Report Number: WUCSE-2013-2

2013

Simple Analytic Performance Models for Streaming Data Applications Deployed on Diverse Architectures

Jonathan C. Beard, Roger D. Chamberlain, and Mark A. Franklin

Modern hardware is inherently heterogeneous. With heterogeneity comes multiple abstraction layers that hide underlying complex systems. While hidden, this complexity makes quantitative performance modeling a difficult task. Designers of high-performance streaming applications for heterogeneous systems must contend with unpredictable and often non-generalizable models to predict performance of a particular application and hardware mapping. This paper outlines a computationally simple approach that can be used to model the overall throughput and buffering needs of a streaming application on heterogeneous hardware. The model presented is based upon a hybrid maximum flow and decomposed discrete queueing model. The utility of the model... **Read complete abstract on page 2.**

Simple Analytic Performance Models for Streaming Data Applications Deployed on Diverse Architectures

Jonathan C. Beard, Roger D. Chamberlain, and Mark A. Franklin

Complete Abstract:

Modern hardware is inherently heterogeneous. With heterogeneity comes multiple abstraction layers that hide underlying complex systems. While hidden, this complexity makes quantitative performance modeling a difficult task. Designers of high-performance streaming applications for heterogeneous systems must contend with unpredictable and often non-generalizable models to predict performance of a particular application and hardware mapping. This paper outlines a computationally simple approach that can be used to model the overall throughput and buffering needs of a streaming application on heterogeneous hardware. The model presented is based upon a hybrid maximum flow and decomposed discrete queueing model. The utility of the model is assessed using a set of real and synthetic benchmarks with model predictions compared to measured application performance.

2013-2

Simple Analytic Performance Models for Streaming Data Applications Deployed on Diverse Architectures

Authors: Jonathan C. Beard, Roger D. Chamberlain and Mark A. Franklin

Corresponding Author: jbeard@wustl.edu

Abstract: Modern hardware is inherently heterogeneous. With heterogeneity comes multiple abstraction layers that hide underlying complex systems. While hidden, this complexity makes quantitative performance modeling a difficult task. Designers of high-performance streaming applications for heterogeneous systems must contend with unpredictable and often non-generalizable models to predict performance of a particular application and hardware mapping. This paper outlines a computationally simple approach that can be used to model the overall throughput and buffering needs of a streaming application on heterogeneous hardware. The model presented is based upon a hybrid maximum flow and decomposed discrete queueing model. The utility of the model is assessed using a set of real and synthetic benchmarks with model predictions compared to measured application performance.

Type of Report: Other

Simple Analytic Performance Models for Streaming Data Applications Deployed on Diverse Architectures

Jonathan C. Beard, Roger D. Chamberlain and Mark A. Franklin

Dept. of Computer Science and Engineering

Washington University in St. Louis

{jbeard,roger,jbf}@wustl.edu

Abstract

Modern hardware is inherently heterogeneous. With heterogeneity comes multiple abstraction layers that hide underlying complex systems. While hidden, this complexity makes quantitative performance modeling a difficult task. Designers of high-performance streaming applications for heterogeneous systems must contend with unpredictable and often non-generalizable models to predict performance of a particular application and hardware mapping. This paper outlines a computationally simple approach that can be used to model the overall throughput and buffering needs of a streaming application on heterogeneous hardware. The model presented is based upon a hybrid maximum flow and decomposed discrete queueing model. The utility of the model is assessed using a set of real and synthetic benchmarks with model predictions compared to measured application performance.

1. Introduction

In search of ever higher performance, computer architectures have diversified to include a wide variety of heterogeneous hardware such as traditional multicore processors, field-programmable gate arrays

(FPGAs) and general purpose graphics processing units (GPGPUs). Presented with multiple architectural platforms on which to run an application, developers need reliable and computationally feasible models to predict performance. One performance metric of interest to many "big-data" applications is overall throughput. This paper explores an analytic model that is both computationally simple and widely applicable to applications that are formulated as directed acyclic graphs (i.e., they can be considered to be in the streaming data paradigm). Validation is performed across multiple heterogeneous resources, a pair of real streaming applications, and multiple synthetic streaming applications.

Given a set of compute resources and a streaming application, how does an application developer model the overall throughput of the application for a specific hardware mapping? How does a developer determine the size of buffers to allocate based on a target (obtainable) throughput? For example, if an application developer is tasked with developing a streaming JPEG encode application as shown in Figure 1, how is that developer going to assign (map) the compute kernels to the available resources? There are several choices, every kernel labeled with SW (compiled software) can be mapped to a general multicore processor, and those labeled with HW (synthesized hardware) can be mapped to an FPGA. The most obvious, albeit time consuming, approach is simply to do an exhaustive empirical measurement of all possible combinations and chose the best performing one. An alternative approach is to develop a model that reflects the changes in performance that result from alternative mappings, and search over the model space to yield a mapping. This alternative approach has the potential to be much faster than exhaustive empirical search. However, the quality of the final result is strongly influenced by the effectiveness of the model. The model's predictions should reasonably correspond to the actual application performance for this approach to be effective.

Performance models for multicore and heterogeneous systems in general are nothing new. Various approaches exist in practice that use everything from execution histories [1] to the roofline model [31] to help decide how to place a compute kernel and how to modify it for maximum performance. This paper focuses on the use of relatively simple analytic performance models to analyze the obtainable overall throughput and the necessary buffering to obtain it. We are interested in assessing the applicability of straightforward flow modeling techniques to streaming applications deployed on architecturally diverse

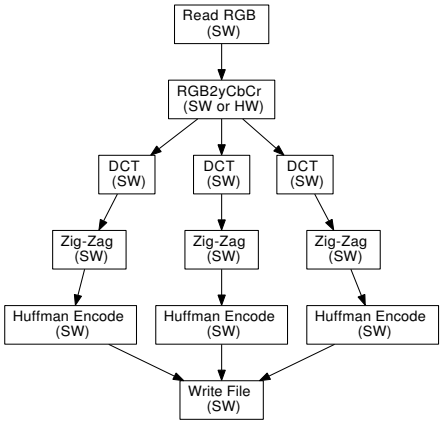

Figure 1. Application topology for JPEG encode [12] expressed as a streaming application. The SW and HW in parenthesis indicate an implementation is available on a multicore processor and/or FPGA respectively.

systems.

The technique is computationally efficient, with a polynomial time solution [10]. It is also usable even when some factors, such as internal interconnect buffering, are unknown. What follows is a brief introduction of background material and then presentation of a computationally simple hybrid maximum flow / queueing network model that incorporates multiple hardware sharing models. Experimental results in Section 5 validate the proposed modeling approach concomitant with individual results for the resource sharing models.

2. Background and Related Work

Stream processing is a computing paradigm that views applications as sets of pipelined kernels connected by streams of data. Each kernel performs specific operations on the data stream before sending it out along an explicit communications link. An example application is shown in Figure 1, in which the boxes represent compute kernels and the arrows represent communications links. Stream processing has been around in various forms for decades [26]. Academic systems include Auto-Pipe [6], Brook [2], Cg [21], S-Net [11], StreamIt [30], and Streams-C [9], while commercial systems include Impulse C [22] and IBM's System S [8]. Examples of application domains that exploit stream processing include media [16], data mining [7], signal processing [24], computational science [20], and others [29].

Streaming applications can be thought of as a series of queues and servers. Each compute kernel is

modeled as a server which draws data from a queue. Each edge is also modeled as a server, representing delivery of data from one kernel to the next. Work by Dor et al. [5] shows that simple queueing networks can accurately model the performance of a heterogeneous streaming application. Many earlier works, including Schweitzer [25], demonstrated that maximum throughput can be determined analytically for a finite-capacity open queueing network. These works have shown that queueing networks can be used for modeling throughput, however they do assume that the queueing capacity is known.

In between each compute node is a communications link. Communications links themselves are often constructed from multiple series of servers and queues, or a "virtual queue." Lancaster et al. [18] showed that these virtual queues have many of the same properties as a single abstract queue and associated server. The model presented here solves for maximum throughput assuming unbounded buffering capacity and then follows with an analysis of the buffering required to maintain that throughput.

Queueing networks have a close relationship with flow networks. Recent work by Boudec [19] considers not individual jobs on a network but flows of jobs within a network. Work by Pourbabai [23] utilizes a maximum flow model to solve a queuing network with side constraints. Unlike the target of these works, most real applications have data-flow routing requirements that are critical to the correctness of the application. For example, the RGB2YCbCr module in Figure 1 takes in a stream of RGB data and outputs three separate streams of Y, Cb and Cr data. A typical formulation of the maximum flow problem, including those mentioned to this point, assume that any path from source to sink can be taken. That is they assign maximum flow to a graph without regard to application imposed data distribution requirements. Using a standard maximum flow model with no further constraints might result in all the data being sent along the Y channel but none to the Cb and Cr channels. The flow model used here places volume constraints on each out edge that are derived directly from data routing requirements of the underlying application.

Many applications exhibit some form of data filtering, that is they change the form of the data from that which is originally received. Two ways in which applications can change data include: increasing or decreasing the volume of data (e.g., a basic block that calculates matrix eigenvalues might take in a grid of data and output a short vector of values) or changing the width of the individual data elements

(e.g., expanding a single unsigned byte to a 64-bit floating point value). Filtering presents an interesting problem for standard maximum flow algorithms. Work by Jewell [15] outlined algorithms for calculating a maximum flow of a network with gain or loss. Using the theoretical work of Jewell the flow model used here is a generalized gain/loss flow network with a fixed branching probability at each out-edge. The model relies on several simple resource sharing models which are empirically validated.

More advanced methods of modeling behavior of applications on shared resources have been used and shown to be relatively effective [3, 27]. Contrary to these complex models, this paper demonstrates that simple models can be as effective as the former complex ones for certain classes of applications, specifically streaming data applications.

3. The Model

3.1. Description

Given the throughput capacity into and out of each compute kernel within an application and the throughput achievable by each communications link, the model presented here calculates maximum data flow through the overall network. Using a constrained generalized maximum flow network the model determines maximum flow through an application topology given a set of constraints. Utilizing a simple $M/M/1$ queuing model, it also estimates the minimum required buffering capacity for each communication edge within the application. What follows is a description of the path from streaming application topology to flow network model, including a queueing network model. Model notation is summarized in Table 1.

An application graph topology G_A (Figure 2) is a connected directed graph consisting of each compute kernel within an application as a vertex V_i and every data-flow dependency (communications link) as an edge $\overrightarrow{V_iV_j}$. An application topology also defines a (pseudo-) data source s and sink t as start and end nodes. Since application topologies can have more than one actual data source and sink, the model inserts the node s with outbound links to all application kernels that do not yet have inbound edges and inserts the node t with inbound edges from all application kernels that do not yet have outbound edges. Nodes s and t are modeled has having infinite capacity so as to not influence the throughput achievable

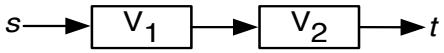

Figure 2. Initial application graph G_A with two compute kernels V_1 and V_2, a data source s, and a data sink t

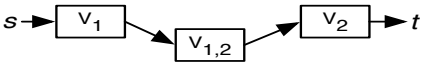

Figure 3. Addition of a communications vertex ($V_{1,2}$) to G_A between compute kernel vertices V_1 and V_2

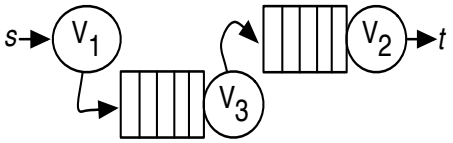

Figure 4. The queueing network G_Q that arises from the topology depicted in Figure 3

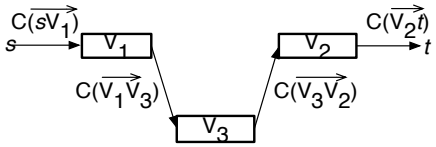

Figure 5. The overall flow graph G_F with capacities C at each edge

in the network.

The model views every communications link as a distinct resource with its own service rate. To model this behavior the application topology (G_A) is transformed by adding additional vertices for each communications link as shown in Figure 3. The transformed application topology of Figure 3 can be directly modeled as a queueing network. The queueing network is defined as the directed graph G_Q (Figure 4). Every application kernel is a queue and server pair in G_Q. Every communications link between compute kernels is also a queue and server pair in G_Q. As illustrated in Figure 4, the nodes modeling communication links in the modified application graph have been renamed so as to simplify the notation. Each communications link could conceivably be reasoned about as being comprised of many sub-queues, however in this work an overall "virtual queue" subsuming the sub-queues will be assumed. Formally G_Q is defined by the 4-tuple:

$$G_Q = (V_Q, E_Q, s \in V_Q, t \in V_Q)$$

where s is the source node and t is the termination (sink) node.

In a queueing network the two main parameters that characterize the performance of the network are $\lambda(V_i)$, the mean arrival rate of data at node V_i, and $\mu(V_i)$, the mean service rate at node V_i. For nodes in V_Q that represent compute kernels, we will measure their service rates empirically, by detaching the

kernel from the application and measuring it in isolation. A compute kernel when detached from its queueing network is simply a single queue and server. That single queue and server is assumed to have an infinite supply of data giving it non-blocking read behavior. Its outbound data port is assumed to always be empty such that outbound writes are also non-blocking. At equilibrium with no gain or loss a server's $\mu(V_i)$ is equal to its aggregate data ingest rate (with units of Bytes/s). The service rates of nodes in V_Q that represent communication links are determined from first principles (i.e., from performance figures published in the literature). The arrival rates $\lambda(V_i)$ will be derived from the flow model described below.

A flow graph is defined as a directed acyclic graph G_F (Figure 5) where each server in the queueing network (Figure 4) is represented as a vertex. G_F is constructed from G_Q by removing the queues on each edge $\overrightarrow{V_iV_j} \in G_Q$. This is reasonable since the queueing model represented here is actually a case of an open Jacksonian network [13, 14]. Formally the flow graph is defined as a 7-tuple:

$$G_F = (V_F, E_F, s, t, C, \gamma, R)$$

$$V_F = V_Q, \quad E_F = E_Q$$

where $C : E_F \rightarrow \Re_+$ represents the flow capacity of each edge (determined as described below), and $\gamma : V_F \rightarrow \Re_+$ represents the data volume gain or loss associated with each node. It is defined as the ratio of the mean data volume out of a node relative to the mean data volume in. If $\gamma < 1$ then there is data loss in the node (e.g., data compression) and if $\gamma > 1$ there is gain in the node (e.g., data expansion). For nodes that represent compute kernels, these values will be determined empirically, and for nodes that represent communication links, $\gamma = 1$. For nodes with more than one outbound edge, $R : E_F \rightarrow (0, 1]$ represents the routing fraction associated with each outbound edge $\overrightarrow{V_iV_j}$ of node V_i. For nodes V_i with only one outbound edge $\overrightarrow{V_iV_j}$, $R(\overrightarrow{V_iV_j}) = 1$.

Given $\mu(V_i)$, $\gamma(V_i)$, and $R(\overrightarrow{V_iV_j})$ for each vertex and edge, the capacity C associated with each edge

can be computed using Equation (1).

$$C(\overrightarrow{V_iV_j}) = \mu(V_i) \times \gamma(V_i) \times R(\overrightarrow{V_iV_j}) \tag{1}$$

Each edge in a flow graph is constrained by the capacity $C(\overrightarrow{V_iV_j})$. Note that the above makes the implicit assumption that each compute kernel has been mapped to a dedicated compute resource. This will be extended to reflect resource sharing in the section below.

To calculate the maximum stable throughput the model maximizes Γ (the overall throughput through the application) and f (the flow at every edge within the graph) subject to the following constraints:

$$\sum_{j|(i,j)\in E_F} f(\overrightarrow{V_iV_j}) - \sum_{j|(j,i)\in E_F} f(\overrightarrow{V_jV_i}) = \begin{cases} + & i = s \\ 0 & i = \text{circulation} \\ - & i = t \end{cases} \tag{2}$$

$$\gamma(\overrightarrow{V_iV_j}) = \frac{1}{\gamma(\overrightarrow{V_jV_i})} \tag{3}$$

$$f(\overrightarrow{V_iV_j}) \leq C(\overrightarrow{V_iV_j}) \tag{4}$$

$$\frac{f(\overrightarrow{V_iV_j})}{\sum_{x=1}^{N} f(\overrightarrow{V_iV_x})} = R(\overrightarrow{V_iV_j}) \tag{5}$$

Equation 2 states that flow must be conserved across all edges and that the only vertices with positive or negative flow can be s and t. Gain or loss as shown in Equation 3 is also conserved. As in a standard maximum flow model, flow must be less than or equal to the capacity as shown in Equation 4. To maintain correct data routing, Equation 5 ensures that the volumes are maintained across each edge.

To bound queue size, the model can be further constrained by ensuring a smaller $\rho = \lambda/\mu$ at each queueing station. The corresponds to maximizing Γ with the following additional constraint:

$$\rho(V_i) \leq \phi \tag{6}$$

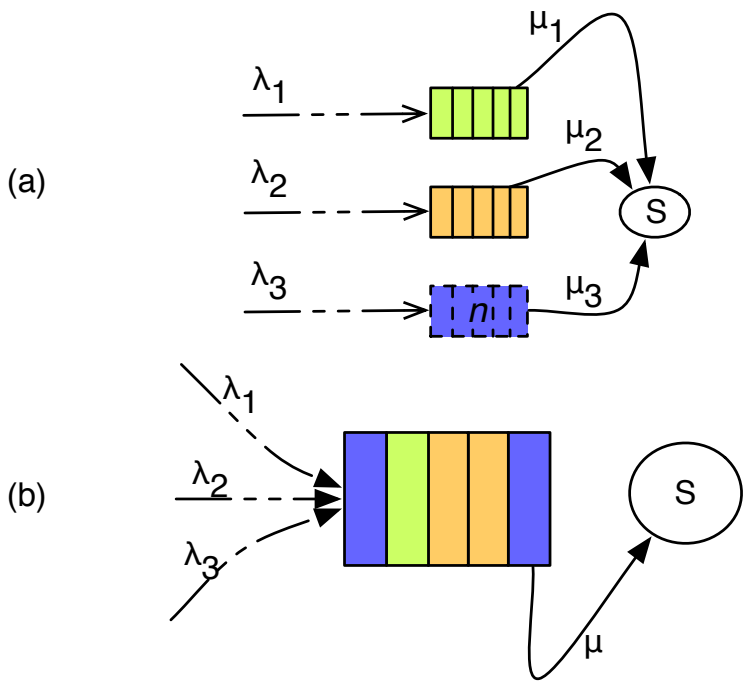

Figure 6. Many compute kernels have multiple incoming communication links, as illustrated in (a). In order to model this behavior, the incoming data streams are combined as shown in (b), and the links are modeled using a single queue.

For the results presented here, ϕ is set to 0.99998, however it could be any value ≤ 1. If equal to 1, then there will not be a queue size bound on the bottleneck node(s), although the technique could still be used to bound the queue size of other nodes in the network.

Once maximal values of $f(\overrightarrow{V_iV_j})$ have been calculated for every $\overrightarrow{V_iV_j} \in E_F$, these values can be used within the queueing model to determine the necessary buffering for the system at the calculated flow. To do this the relationship must be shown between $f(\overrightarrow{V_iV_j})$ and the queueing model parameters $\lambda(V_i)$. For queueing stations with multiple inbound edges, Figure 6 illustrates this circumstance.

In Figure 6 (a) multiple queues are shown with a single server. Each of these queues are treated as sub-queues of one larger queue as shown in Figure 6 (b). The relationship between maximized flows along each edge and λ is therefore

$$\lambda(V_j) = \sum_i f(\overrightarrow{V_iV_j}) \tag{7}$$

Our hypothesis is that the $M/M/1$ model gives an upper estimate of the queue occupancy, since we

Notation	Description
V_i	vertex i
$\overrightarrow{V_iV_j}$	an edge between vertices i and j
$\mu(V_i)$	service rate at vertex i (in Bytes/s)
$\mu_s(V_i)$	shared service rate at vertex i (in Bytes/s)
$\lambda(V_i)$	arrival rate to queue for vertex i (in Bytes/s)
$\rho(V_i)$	utilization of server at vertex V_i
$R(\overrightarrow{V_iV_j})$	fraction of data outbound from V_i routed across $\overrightarrow{V_iV_j}$
$\gamma(V_i)$	gain function across vertex V_i
$C(\overrightarrow{V_iV_j})$	flow capacity of edge $\overrightarrow{V_iV_j}$ (in Bytes/s)
Γ	overall throughput (in Bytes/s)
$f(\overrightarrow{V_iV_j})$	flow along edge $\overrightarrow{V_iV_j}$ (in Bytes/s)
ϕ	constraint on ρ
$K(V_i)$	estimated buffering capacity associated with vertex V_i (in Bytes)

Table 1. The above notation is used to describe the model (where noted, the terminology is applicable to the queueing network, flow graph or both).

expect the actual service time distributions to have a lower coefficient of variation than an exponential distribution. An estimation of the buffering necessary at each queue is determined by solving for the queue occupancy K at a probability P_K that is close to zero as in Equation 8.

$$K(V_i) = \frac{\log(\frac{P_K}{1-\rho(V_i)})}{\log(\rho(V_i))} - 1, \text{where } P_K = 10^{-7} \tag{8}$$

The extent to which the assumptions made for the $M/M/1$ model hold true will be investigated in Section 5.

3.2. Sharing Models

Sharing of resources and resource contention is a function of several parameters. Schedulers are often involved, either from the operating system or built into the hardware. A resource such as an FPGA is typically not shared in time, but shared as a function of area. Diversity in the underlying behavior of sharing across platforms drives the complexity and specificity of sharing models. The models presented here are specific by necessity but intentionally simple.

For multicore processors the sharing model is simply the service rate for a compute kernel executing

in isolation divided by the number of kernels running on the same processor core (Equation 9).

$$\mu_s(V_i) = \mu(V_i)/n, \quad n = \text{\# processes} \tag{9}$$

FPGAs are assumed to be shareable in area, but not temporally. The sharing equation reflects that by giving each compute kernel mapped to an FPGA its full μ until all available gates are exhausted (Equation 10).

$$\mu_s(V_i) = \mu(V_i) \times a_i \tag{10}$$

where $a_i = 1$ if $\sum_{i=1}^{N} Area_i \leq$ Available Area , else $a_i = 0$.

The Virtex-4 FPGAs used for empirical measurement in this paper communicate with multicore processors over a PCI-X bus. The sharing model for this reflects a fair sharing policy on the part of the controller until the bandwidth limit is reached.

$$\mu_s(V_i) = \mu(V_i)/n, \quad n = \text{\# communication links sharing bus} \tag{11}$$

3.3. Modeling Assumptions

The model presented above makes the following assumptions about the applications, graph topology and underlying hardware:

1. The application is assumed to in equilibrium: The streaming computation paradigm is typically used in application domains that require high-throughput, high volume computation. On initial startup and termination non-steady state behavior is exhibited, however during the majority of the execution steady state behavior is typical.

2. The data volume into and out of each edge is measureable on the compute kernel in isolation (i.e., separated from the rest of the application topology.

3. Only non-blocking behavior exists: All compute nodes (servers) are allowed to process data as soon as it is present on its queue.

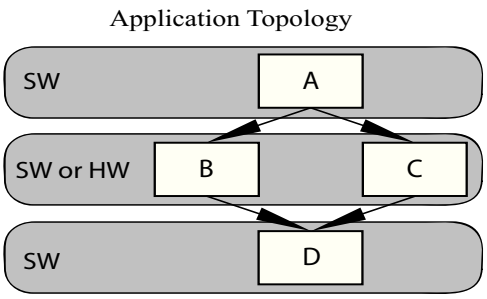

Figure 7. Example application topology

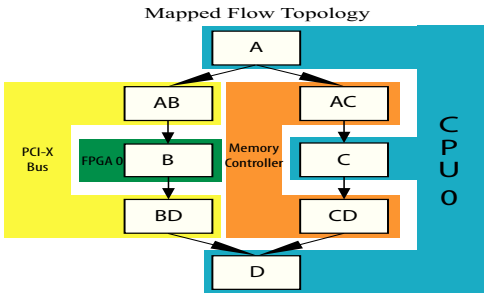

Figure 8. A mapping of the application topology in Figure 7

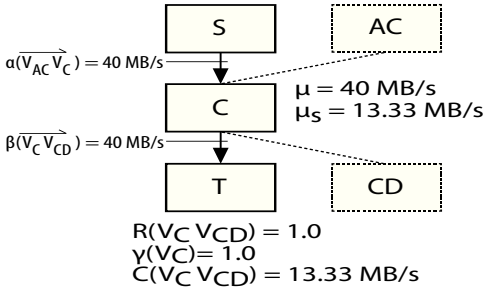

Figure 9. In order to measure the unshared throughput of compute kernel 'C', the kernel is taken out of its application network and given artificial data sources S and T for each in- and out-edge. The dotted lines show the edges where kernel 'C' would have connected in the application which are replaced by the solid lines from S and T. For this kernel the measured input rate for the edge α is 40 MB/s and the measured output rate at edge β is 40 MB/s. The routing fraction is 1.0 as there is only one out-edge and gain (γ) is also 1.0 since there is no gain or loss of data at this compute kernel. The output link capacity C is 13.33 MB/s after application of the sharing model in Equation 9.

4. Data routing is independent of the state of the system: External signals don't drive a server to remove items from a queue, nor do they influence $R(\overrightarrow{V_i V_j})$.

5. All compute kernels are work conserving: When two compute kernels are mapped to the same resource, the work that is done by the compute kernel does not decrease. This is a reasonable assumption for combining individual servers onto a resource. If two compute nodes were combined in such a way that overall work is less for the combined kernel than the two separate nodes then this is non-work conserving.

3.4. Example

In summary the approach presented (illustrated in Figures 7 to 11) begins with a streaming application whose data-flow topology is acyclic. It takes empirical measurements of each compute kernel through

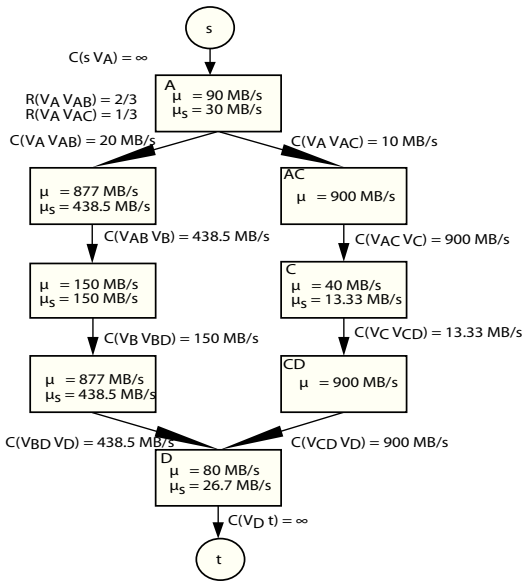

Figure 10. A complete flow graph from the application and mapping in Figure 8

Flow Solution

f(S VA)	26.7 MB/s
f(VAVAB)	17.84 MB/s
f(VAVAC)	8.92 MB/s
f(VABVB)	17.84 MB/s
f(VACVC)	8.92 MB/s
f(VBVBD)	17.84 MB/s
f(VCVCD)	8.92 MB/s
f(VBDVD)	17.84 MB/s
f(VCDVD)	8.92 MB/s
f(VDT)	26.7 MB/s

Figure 11. Solution to flow model in Figure 10

each in-edge and out-edge. The model uses these unshared, unconstrained measurements to calculate mean service rate μ, routing fraction R, and gain γ, associated with each kernel. These metrics are used in the generalized maximum flow model to calculate a maximum flow for the data-flow topology on a specific set of resources. The flows predicted by the flow model are used directly in the $M/M/1$ queueing model to calculate necessary buffering capacity.

4. Model Evaluation Approach

In order to evaluate the model, two approaches are taken. First a pair of real applications are used: a JPEG encode application implemented to the ISO specification (and decomposed as shown in Figure 1) and a DES encrypt application. Second, a set of synthetic applications are generated using a widely used topology generator [4].

For each application, both real and synthetic, random mappings of application kernels to compute resources are generated and run on the hardware enumerated in Table 2. The subsections below describe the tools, hardware, and methods used to evaluate the modeling approach.

4.1. Tools

The Auto-Pipe development environment [6] is used for all experiments. Auto-Pipe supports streaming data applications deployed on heterogeneous compute platforms. In order to make accurate measurements of queue occupancies and edge throughput, the TimeTrial [17] low-impact performance monitor is used.

A graph mapping, modeling and code generation tool called GraphModeler (developed locally) is used as the platform for mapping compute kernels, executing the models described in Section 3 and assessing the effectiveness of the models.

All applications and compute kernels (both real and synthetic) are expressed in combinations of C and VHDL and compiled with the GNU C compiler or synthesized with Synopsys Synplify Premier DP respectively.

4.2. Hardware

There are two distinct hardware platforms used for empirical testing. Each platform is referred to by the heading shown in Table 2.

Table 2. Hardware used for empirical measurement

Name	Machine 1	Machine 2
CPU	12 x 2.4GHz AMD Opteron	4 x 3.1GHz Intel Xeon E3
FPGA	2 x Virtex-4 LX100	None
RAM	32GB DDR2	8GB DDR3

4.3. Empirical Testing

As detailed in Section 3.1, the model needs measurements of each compute kernel running on its assigned hardware as input. To accomplish this each compute kernel is instantiated in isolation and a test bench is produced by GraphModeler that provides high volume input to each input edge and consumes all data on each output edge. Throughput is measured using the TimeTrial measuring system and recorded. These measurements are designated α and β and are shown in Figure 12.

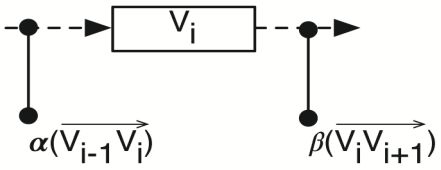

Figure 12. Solid lines with dots indicate where TimeTrial measures throughput along an edge of an application

4.4. Selecting Compute Resources and Mapping Application Kernels

For a given application each compute kernel can be run on many potential resources. This is true for both the real and synthetic applications. In order to select which hardware resource to use for each compute kernel, a series of random walks select resources from the set in Table 2 based on a uniform random distribution. This produces a set Ω of chosen resources for this mapping.

Once the resource set Ω is selected, an application's compute kernels must be mapped to it. To map application kernels to Ω a random compute kernel (drawn uniformly from the set of kernels) is selected and assigned to $\omega \in \Omega$ (again, drawn uniformly from Ω). This process continues until each resource in Ω has one compute kernel mapped to it. The mapping algorithm then assigns compute kernels to resources by randomly walking the in- and out-edges of previously mapped compute kernels until all compute kernels are mapped. A constraint checking algorithm checks user provided constraints on resources while mapping to ensure that the finished kernel/hardware mapping will compile/synthesize and run (e.g., ensuring that the FPGA is not over-utilized). This process is intended not to generate optimal mappings, but rather to generate a range of reasonable mappings for the purpose of assessing the model.

4.5. Synthetic Benchmarks

Whenever the verification of a model is based principally on empirical evidence, a primary consideration is the extent to which the test sets used are truly representative of the overall universe of possibilities. That concern is addressed here through the use of several synthetically generated benchmarks. In order to produce synthetic applications, topologies are generated using the Task Graphs for Free (TGFF) tool [4].

During the topology generation process, the following parameters were used to control TGFF:

1. The number of compute kernels (nodes) in the application topology ranges from 1 to 80 with a uniform distribution.

2. The in-degree and out-degree of nodes is varied from 1 to 4, again uniformly distributed. Sources and sinks have only out- and in-edges respectively.

In order to produce applications from the TGFF generated topologies, GraphModeler uses the following parameters for code generation:

1. Mean execution time is set to 20 μs $\pm$ 10%. Execution time varies dynamically with an exponential distribution.

2. Input data volume for a vertex is statically set with a value chosen between 1 and 64 data bytes. The volume is distributed uniformly.

3. Edges in the graph are constrained so that data volumes are matched between in and out edges.

4.6. Real Applications

The JPEG encode application as shown in Figure 1 is implemented according to the specifications in [12]. The DES encrypt application depicted in Figure 13 is implemented according to FIPS (46-3) standard published by NIST. The topology of each application is specified in the X language [6] which serves as input for the GraphModeler application. GraphModeler takes the pre-coded compute kernel implementations and maps them to hardware resources in the same manner as the synthetic applications mentioned above.

5. Empirical Results

5.1. Processor Sharing Model

Under the processor sharing model (Section 3.2), when multiple compute kernels are mapped to a processor core $\mu(V_i)$ is de-rated according to the number of kernels sharing a given core. A test application designed to run multiple processes on a single core is used to validate this aspect of the model. Each

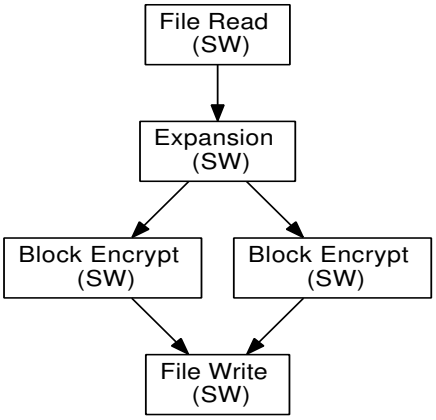

Figure 13. Application topology for the DES encryption algorithm expressed as a streaming application. All compute kernels are implemented in software.

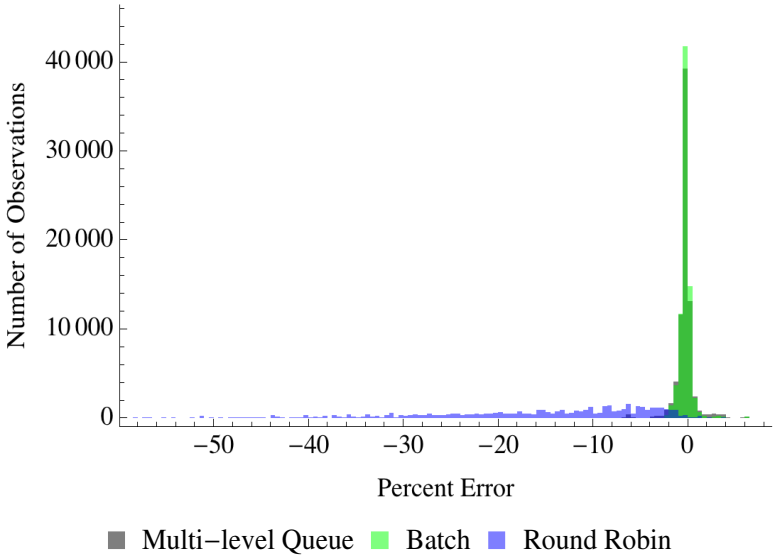

Figure 14. Percent error for processor sharing model from Equation 9 with three different scheduling algorithms (multi-level queue, batch and round robin). All metrics are over 1 through 40 processes on one processor core. Model predicts executions per second. Error is calculated as $\frac{(\text{modeled rate} - \text{observed rate})}{\text{observed rate}}$. R^2 values for each scheduler are .999854, .999952, and .766693 for multi-level queue, batch, and round robin respectively.

process is synchronized to start concurrently with all the other processes within a single experiment (i.e., if 30 processes then all 30 processes are launched together). Each process runs exactly 2 minutes according to the system wall-clock. Each time quantum is consumed by looping for 200 no-op instructions and incrementing a register counter. Tests were run on both machines listed in Table 2. Three different scheduling algorithms (multi-level queue, batch and round robin) were chosen as they are representative of most modern systems [28].

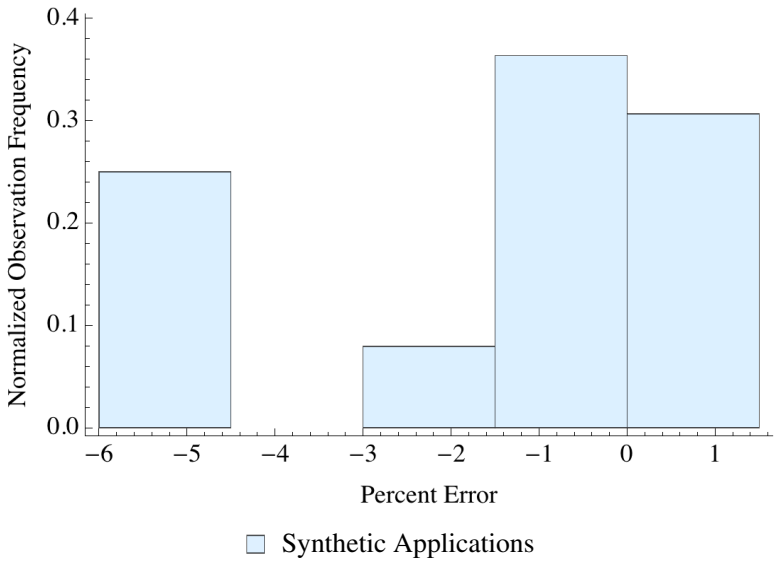

Figure 15. Percent error for gain/loss flow model for the synthetic application set, calculated as $\frac{(\text{modeled flow} - \text{observed flow})}{\text{observed flow}}$. Kernels executed on FPGA and multicore CPUs. Histogram bin size equal to 1.5%.

In Figure 14 the processor sharing model validation percent error distribution is shown for the predicted executions per second. The overall model vs. observed fit is quite good. As expected the round robin scheduler resulted in more variation than the other two scheduling algorithms due to fixed quantum sizing. The fairest schedulers that closely match the assumptions the model makes are the multi-level queue and the batch scheduler.

5.2. The Flow Model

Validation of the flow model proceeds using the set of applications described in Section 4. Forty synthetic applications with 3 through 82 compute nodes were tested on Machine 1 (see Table 2). The results of flow predictions for each edge versus empirically measured flow are shown in Figure 15. Linear regression of the model and measured synthetic application results gives an R^2 value of .9999. The distribution of JPEG encode and DES encrypt application data is very similar to that of the synthetically generated applications as shown in Figures 16 and 17.

Not shown is the data that indicates where the flow model can fail. Firstly, if any of the assumptions are violated, this model's results cannot be trusted. Second, as the number of processes on a single core increases, the error inherent in the simple model grows as well. In our experiments we observed a strong

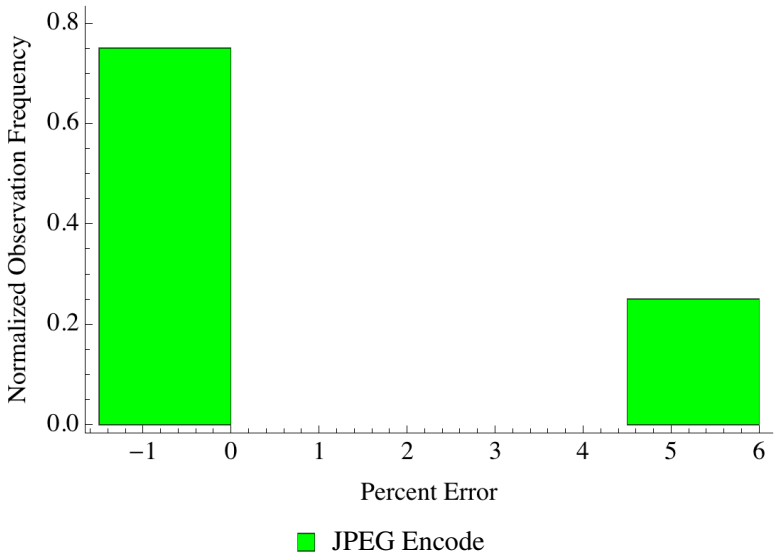

Figure 16. Percent error for gain/loss flow model for the JPEG encode application, calculated as $\frac{(\text{modeled flow} - \text{observed flow})}{\text{observed flow}}$. Histogram bin size equal to 1.5%.

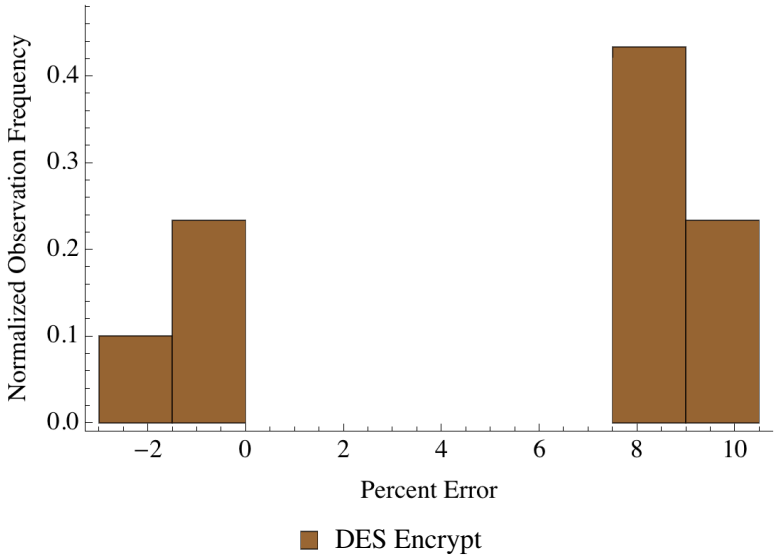

Figure 17. Percent error for gain/loss flow model for the DES encode application, calculated as $\frac{(\text{modeled flow} - \text{observed flow})}{\text{observed flow}}$. Histogram bin size equal to 1.5%.

correlation between increasing percent error and the number of processes per core. Future work will investigate this relationship and perhaps explore the effectiveness of more complex sharing models.

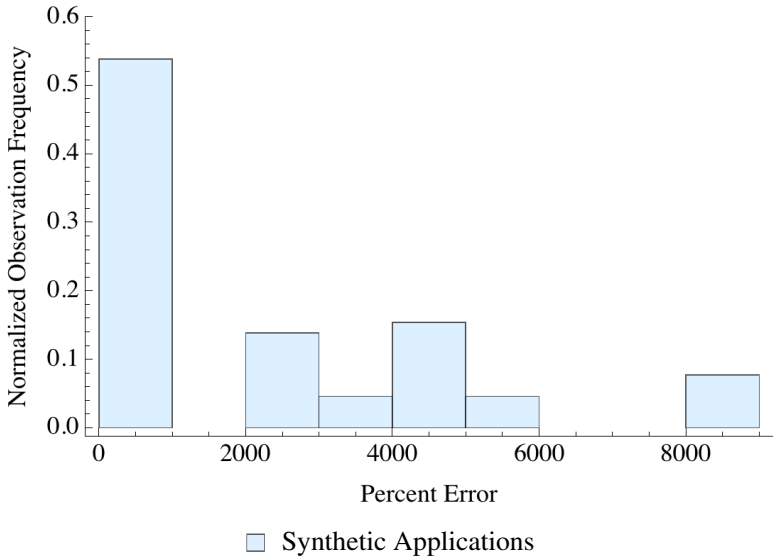

Figure 18. Synthetic application error for modeled queue maximum occupancy vs. measured queue maximum occupancy. For all synthetic applications measured the modeled capacity is always greater. Empirically this shows that for this set of applications the $M/M/1$ queueing model provides a loose upper bound on buffering capacity. Percent error calculated as $\frac{(\text{modeled occupancy} - \text{observed occupancy})}{\text{observed occupancy}}$. Histogram bin size equal to 1000%.

5.3. The Queueing Model

The results for the synthetic, JPEG, and DES applications for an upper bound on queueing capacity are shown in Figures 18, 19, and 20. These figures confirm that our model is conservative for estimating buffering capacity allocations. The modeling assumes exponentially distributed arrival rates and service rates, while real service distributions are typically closer to deterministic (i.e., have a much lower coefficient of variation than an exponential), even if not fully deterministic. It is this distinction that yields conservative estimates for buffer requirements. Note, however, that while conservative, the buffering estimates can be excessive, due to the non-linearity of the queue occupancy relative to server utilization.

6. Conclusions

With multicore chips, FPGAs, general purpose graphics processors and other resources to choose from; application designers have a very difficult set of choices when selecting the best execution platform for a given application. A metric that is of particular interest to "big-data" applications is throughput. The analytic model presented in this paper aims to provide an easy to use method for application devel-

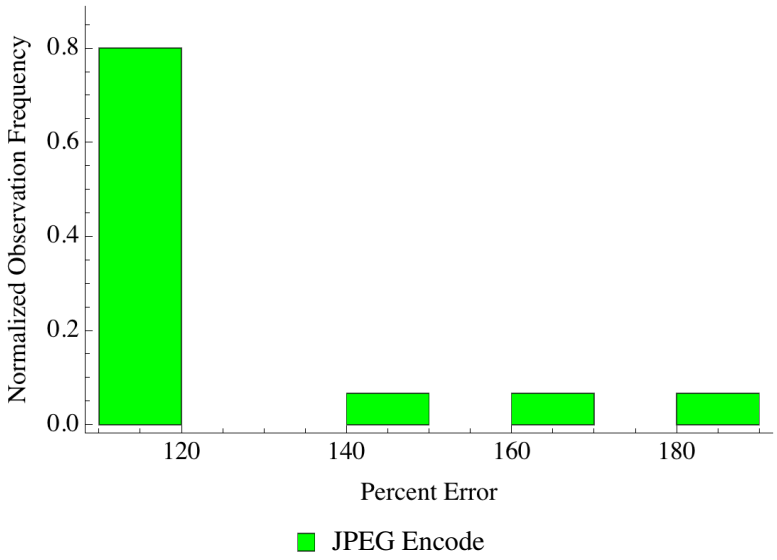

Figure 19. JPEG encode percent error for modeled queue maximum occupancy vs. measured queue maximum occupancy. Three mappings are used across hardware and software. Percent error calculated as $\frac{(\text{modeled occupancy} - \text{observed occupancy})}{\text{observed occupancy}}$. Histogram bin size equal to 10%.

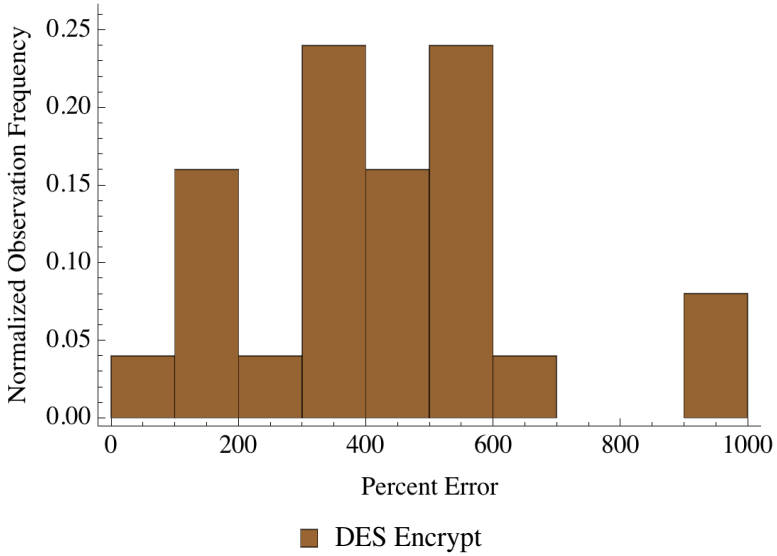

Figure 20. DES encrypt percent error for modeled queue maximum occupancy vs. measured queue maximum occupancy. Four mappings are used on Machine 1, hardware on-chip encryption is not used. Percent error calculated as $\frac{(\text{modeled occupancy} - \text{observed occupancy})}{\text{observed occupancy}}$. Histogram bin size equal to 100%.

opers to find the throughput for an application on a particular set of hardware resources while placing a conservative upper bound on required queueing capacity.

The model was tested using several synthetically generated applications, a JPEG encode application

and a DES encrypt application. The empirical measurements show how the model performs under several conditions and how it can be used to solve for throughputs that are typically within 10% of reality and frequently much closer. This is quite impressive for a set of models that are explicitly trying to stay simple. A unique feature of the model presented is that it can be used across hardware and software platforms. Future work includes testing the boundaries of where these models fail, adding further side constraints to the model so that tighter buffering bounds can be calculated, and exploring the applicability of this model to automated optimization strategies.

Acknowledgment

This work was supported by NSF grants CNS-0905368 and CNS-0931693 and by Exegy, Inc.

References

[1] C. Augonnet, S. Thibault, and R. Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *Proc. of Euro-Par 2009–Parallel Processing Workshops*, pages 56–65, 2010. 2

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics*, 23(3):777–786, 2004. 3

[3] T. Casavant and J. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. on Software Engineering*, 14(2):141–154, 1988. 5

[4] R. Dick, D. Rhodes, and W. Wolf. TGFF: Task graphs for free. In *Proc. of 6th Int'l Workshop on Hardware/Software Codesign*, pages 97–101, 1998. 13, 15

[5] R. Dor, J. M. Lancaster, M. A. Franklin, J. Buhler, and R. D. Chamberlain. Using queuing theory to model streaming applications. In *Proc. of Symp. on Application Accelerators in High Performance Computing*, July 2010. 4

[6] M. Franklin, E. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-Pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006. 3, 14, 16

[7] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. *SIGMOD Rec.*, 34(2):18–26, 2005. 3

[8] B. Gedik, H. Andrade, K. Wu, P. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *Proc. of ACM SIGMOD Int'l Conf. on Management of Data*, pages 1123–1134, 2008. 3

[9] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 49–56, 2000. 3

[10] D. Goldfarb, Z. Jin, and J. Orlin. Polynomial-time highest-gain augmenting path algorithms for the generalized circulation problem. *Mathematics of Operations Research*, 22(4):793–802, 1997. 3

[11] C. Grelck, S.-B. Scholz, and A. Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008. 3

[12] International Telegraph and Telephone Consultative Committee et al. Information technology-digital compression and coding of continuous-tone still images-requirements and guidelines. *Rec. T*, 81, 1992. 3, 16

[13] J. Jackson. Network of waiting lines. *Management Science*, 5(4):518–521, 1957. 7

[14] J. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, 1963. 7

[15] W. Jewell. Optimal flow through networks with gains. *Operations Research*, pages 476–499, 1962. 5

[16] B. Khailany, W. Dally, S. Rixner, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, and A. Chang. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, March/April 2001. 3

[17] J. M. Lancaster, E. F. B. Shands, J. D. Buhler, and R. D. Chamberlain. TimeTrial: A low-impact performance profiler for streaming data applications. In *Proc. of 22nd IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, pages 69–76, Sept. 2011. 14

[18] J. M. Lancaster, J. G. Wingbermuehle, J. C. Beard, and R. D. Chamberlain. Crossing boundaries in TimeTrial: Monitoring communications across architecturally diverse computing platforms. In *Proc. of Ninth IEEE/IFIP Int'l Conf. on Embedded and Ubiquitous Computing*, pages 280–287, Oct. 2011. 4

[19] J. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001. 4

[20] Y. Liu, N. Vijayakumar, and B. Plale. Stream processing in data-driven computational science. In *Proc. of IEEE/ACM Int'l Conf. on Grid Computing*, pages 160–167, 2006. 3

[21] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. on Graphics*, 22(3):896–907, July 2003. 3

[22] D. Pellerin and S. Thibault. *Practical FPGA Programming in C*. Prentice Hall, 2005. 3

[23] B. Pourbabai, J. Blanc, and F. Van der Duyn Schouten. Optimizing flow rates in a queueing network with side constraints. *European Journal of Operational Research*, 88(3):586–591, 1996. 4

[24] J. W. Romein, P. C. Broekema, E. van Meijeren, K. van der Schaaf, and W. H. Zwart. Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer. In *Proc. of ACM Symp. on Parallelism in Algorithms and Architectures*, pages 59–66, 2006. 3

[25] P. Schweitzer. Maximum throughput in finite-capacity open queueing networks with product-form solutions. *Management Science*, pages 217–223, 1977. 4

[26] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997. 3

[27] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. on Software Engineering*, 3(1):85–93, 1977. 5

[28] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2007. 17

[29] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 365–376, 2010. 3

[30] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In R. Horspool, editor, *Proc. of Int'l Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 49–84. 2002. 3

[31] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. 2