau_{i,j}$'s inter-output time, in terms of intervals; it measures the time between two successive outputs.

5. *Number of subframes per chain* (s_i): The number of subframes in a frame that chain Γ_i processes. If the number of K subframes are processed by a functional unit, and the unit is split into D parallel slices - of which Γ_i is one - then $s_i = K/D$.

Again, we assume subframes arrive synchronously, which simplifies our analysis considerably. Then, we adjust the *age* distribution to account for the initial buffering required. Finally, the validation of this approximation is done via simulation.

We compute the distribution of $\mathtt{age}_{i,j}$ via the following recurrence relation:

$$\mathtt{age}_{i,j} = \mathtt{age}_{i,j-1} + \mathtt{W}_{i,j} + \Psi_{i,j}$$

where $\mathtt{age}_{i,0}$ depends on its position in the pipeline. We assume that input arrivals are synchronized at head chains; hence, at the IQ phase the **age** is set to 0 or 1, depending on whether arrivals are read in before or after the first service period. As for the corner-turn phase, we set the incoming **age** distribution to be that of the synthesized compression chains.

Within a chain, we *approximate* the entire $\mathtt{age}_{i,j}$ distribution by assuming the three constituent variables to be independent, i.e.,

$$\Pr[\mathtt{age}_{i,j} = k] \cong \sum_{k_1+k_2+k_3=k} \Pr[\mathtt{age}_{i,j-1} = k_1] \times \Pr[\mathtt{W}_{i,j} = k_2] \times \Pr[\Psi_{i,j} = k_3]$$

As for a join-point, the latency is synthesized from all incoming chains - and indeed, is extracted by analyzing the behavior of the subframe arriving last in each chain. Assume Γ_i processes s_i subframes per frame, and the latency of the s_i-th subframe is characterized by $\mathtt{age}_i(s_i)$. Now assume Γ_{i+1} flows into the same sync point as Γ_i, and it also processes s_i subframes. Then $\mathtt{age}_F^m$, the joint incoming latency of the entire frame, is just extracted by forming their joint probability distribution.

$$\begin{aligned} Pr[\mathtt{age}_F^m = t] \quad \cong \quad & Pr[\mathtt{age}_i(s_i) = t] * Pr[\mathtt{age}_{i+1}(s_i) < t] + Pr[\mathtt{age}_i(s_i) < t] * Pr[\mathtt{age}_{i+1}(s_i) = t] \\ & + Pr[\mathtt{age}_i(s_i) = t] * Pr[\mathtt{age}_{i+1}(s_i) = t] \end{aligned}$$

Likewise, the end-to-end latency of a frame is derived from the latency distributions of the subframes at the end of the channel; when multiple chains join at the end of the channel, frame latency is approximated as above.

Now we need to derive the latency distributions for connected chains, from top-to-bottom.

Case 1: Head Chain

In head chains, an input subframe arrives periodically, and the latency distribution is set as explained above. From the perspective of successor chains, subframes are batched, and considered as single inputs.

Case 2: Non-Head Chain

We approximate the latency of the first subframe using the same technique as outlined above. That is, we first estimate the latency for the first subframe in a frame, $\mathtt{age}_{i,j}(1)$. Then $\mathtt{age}_{i,n}(k)$ is approximated by adding the *extra* waiting time that the k-th subframe should suffer due to the preceding subframes queued up. The *extra* waiting time $\mathtt{W}'_i(k)$ of k-th subframe is approximated as the sum of the $k-1$ independent execution times of the head task $\tau_{i,1}$.

$$Pr[\mathtt{W}'_i(k) = t] \quad \cong \quad \prod_{l=1}^{k-1} Pr[\Psi_{i,1} = t_l] \quad (\text{s.t.} \sum_{1 \le l < k} t_l = t)$$

The age of the k-th subframe is approximated as the sum of the age of the first subframe and the extra waiting time of the k-th subframe:

$$Pr[\mathtt{age}_{i,n}(k) = t] \cong Pr[\mathtt{age}_{i,n}(1) = t'] * Pr[\mathtt{W}'_i(k) = t - t']$$

Waiting Time. Obtaining reasonable waiting-time metrics at each stage of a chain is a non-trivial affair, due to arbitrary execution time PDFs, and scale of the problem. In carrying out the analysis, we use a stochastic process to characterize $\tau_{i,j}$'s total remaining work.

Let $X_{i,j}$ be an imbedded Markov Chain corresponding to input arrival events for $\tau_{i,j}$. This chain is fully connected, and in general it is infinite - as is shown in Figure 7. The state-transitions

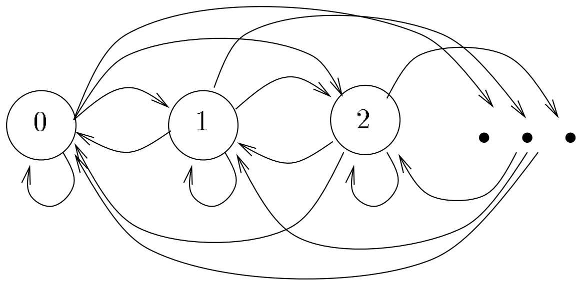


Figure 7: Infinite Markov Chain $X_{i,j}$

occur at the new input arrival instants, and the states represents the total remaining work for the task, excluding that of the new arrival.

By solving the following set of equations, where $\mathbf{P}$ denotes the one-step transition matrix corresponding to $X_{i,j}$, we obtain the steady state probabilities. Here π_t is the steady state probability of being in state t.

$$\vec{\pi} \times \mathbf{P} = \vec{\pi}, \qquad \sum_{t \geq 0} \pi_t = 1 \tag{Eq2}$$

Moreover, the Waiting Time Distribution is exactly that of the steady state probability distribution.

$$Pr[\mathtt{W}_{i,j} = t] = \pi_t$$

As for the Markov chain's transition probabilities, note that a transition from state k to l represents 2 facts: (1) a new input was just received; and (2) it will experience waiting time for l intervals. Depending on the destination state, there are two cases:

Case A : Destination state is 0.

$$\begin{aligned} \mathbf{P}[t,0] &= Pr[\Psi_{i,j} + t \leq \Delta\mathtt{Out}_{i,j-1}] \\ &= \sum_{t_1 > 0} Pr[\Psi_{i,j} = t_1] \times Pr[t + t_1 \leq \Delta\mathtt{Out}_{i,j-1}] \end{aligned}$$

Case B : Destination state $t_1 > 0$.

$$\begin{aligned} \mathbf{P}[t,t_1] &= Pr[t + \Psi_{i,j} - t_1 = \Delta\mathtt{Out}_{i,j-1}] \\ &= \sum_{t_2 > 0} Pr[\Psi_{i,j} = t_2] \times Pr[\Delta\mathtt{Out}_{i,j-1} = t + t_2 - t_1] \end{aligned}$$

Now we show how to derive task $\tau_{i,j}$'s inter-output distribution, $\Delta\mathtt{Out}_{i,j}$. First, note that for all tasks in a chain Γ_i, the **average** inter-arrival time of subframes, $\mathtt{SP}_i$, depends on (1) the average inter-arrival time of a frame to Γ_i ($\mathtt{FP}_i$), and (2) the number of subframes Γ_i should process, or s_i.

$$\mathtt{SP}_i = \frac{\mathtt{FP}_i}{s_i}$$

For example, consider Figure 2. Denote the set of range-compression chains as $\{\Gamma_1, \Gamma_2\}$, and denote the set of pulse-compression chains as $\{\Gamma_3, \Gamma_4, \Gamma_5, \Gamma_6\}$. Since our datapath is assumed to be partitioned into 8 subframes, then $s_1 = s_2 = 4$, and $s_3 - s_6$ are all set at 2. If pulses arrives at 556Hz, the period of the subframe arrival for the head chains ($\mathtt{SP}_1$-$\mathtt{SP}_2$) is

$$\frac{512}{556 \times 4} = 0.230216s$$

.

We calculate the inter-output times for each task as follows. First, for head-tasks, we quantizing the inter-arrival period with service interval of size $\mathtt{I}_1$, as follows:

$$\begin{aligned} Pr[\Delta\mathtt{Out}_{i,0} = \lfloor \frac{\mathtt{SP}_i}{\mathtt{I}_1} \rfloor] &\cong 1 - \frac{\mathtt{SP}_i \bmod \mathtt{I}_1}{\mathtt{I}_1} \\ Pr[\Delta\mathtt{Out}_{i,0} = \lceil \frac{\mathtt{SP}_i}{\mathtt{I}_1} \rceil] &\cong \frac{\mathtt{SP}_i \bmod \mathtt{I}_1}{\mathtt{I}_1} \end{aligned}$$

In Figure 2, when the service interval is 0.01 seconds, $Pr[\Delta\mathtt{Out}_{1,0} = 23] = 0.9784$ and $Pr[\Delta\mathtt{Out}_{i,0} = 24] = 0.0216$, for the head chain Γ_1.

This is an approximation; when the period of input arrival is not divisible by the service interval, the traffic denoted by $\Delta\mathtt{Out}_{i,0}$ fails to model the periodic arrival. This statistical approximation may cause overestimating the latency.

For the other tasks, inter-output times $\Delta\mathtt{Out}_{i,j}$ are calculated via using the execution time $\Psi_{i,j}$ and accounting for the bubble between two outputs from $\tau_{i,j}$. This bubble $\mathtt{B}_{i,j}$ is inserted when $\tau_{i,j}$ is idle.

$$\begin{aligned} Pr[\mathtt{B}_{i,j} = t] &\cong Pr[\mathtt{W}_{i,j} = t_1] \times Pr[\Psi_{i,j} = t_2] \times Pr[\Delta\mathtt{Out}_{i,j-1} = t_1 + t_2 + t] && \textit{(Eq3)} \\ Pr[\Delta\mathtt{Out}_{i,j} = t] &\cong Pr[\mathtt{B}_{i,j} = t_1] \times Pr[\Psi_{i,j} = t - t_1] && \textit{(Eq4)} \end{aligned}$$

Remarks: Several design thresholds were selected for the SAR. For example, in solving (Eq 2), we restricted the number of states to 500; and we aggregated the (small) probabilities for higher-order transitions into those for state 499.

$$\begin{aligned} \mathbf{P}[t, 499] &\cong Pr[t + \Psi_{i,j} - \Delta\mathtt{Out}_{i,j-1} \geq 499] \\ &= \sum_{k>0} Pr[\Psi_{i,j} = k] \times Pr[\Delta\mathtt{Out}_{i,j-1} \leq t + k - 499] \end{aligned}$$

Also, we found that waiting times could be over-estimated, due to the approximation of the

$\Delta\texttt{Out}_{i,j}$. While inputs are bursty, they do have a regularity - which $\Delta\texttt{Out}_{i,j}$ doesn't account for. This can be shown with one task chain as an example, where the inputs arrive periodically, every two quanta. The task's execution time can be either 1 or 2 intervals, with a 50-50 likelihood of each. By solving the equations shown above, we have the following:

$Pr[\texttt{W} = 0]$	$Pr[\texttt{B} = 0]$	$Pr[\texttt{B} = 1]$	$Pr[\Delta\texttt{Out} = 1]$	$Pr[\Delta\texttt{Out} = 2]$	$Pr[\Delta\texttt{Out} = 3]$
1.0	0.5	0.5	0.25	0.5	0.25

In this approximation, n outputs can be produced in n consecutive intervals, with probability 0.25^n. However, we know this is impossible - this sort of batched burstiness cannot occur, due to the regular nature of the arrival process. This conservatism leads to overestimating the waiting time for a task; for longer chains, the approximation gets worse.

To avoid this sort of amplification, we also tried another technique - simply assuming a re-regulated traffic flow, all the way through the pipeline. That is, in this alternative, the following is assumed to hold:

$$\begin{aligned} Pr[\Delta\texttt{Out}_{i,0} = \lfloor\frac{\texttt{SP}_i}{\texttt{I}_1}\rfloor] &\cong 1 - \frac{\texttt{SP}_i \bmod \texttt{I}_1}{\texttt{I}_1} \\ Pr[\Delta\texttt{Out}_{i,0} = \lceil\frac{\texttt{SP}_i}{\texttt{I}_1}\rceil] &\cong \frac{\texttt{SP}_i \bmod \texttt{I}_1}{\texttt{I}_1} \\ \forall_{\tau_{i,j}\ in\ \Gamma_i}\ \forall_k\ Pr[\Delta\texttt{Out}_{i,j} = k] &= Pr[\Delta\texttt{Out}_{i,0} = k] \end{aligned}$$

This alternative fails to consider bursty outputs, and it should be used with care. It could, in essence, lead to underestimating the waiting time of each task, and result in optimistically approximating end-to-end latency. However, we argue that as the end-to-end delay constraints get tighter, the simplification makes more sense. As the constraints get tighter, a task is given more utilization of a resource, and the inter-output traffic gets less bursty.

In the sequel, this alternative is used to generate our results. Then the numbers are checked against a simulation model, as shown in Section 6.

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. As stated in the introduction, the design problem may be viewed as inter-related sub-problems:

1. **Load Assignment.** How should the CPU and network load be partitioned among the set of tasks, so that the latency requirements are met?

2. **Interval Assignment.** Given a load-assignment to the tasks, what is the optimal service interval to maximize effective throughput?

Synthesize(): returns $\{(\mathbf{U}_l, \mathtt{I}_\mathtt{l}, \mathtt{L}_\mathtt{l}, \mathrm{F}_\mathtt{l}) : \mathtt{1} \le \mathtt{l} \le \mathtt{n}\}$

foreach Channel C_l, ($1 \le l \le n$) {
    $\mathtt{I}_\mathtt{l} \leftarrow \mathtt{min(SP_i)}$ (for all Γ_i in C_l)
    $\mathrm{F}_l \leftarrow \mathrm{LF}_l$
    foreach Chain Γ_i in the Channel C_l {
        $u_{i,j} \leftarrow \frac{E[t_{i,j}]}{\mathrm{SP}_i}$ (for all $\tau_{i,j}$ in Γ_i in C_l)
    }
}
$\rho_k \leftarrow \sum_{resource(\tau_{i,j})=k} u_{i,j}$ (for all $1 \le k \le m$)
$\mathrm{S} \leftarrow \{C_l : 1 \le l \le n\}$
while ($\mathrm{S} \neq \emptyset$) {
    Find the $\tau_{i,j}$ in Γ_i in $C_i \in S$ s.t.
      $r_k = resource(\tau_{i,j})$ which maximizes
      $w_{i,j} \leftarrow H(\mathtt{ML_l}, \mathtt{L_l}, \mathtt{E[W_{i,j}]})$
    if($\rho_k \ge \rho_k^m - \Delta$)
      return Failure
    $u_{i,j} \leftarrow u_{i,j} + \Delta$
    $\rho_k \leftarrow \rho_k + \Delta$
    $(\mathtt{L}_l, \mathtt{I_l}) \leftarrow$ Get_Interval(C_l, $\mathtt{I_l}$, $\mathbf{U}_l$, F_l)
    while ($\mathtt{L}_l \le \mathtt{ML_l}$ **and** $\mathrm{F}_l \le \mathrm{UF}_l$)
        $\mathtt{L}'_l \leftarrow \mathtt{L}_l$; $\mathbf{U}'_l \leftarrow \mathbf{U}_l$; $\mathtt{I}'_\mathtt{l} \leftarrow \mathtt{I_l}$; $\mathrm{F}'_l \leftarrow \mathrm{F}_l$
        $\mathrm{F}_l \leftarrow \mathrm{F}_l + 1$
        $(\mathtt{L}_l, \mathtt{I_l}) \leftarrow$ Get_Interval(C_l, $\mathtt{I_l}$, $\mathbf{U}_l$, F_l)
    }
    if ($\mathrm{F}_l > \mathrm{UF}_l$)
        $\mathrm{S} \leftarrow \mathrm{S} - \{C_l\}$
}
return($\{(\mathbf{U}'_l, \mathtt{I}'_\mathtt{l}, \mathtt{L}'_\mathtt{l}, \mathrm{F}'_\mathtt{l}) : \mathtt{1} \le \mathtt{l} \le \mathtt{n}\}$)

Get_Interval(C_l, $\mathtt{I_l}$, $\mathbf{U_l}$) : returns ($\mathtt{L}_l$, $\mathtt{I_l}$)

$\mathtt{L}_l \leftarrow get\_latency(C_l, \mathtt{I_l}, \mathbf{U_l})$
for ($t \leftarrow \mathtt{I_l} - 1$; $t > 0$; $t \leftarrow t - 1$) {
    if (($\mathtt{L}_l$ **mod** t) $< (\alpha \times \mathtt{L}_l)$) {
        $\mathtt{L'}_l \leftarrow$ get_latency(C_l, t, $\mathbf{U}_l$)
        if ($\mathtt{L'}_l < \mathtt{L}_l$) {
            $\mathtt{L}_l \leftarrow \mathtt{L'}_l$
            $\mathtt{I_l} \leftarrow \mathtt{t}$
        }
    }
}
return ($\mathtt{L}_l$, $\mathtt{I_l}$)

Figure 8: Synthesis Algorithm

3. **Input Frequency Setting.** If the highest pulse-rate cannot be met, what is the ideal pulse rate for this problem.

Note that load-allocation is the main "inter-channel" problem here, and interval-assignment is an intra-channel problem. With our time-division abstraction, altering a tasks's service interval will not effect the (average) rates of other chains in the system.

The design procedure shows the basic idea of how to solve the problems.

In general, input frequency is the only parameter that is to be synchronized among those 3 channels while designing. The other parameters, such as service interval of each chain, may be set differently. Here we show the design of only 1 channel of the SAR.

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 at it, the frequency is increased, as is shown in line (18) of algorithm *Synthesize()* in Figure 8. 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, LF_l = 200Hz, and UF_l = 556Hz. The largest input frequency with a feasible load allocation is returned, which is the best achievable performance that the tool 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[2] algorithm terminates when the latency for all channels meet their performance requirements – or when it discovers that no solution is possible. We do not employ backtracking, and a task's load is never reduced. This means the solution space is not searched *totally*, and in some tightly constrained systems, potential feasible solutions may not be found.

Load-assignment is task-based, i.e., it is driven by assigning additional load to the task estimated to need it the most. The heart of the algorithm can be found on lines (9)-(10), where all of the remaining unsolved channels are considered, with the objective of assigning additional load to the "most deserving" task in one of those channels. This selection is made using a heuristic weight $w_{i,j}$, reflecting the potential benefit of increasing $\tau_{i,j}$'s utilization, in the quest of increasing the channel's end-to-end performance.

The weight actually combines two factors, each of which plays a part in achieving feasibility: (1) additional latency improvement required, normalized via range-scaling to the interval [0,1]; (2) the average waiting time $E[\mathtt{W}_{i,j}]$ of the task with current load assignment. For the results in this paper, the heuristic we used was:

$$w_{i,j} \leftarrow H(\mathtt{ML_l}, \mathtt{L_l}, \mathtt{E[W_{i,j}]}) = \frac{\mathtt{L_l} - \mathtt{ML_l}}{\mathtt{ML_l}} \times \mathtt{E[W_{i,j}]}$$

Then the selected task gets its utilization increased by some tunable increment Δ. Smaller increments will obviously lead to a greater likelihood of finding feasible solutions; however, they also incur a higher cost. (For the results presented in this paper, we set Δ = .05.)

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 input frequency, it

[2]Clearly, in this example the degree of parallelism is two, and the computation with more than two chains is similar.

can be removed from further consideration.

Interval Assignment. "Get_Interval" derives a feasible service interval (if one exists). While the problem of interval-assignment seems straightforward enough, there are a few non-linearities to surmount: First, 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. Second, in our analysis, we assume that a task finishes only at the end of the service interval, which errs on the side of conservatism. Third, utilization factor of task $\tau_{i,j}$, which is $\frac{E[\Psi_{i,j}]}{E[\Delta \mathtt{Out}_{i,j-1}]}$, may vary with the service interval.

The negative effect of the second factor is likely to be higher at larger intervals, since it results in adding the fractional part of a computation's final interval. On the other hand, the first factor becomes critical at smaller intervals. The third factor depends on the PDF of the task, and the service interval. The approximation utilizes a few simple rules. Moreover, to speed up the computation in the algorithm in Figure 8, we restrict the search to situations where our interval-based delay estimate truncates no more than $\alpha \times 100$ percent of the continuous-time deadline, where $\alpha << 1$. Subject to these guidelines, intervals are evaluated via the latency analysis presented in Section 4 – which determines the current $\mathtt{L}_l$ metric.

Slack Distribution Slack can be used either for fault-tolerance, or for increasing performance. If the latter is desired, one need only re-run "Constraint Satisfaction" algorithm, with a higher target performance. Here 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 that the relocated tasks are in. When a resource has slack larger than the sum of the load thresholds of the relocated tasks to it, (or when system would not be overloaded even after activating those relocated tasks), all the tasks are given their load thresholds. There may not be sufficient slack to maintain the original performance; hence some decisions need to be made. In general, we use the following rules to adjust the load of existing tasks:

rule 1: The loads of the tasks in channels unaffected by the fault are preserved. This is necessary to prevent the effect of the fault from propagating to other channels.

rule 2: In an overloaded resource (due to the relocated tasks), the loads of these tasks are reduced. This "shares" the effect of the fault throughout the channel, evenly in all multi-threaded resources. In the implementation, we gradually reduce the load shares of the old tasks, and re-distribute it to the relocated tasks. This is done evenly, and ensures that thresholds in the resources are not overloaded.

rule 3: Some resources may be highly loaded at this point, the effect of which could produce bubbles in the pipeline. However, often there is spare slack on other resources hosting successor

A. Synthesized Solutions for Channels.

$\mathtt{I_i}$	$\mathtt{L}_i(A)$	$\mathtt{L}_i(S)$	u_{iq}	u_{n1}	u_{rfft}	u_{n2}	u_{ct}	u_{n3}
36	2.6557	2.887	0.423	0.140	0.541	0.140	0.113	0.086
			u_{n4}	u_{afft}	u_{n5}	u_{n6}	$u_{\mathrm{km/aidft}}$	u_{n7}
			0.086	0.437	0.086	0.086	0.539	0.086

B. Resource Capacity Used by System.

ρ_{r1}	ρ_{r2}	ρ_{r3}	ρ_{r4}	ρ_{r5}	ρ_{r6}	ρ_{r7}	ρ_{r8}	ρ_{r9}	ρ_{r10}	ρ_{r11}	ρ_{r12}
0.846	0.541	0.541	0.226	0.437	0.437	0.437	0.437	0.539	0.539	0.539	0.539
ρ_{n1}	ρ_{n2}	ρ_{n3}	ρ_{n4}	ρ_{n5}	ρ_{n6}	ρ_{n7}					
0.281	0.281	0.171	0.171	0.171	0.171	0.343					

Figure 9: Synthesized Solution of the design I.

tasks; if these tasks are given more load share, they can potentially compensate for the bubble. Hence, we re-run the "Constraint Satisfaction" process, to re-adjust the loads in these successor tasks. Often we can achieve a reconfiguration with the original performance, or only a sight degradation.

5.1 Design Examples Revisited

Figure 9 and Figure 10 show the load allocations for each task in our two designs, which were produced by the synthesis algorithm. In those tables, $\mathtt{L}_i(A)$ denotes the latency estimated analytically, and $\mathtt{L}_i(S)$ denotes the latency measured in the simulation model. Note that we found feasible load allocations and service intervals at the 556 Hz for both layouts.

As for the reconfiguration of Design I: Note that resource r2 could accommodate the load thresholds of the relocated tasks $\mathrm{CT}_{1(1)}$, and $\mathrm{CT}_{1(2)}$. The load of processor "r2" becomes 0.917, which is high - but less than its maximum allowable load of 0.95.

The reconfiguration of Design II shows a different situation. Resource "r4" did not have 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 now achieve our threshold of 0.95 - the peak allowed capacity. Now both tasks are given half of the load - however, no service interval can be found that satisfies latency constraints at 556 Hz input frequency.

However, now rule 3 kicks in. The loads of other tasks are increased, to help compensate for delays at the r4 bottleneck. The result is a sampling frequency of 506 Hz - not the peak frequency, but still falling within the SAR guidelines. Figure 12 shows the results of this reconfiguration.

A. Synthesized Solutions for Channels.

$\mathtt{I_i}$	$\mathtt{L}_i(A)$	$\mathtt{L}_i(S)$	u_{iq}	u_{n1}	u_{eq}	u_{n2}	u_{rfft}	u_{n3}	u_{ct}
25	2.65	2.9047	0.372	0.090	0.159	0.090	0.591	0.090	0.163
			u_{n4}	u_{afft}	u_{n5}	u_{km}	u_{n6}	u_{aidft}	u_{n7}
			0.121	0.924	0.121	0.201	0.121	0.924	0.171

B. Resource Capacity Used by System.

ρ_{r1}	ρ_{r2}	ρ_{r3}	ρ_{r4}	ρ_{r5}	ρ_{r6}	ρ_{r7}	ρ_{r8}	ρ_{r9}	ρ_{r10}
0.743	0.317	0.591	0.591	0.326	0.924	0.924	0.403	0.924	0.924
ρ_{n1}	ρ_{n2}	ρ_{n3}	ρ_{n4}	ρ_{n5}	ρ_{n6}	ρ_{n7}			
0.181	0.181	0.181	0.243	0.243	0.243	0.342			

Figure 10: Synthesized Solution of the design II.

6 Simulation

Since the latency analysis uses some key simplifying approximations, we validate the resulting solution via a simulation model. This is a part of our synthesis tool, as shown in Figure 5.

We first review the main source of the approximations. First, we assume periodic arrivals of inputs to all tasks; in reality, inputs may be bursty. Second, we use an approximate joint probability calculation to determine latency at synchronization points. In reality, there may be a lot of data-dependent correlation between the response-times, and this is ignored in the approximation. Third, statistical modeling of periodic input arrivals is highly inaccurate, due to quantization by the service interval. Fourth, 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. A further approximation is inherent in our compositional data-age calculation.

Some of these approximations are conservative; others lead to optimistic results. However, the simulation model discards these approximations, and keeps 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. Also, the simulation model schedules resources using a modified deadline-monotonic dispatcher (where a deadline is considered the end of an interval), so higher-priority tasks will get to run earlier than the analytical method assumes. Recall that the analysis implicitly assumes that computations may take place as late as possible, within a given interval.

On the other hand, the simulator does inherit some other simplifications used in our analysis model. For example, input reading is assumed to happen at the start of an interval. As in

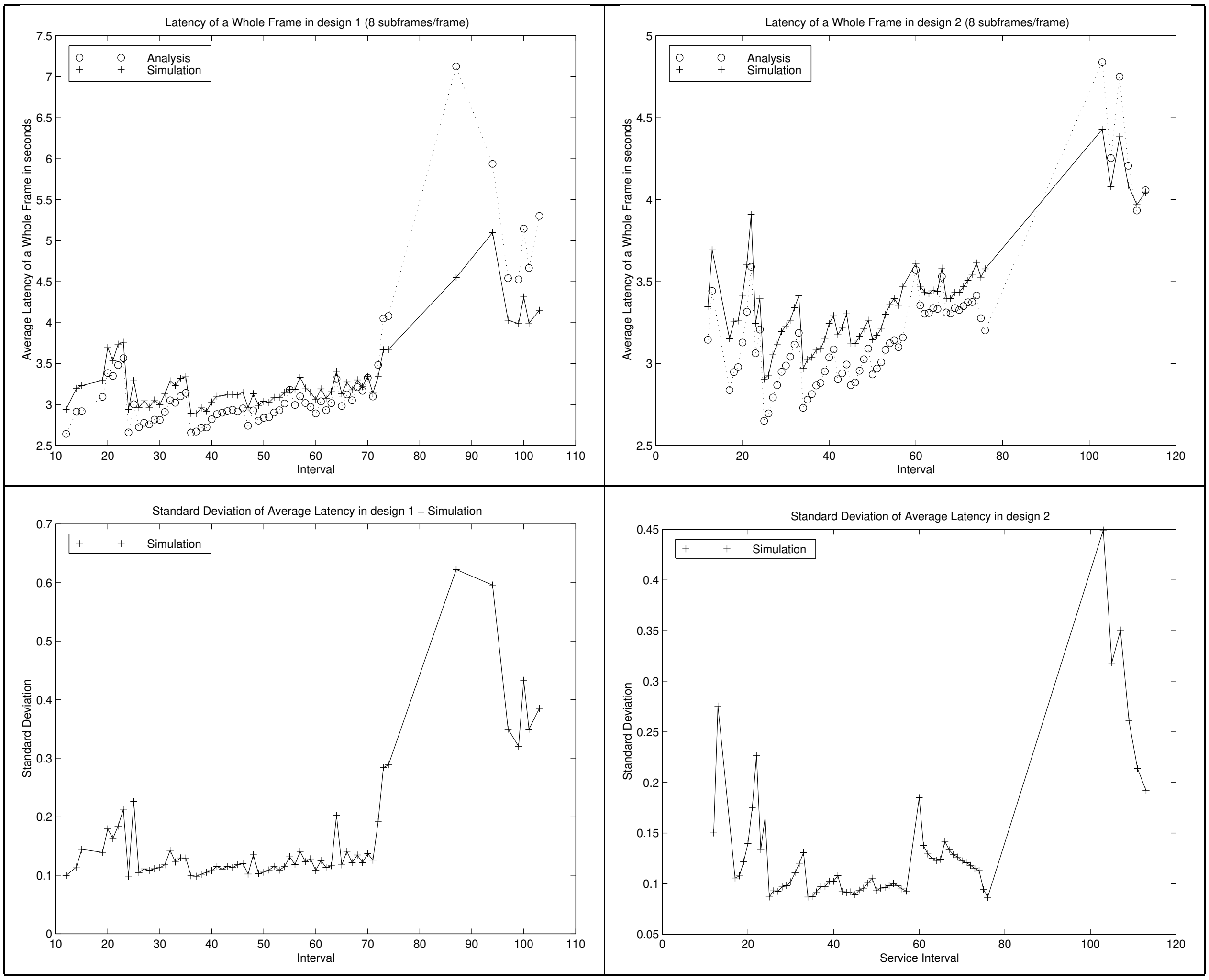


Figure 11: Average latency and its standard deviation of design I and design II at different service intervals

the analysis, context switch overheads are not considered; rather, they are charged to the load distributions. In the simulator, we did not implement infinite-sized buffers; rather, they were restricted to have 2000 slots. However, given the 3 second end-to-end delay, there is little chance that such a huge buffer would fill up - and indeed, it did not in the simulations.

Each simulation trial runs for 20000 frame inputs, which corresponds approximately to 20000 seconds running in real radar processing. The confidence interval of each run with 95 percent is $\pm$ 2 milliseconds.

Figure 11 shows the latencies of Design I and Design II estimated by analysis, and measured by simulation at different service intervals. The load allocation of each task is shown in Figure 9 and Figure 10. Figure 12 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 or lower than a 505Hz input frequency.

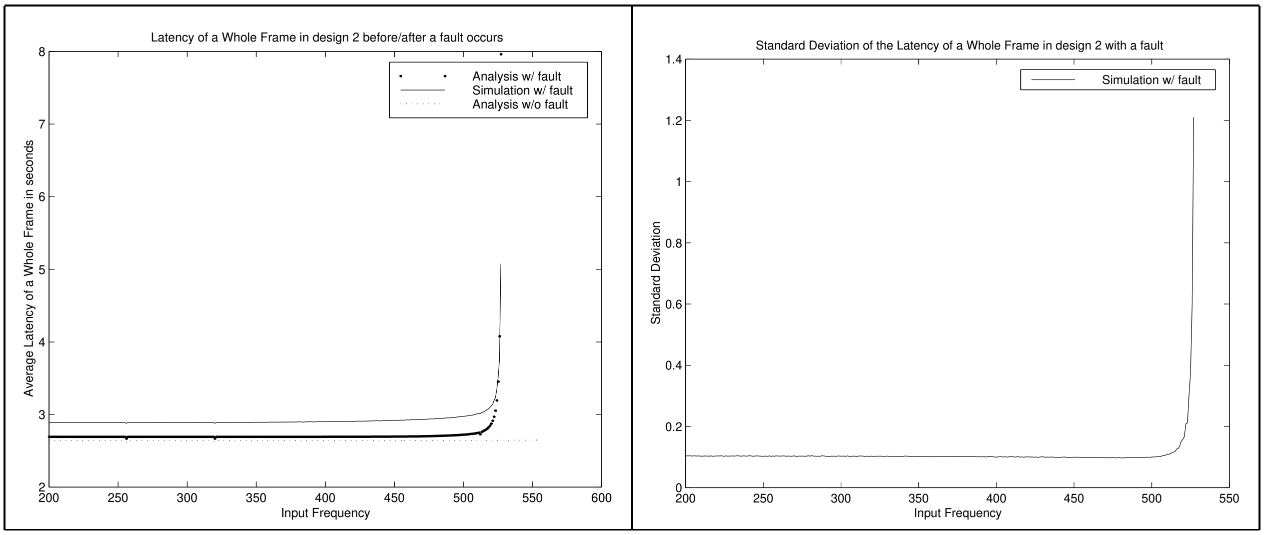


Figure 12: Average latency of the reconfigured and the original design II at 25ms service interval as input frequency changes. And standard deviation of the simulation of the reconfigured design.

From Figure 11 and Figure 12, we note that analysis crosses between conservative estimation to optimistic estimation. The main source of the conservatism comes from the high utilization factors of the tasks in the chain. The utilization of a task may vary with its service interval size. Consider Design I, and $\mathrm{RDFT}_{1(1)}$. At service intervals of 12ms and 84ms, its utilizations are 0.9026 and 0.9946 respectively. At such high utilizations, the system gets less stable, and the statistical approximations for inter-output times start deviating from the true times. We conjecture that this is why service-interval graphs possesse some conservative spikes. In the experiments we ran, however, more than 85 percent of the analytical estimations were within 10 percent of the simulated results.

7 Conclusion

We presented a semi-automated design synthesis technique for calibrating resources in an embedded signal processing system. We showed how analytical techniques can help automate design, and also account for stochastic effects common in COTS systems. We showed how a large SAR could exploit a simple software fault-tolerance scheme - while still giving designers a priori confidence in the stability of their system. Our synthesis tool uses a variety of simple analytic techniques to estimate latency, in tandem with heuristic search algorithm to find a feasible load partitioning.

Though approximation is used, the result is promising. Our four example layouts consist of more than 25 tasks, and 15 shared resources, with compute-times modeled in a variety of different ways. 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. Ultimately, one needs to build the application, and calibrate the kernels and drivers via our analytically-derived parameters. At that point, the system gets subjected to the most important validity test of all: on-line profiling. Even with a careful synthesis strategy, testing usually leads to some additional system tuning – to help compensate for the imprecise modeling abstractions used during static design.

In this regard, we plan on implementing a full-scale version of the RASSP SAR benchmark on the network of workstations; specifically we plan to use off-the-shelf PentiumII processors, connected via a gigabit-ethernet, and with standardized runtimes. In designing a signal processor 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.

References

[1] 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.

[2] 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.

[3] Ladan Gharai and Richard Gerber. Multi-platform simulation of video playout performance. In *Proceedings of SPIE/IS&T Multimedia Computing and Networking (MCMN98)*, 1998.

[4] 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.

[5] Wu chun Feng and Jane W.-S. Liu. Algorithms for scheduling real-time tasks with input error and end-to-end deadlines. *IEEE Transactions on Software Engineering*, 23(2):93–106, February 1997.

[6] R.L. Cruz. A calculus for network delay, part i : Network elements in isloation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

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

[8] 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.

[9] Norival R. Figueira and Joseph Pasquale. Leave-in-Time: A New Service Discipline for Real-Time Communications in a Packet-Switching Network. In *Proceedings of ACM SIGCOMM*, pages 207–218. ACM Press, October 1995.

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

[11] Lucia Franco. Transmission scheduling for fieldbus: A strategy to schedule data and messages on fieldbus with end-to-end constraints. In *Proceedings of IEEE International Symposium on Intelligent Systems /Automation and Robotics (IAR)*. IEEE Computer Society Press, December 1996.

[12] S. Goddard and Kevin Jeffay. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*. IEEE Computer Society Press, June 1997.

[13] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 107–121, October 1996.

[14] Pawan Goyal and Harrick M. Vin. Network Algorithms and Protocol for Multimedia Servers. In *Proceedings of IEEE INFOCOM*. IEEE Computer Society Press, March 1993.

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

[16] Namyun Kim, Minsoo Ryu, Seongsoo Hong, Manas Saksena, Chong-Ho Choi, and Heonshik 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, December 1996.

[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] 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*. IEEE Computer Society Press, December 1996.

[20] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Management. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI '94)*, November 1994.

[21] Carl A. WAldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

[22] David Yates, James Kurose, Don Towsley, and Michael G. Hluchyj. On Per-session End-to-end Delay Distributions and the Call Admission Problem for Real-time Applications with QOS Requirements. In *Proceedings of ACM SIGCOMM*. ACM Press, September 1993.

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

[24] Hui Zhang and Edward W. Knightly. Providing End-to-End Statistical Performance Guarantees with Bounding Interval Dependent Stochastic Models. In *ACM SIGMETRICS*. ACM Press, May 1994.

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

[26] Zhi-Li Zhang, Don Towsley, and Jim Kurose. Statistical Analysis of Generalized Processor Sharing Scheduling Discipline. In *Proceedings of ACM SIGCOMM*, pages 68–77. ACM Press, August 1994.

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