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 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.

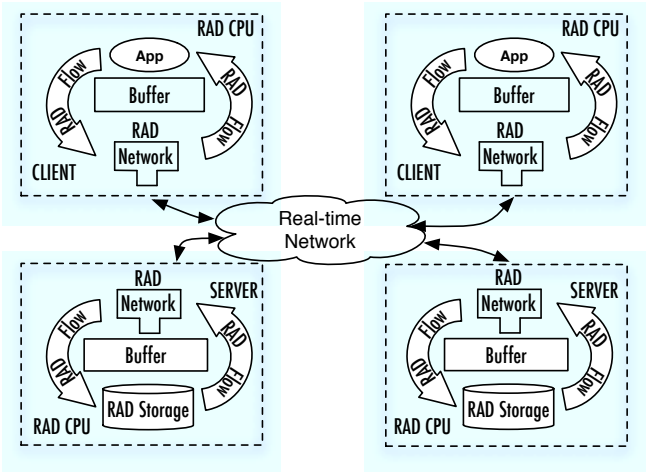


Figure 1: RAD Architecture. End–to–end predictable system.

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 wost-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 common 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 (as well as best–effort) 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 intercommunicating 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 $\text{RAD}_M = \text{RAD}'_M$.
- **Long–Term:** M contacts other system components before it responds. Then, it is possible that $\text{RAD}_M \neq \text{RAD}'_M$.

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 buffering, 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 ($\text{RAD}_{in} = (r_{in}, p_{in})$, $\text{RAD}_{out} = (r_{out}, p_{out})$) 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

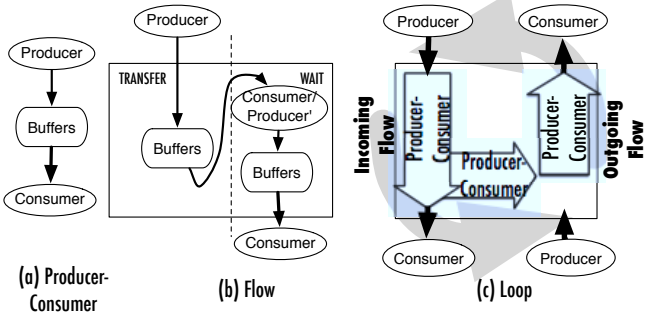


Figure 2: Modeling Operation Flows: The producer–consumer, characterizes operations moving across components. A flow, comprised of one or more producer–consumers joined together, characterizes how operations move across a component. A loop correlates requests/responses.

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 consumer. When the consumer consumes an operation, its relate 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 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 later 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 acknowledgement confirming that a write has been stored in the remote storage device. The buffer space needed for the WAIT building block depend 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 (eg. 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 $i \in [1,\ldots,n]$, is given by the sum of the buffers needed by each building block, $\sum_{i=1}^{n} \textit{Buffering Space}(\text{b}_i)$. Similarly, the buffering time of the flow is given by the sum of the buffering times of its building blocks, $\sum_{i=1}^{n} \textit{Buffering Time}(\text{b}_i)$.

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 turn 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 (flow_i, for $i \in [1,\ldots,n]$) is given by the sum of the buffers needed by each flow, $\sum_{i=1}^{n} \textit{Buffering Space}(\text{flow}_i)$. Similarly, the buffering time of the loop is given by the sum of the buffering times of its flows, $\sum_{i=1}^{n} \textit{Buffering Time}(\text{flow}_i)$.

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^{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 RAD_P, 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 $r_\text{c} \geq \frac{p_\text{p}}{\left\lfloor \frac{p_\text{p}}{p_\text{c}} \right\rfloor p_\text{c}} r_\text{p}$ if $p_p > p_c$, or $r_\text{c} \geq \frac{\left(\left\lceil \frac{p_\text{c}}{p_\text{p}} \right\rceil + 1\right) p_\text{p}}{p_\text{c}} r_\text{p}$ 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 ith 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 ith period of the producer.

Theorem 5.4: Provided that Assumption 5.1 holds, to allow continuous communication between a producer and a consumer given $\text{RAD}_\text{P} = (r_\text{P}, p_\text{P})$ and $\text{RAD}_\text{C} = (r_\text{C}, p_\text{C})$, the buffering space B and buffering time T is bounded by the inequalities below:

i- if $p_\text{p} \leq p_\text{c}$ then *Buffer Space* $B \leq 2\left(\left\lceil \frac{p_\text{c}}{p_\text{p}} \right\rceil + 1\right) r_\text{p} p_\text{p} - r_\text{p} p_\text{p}$ and *Buffer Time* $T \leq 2 p_\text{c}$

ii- if $p_p > p_c$ then *Buffer Space* $B \leq 2r_p p_p + \max(0, r_p p_p - (\lfloor \frac{p_p}{p_c} \rfloor - 1) r_c p_c)$) and *Buffer Time* $T \leq 3p_p$

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^{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^{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

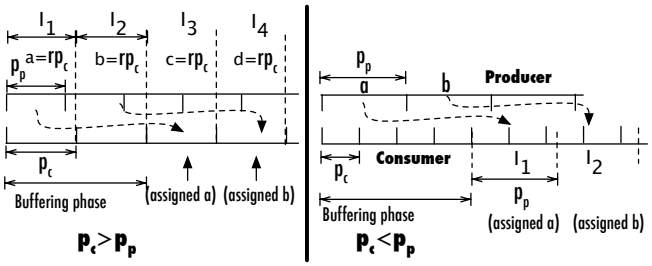


Figure 3: Pre–buffering: Analysis of PRE-BUF building block.

capabilities $\text{RAD}_P = (r, p_P)$ and $\text{RAD}_C = r, p_C$ of P and C, respectively.

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 $(i+2)^{nd}$ 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^{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^{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 vise versa, if we allow a buffering phase of length $max(p_c, p_p)$ and there is a buffer with buffering space $B \leq 2rmax(p_c, p_p)$ and buffering time $T \leq 2max(p_p, p_c)$.

Proof Sketch Let's divide the execution into intervals of length $max(p_p, p_c)$. The operations produced during the i^{th} interval will be consumed during the $(i+1)^{st}$ 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 RAD_N. The data remains in the buffers of the left TRANSFER building block until it get consumed by M, with processing capabilities 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

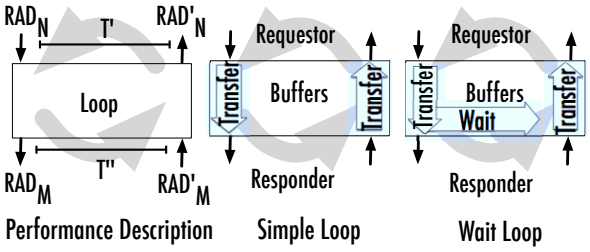


Figure 4: Request–Response Loop: Solutions to direct communication.

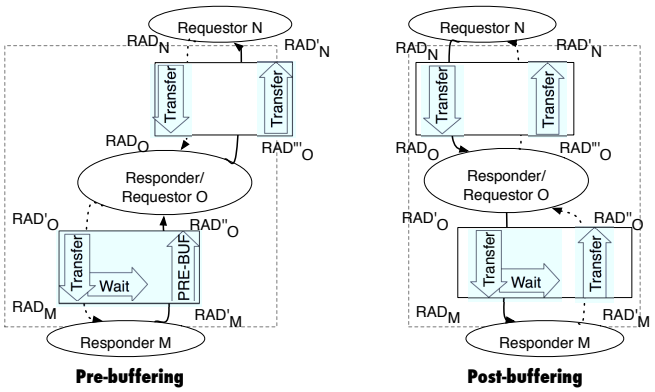


Figure 5: Pre–Buffering and Post-Buffering solutions.

capabilities RAD'_M and get stored in the buffers of the right TRANSFER building block until they are consumed by N with processing capabilities 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.

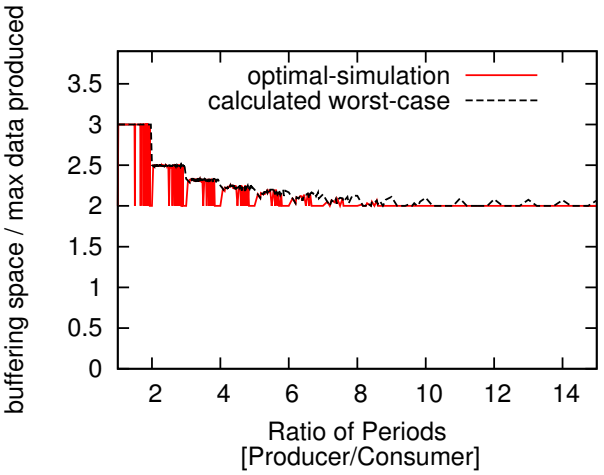
The pre-fetch loop is responsible for providing the right data that will be used by the simple loop to create responses on its own. The simple loop behaves exactly as specified by the algorithm for short–term direct communication with O being the node that consumes requests and produces responses. The 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.

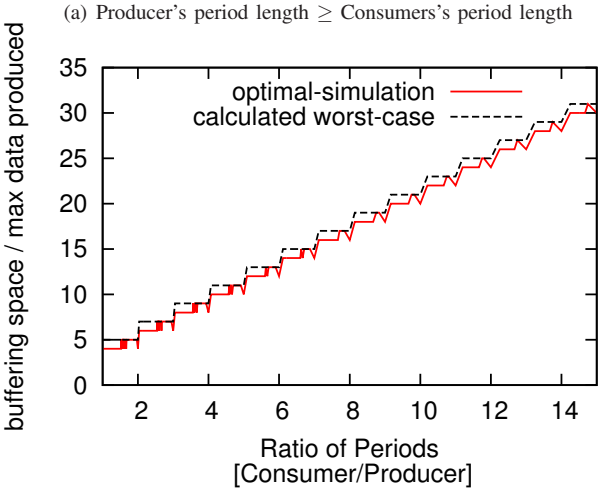
Post-buffering Algorithm: In this paper, we address the solution for immediate writeback, where the intermediate component does not retain any operation in the buffer. Instead the intermediate component forwards the operations immediately to their destination. For this solution, post-buffering time and 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}.



(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.

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 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 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 torwards infinity, the production happens almost uniformly. That allows the consumer to maximize consumption without needing extra buffering space due to unaligment 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. Malthzan, and S. A. Brandt, "RAD-FLOWS: Buffering for Predictable Communication," http://soe.ucsc.edu/~rpineiro/rtas11-appendix.pdf, 2010, [Online; accessed 8-October-2010].

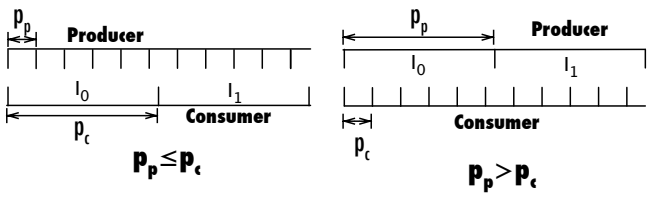


Figure 7: Producer–Consumer. Unaligned producer–consumer period of different length.

APPENDIX

In the appendix we provide all formal proofs of lemmata and theorems that did not appear in Section V due to lack of space.

A. Formal Proof for TRANSFER building block

Lemma A.1: Provided that Assumption 5.1 holds, if $p_{\mathrm{p}} \leq p_{\mathrm{c}}$, the data produced during I_i is guaranteed to be consumed during $I_i \cup I_{i+1}$, where I_i is the ith period of the consumer.

Proof: We will prove the lemma by strong induction on i. Let x_i be the operations produced during period I_i.

Operations x_0 are the only operations produced by the end of I_0 and they are either consumed during I_0 (i.e., the 1st consumer period) or are pending at the beginning of I_1. By the definition of RAD reservations and the fact that $p_{\mathrm{p}} \leq p_{\mathrm{c}}$, then $\|x_0\| \leq (\lceil \frac{p_{\mathrm{c}}}{p_{\mathrm{c}}} \rceil + 1) r_{\mathrm{p}} p_{\mathrm{p}}$. But by the assumption of the statement (i.e., $r_{\mathrm{c}} p_{\mathrm{c}} \geq (\lceil \frac{p_{\mathrm{c}}}{p_{\mathrm{p}}} \rceil + 1) r_{\mathrm{p}} p_{\mathrm{p}}$), we conclude that $\|x_0\| \leq r_{\mathrm{c}} p_{\mathrm{c}}$. Hence, $\|x_0\|$ operations can be consumed during one period of the consumer, and since those operations are available at the beginning of I_1, they are guaranteed to be consumed during I_1 because the buffer behaves as a simple FIFO queue.

We assume that the lemma holds for all $i \in [1, k-1]$ and we will prove the statement for i=k.

We will prove that x_k (i.e., the operations produced during I_k, will be consumed during $I_k \cup I_{k+1}$. By the inductive hypothesis, we get that all operations produced during the execution up to and including period I_{k-1} have been consumed by the end of period I_k. Hence at the end of period I_k the only operations that may be pending (i.e., not yet consumed and at the beginning of the FIFO queue) are the ones that have been produced during I_k (i.e., x_k). We will show if those operations are not consumed during I_k, there will be consumed during I_{k+1}. During I_k there can be at most $(\lceil \frac{p_{\mathrm{c}}}{p_{\mathrm{p}}} \rceil + 1) r_{\mathrm{p}} p_{\mathrm{p}}$ operations produced. By the statement of the lemma, those are less than $r_{\mathrm{c}} p_{\mathrm{c}}$ which is what can be consumed within a consumer's period. We conclude that all operations produced during I_k will be consumed by the end of I_{k+1}. ■

Lemma A.2: Provided that Assumption 5.1 holds, if $p_{\mathrm{p}} > p_{\mathrm{c}}$, the data produced during I_i is guaranteed to be consumed during the consumer periods that start in I_{i+1} if not earlier, where I_i is the ith period of the producer.

Proof:

Let x_i be the operations produced during period I_i. We will prove the lemma by strong induction on i.

For $i = 0$, we get that during the first producer's period, I_0, $\|x_0\| \leq r_{\mathrm{p}} p_{\mathrm{p}}$ data is produced. There is no previous data pending in the system hence, by the end of I_0, exactly $\|x_0\|$ operations are available for consumption (at the beginning of the FIFO queue) or less if some operations in x_0 are already consumed during I_0. Time period I_1, contains the beginning of at least $\lfloor \frac{p_{\mathrm{p}}}{p_{\mathrm{c}}} \rfloor$ (consecutive) periods of the consumer (each of length p_{c}). During each of these periods the consumer can consume at most $r_{\mathrm{c}} p_{\mathrm{c}}$ operations (if available). Hence during those periods the consumer is guaranteed to consume up to $\lfloor \frac{p_{\mathrm{p}}}{p_{\mathrm{c}}} \rfloor r_{\mathrm{c}} p_{\mathrm{c}}$ operations. Because by assumption in the lemma statement, $\lfloor \frac{p_{\mathrm{p}}}{p_{\mathrm{c}}} \rfloor r_{\mathrm{c}} p_{\mathrm{c}} \geq r_{\mathrm{p}} p_{\mathrm{p}}$ and $\|x_0\| \leq r_{\mathrm{p}} p_{\mathrm{p}}$, then the part of x_0 that is not consumed in I_0 it will be consumed during the consumer periods that start during I_1 since those operations are already in the queue before the beginning of each of those consumer's period.

We assume that the lemma holds for $i \in [1, k-1]$ and we will prove the lemma for i=k.

We will prove that x_k (i.e., the operations produced during I_k, will be consumed during the consumer's periods initiated during I_{k+1}, if not earlier. By the inductive hypothesis, we get that all operations produced during the execution up to and including period I_{k-1} have been consumed by the end of the consumer periods that were initiated during I_k. Let that time be t. Hence at the end of those periods (i.e., at time t) the FIFO queue contains at its beginning the subset of the operations that have been produced during I_k (i.e., x_k) and not yet consumed. Time t is in I_{i+2}. In particular, time t is the beginning of the first consumer period that starts during I_{i+2}. We will show that the pending operations among x_k at time t will be consumed by the end of the consumer periods initiated during I_{k+1} (i.e., initiated at time t).

Time period I_{k+1}, contains the beginning of at least $\lfloor \frac{p_{\mathrm{p}}}{p_{\mathrm{c}}} \rfloor$ (consecutive) periods of the consumer (each of length p_{c}). During each of these periods the consumer can consume at most $r_{\mathrm{c}} p_{\mathrm{c}}$ operations (if available). Hence during those periods the consumer is guaranteed to consume up to $\lfloor \frac{p_{\mathrm{p}}}{p_{\mathrm{c}}} \rfloor r_{\mathrm{c}} p_{\mathrm{c}}$ operations. Because by assumption in the lemma statement, $\lfloor \frac{p_{\mathrm{p}}}{p_{\mathrm{c}}} \rfloor r_{\mathrm{c}} p_{\mathrm{c}} \geq r_{\mathrm{p}} p_{\mathrm{p}}$ and $\|x_k\| \leq r_{\mathrm{p}} p_{\mathrm{p}}$, then the part of x_k that is not consumed by time t it will be consumed during the consumer periods that start during I_{k+1} since those operations are already in the queue at time t. ■

Lemma A.3: Provided that Assumption 5.1 holds, if $p_{\mathrm{p}} > p_{\mathrm{c}}$, 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 ith period of the producer.

Proof: By Lemma A.2 the data produced during I_i is guaranteed to be consumed during the consumer periods that start in I_{i+1} if not earlier, where I_i is the ith period of the producer. But those consumer periods will terminate before the end of I_{i+2} because the consumer's period is smaller than the producer's period. The Lemma follows. ■

Theorem A.4: Provided that Assumption 5.1 holds, to allow continuous communication between the producer and a consumer given $\text{RAD}_\text{P} = (r_\text{P}, p_\text{P})$ and $\text{RAD}_\text{C} = (r_\text{C}, p_\text{C})$, the buffering space B and buffering time T is bounded by the inequalities below:

i- if $p_\text{p} \leq p_\text{c}$ then *Buffer Space* $B \leq 2\left(\left\lceil \frac{p_\text{c}}{p_\text{p}} \right\rceil + 1\right) r_\text{p} p_\text{p} - r_\text{p} p_\text{p}$ and *Buffer Time* $T \leq 2p_\text{c}$

ii- if $p_\text{p} > p_\text{c}$ then *Buffer Space* $B \leq 2r_\text{p} p_\text{p} + \max(0\,, r_\text{p} p_\text{p} - (\lfloor \frac{p_\text{p}}{p_\text{c}} \rfloor - 1) r_\text{c} p_\text{c})$) and *Buffer Time* $T \leq 3p_\text{p}$

Proof: Let x_i be the of operations produced during period I_i. We prove the theorem for the following two cases:

If $p_\text{p} \leq p_\text{c}$ then by Lemma A.1, we conclude that the operations consumed by some time t, include all operations that have been produced up to period I_{i-2}, where i is such that $t \in I_i$ and I_i is the ith period of the consumer. Therefore, in the worst case, the operations pending at time t, include the operations that can be produced during $I_{i-1} \cup I_i$. This provides an upper bound on the size of the buffer needed since the buffer's role is to hold the operations that are pending at any time. Next we calculate an upper bound on the data that can be produced during $I_{i-1} \cup I_i$. Let v be that data, then the buffer space needed is at most equal to v (i.e., $B \leq v$). During any two consecutive consumer's periods there can be at most $2(\lceil \frac{p_\text{c}}{p_\text{p}} \rceil + 1) r_\text{p} p_\text{p} - r_\text{p} p_\text{p}$ operations produced because there can be at most $2(\lceil \frac{p_\text{c}}{p_\text{p}} \rceil + 1) - 1$ overlapping periods of the producer. This is because every period of the consumer can overlap at most $(\lceil \frac{p_\text{c}}{p_\text{p}} \rceil + 1)$ producer periods but then when this value is reached one of them would be shared between this consumer period and the next one. Hence during any two consecutive consumer periods there can be at most $2(\lceil \frac{p_\text{c}}{p_\text{p}} \rceil + 1) - 1$ overlapping producer periods. Since there can be at most $r_\text{p} p_\text{p}$ operations produced during each producer periods, then during any two consecutive consumer periods, there can be at most $(2(\lceil \frac{p_\text{c}}{p_\text{p}} \rceil + 1) - 1) r_\text{p} p_\text{p} = 2(\lceil \frac{p_\text{c}}{p_\text{p}} \rceil + 1) r_\text{p} p_\text{p} - r_\text{p} p_\text{p}$ operations produced and $v \leq 2(\lceil \frac{p_\text{c}}{p_\text{p}} \rceil + 1) r_\text{p} p_\text{p} - r_\text{p} p_\text{p}$. Considering the buffering time T each operation may delay to be consumed by at most 2 periods of the consumer, hence $T \leq 2p_\text{c}$. This follows directly from Lemma A.1.

Otherwise $p_\text{p} > p_\text{c}$. By Lemma A.3, we conclude that the operations consumed by some time t, include all operations that have been produced up to period I_{i-3}, where i is such that $t \in I_i$ and I_i is the ith period of the producer. Therefore, in the worst case, the operations pending at time t, include the operations that can be produced during $I_{i-2} \cup I_{i-1} \cup I_i$ minus a subset of those operations that are guaranteed to be consumed by time t. This provides an upper bound on the size of the buffer needed since the buffer's role is to hold the operations that are pending at any time. Next, we calculate an upper bound on the data that can be produced during these three producer's periods. Let v be that data, then the buffer space needed is at most equal to v (i.e., $B \leq v$). During I_j for any j, the producer can produce at most $r_\text{p} p_\text{p}$ operations. The operations produced during $I_{i-1} \cup I_i$ may not be consumed by time t. Hence considering the period $I_{i-1} \cup I_i$, in the worst case, there will be $2r_\text{p} p_\text{p}$ operations produced and not yet consumed by time t. It remains to calculate the operations produced during I_{i-2} that will not be consumed by time t. As we will show next, there are at most $\max(0\,, (r_\text{p} p_\text{p} - \lfloor \frac{p_\text{p}}{p_\text{c}} \rfloor - 1) r_\text{c} p_\text{c})$ operations produced during I_{i-2} and not consumed at time t. This implies that the operations produced during $I_{i-2} \cup I_{i-1} \cup I_i$ and not yet consumed by time t are $v \leq 2r_\text{p} p_\text{p} + \max(0\,, r_\text{p} p_\text{p} - (\lfloor \frac{p_\text{p}}{p_\text{c}} \rfloor - 1) r_\text{c} p_\text{c})$.

Next, we show that there are at most $r_\text{p} p_\text{p} - \max(0\,, (\lfloor \frac{p_\text{p}}{p_\text{c}} \rfloor - 1) r_\text{c} p_\text{c})$ operations produced during I_{i-2} and not consumed at time t. By Lemma A.2, we know that all operations produced by the end of I_{i-3} will be consumed in consumer's periods that start in I_{i-2}. Those periods will end before the beginning of the first consumer period that starts in I_{i-1}, let's say at time t'. Hence, if there were some operations produced during I_{i-2} then those are guaranteed to be on top of the FIFO queue at time t'. There are at least $(\lfloor \frac{p_\text{p}}{p_\text{c}} \rfloor - 1)$ consumer periods that start at time t' and complete before time t. During each of these periods the consumer can consume up to $r_\text{c} p_\text{c}$ operations. Hence during time $[t', t]$, $(\lfloor \frac{p_\text{p}}{p_\text{c}} \rfloor - 1) r_\text{c} p_\text{c}$ operations can get consumed. If the operations produced during I_{i-2} are less than or equal $(\lfloor \frac{p_\text{p}}{p_\text{c}} \rfloor - 1) r_\text{c} p_\text{c}$, then all those operations will be consumed by time t', and hence, all operations produced during I_{i-2} (if any) will be consumed by time t. Otherwise, the operations produced during I_{i-2} are more than the operations that are guaranteed to be consumed by time t. In this case there will be some left over of at most $(r_\text{p} p_\text{p} - (\lfloor \frac{p_\text{p}}{p_\text{c}} \rfloor - 1) r_\text{c} p_\text{c})$, since the producer will produce at most $r_\text{p} p_\text{p}$ operations during I_{i-2}.

Considering the buffering time T each operation may delay be consumed by at most 3 periods of the producer, hence $T \leq 3p_\text{p}$. This follows directly from Lemma A.3. ■

B. Formal Proof for PRE–BUF building block

Theorem A.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 reservations.

Proof: We will prove the theorem by contradiction. Assume that there is a solution for which no buffering happens at the beginning. Let t be a finite time after which the consumer must meet its target reservations. We will construct an execution that renders this impossible.

For our execution, we assume that the periods of the consumer and the producer are the same. Let $p = p_\text{p} = p_\text{c}$. The producer produces rp operations during each period but not at its beginning. The consumer consumes those operations immediately once produced up to beginning of the first period that starts after time t. Let this be the ith period of the execution. At the beginning of period i, the cache is empty because everything produced is consumed immediately so far in the execution. Recall that a consumer is guaranteed to consume some operations only if those

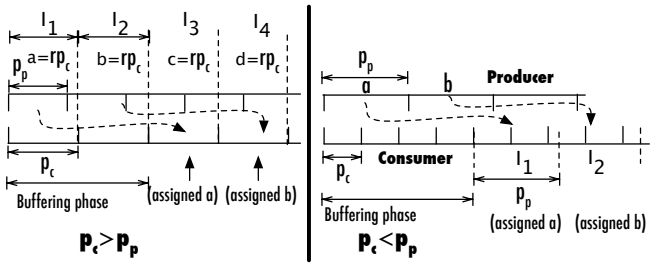


Figure 8: Pre–buffering: Analysis of PRE–BUF building block.

are available at the beginning of its period (assuming that those are less than rp). If operations become available for consumption within the period but after its beginning they may or may not be consumed during that period. Hence, we can construct the period i in such a way so that nothing gets consumed during period i, violating the target reservations for the consumer for period i. Since period i starts after time t we reached a contradiction to the fact that the solutions was guaranteeing the target after time t. ■

Theorem A.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:

Let's divide the producer's periods in time frames such that each time frame is defined by the starting and ending of the consumer's periods as illustrated in Figure 3(left). In particular, the ith time frame, I_i starts at the beginning of the ith consumer period and ends when this period ends, hence the length of I_i is equal to p_c (for all $i \geq 1$).

If the production of operations was happening uniformly, then everything produced in I_i would be available for consumption in the $(i+2)$nd period of the consumer for $i \geq 1$. Since the length of $I_i = p_c$, then during I_i there would be rp_c data produced which is exactly the data that is consumed within a consumer period. Hence, if the producer was producing uniformly operations then the operations produced during I_i would be consumed during the $(i+2)$nd period which in turn proves that the target reservation is met starting from the 3rd consumer period. Recall that during the first two consumer periods there is no consumption because of our choice of buffering phase. We will show that the fact that the data is not uniformly produced does not affect the statements above.

During each period of the producer there are rp_p operations produced. Let's split those operations so that exactly rp_c operations are assigned per each I_i interval in the following way: All operations produced during complete periods of the consumer that belong in I_i are assigned to I_i; if I_i contains incomplete periods of the producer, then we assign part of the operations produced during those periods based on the overlapping part as if the operations were produced uniformly during those periods. There are exactly rp_c operations assigned to each period I_i and there is no operation produced which is not assigned to some time frame.

We claim that the operations assigned to I_i are consumed during the $(i+2)$nd period of the consumer. This implies that the buffering time $T \leq 3p_c + p_p$ because the operations assigned to I_i are either produced during I_i or in some period of the producer that overlaps I_i. To prove the above statement, first, recall that the consumer can only consume rp_c operations per period, hence there will be an exact matching of the time frames I_i to the periods of the consumer. It remains to prove that the operations assigned to I_i are available for consumption at the beginning of the $(i+2)$nd consumer period. The operations assigned to I_i are either produced within I_i or they are produced within producer's periods which have some overlap with I_i. Since $p_c > p_p$, all producer periods from which we assign operations to I_i will finish within I_{i+1}. Hence, all operations assigned to I_i are produced before the beginning of I_{i+2}. Since the beginning of I_{i+2} is the same of the beginning of the $(i+2)$nd consumer period, the lemma follows.

Next, we will calculate the bound on the size of the buffer component. Let t be a time in the $(i+2)$nd consumer period. All operations assigned up to the beginning of I_i have been consumed by time t because they have been consumed by the $(i+1)$st consumer period. Hence, the operations pending at time t are bounded by the operations assigned to $I_i \cup I_{i+1}$ plus the operations produced during I_{i+2}. The operations assigned to $I_i \cup I_{i+1}$ are $2rp_c$. The operations produced during I_{i+2} are bounded by $(\lceil \frac{p_c}{p_p} \rceil + 1)rp_p$. In total, the pending operations at time t are bounded by $2rp_c + (\lceil \frac{p_c}{p_p} \rceil + 1)rp_p$ and this is the bound of the size of the buffer component.

■

Theorem A.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 $4rp_p + rp_c$ and buffering time $T \leq 4p_p + p_c$.

Proof:

By the definition of the buffering phase, the consumption starts from the beginning of the first consumer period after $p_c + p_p$ time. Starting at that point and on, we divide the execution of the consumer into intervals I_i each of length p_p for $i \geq 1$ as illustrated in Figure 3(right). We divide the operations produced into sets that get assigned to intervals I_i as follows: Interval I_i gets assigned exactly rp_p operations which have been produced during the ith producer period. We further divide those operations to assign exactly rp_c operations per consumer period after the buffering phase. This assignment happens by uniformly distributing the rp_p operations assigned to each interval I_i so that each consumer's period that overlaps I_i gets proportional amount of operations according to the length of its overlapping part

with I_i (i.e., if a consumer period overlapping length with I_i is x then it gets assigned exactly rx operations from interval I_i).

Next, we prove that all assigned operations to each of the consumer periods (after the buffering phase) are available for consumption at the beginning of each such period. Then, since there are exactly rp_c operations assigned to each consumer period, then during each consumer period rp_c operations get consumed, and the target reservations are met. Let's pick any period p of the consumer which starts after the buffering phase. First, assume that p belongs completely to an interval I_i. In that case, the operations assigned to p are operations produced during the ith producer period. By the choice of buffering phase, I_i starts at least p_c time after the end of the ith period period. Hence, the operations assigned to p are available for consumption before the start of I_i and hence at the beginning of p. Second, let p be a consumer period that is divided between two intervals I_i and I_{i+1}. In this case, the operations assigned to p are produced during both producer periods i and $(i+1)$. Similarly to above, since the ith period of the producer completes before I_i then those operations are available at the beginning of p. We will now prove that the $(i+1)$st producer period is also completed before p starts. This completes the proof because this implies that the operations assigned to p are available for consumption at the beginning of p. Let t be the time where I_i ends (and I_{i+1} begins). Because p is split between I_i and I_{i+1}, p is the last period which overlaps I_i. Since p is a consumer period with length p_c then p starts after time $t-p_c$. Because the length of the buffering phase is at least p_p+p_c and the length of the interval I_i is p_p, the producer period $(i+1)$ starts before or at $t-(p_p+p_c)$. Since the length of the producerś periods is p_p, then the producer period $(i+1)$ ends before or at time $t-p_c$. Hence, since p starts after time $t-p_c$ (as we showed above), p starts after the $(i+1)$st producer period is completed. This completes the proof that the target consumer reservations are met after the buffering time of p_c+p_p.

Next, we calculate an upper bound on the buffer size that allows the buffering phase of $\lceil\frac{p_c+p_p}{p_c}\rceil p_c$. Let time t be during the buffering phase. Then up to 3 producer periods can be accumulated for consumption at time t. This is because the buffering phase may overlap with at most 3 producer's periods. Let time t be within some interval I_i (i.e., after the buffering phase). Interval I_i overlaps with at most two producer periods whose operations may be pending. To accommodate those the buffer needs at most $2rp_p$ size. It remains to calculate the data that has been produced before those two producer periods which have not been consumed by time t. The operations assigned to consumer periods overlapping I_i are produced in the ith and possibly the $(i-1)$st producer period. Since there can be at most $\lceil\frac{p_c+p_p}{p_c}\rceil p_c$ distance between the beginning of I_i and the beginning of the ith producer period (by the buffering phase length), then there can be at most another 2 producer's periods between I_i and the ith producer period. The operations of those producer's periods may be pending at time t. To store those, we would require an additional buffer size of $2rp_p$. Finally, the operations of the $(i-1)$th period of the producer possibly assigned to a period of I_i are the only remaining operations that can be pending at time t as all other operations are already consumed before I_i starts. Next, we show that the operations produced during the $(i-1)$st producer's period that are assigned to a consumer's period in I_i are at most rp_c. This is because since a consumer period of I_i gets assigned operations from the $(i-1)$st producer period, that means that this consumer's period overlaps both I_{i-1} and I_i. The overlapping part of this period to I_{i-1} is of length at most p_c, hence since the assignment of data happens in a uniform distribution, then the data assigned to this overlapping part of this consumer's period is at most rp_c. In total the pending data at time t is then bounded by $4rp_p+rp_c$.

Finally, the buffering time T is at most $4p_p+p_c$ because each operation produced during period i will be consumed during a producer's period overlapping interval I_i as explained above and the separation of those two is at most $4p_p+p_c$.

■

Theorem A.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 vise versa, if we allow a buffering phase of length $max(p_c,p_p)$ and there is a buffer with buffering space $B\leq 2rmax(p_c,p_p)$ and buffering time $T\leq 2\max(p_p\,,p_c\,)$.

Proof: Let's divide the execution into intervals, I_i of length $p=max(p_c,p_p)$ each (for $i\geq 1$). During I_1 no consumption happens because of the buffering phase.

If $p=p_p$, then I_i is aligned to the producer's period. The rp operations produced during I_i are available for consumption at the beginning of I_{i+1} for all $i\geq 1$. Let $p_p=kp_c$ for some natural number k. During I_{i+1} there are k periods of the consumer each consuming rp_c operations. Therefore, during I_{i+1} there will be $krp_c=rp_p$ operations consumed if they are available at the beginning of I_{i+1}. Since those are produced during I_i, they are available at the beginning of I_{i+1}, hence the target reservation of the consumer is met.

If $p=p_c$ then I_i is aligned to the consumer's period. Let $p_c=kp_p$ for some natural number k. During I_{i+1} there are k periods of the producer each producing rp_p operations. Therefore during I_{i+1} there will be $krp_p=rp_c$ operations produced. The operations produced during I_i will be available for consumption at the beginning of I_{i+1} (which is a consumer's period) for $i\geq 1$. Therefore, at all consumer's periods with the exception of the first that is the buffering phase, the consumer will be consuming exactly rp_c operations, which is its target reservation.

At any given time t in I_i the operations pending are the ones produced during $I_{i-1}\cup I_i$. This is because the

operations produced during one interval will be consumed during the next. At each interval $max(p_c, p_p)$ operations will be produced. Hence, it suffices to have buffer of size $2r\max(p_c, p_p)$. Finally, the buffering time is at most $2\max(p_p, p_c)$ which is the length of two consecutive intervals as all operations of one interval will be consumed in the next. ■