

# RAD-FLOWS: Buffering For Predictable Communication

Roberto Pineiro, Kleoni Ioannidou, Scott A. Brandt, Carlos Maltzahn  
Computer Science Department  
University of California, Santa Cruz  
{rpineiro,kleoni,scott,carlosm}@cs.ucsc.edu

*Abstract*— Real-time systems and applications are becoming increasingly complex and often comprise multiple communicating tasks. The management of the individual tasks is well-understood, but the interaction of communicating tasks with different timing characteristics is less well-understood. We discuss several representative inter-task communication flows via reserved memory buffers (possibly interconnected via a real-time network) and present RAD-Flows, a model for managing these interactions. We provide proofs and simulation results demonstrating the correctness and effectiveness of RAD-Flows, allowing system designers to determine the amount of memory required based upon the characteristics of the interacting tasks and to guarantee real-time operation of the system as a whole.

## I. INTRODUCTION

Many real-time applications that were previously executed on dedicated systems in order to meet their needs of predictable performance, now share execution on systems with many other applications. For example, consider a smart phone device, in which an audio application may run simultaneously to a GPS map application. At the same time, many real-time tasks interact with and share data with other real-time tasks or subsystems. The aforementioned GPS application, for example, deals with data sets much larger than can fit into primary storage memory, and must access a (possibly shared) storage device in real-time. Real-time data capture and processing from satellite, video surveillance, radio telescopes, and sensor networks, similarly produce large data sets which must be processed by, transferred to, or stored by other tasks or subsystems in real-time. This creates the need for real-time memory management to support the real-time buffering, transfer, caching, and processing needs of such applications.

Real-time resource managers and schedulers have been developed for many critical system resources including the CPU, network, disk, and others. However, memory is still typically managed by static reservation and overprovisioning. Each application’s memory footprint is determined offline and sufficient memory is built into the system to support all required applications. Where an application’s footprint is too large, it is typically up to the application to manage its own memory usage. Our goal is to develop and implement a general model for real-time memory management that will work together with our real-time CPU, disk, and network managed to ensure predictable performance throughout a shared, distributed, real-time computing environment with

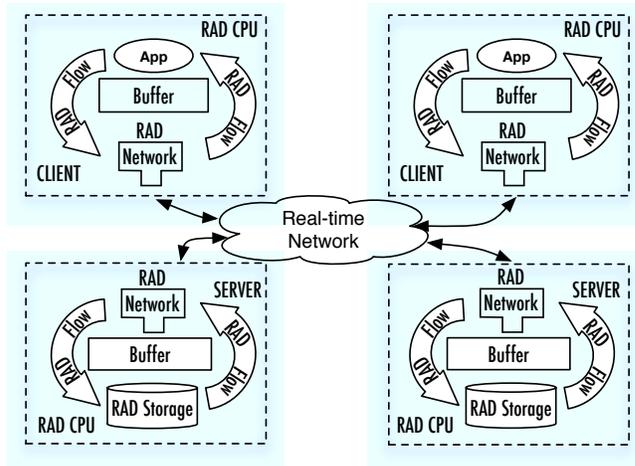


Figure 1: RAD Architecture. End-to-end predictable system.

large amounts of data being processed, buffered, and transferred among tasks on the same and different compute nodes. A driving application is a large, distributed real-time data storage system used for ultra large-scale data capture, filtering, and processing, such as might be used with internet monitoring or radio telescope data capture.

To ensure that a system would behave predictably (i.e., that it would complete a given amount of work in a given amount of time) [1], [2], [3], [4], we have initiated the design of common system schedulers for CPU, disk, and network that ensures that each component behaves predictably according to a unified model. Our goal in this work is to ensure that operations move from component to component in a predictable manner in a multi-task or multi-node computing system. This would allow us to calculate bounds on the total execution time by simply adding the worst-case times spent at each component along the system.

This paper provides the link for those predictable system components. We show how to combine different predictable system components so that operations move from one to the other without compromising predictability. We investigate how to allow a continual flow of operations across system components with different processing capabilities without having to block them due to lack of buffer space to store data. One key aspect to the solution is to account for

buffer space used to communicate between those system components.

Our goal is to provide the tools for system designers to assess how much buffer space is needed—as a function of the processing capabilities of the system components—and how this buffer space can be used to facilitate and guarantee commonly used communication patterns between system components. This accounting of buffer space is needed to ensure hard bounds [5] on the overall time that operations will take. Without it, we could potentially delay delivery of data between real-time components, missing overall system guarantees. For example, using a buffer that can only hold one operation at a time would require us to block applications so that each operation is completed before another one is issued, causing significant delays in the overall execution, dropped data, or worse.

In this paper, we address buffer management to enable predictable delivery of data for distributed hard real-time applications and systems. Given system components whose processing capacities are characterized by a processing rate over some time granularity (called period), we provision enough buffer space to guarantee those performance requirements at all times. Applications and system components run continually according to their processing capabilities (i.e., we do not cause blocking). We chose to describe processing capabilities by a rate and period, according to the RAD (Resource Allocation/Dispatching) model [1], [2], [3], [4], which we already use for CPU, disk, and network resource management. We call our memory management model RAD-Flows, which are designed for end-to-end predictable systems as illustrated in Figure 1.

To the best of our knowledge, RAD-Flows provides the first general framework describing predictable flows of data between components with predictable behavior. We capture a wide range of commonly used communication patterns between communicating tasks or system components. We address simple buffering where data moves from a component to another through a shared queue and buffering for I/O storage operations for which there is a response correlated to each request. For the latter, we distinguish between direct communication, where responses are produced by a predictable system component directly, and indirect communication, which allows the memory shared by the system components that communicate to produce responses using mechanisms such as prefetching and write-back. Our model also applies to systems of multi-level storage hierarchies.

The main contributions of this paper are:

a) A formal definition of predictability for both system components and communication between such components. (Section III).

b) The RAD-Flows model, which consists of a set of basic communication patterns that we show can be combined to describe communication patterns commonly seen in practical scenarios (such as simple buffering, direct, and indirect communication). (Section IV).

c) Theoretical analysis of the RAD-Flows model. We relate the buffer space needed for a certain communication to the processing capabilities of the communicating components and to the time that it would take for this communication to complete. Calculating the time of completion of an operation is useful as some applications may impose bounds on that time. (Section V).

d) Algorithms that use the RAD-Flows model to provide predictable communication together with proofs of their correctness for all types of communication: simple buffering, direct, and indirect. (Section VI).

e) A simulation of a system that verifies our proven bounds on buffer space for the RAD-Flows model while it also helps close the small gap (calculated in our analysis to be a small constant factor) between our solution and the optimal buffer allocation. To construct this simulation we provide insight in what would cause a worst-case execution (which is what we try to trigger). Note that this simulator can be extended in the future to model an end-to-end workload execution for more complex systems. (Section VII.)

Finally, the RAD-FLOW model is extensible and reusable. It can be easily extended to accommodate other communication behavior and the components in the model and analysis can be reused for analyzing other systems involving buffer management and flows (*e.g.* buffer-cache, remote/local copying services, RAID controllers, etc.).

## II. RELATED WORK

The design of inter-task communication for hard real-time systems is challenging. Our goal is to find the maximum bound on the time needed to deliver operations throughout a set of inter-communicating task or nodes with predictable behavior. In our system, operations in the form of requests are delivered and executed. Then they are transformed into responses which are delivered back. The ability to consume/produce operations of each task/node is characterized by their maximum (or minimum) production (or consumption) rate over period of time. We have opted to compute the maximum bound as the summation of the maximum execution time throughout each component along the path. To do so our design must eliminate any dependency between components along the path. These dependencies are introduced due to lack of space to capture data or due to lack of operations readily available in the buffer for consumption. We are ensuring that there is always sufficient space to store operations or data that is available for consumption so that tasks do not block. Note that such blocking would have caused undesirable dependencies which could also lead to missing deadlines.

In our approach, we avoid blocking real-time tasks that are capable of producing (or consuming) specific amount of operations in a given amount of time. We have provided a model that enables system designers to identify the maximum bound on the time operations take to move from component to component. Also, we identify the minimum

buffering space needed to enable each node to process operations at its own rate without having to block tasks/nodes. In the following we will provide a detailed comparison of our contributions to most related existing work.

Asynchronous communication between real-time tasks have been subject of study by [6]. Their work has focused on non-blocking real-time tasks communicating through shared memory by avoiding blocking introduced due to mutex, semaphores, etc. Our approach is complementary to their approach. In our case, we avoid tasks from blocking by providing sufficient resources to store data or by feeding enough data into the buffer to keep the consumer busy. We explore two cases where the producer does not have to block due to lack of empty space to store data (the consumer is allowed to stop without under flowing the buffer), and the case where neither the producer nor the consumer have to stop due to lack of empty space, or lack of operations to consume in the buffer, respectively. The techniques presented in [6] can be applied on top of our analysis to deal with shared memory accesses, while non-blocking is preserved.

Characterizing the amount of time operations remain in the buffer, and the buffer space needed to enable inter-task communication have been subject of study by queueing theory [7]. However, traditional queueing theory does not take into consideration task's period. Although further development of queueing theory for real-time [8] do account for timing considerations, the analysis is focused to task with specific inter-arrival and service time distributions. In our approach, we are particularly interested in bounding the worst-case time operations remain in the buffer between a pair of communication tasks. Those results apply to chains of more than two tasks/nodes as well. Also, we are interested in the worst-case buffering space (maximum queue size or minimum buffer space needed) in order to avoid blocking the inter-communicating tasks. Our analysis is independent of inter-arrival or service time distribution. Furthermore, although queueing theory might be used for predictable communication for soft performance guarantees, our main concern in this paper is to provide hard performance guarantees needed by hard real-time applications. Finally, in our design we accommodate for more complex communication patterns that emerge in practical scenarios in the context of predictable I/O storage access. In some of our analysis we need absolute assurance that no buffer overflow (or underflow) ever occurs. Also, we characterize scenarios where operations aren't removed from the buffer until acknowledgements for completion from a remote component have been received. To our knowledge, those are not captured by any analysis provided by related queueing theory results.

Pipelined systems have been studied before by [9], however they focus on the problem of schedulability of tasks. We aim at providing an alternative approach, where we explore composition of task/node by implementing a com-

mon interface characterizing production/consumption over a specific amount of time (period). The communication pattern that is considered in [9], corresponds to one of the communication patterns that we consider, called the loop. In addition, we also look at other communication patterns that occur when intermediate components are involved in the communication. We also account for more complex communication patterns that emerge in practical scenarios in the context of predictable communication for I/O storage, such as waiting for acknowledgement before releasing operations from the buffer, proactively moving data closer to the application through pre-buffering, and postponing processing of operations, etc.

Similarly to the pipeline approaches [9], we are envisioning an end-to-end system that performs predictably. In contrast to improving utilization and studying schedulability as in [9], we focus on providing a system that is composable through a common interface that will provide hard guarantees. Our idea is to enable system components which enforce predictable behavior, characterized by a common interface, to be plugged into the system and provide end-to-end hard performance guarantees. In particular, our model can be combined with existing work [2], [4], [1], [3] to implement an end-to-end system that enforces hard guarantees while all processing capabilities are no longer expressed by deadlines but by a representation called RAD [2]. We focus on RAD characterizations for the performance of the system components because of the flexibility it introduces while managing resources: it separates rate of resources needed for execution from the point in time when those resources are needed. This enables us to vary these two components independently, while allowing concurrent support of non real-time tasks, as well as real-time tasks with needs for hard and soft performance guarantees. RAD scheduling has been proved useful for CPUs [1] and DISKS [4], [3] as it provides flexible management of resources and has improved performance on those devices, while ensuring hard performance guarantees. Similarly, Shewmaker et al. [1] incorporates RAD based schedulers for networks. Our work is the missing link that will allow construction of an end-to-end system that uses RAD throughout as illustrated in Figure 1. Since each of the RAD components of these systems has better performance compared to related work, our careful accounting of buffering needed for communication of these components is expected to lead us to an end-to-end system that does not over provision resources while it guarantees the performance needs of applications.

Predictable communication for soft real-time applications has been an extensive area of research, specially for multimedia [10], [11], [12], [13], [13]. The communication patterns presented in this paper have been explored before, in the context of soft performance guarantees. However, we are particularly interested in developing models that enable hard performance guarantees needed for hard real-time tasks/nodes. For that, we need detailed models, based on formal

analysis that characterize the fundamental interactions which emerge between inter-communicating tasks/nodes. We build upon these models in order to provide hard performance guarantees along the composition of these primitive communication patterns. This allow us to build more sophisticated communication patterns (such as pre-buffering, waiting for acks) which also provide hard performance guarantees.

### III. PREDICTABLE COMMUNICATION

We consider a system of nodes that directly communicate by sharing buffering components. We explore how this communication should happen to enable the nodes to continuously process operations at their specified processing capabilities without having to stop. We ensure continual flow of operations by accounting for sufficient buffer space without allowing buffer overflow. Data related to operations may remain in the buffer while still needed by applications. The amount of buffers needed is described by a parameter that we call *buffering space*. Each operation's data is maintained in the buffering component for some time that we call *buffering time*. More formally, a *predictable* node (or component) is capable of processing operations according to its processing capabilities. These capabilities characterize the amount of operations that might be processed within some amount of time. We consider predictable nodes and we want to enable predictable communication, formally defined as follows:

**Predictable Communication** A communication between two predictable nodes is predictable if there is enough finite buffer space available in their shared buffering component to accommodate data produced while both nodes are allowed to operate according to their processing capabilities, without having to stop and no useful data ever gets lost.

In Subsection III-A, we describe how we model each node's processing capabilities. In the following subsections, we formalize different predictable communication patterns commonly seen in practice. The communication we consider include simple buffering, direct communication during which an application gets responses directly produced by the system, and indirect communication where an intermediate node (which could be the buffer itself) get involved in the communication and produces responses asynchronously. We describe those communication patterns in detail in the following sections.

#### A. RAD Based Resource Management

The RAD (Resource Allocation/Dispatching) model initially was introduced to manage CPU [2] and later extended to manage disks [3], [4]. This model provides the foundations for predictable resource management by decoupling how many resources are needed, from when those resources are needed.

The model initially introduced device time utilization as the metric for guaranteed performance. This makes sense for system components such as disk and CPU, but for buffers

the RAD metric is rate of operations that can be produced or consumed and the period (time) when this rate must be enforced. Hence, according to RAD for buffers, if a node produces operations with rate  $r$  over periods of length  $p$ , then  $rp$  operations are produced during any period. We use RAD (rate and period) to characterize the processing capabilities of nodes in the system.

#### B. Simple Buffering

Let  $N$  and  $M$  be two nodes that *communicate by simple buffering*.  $N$  produces operations (according to its processing capabilities,  $RAD_N$ ), and those are consumed by a node  $M$  (according to its processing capabilities  $RAD_M$ ) who does not need to send responses. Until the later happens, the data associated with those operations is maintained in the shared buffers between  $N$  and  $M$ . This data occupies some buffering space  $B$  and each data stays in the buffer for some buffering time  $T$ . We need to relate  $RAD_N$ ,  $RAD_M$ ,  $B$ ,  $T$  for the above communication to be possible. The relationship given by a solution to predictable simple buffering would be most valuable if optimal (i.e., it would provide minimum buffer space).

**Predictable Simple Buffering Problem** Given two predictable nodes  $N$  and  $M$  that communicate by simple buffering, our goal is to specify a relation between the processing capabilities of  $N$  and  $M$ , buffering space, and buffering time that makes this communication predictable.

#### C. Direct Communication

We define *direct communication* between two nodes as follows: A node a node  $N$  produces operations according to processing capabilities  $RAD_N$  and expects responses according to possibly different processing capabilities  $RAD'_N$ , with a maximum time separation  $T_{max}$  between these two events. Let  $M$  be the node that consumes the operations produced by  $N$  according to its processing capabilities  $RAD_M$  (that may be different than  $RAD_N$ ). Then  $M$  produces the corresponding responses according to processing capabilities  $RAD'_M$ . There are two cases to consider:

- **Short-Term:**  $M$  consumes the operations and produces responses immediately without any other system component being involved. Then  $RAD_M = RAD'_M$ .
- **Long-Term:**  $M$  contacts other system components before it responds. Then, it is possible that  $RAD_M \neq RAD'_M$ .

Between the time an operation is produced by  $N$  and a corresponding response is received by  $N$ , all related data are kept in the buffering component that is shared between  $N$  and  $M$ . To allow predictable direct buffering (formally described below), we need to relate the processing capabilities of  $N$  and  $M$  to buffering space  $B$  and buffering time  $T$ . An optimal solution would require minimum buffering space.

**Predictable Direct Buffering Problem** Given two predictable nodes  $N$  and  $M$  that communicate by direct buffer-

ing, our goal is to specify a relation between the processing capabilities of  $N$  of producing operations and receiving responses, the processing capabilities of  $M$  of consuming operations and producing responses, buffering space, and buffering time  $T \leq T_{max}$  that makes this communication predictable.

#### D. Indirect Communication

Let  $N$  be a node that initiates an indirect communication with node  $M$ . Node  $N$  produces operations that initiate responses not produced by  $M$  but by the buffering component itself, that we call *intermediate node*  $O$ . There are two specific instances of indirect communication we will explore: pre-buffering and post-buffering.

1) *Pre-buffering*: Consider a node  $N$  that initiates operations and expects some amount of responses above some threshold according to its processing capability  $RAD_N$ , with a maximum time separation  $T_{max}$  between these two events. If the system supports pre-buffering, then data is fetched into the buffering component by node  $O$  that can produce those operations and present them to node  $M$  before  $N$  requests the corresponding data. For this to be possible, we assume that the required data by  $N$  is known in advance (*predictable access*, e.g. multimedia applications).

Let *pre-buffering time* correspond to the time that data in  $M$  is fetched by  $O$  before  $N$  is allowed to request it. Let *response time*  $T$  be the maximum time it takes for  $N$  to get a response to an operation ( $T \leq T_{max}$ ). Let *pre-buffering space* correspond to the amount of data in  $M$  that is fetched by  $O$  before  $N$  is allowed to request operations. The *buffering space* is the maximum buffer size needed at any time during the procedure of pre-buffering. Our goal is to specify how much data and when it should be prefetched by  $O$  to satisfy the requirements presented by applications while taking into account the system's capabilities described by the processing capabilities of  $O$ .

**Predictable Pre- Buffering Problem** Given two predictable nodes  $N$  and  $M$  and intermediate node  $O$  such that  $N$  initiates a pre-buffering operation served by  $M$ , our goal is to specify a relation between the processing capabilities of  $N$ ,  $M$ , and  $O$ , buffering space, pre-buffering space, pre-buffering time, and response time  $T \leq T_{max}$  that makes this communication predictable.

2) *Post-buffering*: Consider a node  $N$  that initiates operations and expects responses according to some processing capability  $RAD_N$ , with a maximum time separation  $T_{max}$  between these two events. If the system supports post-buffering, then data is stored into the buffering component and an intermediary node  $O$  consumes those operations and responds to  $N$ . Eventually,  $M$  receives this data by operations initiated by  $O$  (e.g. real-time data capture).

Let *post-buffering time* correspond to the time that data in  $M$  are stored by  $O$  after  $N$  created it. Let *response time*  $T$  be the maximum time it takes for  $N$  to get a response to an

operation ( $T \leq T_{max}$ ). Let *post-buffering space* correspond to the maximum amount of data temporarily stored by  $O$ . *Buffering space* is the maximum buffer size needed at any time during the procedure of post-buffering. When studying the problem of post-buffering, we would like to specify how much data and when processing of data will begin by  $O$ .

**Predictable Post- Buffering Problem** Given two predictable nodes  $N$  and  $M$  and an intermediate node  $O$  such that  $N$  initiates a post-buffering operation served by  $M$ , our goal is to specify a relation between the processing capabilities of  $N$ ,  $M$ , and  $O$ , buffering space, post-buffering space, post-buffering time, and response time  $T \leq T_{max}$  that makes this communication predictable.

## IV. RAD-FLOWS MODEL

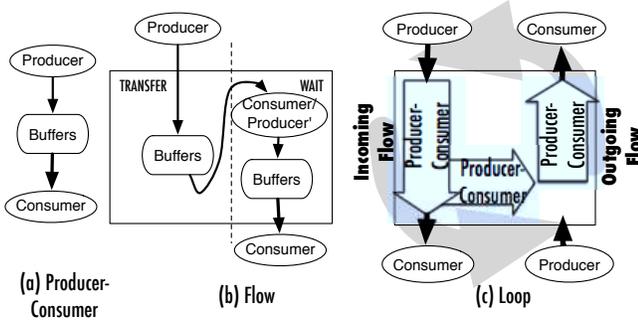
This section introduces the RAD-Flows model that characterizes flow of operations across components with predictable behavior. In the following subsections, we describe how RAD (rate and period) characterizes the processing capabilities of predictable nodes. Then, we introduce the producer-consumer model, which is the core element used to model how information moves throughout system components under worst-case conditions. We show how to chain together producer-consumer models to create flows, which are higher level communication abstractions. Finally, we group flows to create the loop which characterizes how correlated requests/responses move vertically across various system layers.

### A. RAD Performance Interface

Each predictable component's processing capabilities are described by its RAD interface which consists of two RAD characterizations ( $RAD_{in} = (r_{in}, p_{in})$ ,  $RAD_{out} = (r_{out}, p_{out})$ ) and an upper bound for buffering time,  $T_{max}$  (i.e., bound on time between arrival and departure of operations from the component).  $RAD_{in}$  describes the rate and period of information going in the component.  $RAD_{out}$  describes the rate and period of information going out of the component. In particular, the component may receive with up to, but no more, than  $r_{in}p_{in}$  operations per period  $p_{in}$  and it may generate up to, but no more, than  $r_{out}p_{out}$  operations per period  $p_{out}$ . This RAD interface applies to every component introduced in this section.

### B. Producer-Consumer Model

The *producer-consumer model*, illustrated in Figure 2 (a) which characterizes unidirectional communication between a producer and a consumer. It consists of a *producer*, a *consumer*, and memory buffers used to deliver operations between them. A producer (or consumer) might be a software or hardware component, (logical or physical) capable of producing (or consuming) operations with predictable behavior according to its processing capabilities  $RAD_{in}$  (or  $RAD_{out}$  for the consumer). Operations produced by the producer are stored in buffers while being delivered to the



**Figure 2: Modeling Operation Flows:** The producer–consumer, characterizes operations moving across components. A flow, comprised of one or more producer–consumers joined together, characterizes how operations move across a component. A loop correlates requests/responses.

consumer. When the consumer consumes an operation, its related data is removed from the buffer.

Many communication patterns, commonly used across different systems, can be modeled by the producer–consumer model. From all those cases, we extracted three basic patterns which are repeated across many configurations. We call those (*communication*) *building blocks* and they are presented in Table I.

The building block called TRANSFER serves queuing operations between components with possibly different processing capabilities. The producer never stops (or blocks) due to lack of space but it is allowed to produce operations with a variable rate up to a certain upper bound specified by its processing capabilities. Note that in this case, the consumer is allowed to stop if no data is available due to the producer’s possibly reduced production.

The building block PRE–BUF describes how early and how much data needs to be prefetched to meet a consumer’s minimum processing capabilities. The producer never stops (or blocks) due to lack of space and it produces operations according to a fixed rate specified by its processing capabilities. The consumer never stops (or blocks) due to lack of available operations to consume. The consumer will never happen due to the fact that we allow certain operations to accumulate in the buffer before initiating consumption and also the producer never stops. We use PRE–BUF to solve pre-buffering.

Finally, the WAIT building block is an extension of the TRANSFER building block which allows data to reside in the buffer after they have been consumed in anticipation of some additional event to happen such as receiving an acknowledgment confirming successful completion of the operation. An example would be waiting for an acknowledgment confirming that a write has been stored in the remote storage device. The buffer space needed for the WAIT building block depends on the maximum amount of time the corresponding responses take to arrive (i.e., time

to serve the operation on a remote system). This time may depend on factors such as system topology. We defer the analysis of the WAIT building block to future work that considers buffers in memory hierarchies. The buffering time and space of TRANSFER and PRE–BUF is analyzed in Section V.

Building Block	Description
TRANSFER	accounts for buffering (or queuing) space, and worst-case queuing time while moving operations across components with possibly different RAD processing capabilities.
PRE–BUF	accounts for pre-buffering time and space, in order to meet the minimum performance requirements of a consumer.
WAIT	accounts for buffer space needed while retaining operations in the buffering components.

**Table I: Basic building blocks:** Typical producer–consumer instances used by I/O buffer–cache.

### C. Flow of Operations

Flows, depicted in Figure 2 (b), describe the path of operations as they move through the system: into a component, out of a component, or across one or more internal or external components. Each flow is comprised of one or more stages, each modeled by a building block. Flows capture different behavior throughout the path of an operation (e.g. transferring requests, waiting for responses, etc). Figure 2(a and b) illustrates two commonly used flows consisting of one and two stages, respectively. For example, the one stage flow might be used to characterize operations moving from one component into another where resources are relinquished immediately after transferring operations (e.g. when I/O updates are directly transferred into a local storage device). In contrast, the flow with two stages might be used to characterize data moving into a component which may take some time to relinquish the resources (e.g. where I/O updates remain in the buffer until an acknowledgement is received back from the remote storage device).

**Space and Time Requirements:** The buffering space needed for a flow consisting of  $n$  building blocks,  $b_i$ , for  $i \in [1, \dots, n]$ , is given by the sum of the buffers needed by each building block,  $\sum_{i=1}^n \text{Buffering Space}(b_i)$ . Similarly, the buffering time of the flow is given by the sum of the buffering times of its building blocks,  $\sum_{i=1}^n \text{Buffering Time}(b_i)$ .

### D. Request–Response Loops

Our next level of abstraction is a Request–Response loop which uses flows to construct bidirectional communications between components. *Request–response loops* characterize how requests move from a requestor down to a responder, where requests are turned into responses and then these responses move back to the requestor. The *requestor* generates requests and consumes responses, and the *responder* consumes requests and turns them into responses. The requestor

(as well as the responder) might be a hardware, software, or system component. Requests (as well as responses) are operations which may have associated data in either direction or both directions at the same time.

The RAD interface of the loop consists of four RAD performance descriptions. The first two characterizes operations moving in and out of the loop from the requestor on top of the loop. The remaining two characterize operations moving in and out of the responder. Loops consists of two flows chained one after the other, as shown in Figure 2(c). The first flow consists of two chained building blocks characterizing (vertical and horizontal) the flow of data in one direction and the second flow is the building block characterizing the flow of data in the opposite direction. The buffering time of the loop, characterizes the maximum time separation between the point requests arrive into the loop (produced by the requestor) and when responses come out of the loop (consumed by the requestor).

Request–response loops are used to model direct unidirectional as well as bidirectional data transfer. This might particularly useful in direct I/O reads/writes/read–writes. The loop might be used to model stateful delivery of operations as well. Finally, loops can be combined to form hierarchies of loops that can model indirect communication as we show in Section VI.

**Space and Time Requirements:** The buffering space needed for a loop consisting of  $n$  flows ( $\text{flow}_i$ , for  $i \in [1, \dots, n]$ ) is given by the sum of the buffers needed by each flow,  $\sum_{i=1}^n \text{Buffering Space}(\text{flow}_i)$ . Similarly, the buffering time of the loop is given by the sum of the buffering times of its flows,  $\sum_{i=1}^n \text{Buffering Time}(\text{flow}_i)$ .

## V. ANALYSIS OF BUILDING BLOCKS

In this section, we analyze buffering time and space of the basic building blocks. Periods of the producer (or consumer) start immediately after the producer (or consumer) starts operating. The beginning of a period is marked by the end of the previous period but no periods overlap (i.e., if the  $i^{\text{th}}$  period starts at time  $t_i$ , then this period is  $[t_i, t_{i+1})$ ). Considering a set of operations that could be consumed during a period of the consumer (according to its processing capabilities), those are guaranteed to be consumed during that period only if they are available to the consumer at the beginning of the period. The consumer may consume operations that are arriving after the beginning of its periods but this is not guaranteed.

In the following analysis, we assume that both producer and consumer start operating at the same time. This ensures that initially, the periods of the consumer and the producer are aligned. This may not be possible for all practical scenarios as in some cases, a producer could start operating before a consumer starts. To deal with this case, we can keep in a separate buffer all operations produced by the producer until the consumer is ready to start operating. Then we would release the operations of the producer from that buffer in

the same order and distribution as produced earlier. This shifting of operations produced (by the difference between the starting time of the producer and consumer) would cause initial alignment of the periods of the consumer and producer. Hence, using our analysis that works given initial alignment we can solve all cases of unaligned periods by adding this special buffer to capture the data produced while the consumer is not operating. Due to lack of space we only present sketches of some proofs in this section, and we refer the reader to [14] for the formal proofs.

### A. Analysis of TRANSFER Building Block

In this section, we perform the analysis of the TRANSFER building block. A producer  $P$  produces operations with processing capabilities  $\text{RAD}_P = (r_P, p_P)$  (i.e., it produces at most  $r_P p_P$  operations at each of its periods of length  $p_P$ ) and a consumer  $C$  consumes operations with processing capabilities  $\text{RAD}_C = (r_C, p_C)$  (i.e., it consumes at most  $r_C p_C$  operations at each of its periods of length  $p_C$ ). Next, we describe the relation between  $\text{RAD}_P$ ,  $\text{RAD}_C$ , the buffering space  $B$  and the buffering time  $T$  of the TRANSFER building block.

Although the periods of a producer and a consumer may differ arbitrarily, this is not the case for their rates. The consumer must consume at least what is produced to avoid overflow of the buffers in between. For some cases, where the periods are not a multiple of each other then the rate of the consumer may be larger by a small constant factor (smaller than 3) above the rate of the producer, as expressed by the inequality in the following assumption.

*Assumption 5.1:* Let  $r_C \geq \frac{p_P}{\lfloor \frac{p_C}{p_P} \rfloor} r_P$  if  $p_P > p_C$ , or  $r_C \geq$

$\frac{(\lceil \frac{p_C}{p_P} \rceil + 1) p_P}{p_C} r_P$  otherwise.

The analysis of the TRANSFER building block appears in Theorem 5.4 following some preliminary lemmata that specify when operations are consumed in the worst case.

*Lemma 5.2:* Provided that Assumption 5.1 holds, if  $p_P \leq p_C$ , the data produced during  $I_i$  is guaranteed to be consumed during  $I_i \cup I_{i+1}$ , where  $I_i$  is the  $i$ th period of the consumer.

*Lemma 5.3:* Provided that Assumption 5.1 holds, if  $p_P > p_C$ , the data produced during  $I_i$  is guaranteed to be consumed during  $I_i \cup I_{i+1} \cup I_{i+2}$ , where  $I_i$  is the  $i$ th period of the producer.

*Theorem 5.4:* Provided that Assumption 5.1 holds, to allow continuous communication between a producer and a consumer given  $\text{RAD}_P = (r_P, p_P)$  and  $\text{RAD}_C = (r_C, p_C)$ , the buffering space  $B$  and buffering time  $T$  is bounded by the inequalities below:

- i- if  $p_P \leq p_C$  then *Buffer Space*  $B \leq 2 \left( \left\lceil \frac{p_C}{p_P} \right\rceil + 1 \right) r_P p_P - r_P p_P$  and *Buffer Time*  $T \leq 2 p_C$
- ii- if  $p_P > p_C$  then *Buffer Space*  $B \leq 2 r_P p_P + \max(0, r_P p_P - (\lfloor \frac{p_P}{p_C} \rfloor - 1) r_C p_C)$  and *Buffer Time*  $T \leq 3 p_P$

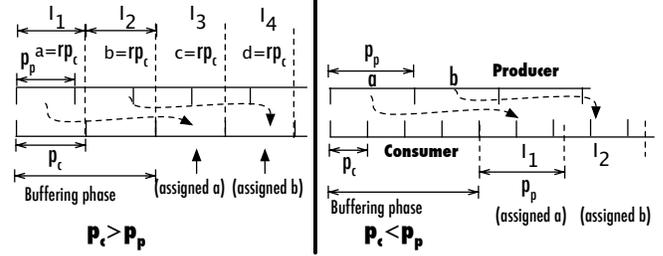
**Proof Sketch** To calculate the buffering space, we calculate the maximum data that can coexist in the buffers. If  $p_p \leq p_c$  then, by Lemma 5.2, the operations pending at time some time in the  $i^{\text{th}}$  consumer's period  $I_i$  are bounded by the operations that can be produced during  $I_{i-1} \cup I_i$  (i.e. the 2 previous periods of the consumer). Otherwise, if  $p_p > p_c$ , then by Lemma 5.3, in the worst case, the operations pending at some time in the  $i^{\text{th}}$  producer period  $I_i$  are the operations that can be produced during  $I_{i-2} \cup I_{i-1} \cup I_i$  (i.e., the operations produced during the last three periods of the producer). We calculated our bounds by adding the maximum number of operations produced during these periods minus the operations that are guaranteed to be consumed considering any possible execution. Considering buffering times, for the case of  $p_p \leq p_c$ , Lemma 5.2 implies that each operation may delay to be consumed by at most 2 periods of the consumer, hence  $T \leq 2p_c$ . Similarly, if  $p_p > p_c$ , then from Lemma 5.3, we get that each operation may delay to be consumed by at most 3 periods of the producer.

Theorem 5.4 can be intuitively explained because if the consumer's period is larger than the producer's period, then the consumer is behaving "lazily" compared to the producer and the buffer pays for it (in amounts proportional to the consumer's period). Otherwise, the buffer's size is independent of the consumer's period because the consumer takes care of the workload in a more "intensive" way compared to the producer. Our simulation in Section VII shows that our bounds on buffering space calculated in Theorem 5.4 are optimal. Analytically optimality can be shown by simply describing this worst case execution of the simulation that requires the bounds of buffer space expressed by Theorem 5.4.

### B. Analysis of PRE-BUF Building Block

Consider a producer  $P$  that produces operations according to processing capabilities  $\text{RAD}_P = (r_p, p_p)$  and a consumer  $C$  that must achieve a consumption of rate of operations  $r_c$  every period  $p_c$ . We call  $(r_c, p_c)$  the *target performance* for the consumer. Hence, the consumer needs to consume  $r_c p_c$  operations during each of its periods to achieve its goal. To make that possible the producer's rate must be the same as the consumer's rate. Otherwise, if the producer's rate is larger the buffer will overflow, and if it is smaller, there will not be sufficient operations to be consumed for the consumer to reach its target performance. Let  $r = r_p = r_c$  be the fixed rate of production/consumption.

In Theorem 5.5, we show that to guarantee the target performance of the consumer, it is necessary to have an initial phase where data produced is being accumulated in the buffer and not consumed. We call this initial phase, *buffering phase*. Then, in the following theorems, we show how the buffering phase, buffer space, and buffer time of the PRE-BUF building block should relate to the processing capabilities  $\text{RAD}_P = (r, p_p)$  and  $\text{RAD}_C = r, p_c$  of  $P$  and  $C$ , respectively.



**Figure 3: Pre-buffering:** Analysis of PRE-BUF building block.

*Theorem 5.5:* If there is no buffering phase, then it is impossible to ensure that after some finite time any execution will allow the consumer to meet its target performance.

*Proof:* We assume that there is a time  $t$  after which the consumer must meet its target performance while no buffering phase exists and get a contradiction. It suffices to describe an execution that fails the target performance after time  $t$ . For this execution, producer and consumer have the same periods and everything produced is directly consumed up to time  $t$ . After time  $t$  the producer produces some operations at the middle of a consumer's period which are not consumed until the next period of the consumer. During this period, the consumer has no data to consume which contradicts the fact that after time  $t$  the consumer should meet its target performance. ■

*Theorem 5.6:* Given a target performance  $(r, p_c)$  of a consumer, a producer can have processing capabilities  $(r, p_p)$  such that  $p_c > p_p$ , if we allow a buffering phase of length  $2p_c$  and there is a buffer with buffering space  $B \leq 2rp_c + (\lceil \frac{p_c}{p_p} \rceil + 1)rp_p$  and buffering time  $T \leq 3p_c + p_p$

**Proof Sketch** The solution is illustrated in Figure 3(left), where we divide the execution of the producer into time frames  $I_i$  of length  $p_c$ . We group the data produced as if they were produced uniformly throughout the execution so that each time frame  $I_i$  gets assigned exactly  $rp_c$  produced operations. Note that those operations may not all be produced during time frame  $I_i$  but we can show that those operations will be available at the beginning of the  $(i+2)^{\text{nd}}$  period of the consumer, where they are guaranteed to be consumed. This shows that the buffering phase of  $2p_c$  suffices. Using the above construction which indicates where operations get consumed in the worst case, we then calculate buffering space and time.

*Theorem 5.7:* Given a target performance  $(r, p_c)$  of a consumer, a producer can have processing capabilities  $(r, p_p)$  such that  $p_c < p_p$ , if we allow a buffering phase of length  $\lceil \frac{p_c + p_p}{p_c} \rceil p_c$  and there is a buffer with buffering space  $B \leq 4rp_p + rp_c$  and buffering time  $T \leq 4p_p + p_c$ .

**Proof Sketch** The main idea of the proof is to divide the consumer's execution into time frames  $I_i$  of length  $p_p$  starting immediately after the buffering phase as illustrated in Figure 3(right). Then we show that the operations produced during the  $i^{\text{th}}$  producer period will be consumed during

the consumer's periods that overlap  $I_i$ . It is important to note that correctness follows because the buffering space is large enough to separate the  $i^{\text{th}}$  producer's period to the consumer's periods that overlap  $I_i$ , so that all operations that are supposed to be consumed during those consumer periods are available at the beginning of the first consumer period overlapping  $I_i$ . Based on this framework that specifies where operations are consumed in the worst case, we calculate the bounds on buffering space and time.

The above theorems show some bounds that hold for all cases of period transformations. For the case where the one of the periods of the consumer or the producer is a multiple of the other, smaller buffering phases and smaller buffers suffice to solve the problem, as we show next.

**Theorem 5.8:** Given a target performance  $(r, p_c)$  of a consumer, a producer can have processing capabilities  $(r, p_p)$  such that either  $p_c$  is a multiple of  $p_p$  or vice versa, if we allow a buffering phase of length  $\max(p_c, p_p)$  and there is a buffer with buffering space  $B \leq 2r\max(p_c, p_p)$  and buffering time  $T \leq 2\max(p_p, p_c)$ .

**Proof Sketch** Let's divide the execution into intervals of length  $\max(p_p, p_c)$ . The operations produced during the  $i^{\text{th}}$  interval will be consumed during the  $(i+1)^{\text{st}}$  interval.

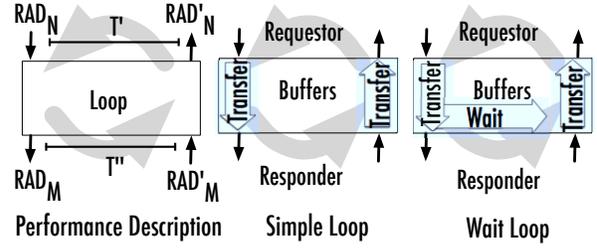
Analytically, it is easy to show that we are within a constant factor from the optimal solution regarding both buffering phase and buffering space. This is possible because if we reduce each by a constant factor we can describe an execution where the target performance fails. This is additionally verified by our simulation although it is not illustrated here due to lack of space.

## VI. SOLUTIONS FOR PREDICTABLE COMMUNICATION

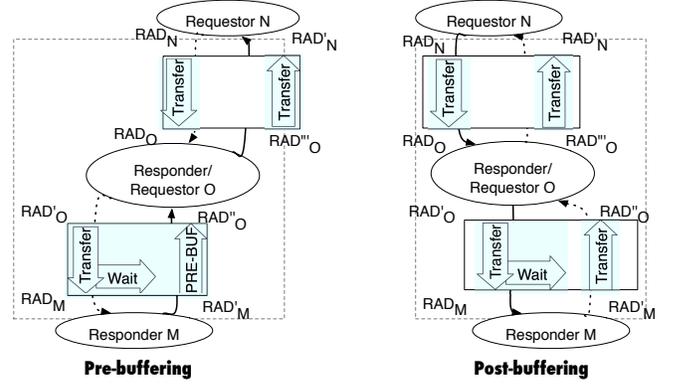
In this section, we show how to use the RAD-Flows model introduced in Section IV, to solve the predictable communication problems described in Section III.

**Simple Buffering Algorithm:** Let  $N$  be a node that produces operations with rate at most  $r_N$  over period  $p_N$ , while  $M$  consumes those operations with rate at most  $r_M$  over period  $p_M$ . We propose a system of a simple TRANSFER building block, where  $N$  is the producer,  $M$  is the consumer with processing capabilities  $(r_N, p_N)$  and  $(r_M, p_M)$ , respectively. Data are stored in the (FIFO) buffers until they are consumed. The buffer's size is specified by Theorem 5.4.

**Direct Algorithms:** The solution to long-term direct problem between two nodes  $N$  and  $M$ , where  $N$  initiates operations served by  $M$ , is the wait loop illustrated in Figure 4.  $N$  is a producer of the left building block and it produces operations with processing capabilities  $\text{RAD}_N$ . The data remains in the buffers of the left TRANSFER building block until it get consumed by  $M$ , with processing capabilities  $\text{RAD}_M$ . Then the data moves into the buffers of the WAIT building block where it remains until responses come back. At this point responses are produced by  $M$  with processing capabilities  $\text{RAD}'_M$  and get stored in the buffers of the right TRANSFER building block until they are consumed by  $N$



**Figure 4: Request-Response Loop:** Solutions to direct communication.



**Figure 5: Pre-Buffering and Post-Buffering solutions.**

with processing capabilities  $\text{RAD}'_N$ .  $N$  expects a response to an operation it produced within time  $T_{max}$ . The system should provide the buffer space needed by this loop (i.e., the sum of buffer spaces of 2 TRANSFER and a WAIT building block). Also this solution works provided that  $T_{max}$  is larger than the buffering time of the loop. The Short-term Direct Communication is solved similarly by the simple loop illustrated in Figure 4. The WAIT building block is not needed as  $M$  produces responses immediately and  $\text{RAD}'_M = \text{RAD}'_M$ .

**Pre-buffering Algorithm:** The solution to pre-buffering is a combination of two loops as illustrated in Figure 5(left). The top loop is a simple loop (that consists of two flows with a TRANSFER building block each) and the bottom loop is a pre-fetch loop (as illustrated in Figure 5 on the left).

There are many possible different variants of this problem that depend on the behavior of  $N$ . Here we assume that  $N$  must consume operations with a fixed rate, hence it also produced operations with the same fixed rate. Alternative scenarios can be similarly solved by our model but are left as future work.

The buffering space of our algorithm is the sum of buffering spaces of all loops involved. The response time of  $N$  in our algorithm is less or equal to the buffering time of the loop involving  $N$  and  $O$ . Pre-Buffering time of the algorithm is at least the time needed to propagate operations from  $O$  to  $M$  characterized by the TRANSFER building

block, plus the pre-buffering phase needed by the PRE-BUF building block.

The pre-fetch loop is responsible for providing the right data that will be used by the simple loop to create responses on its own. The simple loop behaves exactly as specified by the algorithm for short-term direct communication with  $O$  being the node that consumes requests and produces responses. The processing capabilities,  $RAD_O$  of  $O$  have to comply with the capabilities (i.e., speed, etc) of the buffering component.

The consumer  $N$  is not allowed to consume operations from  $O$  until pre-buffering time has elapsed. During that amount of time  $O$  must fetch data from  $M$  at the rate specified by  $N$ . Note that the rates of all the producers through the building blocks are fixed and equal to the rate of  $N$  (which is the target rate of consumption). The rate of the consumers are derived from our analysis of the corresponding building blocks. After pre-buffering time,  $N$  is allowed to consume operations. From that point onward,  $N$  will be able to consume operations at its target fixed until all the workload is finally consumed.

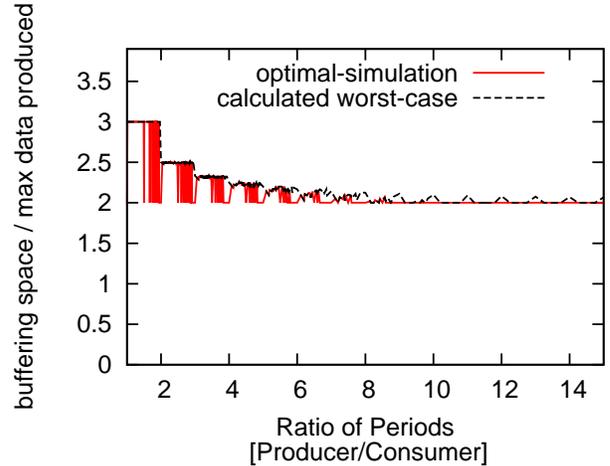
**Post-buffering Algorithm:** In this paper, we address the solution for immediate writeback, where the intermediate component does not retain any operation in the buffer. Instead the intermediate component forwards the operations immediately to their destination. For this solution, post-buffering time and post-buffering space are equal to zero. The buffering space is equal to the space needed for each loop.

The solution to post-buffering consists of two loops: a simple loop on top, and a wait loop on the bottom as illustrated in Figure 5(right). All the producers of the building blocks involved in the solution may produce operations with variable rates bounded by the same maximum value. The rate of the consumers is derived from the analysis of the building blocks. Node  $N$  initiates operations which result in responses back from the intermediate component  $O$ , similarly as described in the algorithm for short-term direct communication. Then  $O$  immediately initiates operations to  $M$  through the bottom loop and eventually  $M$  responds, similarly as explained by the long-term direct communication algorithm.

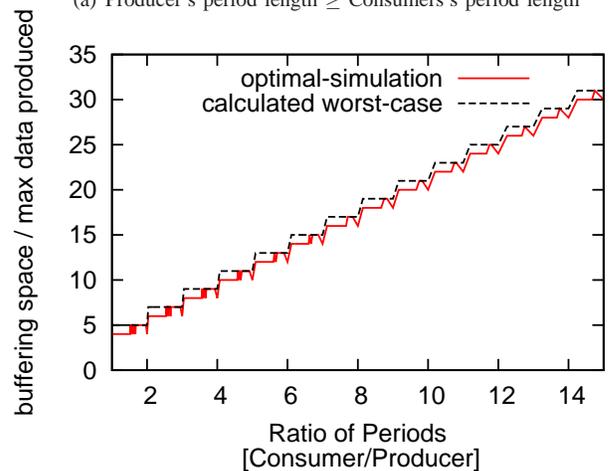
Correctness of all algorithms follows directly from correctness of the building blocks involved in each solution, under the assumption that enough buffer space is available (as specified by our analysis). Since all building blocks work as FIFO queues, no data is replaced unless it is no longer needed. Finally, assuming that  $T_{max}$  is greater or equal to the sum of buffering times of the building blocks of each solution, then all operations will be served within time  $T_{max}$ .

## VII. EMPIRICAL EVALUATION

We have developed a simulation of RAD-Flows to explore worst-case conditions that otherwise would be very difficult to reproduce under a real system. We use our simulator to



(a) Producer's period length  $\geq$  Consumers's period length



(b) Producer period's length  $<$  Consumer's period length

**Figure 6: Buffering space vs ratios of periods.**

evaluate the analytical results regarding the building blocks. Our evaluation confirms the results from the theorems regarding the TRANSFER and PRE-BUF building blocks. Additionally, it verifies that we accounted for the right factors (e.g. buffering phase, buffering space, and buffering time).

In our first set of experiments, we identified maximum number of queued operations (or minimum buffering space) needed to enable communication between predictable components with specific processing capabilities. In our second set of experiments we looked into minimum buffering phase and minimum buffering space needed by the PRE-BUF building block in order to enable a consumption rate above a particular threshold. Not only does our calculated values for all cases (i.e., buffering phase and buffering space) suffice, but in most cases our calculated buffering space is necessary (it matches the bound given by the simulation). This makes our analytical results optimal. Considering all executions, the analytically calculated buffering space is always within

a small (less than 2) constant factor from the buffering space needed according to the simulator.

Due to lack of space, we only present a subset of our experiments. We chose to present results regarding TRANSFER which is the most used building block in our communication patterns. In Figure 6(a), we show that when the consumer's period is slightly smaller than the producer's period, unalignment of periods can cause worst case buffering space (this matches the analytically calculated buffering space). As we decrease the consumer's period, the buffering space needed gets reduced. This is expected because by reducing the consumer's period, we reduce the overhead of unalignment between periods. Eventually, buffer space needed converges to the minimum requirements (within a constant factor from worst case). Based on the relation between producer's and consumer's period, we can provision close to exact the amount of buffering space needed in the experiments.

In Figure 6(b), we illustrate buffering space for the case of the consumer's period being larger than the producer's period. In this case, the worst case calculated by our formula follows the curve of the experiment as it depends on  $p_c$ . The worst case happens again due to unalignment of the periods. As we increase the consumer's period, we see that the worst case and best case buffering spaces converge. This is expected because as the ratio increases towards infinity, the production happens almost uniformly. That allows the consumer to maximize consumption without needing extra buffering space due to unalignment of the periods. This is also the case when the periods have the same size.

## VIII. CONCLUSION AND FUTURE WORK

We introduced the RAD-Flows model to describe predictable flow of operations and used it to ensure predictable communication between nodes that communicate through buffers. RAD-Flows can also be used by algorithms solving different communication patterns that can appear in multi-level environments, in caches etc. By extending our simulation we could evaluate communication patterns that extend across multiple components and could provide insight not just for worst case, but also for average case buffering space. In that case, our model can be used to calculate requirements for applications with soft guarantees. Our work provides the foundation to construct an end-to-end predictable system by linking together existing predictable components.

## REFERENCES

- [1] S. Brandt, C. Maltzahn, A. Povzner, R. Pineiro, A. Shewmaker, and T. Kaldewey, "An integrated model for performance management in a distributed system," *OSPERT*, 2008.
- [2] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, Dec. 2003, pp. 396–407.
- [3] T. Kaldewey, T. M. Wong, R. Golding, A. Povzner, S. Brandt, and C. Maltzahn, "Virtualizing disk performance," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, pp. 319–330, 2008.
- [4] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn, "Efficient guaranteed disk request scheduling with fahrrad," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, pp. 13–25, 2008.
- [5] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.
- [6] K. H. Kim, "A non-blocking buffer mechanism for real-time event message communication," *Real-Time Syst.*, vol. 32, pp. 197–211, March 2006.
- [7] D. Gross and C. M. Harris, *Fundamentals of queueing theory (2nd ed.)*. New York, NY, USA: John Wiley & Sons, Inc., 1985.
- [8] J. P. Lehoczky, "Real-time queueing theory," in *Proceedings of the 17th IEEE Real-Time Systems Symposium*, ser. RTSS '96, 1996, pp. 186–.
- [9] P. Jayachandran and T. Abdelzaher, "Delay composition in preemptive and non-preemptive real-time pipelines," *Real-Time Syst.*, vol. 40, no. 3, pp. 290–320, 2008.
- [10] S. Lim and M. H. Kim, "A real-time prefetching method for continuous media playback," *Database and Expert Systems Applications, International Workshop on*, vol. 0, p. 889, 1999.
- [11] W.-J. Tsai and S.-Y. Lee, "Real-time scheduling of multimedia data retrieval to minimize buffer requirement," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 3, pp. 67–80, 1996.
- [12] S. Oh, B. Kulapala, A. Richa, and M. Reisslein, "Continuous-time collaborative prefetching of continuous media," *Broadcasting, IEEE Transactions on*, vol. 54, no. 1, pp. 36–52, mar. 2008.
- [13] S.-H. Chang, R.-I. Chang, J.-M. Ho, and Y.-J. Oyang, "An optimal cache algorithm for streaming vbr video over a heterogeneous network," *Comput. Commun.*, vol. 28, no. 16, pp. 1852–1861, 2005.
- [14] R. Pineiro, K. Ioannidou, C. Maltzahn, and S. A. Brandt, "RAD-FLOWS: Buffering for Predictable Communication," <http://soe.ucsc.edu/~rpineiro/rtas11-appendix.pdf>, 2010, [Online; accessed 8-October-2010].