Using PCI-Bus Systems in Real-Time Environments

Dissertation

to receive the academical degree "Doktoringenieur (Dr.-Ing.)"

submitted to

Technische Universität Dresden
Department of Computer Science
Institute for System Architecture

by

Dipl.-Inform. Sebastian Schönberg
born November 16, 1973

Dresden, June 2002

Thesis Supervisor: Prof. Dr. rer nat Hermann Härtig (TU Dresden)
Thesis Reviewer : Prof. Dr.-Ing. Gerhard Fettweis (TU Dresden)
Thesis Reviewer : Prof. Dr. Gernot Heiser (UNSW Sydney)

Acknowledgments

I would like to thank my supervisor, Prof. Hermann Härtig, for his continuous support and all members of the Operating Systems Chair at the Dresden University of Technology for the pleasant working environment. I would like to thank my reviewers, Prof. Gernot Heiser and Prof. Gerhard Fettweis for their support and extremely quick response time.

Special thanks for many helpful comments and discussions to Rich Uhlig and his team at the Intel Microprocessor Research Lab in Hillsboro (Oregon), where I spent two fantastic internships, especially Gil Neiger for proof reading and adding many helpful comments.

Furthermore, I would like to thank Claude-Joachim Hamann for his persistent help in getting my formulas right, Marc Shand from Compaq SRC for his support and the opportunity to get access to the Pamette Board, Jochen Liedtke, Volkmar Uhlig, Uwe Dannowski, Erik Cota-Robles, Shane Stephens, Michael Hohmuth and many others for helpful discussions and their valuable input.

Last but not least, I would like to thank my parents for all their support and love during this time.

Contents

List of Figures

List of Tables

1. Introduction

In classical hard real-time environments, buses have never been a problematic resource. To guarantee bandwidth and latency, either dedicated (bus) hardware with real-time support is used or the available bus bandwidth is higher than the bandwidth required by applications. More recently, an increasing number of applications processing a high volume of data such as audio and video have appeared. These modern applications would run well in classical real-time environments, however, they are targeted to run on standard commodity PC hardware. A problem arises since PC hardware is less predictable than dedicated hardware and does not provide support for bus-bandwidth reservation. The goal of this thesis is to explore and describe possibilities and limitations of the buses deployed in such systems and the impact on the processor caused by transfers of data over these buses.

In PC systems, processors and the main memory are tightly coupled via the *memory bus*, sometimes also referred to as *local bus*. To prevent the memory bus from becoming the critical performance bottleneck of the system, this bus should and usually does provide the highest bandwidth in the system. Memory-bus bandwidth is shared among all processors and devices accessing main memory. If the system does not provide mechanisms to assign a certain bandwidth to a device, the device's bandwidth always depends on the actual behavior of all other devices. Since this also holds for processors, one can conclude that the execution times of processor instructions accessing main memory also depend on the behavior of other devices. Hence, the execution time of an entire application can vary and is influenced by other devices.

Peripheral devices, such as adapter cards, are connected via the system's *I/O bus*. An example of such an I/O bus is the *Peripheral Component Interconnect* (PCI) bus [6]. This bus is widely used in consumer systems, as well as in workstations or low-end servers, and applied in a variety of platforms; among them we can find IA-32 (also referred to as x86), Itanium, Alpha and MIPS based computers. Since its introduction and definition in 1992, PCI has been an open industry standard, supervised by the PCI Special-Interest Group (PCISIG). The standard for the PCI architecture specifies all electronic and logical requirements but allows the designer of a system to adapt it to the requirements of the operating field.

To allow devices attached to the I/O bus to exchange data with main memory, a *bus adapter* is responsible for converting the signals on the I/O bus into signals on the memory bus. To connect the PCI bus with the memory bus, a *PCI host bridge* is used. In almost all systems, PCI host bridge and memory bus (controller) are integrated into one chipset and targeted to achieve high performance with a good price-performance ratio. As in many other cases, high performance is considered to be high long-term bandwidth and low average latency but does not imply bandwidth or latency guarantees. However, for real-time systems, bandwidth and latency guarantees are more important than peak and average values. Although the PCI bus specification allows free configurability of the bus system in terms of bus arbitration, very little work has been done on scheduling real-time streams over the PCI bus [7].

The PCI standard has been extended or enhanced to fill the gap between a universal application and a specialization for certain areas. CompactPCI is an extension to the PCI 2.2 standard to make better use of the PCI bus in embedded devices. It covers different form factors to suit rack-mounted devices and connectors better and provides support for additional application-defined side buses such as time-division-multiplex buses for telecommunication systems. The logical and electronic behavior of CompactPCI is identical to the standard PCI. To meet the growing requirements of bandwidth by faster I/O technologies such as Gigabit Ethernet, Fibre Channel, and Ultra3 SCSI, an addendum to the PCI 2.2 specification has been made, called PCI-X. This addendum provides bus bandwidth beyond 1GB/s, and extended information is passed along with each transaction to make host buffer management more efficient [8]. A smooth migration to the new technology is provided by backward compatibility to PCI. Devices can be designed to operate in PCI-X environments as well as in conventional PCI systems.

The availability of powerful commodity PCI-bus-based PC systems, the wide variety of PCI-bus devices, and the moderate price make such systems an interesting research platform. However, many performance-improving features such as caches make PC hardware less predictable than hardware especially designed for real-time systems. This fact requires new schemes to cope with the problems of this hardware.

This Ph.D. thesis is embedded in the context of the Dresden Real-time Operating System (Drops [3, 4]) that provides support for execution of real-time and non-real-time applications at the same time on a single machine. Major research goals of Drops are to discover the possibilities and limitations of commodity PC hardware and a systematic construction of a mixed real-time/time-sharing systems.

1.1. Motivation and Main Objective

A characteristic of all real-time systems is that tasks must be completed within a certain time or no later than a given point in time, called the *deadline*. Depending on the importance of the requirements in meeting the deadline, we distinguish between *hard* and *soft* real-time. Missing a deadline in a hard real-time system can have drastic impacts on the system's environment. In hard real-time systems, the task has no value if it is completed after the deadline. In contrast, a missed deadline in a soft real-time system results only in reduced value, and the system can gracefully adapt to the new situation. Often, these results are an unpleasant interaction with the user such as a flaky video or audio stream.

Both types of real-time systems require appropriate schemes for allocating and assigning resources. To decide whether enough resources are available for an application to deliver the expected results, an *admission test* must be performed. Since the successful admission must reserve and assign the necessary resources to the application, admission and resource allocation are often combined. Resource reservation is distinguished in a manner similar to real-time applications. If a resource with a *hard reservation* cannot be allocated, the application cannot proceed. However, a *soft reservation* can fail under some circumstance such as overload and the application must be able to cope with this situation.

As already mentioned in the introduction, the execution times of processor instructions accessing main memory depend on the behavior of all other devices. Early measurements showed that the impact on the execution time of applications is relevant and cannot be neglected when large amounts of data are transferred [5]. Hence, the naïve assumption that the execution time of an application is not influenced by other devices (as often seen in real-time systems) may result in violations of the given guarantees. To deal with that effect, resource reservations based on the execution time of an application must be adapted to handle the impact caused by bus traffic. However, this is only possible if the operating system handles bus bandwidth as another resource.

It is obvious that, for commodity hardware as used in DROPS, bus bandwidth is a key resource and we would expect to see a wide variety of hardware and software solutions to request, reserve, and assign bus bandwidth on such systems. However, existing real-time operating systems pay attention to various resource types, such as processors, main memory, hard disks, and the second-level cache, but disregard bus bandwidth as a resource. To our knowledge, currently no real-time operating system on such hardware (neither in the commercial area nor in the research community) supports any kind of bandwidth guarantees for buses or considers bus bandwidth as a resource that may influence other resources. Some support for buses is offered where these buses already support reservation. An example is the Universal Serial Bus (USB). Bandwidth is guaranteed (by the USB bus) between USB devices and the USB adapter card, but not between the USB adapter card and the main memory.

The purpose of this Ph.D. thesis is to analyze and predict the affect of load generated by the PCI bus on the execution times of applications, to determine the real-time capabilities of the PCI bus by analyzing the limitations of commercially available PCI-bus systems, and finally, to provide a mechanism to use the PCI bus in a predictable fashion with support for hard and soft reservations and bandwidth and latency guarantees.

1.2. Environment

This section gives a short overview of the hardware and software typically used in the Dresden Real-Time Operating System (DROPS) and describes the properties of the measurement environment.

1.2.1. Hardware

Figure 1.1 shows the structure of a typical PCI-bus-based commercial off-the-shelf system that we use for DROPS. Points with data contention are the memory bus (I) and the PCI bus (II). Both contention points are addressed in this thesis.

To determine whether different processor architectures also have an impact on the relation between I/O load and the execution time of an application, we chose two different platforms: systems based on Intel IA-32 (Pentium Pro, Pentium II) [10] and on DEC Alpha 21164 (EV5) [11] based systems. While Intel IA-32 is a 32-bit CISC instruction-set architecture, Alpha is a 64-bit RISC load–store architecture. For non-intrusive performance measurements, both architectures provide software accessible hardware performance counters to count various hardware events such as TLB-misses, cache misses, retired instructions, and so on. For time measurements, they also provide a time-stamp counter.

Initial tests and measurements showed that IA-32 and Alpha systems behave similarly [5]. Hence, we focused on IA-32 systems for later tests. Comparisons of hardware specifications and experiments indicate that all results and solutions can be applied to both systems.

To generate different PCI-bus loads, we used an adapter card with a programmable PCI interface chip, 2MB memory, and an Intel i960 processor (FORE PCA200e ATM card [12, 13, 14]). PCI bursts of variable length and frequency were created by our own firmware,

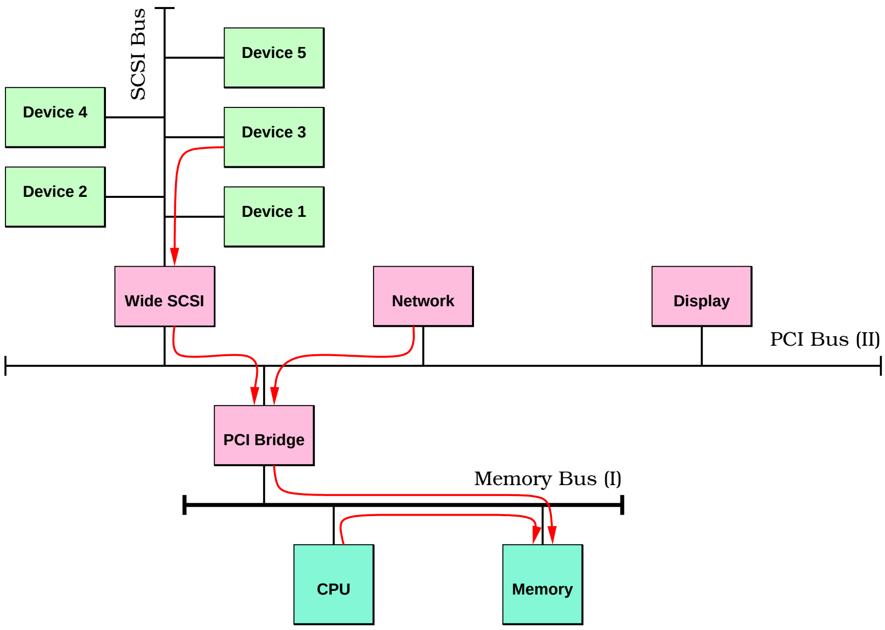


Figure 1.1.: PC hardware architecture used in DROPS

downloaded onto the card using the DROPS framework [15]. To study the impact of a highly utilized PCI bus, we performed measurements with up to five cards in parallel.

For additional PCI-bus operations, we used a *DEC Pamette board* [16, 17]. This board contains five freely programmable gate array (FPGA) chips. The complete PCI-bus access is controlled by one preinitialized FPGA. The other four FPGA chips are ordered in a two-by-two matrix and provide access to on-board memory (32kB SRAM and 16MB DRAM). An important feature is the Pamette's extended *DMA machine* which can generate PCI bursts of variable length. The Pamette was also used to capture continuous PCI-bus traffic to validate our model with real PCI-bus traffic.

All measurements on the IA-32 platform were performed on a single system with a 400MHz Pentium II CPU, 64MB SDRAM100 main memory (10ns), and 256kB second-level cache. The system uses an Intel 440BX PCI host bridge. The Alpha platform is a 433MHz 21164 based system with 64MB EDO RAM, 2MB of L3 cache and a PCI bus with the 21171 core logic chipset[1]. All measurements either use the internal CPU timestamp or performance counters for accurate results with minimal perturbance to the system. The 21171 core logic chipset provides additional performance counters to count bus events such as the number of PCI transactions, target or initiator aborts, bus stalls, etc.

1.2.2. Dresden Real-Time Operating System

The Dresden Real-time Operating System is based on the L4 microkernel [2, 18] as processor abstraction layer. L4 is a microkernel of the second generation and is available on several platforms, such as IA-32, Alpha, MIPS, and StrongARM. The L4 kernel provides multiple address spaces, threads, a fast and flexible inter-process communication (IPC), and support for

[1]The 21171 also contains the PCI host bridge.

user-level memory management. On top of the kernel, several *cooperating resource managers* and applications form the DROPS system architecture. A main feature of this framework is the support of time-sharing and real-time applications on a single system. Figure 1.2 illustrates the typical components of the DROPS system architecture.

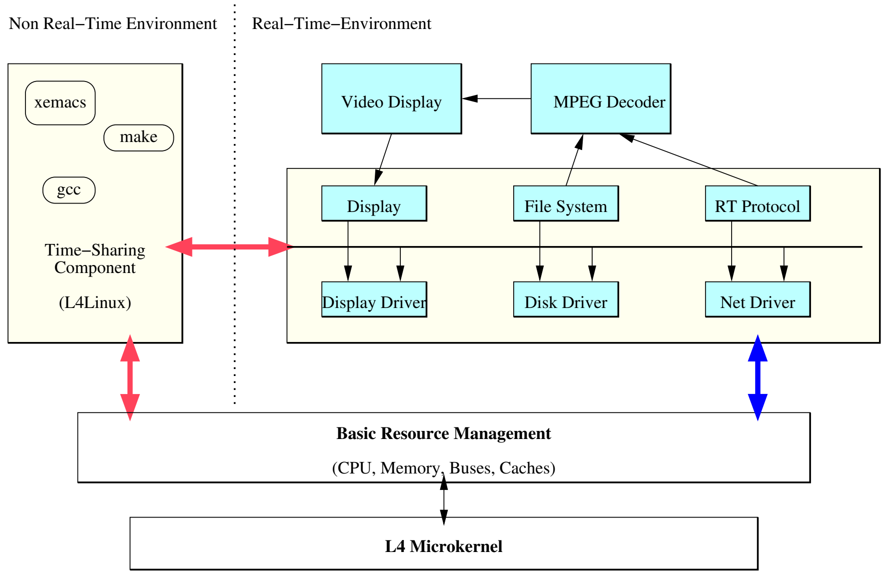


Figure 1.2.: DROPS system architecture with an MPEG player

Cooperating resource managers are the fundamental part of resource management in the DROPS system. They are responsible for handling, controlling, and distributing the hardware and system resources such as memory or L2 caches, and also for enforcing given guarantees. Managers for high-level resources map these resources to low-level resources. The general design principles for such managers are described elsewhere [19].

The time-sharing part of DROPS (L^4Linux [2]) receives unreserved resources or reserved but unused revocable resources. To guarantee basic functionality, it also reserves a certain amount of resources at startup. L^4Linux manages all received resources and handles the requests of its applications by its own policies. The standard Linux system-call interface does not need to be changed, and we can support the execution of native Linux applications without modifications. All real-time components have a separate management interface to reserve and request resources either directly from the resource managers or from other components.

Resource managers in the initial version of DROPS used deterministic reservation strategies with adaptation methods for overload. Reservations based on deterministic schemes require an explicit consideration of the worst case which leads to overreservation. It became apparent that especially modern hardware architectures exhibit a widening gap between normal and worst-case behavior and hence behave poorly with respect to absolute timing guarantees. To address these issues, we extended the deterministic admission schemes by a probabilistic approach [1]. Applications are now able to specify a value for importance of their tasks and a value for quality. By trading a known reduction of quality for a noticeable reduction of resources to be reserved, this scheme allows us to handle situations of temporary overload in soft real-time system far more effectively than deterministic reservation schemes. As a result, resources can be better utilized and a higher number of tasks admitted. Managers with support for probabilistic reservations exist for the processor and SCSI disks. An application that strongly benefits from this approach is our MPEG video player. Since reservations are not based on the worst case but on distributions of the actual resource usage, probabilistic schemes cannot be applied for resource management in hard real-time systems.

1.3. Contributions

This thesis makes several research contributions:

1. We introduce the *slowdown factor* to describe the influence of external bus load on applications executed by the main processor. Maximizing the external load leads to the application's *worst-case slowdown factor*. If we choose an application that is most sensitive to external load, we can determine the *upper-bound worst-case slowdown factor* (Section 3.1).

2. The slowdown factor of an application can be determined based on the instruction mix and the upper-bound worst-case slowdown factor (Section 3.3.1).

3. By means of a quadratic function, the impact of arbitrary load on processor read or write operations can be approximated. We derive a set of formulas to determine the slowdown factor from the instruction mix of an application and the current external bus load (Section 3.3.2).

4. To describe the behavior of devices in an off-the-shelf PCI system, we derived a set of formulas to determine worst-case latency and bandwidth. The formulas are based on a simplified model of the phases of PCI-bus transactions (Section 4.3).

5. We develop a fast proportional-share algorithm to assign arbitrary shares of the PCI bus bandwidth to individual devices (Section 4.5.1). This allows us to use the PCI bus in a predictable fashion and to reserve bandwidth for individual devices (Section 4.5.2).

6. We design a hardware implementation of the fast proportional-share algorithm. By parallelizing the algorithm, we can reduce its complexity for n devices from $O(n)$ to $O(1)$ (Section 4.5.5).

7. An extension to the design supports both hard and soft reservation of PCI-bus bandwidth (Section 4.5.6).

8. We give a set of equations to calculate the shares of each device in relation to the bandwidth the device requests and estimate worst-case latencies (Section 4.5.7).

The following, additional contributions were made:

1. To use the Digital Pamette PCI board under Linux, we developed and implemented a device driver for Linux.

2. To model various bus arbitration schemes and to verify our proportional-share PCI-bus arbiter, we developed and implemented a PCI-bus simulator "PCItrace."

1.4. Synopsis

The thesis is structured as follows. The next chapter describes the most important standard bus systems and their use in real-time systems, and gives an overview of related work. It analyzes scheduling algorithms that are suitable for resource scheduling and considers deterministic and probabilistic schemes for admission control. Chapter 3 describes the impact on applications running on the host processor due to memory-bus load generated by external sources, *e.g.*, devices on the PCI bus. Chapter 4 analyses the PCI bus system. We present a model to describe the bus characteristics and behavior of PCI devices. A new arbitration algorithm is presented to use the standard PCI bus in a predictable manner. Finally, Chapter 5 summarizes the results, gives an outlook for future research, and concludes this thesis.

2. Current Bus Systems and Related Work

In the past, the need for scheduling of buses has been handled in various ways. In real-time operating systems for embedded environments, designed to control dedicated hardware such as actuators and sensors, the amount of data handled and the required bandwidth is relatively low. Only a few control messages and some measurement data have to be exchanged between peripheral devices and the main unit. In this case, it is sufficient to use a bus system with support for a guaranteed transmission latency; bandwidth scheduling is not an issue here. Systems with a demand for higher bandwidth such as aerospace and avionics control, robotics, medical image processing, or video-on-demand servers use bus systems with explicit real-time support such as the VMEbus.

In the recent past, off-the-shelf PC systems for the consumer market—in general with the PCI bus— have become so powerful that "real-time processing" is an often-heard buzzword. Many applications, especially in the scope of audio and video processing, are available. However, such applications often only claim real-time capabilities but do not even run on a real-time operating system. If real-time systems are used, bus bandwidth is completely disregarded as a resource. One reason for this is the assumption that, merely by using powerful hardware, bus bandwidth is always available in the requested amount and is limited only by the hardware-given maximum bandwidth, and that there is never any shortage of this resource. Another reason why bus bandwidth is not seen as an "operating-system manageable" resource is the large difference in the timing base. While buses operate in ranges of nano- to microseconds, the time base of CPU scheduling is in the range of tens to hundreds of milliseconds.

To our knowledge, no real-time operating system for standard hardware, neither in the research community nor in the commercial arena, deals with bus bandwidth as a resource to be managed. Some support is given for external bus systems that provide bandwidth guarantees. However, support is restricted to the directly provided capabilities of the hardware. For example, drivers for the Universal Serial Bus (USB) support bandwidth guarantees for data transfer between USB devices and the USB host adapter (since this is supported by the physical USB hardware), but do not guaranteed that enough bandwidth is also available and reserved on the system's local bus to finally transfer the data into main memory. Small operating systems on predictable hardware adapter cards handle bandwidth to guarantee given commitments for this hardware, but not to manage the local buses of the card. Related work in terms of bus scheduling, especially for the common bus systems considered in this work, is very rare [20, 21].

The remainder of this chapter is structured as follows. In Section 2.1, we describe a variety of alternative bus systems such as Industry Standard Architecture, VersaModule Eurocard Bus, FutureBus and FutureBus+, InfiniBand, Controller Area Network, the Cambridge Desk Area Network, Universal Serial Bus, IEEE-1394 or Firewire, and Accelerated Graphics Port to connect internal and external devices to the CPU and the main memory. We give information about general resource management and resource-scheduling algorithms in Sections 2.2 and 2.3, focusing on algorithms that are potentially adaptable to bus scheduling. In Section 2.4, we describe some bus systems and scheduling schemes used in hard real-time environments. Section 2.5 describes several techniques for admission, including *static admission* schemes, where all decisions are done before the system is started, and *dynamic admission* schemes, which allow us to dynamically add and remove tasks. Dynamic admission schemes include deterministic and probabilistic admission schemes. Section 2.6 describes mechanisms to estimate the execution time of applications and how other systems handle the impact of I/O load on the execution time of applications.

2.1. Bus Systems

In this section we give a short overview of various bus systems and their behavior regarding arbitration and timing constraints. These buses are listed in this section because they are either used in real-time systems, provide capabilities for bandwidth reservation, or have influenced design decisions of the PCI bus. Table 2.1 summarizes the bus width, clock frequency, and maximum bandwidth of common bus systems.

Table 2.1.: Parameter comparison of common bus systems

Bus Name	**Width (bits)**	**Frequency (MHz)**	**Max. Bandwidth (MB/s)**
8-bit ISA	8	8.3	8.3
16-bit ISA	16	8.3	16.6
EISA	32	8.3	33.2
VLB	32	33	132
32-bit PCI	32	33	132
64-bit PCI 2.1	64	66	528
AGP	32	66	264
AGP (x2 mode)	32	66x2	528
AGP (x4 mode)	32	66x4	1,056
USB 1	1 (serial)	N/A	0.1875 (sync) 1.5 (async)
USB 2	1 (serial)	N/A	60
FireWire / IEEE-1394	1 (serial)	N/A	400
FutureBus	32	100	400
FutureBus+	256	100	3,200
VMEbus	64	10 (100ns cycle time)	80
CAN bus	1 (serial)	variable	length $> 100m : \frac{7.5}{\text{bus length}(m)}$

2.1.1. Industry Standard Architecture Bus

The *Industry Standard Architecture* (ISA) bus is the oldest bus system for IBM PC systems. Originally, it was an 8-bit data bus with 16-bit addresses. Extensions of the data path to 16-bit (and partially to 32-bit) were made. The maximum transfer rate of the original version is 8.3MB/s at a clock speed of 8.3MHz. This provides reasonable throughput for devices with low bandwidth requirements. All transactions are actively controlled either by the CPU or a single DMA controller. The CPU and the DMA controller synchronize their bus accesses using a direct wire. All devices are "passive," that is, they cannot actively request the bus and transfer data to main memory autonomously. Hence, an arbitration process is not required.

However, devices can be programmed to announce available data to the DMA controller which then performs the transfer without further CPU interactions. Since the DMA controller is only programmed by the CPU, no uncontrolled data transfer between a device and the main memory can occur. This makes the bus very predictable but does not provide the performance required for multimedia applications. In current systems, the ISA bus is usually connected via a PCI-to-ISA bridge and is present only to support legacy devices. The ISA bus will be removed from systems at some point in the future.

2.1.2. VersaModule Eurocard Bus

The *VersaModule Eurocard* (VME) bus is an asynchronous, master-slave, parallel bus. The original bus [22] is 32 bits wide and supports transfer rates up to 40MB/s, current systems are 64 bits wide and provide a bandwidth of 80MB/s. Recent versions (2eSST) of VMEbus achieve a sustained backplane performance of 1GB/s. VMEbus is mainly used in embedded systems, such as factory automation, avionics, in-flight video servers, cellular-phone base stations, and many others.

Every VMEbus must have a *system controller*, which performs the arbitration. The VMEbus system controller must reside in slot one and provides its functionality independent of the card plugged into that slot. Hence, a master or a slave may coexist with the system controller in the same slot.

Three arbitration modes are available: *priority mode* (PRI), *round robin* (RR), and *single level* (SGL). Selecting the arbitration mode is part of the system initialization process; the arbitration mode cannot be changed while the system is runing. Priority mode provides four bus request levels (BRL) and gives highest priority to bus request level three, then two, one and finally zero. For devices at the same request level, proximity to slot one determines which device gets the bus next. Round-robin mode grants the bus in sequential order, *i.e.*, BRL 3, 2, 1, 0, 3, 2, 1, etc. Single-level arbitration honors only BRL 3, other levels are ignored. A mix of PRI and RR mode is also allowed by the VMEbus specification. Here, BRL 3 works in PRI mode, BRL 2, 1, and 0 in RR mode. Alternatively, the bus allows using PRI mode for BRL 3 and 2, but RR mode for BRL 1 and 0.

While PRI mode is suitable for systems or devices with real-time requirements, RR mode is designed to support applications where fair sharing of the bus is required. SGL mode is provided for simple systems that do not require a sophisticated arbitration scheme.

The basic support for priorities makes VMEbus especially suitable for real-time environments. Since the main target are embedded systems with a relatively static design, priorities can easily be calculated up front (*e.g.*, by using RMS [23]).

2.1.3. FutureBus / FutureBus+

FutureBus+ [24, 25] is an asynchronous high-performance bus and an improvement of the old FutureBus. It is architecture and processor independent. The current standard supports bus widths up to 256 bits. The bus supports both centralized and decentralized arbitration. Each module has a unique 8-bit arbitration number. The module that supplies the highest number is granted bus access by the arbitration logic. Data transfer is burst-oriented—after sending a target address, the entire data burst is sent. Currently, FutureBus and FutureBus+ are rarely used, but they have strongly influenced the development of current state-of-the-art bus systems.

An approach for real-time traffic over FutureBus+ based on rate-monotonic theory is given by Sha *et al.* [26]. This is also the recommended approach in the FutureBus+ System Configuration Manual (IEEE 896.3) [27].

2.1.4. InfiniBand Architecture

The *InfiniBand Architecture* [28] specifies a "first-order interconnect technology," *i.e.*, a system for connecting processors and I/O nodes to build a system area network. The architecture is independent of the host operating system and the processor platform. The InfiniBand architecture is based on a point-to-point, switched I/O fabric. All end-node devices are connected via switch devices. InfiniBand is designed to become the backbone for server systems with support for bandwidth and latency guarantees. Similar to ATM networks, a certain amount of bandwidth is reserved per connection, and the switch fabrics are responsible for guaranteeing and enforcing these reservations. These features make InfiniBand an interesting alternative for future real-time systems. At this time, information is very preliminary and not publicly available.

2.1.5. Controller Area Network Bus

The *Controller Area Network* [29] is a serial communication protocol that supports distributed real-time control systems. Originally designed for connecting components in automotive systems, it has applications in many industrial automation and process-control systems because it is a simple and low-cost system.

The CAN bus allows real-time and non-real-time applications to share the network. The bus-access protocol is based on the carrier sense multiple access / non-destructive bitwise-arbitration scheme (CSMA/NBA). After sending a preamble to indicate the start of a transmission, a priority identifier is sent. Each node continuously senses the bus and checks its state. Each node compares the scanned state on the bus with the state it has put on the bus itself. In case of a mismatch, the device drops out of connection. Finally, the sender transmitting the highest identifier hereafter gains exclusive access to the bus. To avoid performance degradation, this mechanism requires an efficient control mechanism to provide guaranteed access for real-time applications.

Bus scheduling based on earliest-deadline-first (EDF) techniques has been described elsewhere [30, 31]. Compared to static-priority scheduling such as rate-monotonic scheduling or deadline-monotonic scheduling, the proposed approach allows an increase of up to 20% in the feasible network workload. An approach that combines both EDF and fixed-priority scheduling is given by Zuberi and Shin [32].

2.1.6. Cambridge Desk Area Network

The *Cambridge Desk Area Network* (DAN) [33] uses ATM techniques to build a multiprocessor multimedia station around an ATM switch. Various intelligent autonomous components (also called nodes) such as CPU, frame grabber, network cards or independent memory/storage nodes are connected via the DAN. Due to its switching technology, this system scales very well by connecting multiple DANs with each other. The computing nodes, input/output nodes and even the integrated parts of a typical desktop station can be separated. Since it also serves as a local area network, personalized I/O nodes can travel along with the user while the computation nodes always stay connected and online.

By design, the cell-based technology of ATM supports bandwidth reservation; hence additional technologies are not required. Once a "channel" between source and destination has been established under a given quality-of-service parameter set, the switches enforce this service quality.

2.1.7. Universal Serial Bus

Universal Serial Bus (USB) [34] is a serial bus designed to overcome known problems of low- to mid-speed buses, such as inflexibility, limited scalability and extensibility, or missing plug-and-play capabilities. With USB-1, a well-defined interface for hardware and software interconnection allows an easy integration of new devices into current systems. The first version supports bandwidths up to 12Mbps for asynchronous or 1.5Mbps for synchronous data transmission. The increasingly higher performance and capability to process huge data streams in end systems also results in a need for higher speed and bandwidth in the interconnect and bus systems. This is the motivation for USB-2 [35], which now supports transfer rates of up to 480Mbps. USB devices are distinguished based on their bandwidth consumption in *low-speed* (10–100kbps), *full-speed* (500kbps–10Mbps), and *high-speed* (25-480Mbps) devices.

All devices are connected point-to-point to a hub device. The hub device itself can be connected to another hub, building a hierarchy as illustrated in Figure 2.1.

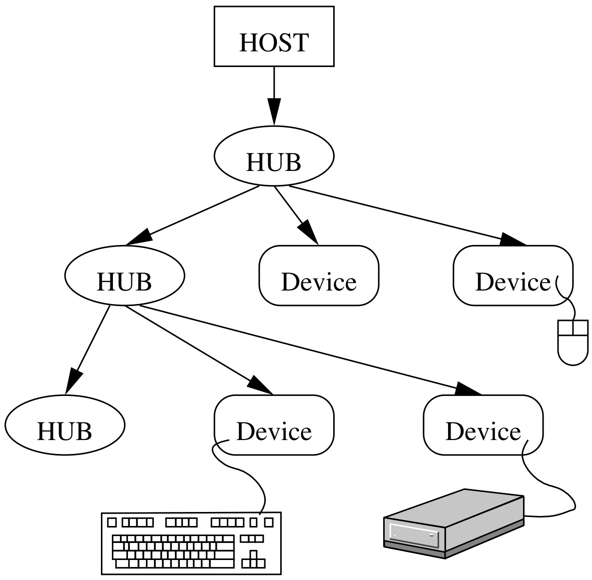


Figure 2.1.: USB device hierarchy

All transfers are based on *pipes*, a point-to-point association between two endpoints, the USB host and the USB device. Each pipe holds information about the transfer type and the requirements of the stream. To support various requirements of devices, USB provides the following transfer types: control transfer, interrupt-data transfer, bulk-data transfer, and isochronous-data transfer. Control transfers are mainly used to configure a device at the time it is attached to the bus but can also be used for other device-specific purposes. The bulk-data transfers have a wide dynamic latitude in transmission constraints and are generated or consumed by devices in relatively large and bursty quantities. Interrupt-data transfers are used for timely critical (low latency) and reliable delivery of data, such as keyboard or mouse events. Isochronous data transfers are targeted to transfer real-time data with constant bit rate. They occupy a prenegotiated amount of USB bandwidth with a prenegotiated delivery latency; bandwidth and latency are guaranteed by the bus.

Time on the bus is divided either into 1ms frames for full-speed or 125μs micro-frames for high-speed segments. Isochronous and interrupt endpoints are given opportunities to access the bus every N (micro)frames. Isochronous transmissions are guaranteed to occur within a prenegotiated frame but may occur anywhere within the frame.

2.1.8. IEEE-1394 Serial Bus / FireWire

The standard described in *IEEE-1394* and *IEEE-1394A*, also known as FireWire or iLink, is a high-performance serial bus targeting high-speed computer peripherals such as video cameras, audio devices, network interfaces, etc.; it also supports low-speed, non-time-critical devices such as printers and hand-held devices. The design and application area of IEEE-1394 is similar to that of USB.

Data rates up to 400Mbps are currently supported. Future plans to support data rates of 3,200Mbps would allow using IEEE-1394 buses as the common backplane within hosts. The main focus of the IEEE-1394 serial bus is an easy connection of external devices to a host. Similar to USB, easy hot-plug-and-play technology provides the ability to add or remove devices at any time.

Devices are addressed by a 64-bit ID that is partitioned into 10 bits for the network ID, 6 bits for the node ID, and 48 bits for memory addressing within each node. Hence, this topology can support up to 1023 networks or buses of 63 devices per bus (node). The address space each individual device can address is 281 terabytes. This direct addressing mode of IEEE-1394 is the major advantage over USB. Devices can be directly connected "back-to-back" without additional hardware. From a device's perspective, data connectivity is a true peer-to-peer connection since each device's memory is addressable from every point in the topology.

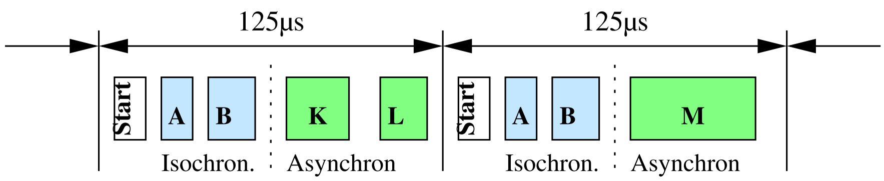


Figure 2.2.: IEEE-1394 bus cycle with isochronous and asynchronous data

In contrast to PCI, where only the electrical mechanism and timing constraints for arbitration are defined, IEEE-1394 also defines the arbitration scheme. In their technical construction, IEEE-1394 and USB are very similar. Time on the bus is divided into 125 μs frames. Figure 2.2 illustrates a typical cycle on the bus. To ensure bandwidth guarantees, up to 80 % of the total bandwidth can be reserved for isochronous (real-time) data streams. The rest is available for non-real-time data transfers, called asynchronous data. A variable-length gap between data packets is an essential part of the arbitration algorithm and indicates the maximum propagation delay. This is required since IEEE-1934 packets are not equally sized within a cycle and bus participants recognize a packet end by means of such a gap. Gaps between isochronous packets are shorter than those between asynchronous packets.

2.1.9. Accelerated Graphics Port

The *Accelerated Graphics Port* [36] (AGP) is a point-to-point connection between the graphics controller and the main memory. It is used exclusively to connect display adaptors and graphics devices; the primary intention was to improve graphics performance.

By providing a real bandwidth improvement between the graphics accelerator and main memory, some of the data structures used for 3D graphics may be moved to main memory. This effectively reduces the costs for accelerated graphics adaptors. Texture data are especially suited to a shift to main memory, since they are generally read-only and therefore do not have special access ordering or coherency problems.

AGP is, in general, based on and derived from the 66MHz PCI-bus specification but provides significant performance extensions such as deeply pipelined memory and demultiplexing of address and data on the bus. Since data transfers are possible at both falling and rising edge of a clock cycle, the transfer rate is twice as high as for the PCI bus.

Given its typical point-to-point connection topology, bandwidth guarantees exist automatically and no contention or conflicts on the AGP can arise. Contention is possible in the host bridge where AGP and the PCI bus are connected to the memory bus. From the bridge point of view, an AGP device has the same properties as an entire PCI-bus subsystem connected to the bridge.

2.2. Resource-Scheduling Schemes

In the following sections, we give an overview of scheduling schemes that can be applied to various types of system resources and that provide features to be considered for bus scheduling. At first, we describe classical schemes for static and dynamic scheduling. Thereafter, we describe schemes based on a proportional sharing of the resource.

2.2.1. Rate-Monotonic and Deadline-Monotonic Scheduling

One of the classical deterministic scheduling strategies is *rate-monotonic scheduling* (RMS) [23]. All tasks are periodic and have a determined runtime per period. RMS is based on hard priorities and tasks receive priorities inversely to the length of their period: the shorter the period the higher the priority.

Another well-known periodic scheme is *deadline-monotonic scheduling* (DMS). Priorities are assigned to tasks according to their relative deadlines: the shorter the relative deadline, the higher the priority. If the relative deadline of every task is proportional to the task's period, DMS is identical to RMS. If the relative deadlines are arbitrary, DMS can sometimes produce a feasible schedule while RMS fails. However, if DMS fails to produce a feasible schedule, RMS will also fail.

Rate-monotonic scheduling is well suited for scheduling tasks with small variations. However, tasks with a high variation in their resource requirements are difficult to handle. To guarantee the successful completion of all jobs, enough resources must be reserved to cover the worst case, independent how often the worst-case occurs. The result is an overreservation resulting in a waste of resources most of the time. This is mainly caused by the high variance between average and worst-case execution times of a resource. Rate-monotonic scheduling has been used for bus scheduling [7, 26]. In combination with bus scheduling, the missing support for non-periodic tasks (as required for non-real-time streams) is the major drawback of RMS and DMS.

2.2.2. Statistical Rate-Monotonic Scheduling

Statistical rate-monotonic scheduling (SRMS) [37] by Atlas and Bestavros is a probabilistically extended version of RMS. In terms of scheduling, SRMS is identical to RMS: hard priorities are assigned inversely proportional to the period length of a task. The basic requirements of RMS, such as periodic tasks with deterministic deadlines, persist, but the deterministic worst-case execution time of a task is replaced by a probabilistic execution time. This allows SRMS to better schedule periodic tasks with highly variable execution times and statistical quality-of-service requirements.

An RMS schedule is always successful if it can be calculated and the resource consumption of a task is less than or equal to the reserved amount. In contrast, SRMS guarantees only a percentage of tasks to be successful; other tasks may fail to meet their deadline.

In contrast to soft real-time, where, in case of overload, no prediction can be made of which resource allocation will fail, SRMS allows calculation of the percentage of tasks that will meet their deadline under the given available resources. However, this calculation cannot be inverted to calculate the amount of resources required to achieve a given percentage of successful tasks. This amount can be determined only by approximation schemes.

The major drawback of SRMS is that the importance of an application is not considered and the user cannot specify a preference between tasks with regard to deadline importance. If distributions of I/O load are known, SRMS provides an applicable scheme to improve bus utilization.

2.2.3. Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) scheduling assigns priorities dynamically to individual jobs in a task according to their absolute deadlines; the job whose deadline is next is executed first. It has been shown that successful execution can be guaranteed for a resource utilization of up to 100% [23]. The major drawback of EDF is the difficulty of predicting which tasks will miss their deadline during an overload. A job that has already missed its deadline has a higher priority than a job with a future deadline, hence, if the execution of a late job is permitted, it may cause other jobs to be late. This unstable behavior makes EDF unsuitable for systems in which overload can occur.

EDF has been proposed and successfully implemented for bus scheduling of the CAN bus [30, 31]. Access to the bus is priority-based and controlled by a CSMA/NBA scheme (see Section 2.1.5). The device with the highest priority gains access to the bus. Since deadlines can be kept locally by the device and no central arbiter needs to be updated, EDF can be efficiently implemented on the CAN bus.

On the PCI bus, the central arbitration scheme makes EDF unsuitable for scheduling. The major problem are the frequently required updates of deadlines in the arbiter. An alternative is using EDF with periodic deadlines where the arbiter can autonomously determine the next deadline of a device. However, this leads to problems similar to those of RMS and DMS, that tasks with high variation in their resource requirements are difficult to handle.

2.2.4. Stride Scheduling

Stride scheduling [38] is a proportional-share resource-scheduling scheme suitable for a wide diversity of resource types. Resource rights are represented by *tickets* that a client acquires when it requests access to a resource. The number of tickets each client receives is proportional to the amount of the resource given to the client. Consequently, the latency is inversely proportional to the number of tickets a client holds. The scheduling mechanism is based on a deterministic rate-based flow-control algorithm for networks. Stride scheduling is suitable for hard real-time systems.

Based on the client's share of the resource, the *stride* is calculated. The stride determines the time a client must wait at least between two successive allocations. Consequently, the client with the smallest stride will be scheduled most frequently. Stride scheduling is an approximation of an ideal system known as "fluid-flow." While resources in an ideal system can be allocated in arbitrarily small units, resources in a real system can be allocated only in multiples of the smallest allocation unit. This problem was investigated by Stoica *et al.* [39].

The difference between the resource share a client should receive and what it actually receives is called *service-time lag*. The service-time lag quantifies the allocation accuracy; the smaller the service-time lag, the higher the accuracy of the allocation. Hence, a goal is to minimize the service-time lag. This is done by tracking and accumulating the service-time lags of clients. At the end of each resource quantum, only those clients with a positive service-time lag are considered. A positive service-time lag means that the client is behind schedule and has received fewer resources compared to the perfect fluid-flow system. A negative service-time lag shows that the client has received more resources than it should. All clients that are behind schedule are eligible to receive resources and will be scheduled using an earliest-deadline-first rule. This allows scheduling of a set of clients such that their service-time lag is bounded by a constant. Bounding the service-time lag is especially important for real-time systems to avoid missing of deadlines due to inexact scheduling.

Since stride scheduling was designed to schedule arbitrary types of resources, it is also possible to apply this scheme to bus scheduling. The basic implementation does not consider the service-time lag. This can lead to an unacceptable high variation between calculated and assigned bandwidth. The modified version considers the service-time lag but requires complex operations. The scheduling of clients that are behind schedule using an earliest-deadline-first scheme makes this variant unsuitable for an implementation in hardware.

2.2.5. Charge-Based Proportional Scheduling

Charge-based proportional scheduling [40] by Maheshwari is a deterministic approach similar to stride scheduling (see Section 2.2.4). It assign resources to clients in a round-robin order. Each time a client receives a *quantum*, the scheduler deducts a charge from the client's account. During scheduling, every client with a negative account is skipped. When no client has a positive account, the scheduler refunds all clients their initial number of shares. The major drawback is the possibility of long starvation. This can happen when clients have a very different number of shares. Here, over time, fewer and fewer clients have a positive account. Finally, only one client receives all resources until this client has a negative account, and then each client receives its initial number of shares. Starvation leads to high latency for clients with a small share, which is not acceptable for bus scheduling. The starvation can be avoided by evenly distributing the scheduling events over the period. This distribution can be achieved by a scheme we propose in this work.

2.3. Resource-Reservation Policies

Scheduling algorithms are responsible for making *ad-hoc* decisions to grant access to a single resource based on actual requests. Classical resource schedulers attempt to assign a resource to applications such that given guarantees hold. Classical resource schedulers do not consider the human user of a system. By starting many applications, one user can monopolize resources and gain unfair advantage over other users. They also do not consider long-term results of the system. Allowing access to an available resource to satisfy one request can result in overall performance worse than denying the access. To solve this problem, long-term considerations of resources have to be made.

The following resource-reservation policies are additions on top of basic resource-scheduling schemes. They can be used to consider bus bandwidth in concert with other resources.

2.3.1. Fair-Share Scheduler

Fair-share schedulers [41, 42] attempt to solve user-based monopolization by allocating resources to a user or a group of users proportional to the share they have been assigned. The main goal is to achieve a fair share of the entire computer over a longer period of time, rather than fulfilling a short-time scheduling decision for one single resource. Hence, a main distinction between proportional-share-based schedulers and fair-share schedulers is the different time base. While proportional-share-based schedulers provide *instantaneous* information according to the shares of the application, fair-share schedulers attempt to influence the system to provide a timed-averaged fair share to the user.

Fair-share schedulers often deploy complex algorithms with a statistical analysis of the usage and utilization of a special resource or the entire machine. They are implemented on top of the conventional, priority-based scheduling system and adjust the priorities determined by the underlying priority scheduler to achieve long-term fairness.

2.3.2. Earliest-Eligible Virtual-Deadline First

Earliest-eligible virtual-deadline first (EEVDF) [43] unifies and extends the proportional share and resource-reservation policies. Instead of characterizing a client exclusively by its weight, as in proportional-share schemes, or by its share as in reservation schemes, this approach uses both characterizations simultaneously. Each client is associated with a tuple, containing the *weight value* and the *share value*. The tuple describes how much and at which rate a client has to "pay" for using a fraction of the full resource.

By setting the weight to a fixed value, the costs of using an unknown amount of the resource are predictable. At any time, the fraction of the resource may change depending on the level of competition for the resource. The client cannot predict how much of the resource it will actually receive. On the other hand, by fixing the share, the client receives a predictable share of the resource, but it cannot predict the costs.

2.3.3. Imprecise Computations

Imprecise computations (IC) [44] were introduced as a way to handle transient overload and enhance the fault tolerance of soft real-time systems. Unlike schemes that calculate priorities only from a task's share or execution time, IC also considers the *importance of an application.*

Imprecise computations handle periodic tasks that consist of one *mandatory part* and one *optional part.* While mandatory parts of all tasks are guaranteed to complete successfully, no statement is made about optional parts. Each application can thus generate, in the mandatory part, a guaranteed approximated basic result that can be used whenever failure or system overload prevents the application from refining this result and producing, in the optional part, a result with better quality.

The relative improvement often decreases with the calculation time of the task. If the improvement can be described depending on the calculation time by a *value-time-function*, the ratio between performance gain and execution time can be used to further improve the scheme. Many tasks can come closer to an optimal result rather than only a few actually reaching it.

In the original version, scheduling of mandatory parts is done using RMS. But all other deterministic scheduling schemes are possible. IC is tailored to the processor resource, but can also be used for other resources. In such cases, the mandatory part gives the share of a resource a client needs to deliver the minimal required result; the optional part improves this result.

2.3.4. Quality-Assuring Scheduling

SRMS and imprecise computations, both achieve an improvement in resource utilization over pure, deterministic scheduling and admission schemes. However, in the soft real-time environment of Drops, both have distinguished drawbacks. While SRMS cannot handle user-given importance of applications, imprecise computations lack the probabilistic part to handle variable execution times.

Our earlier work [1] presented an approach, part of the Drops system and called *quality-assuring scheduling* (QAS), that combines the advantages of both SRMS and IC. Similar to imprecise computations, an application is split into one mandatory and in one or more optional parts.

To admit the mandatory parts, we use their worst-case execution time and an RMS-based admission scheme. Resource reservation for mandatory parts is always guaranteed. Since we also want to assure a requested percentage of successful optional parts, we have to perform an admission test for these parts as well. Reservation for optional parts can be assured by the requested percentage.

Optional parts are started only after all mandatory parts have finished. It would be sufficient to determine the time remaining for optional parts based on the worst-case execution time of the mandatory parts. However, due to the high differences between worst-case and average-case execution time of the mandatory parts, the actual execution times of the mandatory parts is likely to be far less than their worst case used for their admission. Hence, we determine the time available to optional parts by the expected execution time of the mandatory parts. Optional parts are admitted based on their own expected execution time and on the expected execution time of all mandatory parts. The higher the difference between worst case and average case of the mandatory parts, the more time is available to optional parts and the more optional parts can be admitted.

QAS can be applied to various types of resources. Considering bus bandwidth as yet another resource to be managed in such a way, an appropriate bus-scheduling scheme must also support soft and hard reservations.

The concept of quality-assuring scheduling can also be used to reduce the influence of I/O load on applications. While mandatory parts of an application are executed, external load is restricted to a certain threshold thus limiting the impact on application. During the execution of optional parts, external load is not restricted.

2.4. Bus Scheduling

Task scheduling is an operating-system term, if multiple tasks are ready to run, that describes the calculation of scheduling parameters, such as priorities or deadlines, and the decision of which task runs next. When discussing buses, *arbitration* describes the selection of the device that gets access to the bus. It does not include the calculation of parameters to achieve a certain bandwidth or guarantee a worst-case latency. In the following, we use the term *bus scheduling* to describe a behavior identical to operating systems. It includes the determination of parameters as well as the bus arbitration.

The time to perform an arbitration is short, usually only one or a couple of bus cycles. Therefore, it is implemented in hardware and is often static and unmodifiable. From the hardware point of view, bus arbitration plays a very critical rule regarding performance and system stability; modifications to an existing system are expensive and should be avoided whenever possible. For general-purpose systems, simple and reliable schemes such as round robin or static priorities are used. If more complex arbitration schemes are required in embedded environments, *e.g.*, the first device must have the lowest latency, systems will be tailored to solve exactly this problem and are not generally applicable. These systems are often niche products and designed, developed, and deployed by the same commercial vendor; publication of the technical details of such systems is very rare, implying a very small number of related research publications.

2.4.1. Formal Analysis of Bus Scheduling

Lehoczky and Sha [45] evaluate several scheduling algorithms and address problems of hard real-time bus scheduling with fixed priorities. Their paper is one of the few that explicitly points out the similarities between bus and processor scheduling. Transfers and their lengths (*i.e.*, the times for which they need the bus) are similar to tasks and their execution times. Every transfer has a deadline by which it must be completed. Three important issues are described by which bus scheduling is distinguished from processor scheduling: preemption, priority-level granularity, and buffering.

For the PCI bus, preemption at an arbitrary time is not feasible, since abortion of a transaction is an expensive operation. However, since the maximum time a device can use the bus is limited, the bus can still be considered as a preemptible resource.

Some bus systems provide mechanisms to prioritize devices, transaction types, or individual transactions. For the device case, this can either be a static assignment based on the position on the bus or a dynamic assignment. Transaction types are groups such as control, data, or interrupt signalling transactions. The finest granularity possible involves schemes by which each transaction can be individually prioritized. The PCI-bus specification defines the format of transactions but not the arbitration scheme. Hence, assigning individual priorities to devices is possible but to transactions or transaction types is not.

While buffering large amounts of data is a standard technique for the processor, it is not done on bus systems. Low-latency transfers prohibit the use of extensive buffering. A trade-off between bandwidth and latency to improve bandwidth has been made by introducing small buffers in the PCI host bridge.

A set of formal scheduling models for several common system bus architectures were developed and described by Kettler *et al.* [21]. The paper analyzes various schemes such as fixed priority, round-robin and time-division-multiplexed algorithms. It is of special interest for all commodity PCI systems that use a round-robin-based arbitration algorithm.

2.4.2. Slot-Based Scheduling

Slot-based scheduling is used to control access to a shared medium. The time on the bus is divided into *slots* of identical and fixed length, which are assigned to individual devices. A device can access the bus only during the time of its slots, hence all bandwidth must be reserved to obtain an adequate amount of slots. The share of bandwidth a device receives is proportional to its number of slots, allowing us to determine bandwidth and latency precisely. Due to the fixed division into slots, bandwidth must be traded for latency or vice versa. For example, smaller slots mean lower latency, but also lower bandwidth. To support non-real-time data transfers, a device must also reserve a certain share of the bus explicitly for these transfers. This is the major drawback of slot-based schemes. If one device cannot fill its slots with non-real-time data, the slots cannot be used by other devices and remain unused. To overcome this drawback, techniques such as carrier-sense multiple access with collision detection (CSMA/CD) for non-real-time slots could be used.

Using the PCI bus with slot-based access control is possible, but doing so gives up the advantages of high bandwidth utilization and low latency of the access-based PCI bus. The slot-based version is still fully compliant with the PCI specification, and all adapter cards can be used without modifications. The latency timer on all devices must be set to the same value. This limits the access time of each device on the bus to the same value and determines the maximum length of the slots. If devices do not make use of the entire slot, bandwidth is wasted since bus requests of devices are only honored by the arbiter when a new slot starts. The slotted scheme automatically provides guaranteed bandwidth and latency.

2.4.3. Cell-Based Scheduling

Cell-based schemes are used in switched networks with point-to-point connections. Devices can access the medium at any time to send data cells, *i.e.*, packets of identical size. If a direct point-to-point connection does not exist between two devices, switches are responsible for forwarding the data cells. To guarantee bandwidth for constant-bit-rate (CBR) streams across switches, a reservation must be made on each switch along the path by which the stream is routed. This reservation assigns a fixed number of cells-per-time to the stream. In contrast to slot-based schemes, cell-based schemes automatically provide support for streams with variable bit rate (VBR). If no data from CBR streams is available, cells can be filled with data from VBR streams. Hence, all bandwidth that is not reserved or not used by CBR streams can be given to best-effort services. To ensure some progress of the VBR streams, some bandwidth is usually kept back for VBR streams. As with slot-based schemes, latency and bandwidth depend on the size of the cell.

A switched implementation of the PCI interface was proposed by Wilson [46]. A central switching fabric has multiple PCI-bus outlets, each for a single device. If the switching fabric provides also bandwidth reservations, this is an elegant scheme to support guaranteed bandwidth.

2.4.4. Enhanced PCI Bus to support Real-Time Streams

Scottis *et al.* [7] presented an arbitration scheme, called *enhanced peripheral interconnect bus* (EPCI). It is based on rate-monotonic scheduling for CBR real-time streams and VBR time-sharing streams over the PCI bus.

A modified arbitration logic is proposed in which hard priorities can be assigned to each pair of PCI-bus request/grant lines. Priorities can be modified at any time by the *schedule manager*, which is part of the operating system. Priorities are assigned according to the rate-monotonic scheduling algorithm and remain constant until an existing real-time stream is ended or a new one is created.

Applying rate-monotonic scheduling schemes to give bandwidth guarantees on buses was proposed earlier for the FutureBus by Sha *et al.* [26]. The proposed scheme has the same drawback common to all RMS-based schemes. It may be impossible to schedule despite the fact that the bus is not completely utilized. The performance of RMS for resources with a high variation is poor, hence this scheme cannot be efficiently applied to such streams. Additionally, Scottis *et al.* claim that the EPCI provides support for VBR streams. Since it is not possible to support multiple non-periodic tasks with unmodified RMS, it remains open how EPCI supports multiple VBR streams and how bandwidth is distributed among multiple VBR streams.

2.4.5. Memory-Bus Scheduling

Preliminary thoughts about memory-bus scheduling for multiprocessor environments were published by Liedtke *et al.* [20]. This paper analyzes why bandwidth generated by processors on the memory bus cannot be simply summed. In an example, it shows that two processors access the bus in regular patterns, the first for one out of three time units (33%), the second for one out of two time units (50%), do not generate the expected bandwidth consumption of 83%. Due to access conflicts, the bandwidth achieved is only 66%. If all processors access memory during a system-global Δ-*window* only once and also through dense sequences of memory operations, *i.e.*, through bursts, the problem of interleaving requests disappears and bandwidth can be summed. However, it remains open whether applications can be written such that memory accesses always occur in bursts and what hardware support is required.

The same problem exists on the PCI bus. However, an implementation of this idea would require modifications the PCI bus as well as to devices. Since this also implies a change of the specification, this scheme does not seem very practical.

2.5. Admission-Control Schemes

Admission control is part of a real-time operating system that accepts or denies new tasks. The result of an admission is a single *yes–no decision*, not a set of scheduling parameters. A task can be accepted only if (a) all other tasks that have already been accepted will still be able to deliver the guaranteed results, and (b) the new task receives all requested (*i.e.*, necessary) resources to deliver the desired functionality.

A major objective is achievement the highest possible utilization of all available resources under some user-defined optimization criterion. Hard real-time systems require a strict and exact resource reservation scheme based on worst-case assumptions to allow tasks to complete successfully under any circumstance. If average and worst-case resource usage is very different, only a fraction of the reserved resource will usually be used. This is a high cost for guaranteeing the requested quality. Reserved but currently unused resources cannot be used to fulfill requests of other reservations, since they must be available to the real-time task for immediate use at any time. This is acceptable for mission-critical, hard real-time systems, but not for soft real-time systems. In this case, a much higher utilization is traded for some "acceptable loss of quality."

The optimization criterion for admitting tasks depends on the usage of the system. For instance, in a commercial video-on-demand environment, where requests for new streams are queued until the system can admit them, an option for optimization might be to minimize this queue waiting time. Other options are to prefer shorter over longer videos, frequent customers over sporadic, or expensive over cheap videos. An admission for sequentially arriving tasks is simple and straightforward. Every new arriving task will be individually tested. As long as enough resources are available, the task will be instantaneously admitted. However, it is not possible to define a multi-task-spanning goal with this scheme. A solution is to base the admission decision on the properties of a group of several tasks. This increases the admission latency, but allows the system to better fulfill the optimization criterion.

Admission schemes can be divided into two general groups: static and dynamic. With a static admission scheme, all calculations are done before the system is started. Once started, no new tasks can be added to the system. Static admission schemes are mainly used for embedded systems with dedicated functionality. An example for a static system is a controller for antilock brakes in automobiles. In contrast, dynamic admission schemes allow addition and removal of tasks during runtime.

Both schemes can be extended by a probabilistic approach. Resource requirements are described by a distribution function, and resource allocation can be guaranteed with a certain probability. While deterministic schemes try to determine the worst-case resource consumption exactly, probabilistic admission schemes consider variations in resource demand and guarantee results with a certain probability.

In the following, we describe some of the most important admission schemes. Since Drops is targeted to support interactive systems, static admission schemes are not applicable and we focus only on dynamic admission schemes.

2.5.1. Deterministic Admission Control

Rate-monotonic scheduling defines how priorities must be assigned to periodic tasks. It also provides an admission test to check whether a set of tasks can be scheduled with RMS. The advantage of RMS over other, static schemes is the existence of a simple feasibility criterion. To perform an admission cycle, it is sufficient to check this criterion instead of trying to calculate and test a complete schedule.

With time-demand analysis, Lehocky *et al.* [47] provide a formula to calculate whether a task set of n tasks is schedulable. Let e_i be the worst-case execution time and t_i the period of task i. There must exist a t such that:

$$\forall i = 1...n \;\; \exists t \in R \qquad e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{t_i} \right\rceil e_k \leq t. \tag{2.1}$$

The utilization $U(n)$ of a resource with n task can be calculated by the following:

$$U(n) = \sum_{i=1}^{n} \frac{e_i}{t_i}. \tag{2.2}$$

A simplified but sufficient criterion is given by Liu and Layland [23]. A task set of n tasks is called *feasible*, if Equation 2.3 holds:

$$U(n) \leq n(\sqrt[n]{2} - 1). \tag{2.3}$$

This equation (2.3) can be further simplified for large numbers of tasks:

$$\lim_{n \to \infty} U(n) \leq \ln 2. \tag{2.4}$$

A task set is always schedulable if the utilization of the resource is less than $\ln 2$, or 0.693. This admission test also holds for small n, since Equation 2.4 approximates $\ln 2$ from the upper bound. In RMS-based real-time, the test for a utilization less then is the most often used criterion for task admission.

If the utilization is higher than the value given by Equation 2.3, time-demand analysis must be used to individually test the given task set. This test can fail even if the resource is not fully utilized, which is one of the major disadvantages of RMS.

2.5.2. Probabilistic Admission Control

Probabilistic admission and scheduling strategies are a class of solutions that attempt to reduce the amount of resources reserved based on worst-case assumptions. They were introduced to overcome the resource-utilization problems and poor performance that are typical for deterministic admission schemes. By extending conventional schemes with a probabilistic approach, systems are better able to deal with the increasingly high unpredictability of processing times.

Execution times of resources are not given as worst-case values, but instead as probability-distribution functions. Figure 2.3 is an example of such a distribution.

In this example, the worst-case execution time is 100ms, however about 96% of all executions complete within 78ms. Hence, if we make reservations based on an execution time of 78ms, we can calculate that about 4% of all tasks will miss their deadline. The required execution time can also be calculated based on the requested ratio of successful tasks. Using the quantile $Q(0.96)$ of the distribution function, we get an execution time of 78ms. This also illustrates that a probabilistic approach leads to a better resource utilization and reduces the probability of overbooking resources. Accepting 4% of missed deadlines results in a 22% lower resource reservation.

Statistical approaches require from applications and operating systems the ability to cope with missing data. For streaming audio or video applications that use an unreliable network as transport media, this is a typical case and these applications can handle this problem very well. This encourages us to apply probabilistic strategies to various types of resources. It makes no difference whether data is missed due to a network problem, a hard-disk request that missed its deadline, or unavailable buffer space in main memory.

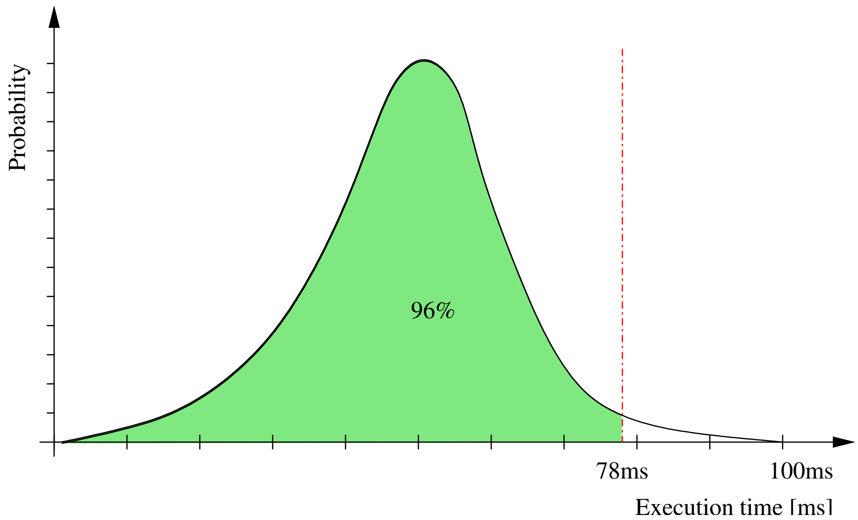


Figure 2.3.: Distribution of a resource's execution times

2.6. Applications and Input/Output Load

This section describes related work that influenced the direction of this thesis. Only two papers known to us describe the impact of input/output load (I/O-load) on the execution time of an application. However, they do not provide a general solution for dealing with this impact and do not exactly quantify it. The techniques described in Section 2.6.2 (Prediction of Execution Times) are used in Section 3.3 to derive an algorithmic approach to determine the impact of I/O load on the execution time of applications.

2.6.1. Impact on Application

Bellosa [48] discusses a simple approach for reducing the influence of time-sharing applications on real-time applications on different processors in a multiprocessor system. If the frequency of non-bandwidth-bound memory accesses of a time-sharing application exceeds a certain threshold, the application is throttled by performing additional no-operation cycles. This reduces the load on the memory bus caused by this application, leading to a smaller impact on the applications executed on other processors.

An analytical approach to describe the impact of I/O load generated by VMEbus devices on real-time applications is described by Huang *et al.* [49]. They observed that all programs are composed of three basic structures: straight-line, conditions, and loops. Each block of a program is individually analyzed with regard to how its execution time is influenced by I/O load. To bound the worst-case execution time of an entire program, the worst-case execution time of each basic block that may access I/O needs to be bounded. Since the bus controller uses a protocol based on the VMEbus specification, which supports hard priorities to regulate the bus contention between the processor and the devices, the model cannot be used without modifications on commodity PC systems.

2.6.2. Prediction of Execution Times

To compare performance of software, processors, or whole computer systems, various techniques such as benchmark programs are available [50, 51, 52]. Since benchmark programs are designed to cover a wide range of application types, the results of one benchmark program are used to compare different machines. However, exact prediction of execution times of individual programs is not possible.

Several approaches for estimating and predicting the performance of applications on the same or other processors have been published [53, 54]. Saveedra and Smith [55] describe an approach by which the performance of an application is predicted based on *abstract operations*. The execution time of an application is the linear combination of the number of times each abstract operation is executed (C_i), multiplied by the time it takes to execute each operation (P_i). The *instruction mix* (C) gives the amount of each individual abstract operation. In an extension [56], they also consider effects of the memory hierarchy, such as caches and translation look-aside buffers (TLB) on the execution time. The execution time (T) of an application using an instruction mix with n individual abstract operation can be determined by:

$$T = \sum_{i=1}^{n} C_i P_i. \tag{2.5}$$

This methodology allows one to analyze the behavior and to characterize individual machines. Applications can be analyzed, and their execution time can be predicted on the characterized machines. Prediction accuracy is very good; often the difference between real and predicted execution time is less than 10%.

To characterize applications regarding their bus-load affinity, we do not need to consider each individual operation. It is sufficient to group instructions into memory and non-memory operations.

2.7. Summary

In this chapter, we gave an overview of bus systems and their capabilities for real-time environments and analyzed schemes and techniques for real-time bus scheduling. We described some proportional-share algorithms, since they seem to be most applicable for a PCI-bus arbiter with support for bandwidth reservations. Additionally, we analyzed two fair-share resource-management schemes. These schemes are not necessary for basic real-time bus scheduling but would allow us to consider a fair sharing of bandwidth among users. We also gave an overview of deterministic and probabilistic admission schemes. For bus-bandwidth reservation in the Drops environment, a scheme supporting hard and soft reservations such as quality-assuring scheduling should be considered. Finally, we described schemes to compare and predict execution times or the impact of external load on applications. To our knowledge, a combination of both (prediction of an execution time under a certain external load) does not exist.

3. Impact of Bus Traffic on Applications

Any data exchanged between the processor and main memory uses the memory bus, sharing it with data exchanged between I/O devices and main memory. If the processor and a device try to transfer data at the same time, an impact can be seen on the processor as well as on the device. As a result, the execution time of an application on the processor may increase due to the memory-bus load generated by I/O devices. This chapter gives a method for describing and quantifying the impact of such load on applications executed by the processor.

Following the terminology of Pentium processor's performance-counter specification, the *internal load* of a specific processor refers to the memory-bus load generated by this processor, and the *external load* refers to the memory-bus load generated by all other possible sources. In the uniprocessor case, external load is always generated by the PCI host bridge. An *internal transaction* of a specific processor refers to a transaction on the memory bus generated by this processor, and an *external transaction* refers to a memory-bus transaction from any other possible sources, respectively. In the uniprocessor case all external transactions are generated by the PCI host bridge.

3.1. Slowdown Factor

Since external load reduces the capacity of the memory bus available to the processor, we can assume also that it increase the execution time of an application. In real-time systems, the correct amount of processor resource must be reserved to successfully execute an application. To consider the impact of external bus load on applications requiring resource reservations, we need a metric to express the strength of the impact. We call the ratio between the execution time under external load and the execution time under no external load the *slowdown factor*. The slowdown factor is always greater than or equal to one. The slowdown factor of an application A under external load L is denoted by $\mathcal{F}(A, L)$. Depending whether the application A or the load L remain fixed, the notation can be abbreviated by $\mathcal{F}_A(L)$ or $\mathcal{F}^L(A)$. The external load can be expressed by the bandwidth or the number of memory-bus transactions per time.

For reservation schemes that consider worst-case execution times—as for hard real-time systems—the worst-case impact is of interest. To determine this worst-case impact, all possible factors such as buffer and queue states in the host bridge, the access pattern of the physical RAM chips, and processor features such as the Pentium's system management mode[1] must be considered in addition to the application itself. A detailed analysis here is beyond the scope of this thesis. For simplicity, we consider increasing the external load to a maximum possible value as an acceptable approximation of the worst case: $\mathcal{F}(A, L \to \max) = \mathcal{F}^{L_{max}}(A)$. We call this value the *worst-case slowdown factor* of an application (WCSF).

[1] For instance, Pentium's system management mode (SMM) is used for the emulation of legacy devices or to work around bugs of the motherboard or in the chipset.

Alternatively to the worst-case load, we can identify an application that is most sensitive to external load (worst-case application). In this case, we denote $\mathcal{F}(A \to \max, L) = \mathcal{F}_{A_{max}}(L)$. The combination of worst-case load and worst-case application leads to an *upper-bound worst-case slowdown factor* for a machine. We describe this value by $\mathcal{F}(A \to \max, L \to \max) = \mathcal{F}^{L_{max}}_{A_{max}}$. To determine the confidence level of these worst-case approximations, analytical and statistical techniques such as those proposed by Edgar *et al.* [57] can be used.

The slowdown factor and the worst-case slowdown factor are metrics that describe the strength of the impact caused by external bus load. For example, an application A characterized by $\mathcal{F}^{L_{max}}(A) = 1.8$ takes, under worst-case load, 80 percent more time to complete. However, $\mathcal{F}(A, 20\text{MB/s}) = 1.2$ means that the application runs 20 percent longer under the given load of 20MB/s. The upper-bound worst-case slowdown factor gives a maximum value by which the most sensitive application can be influenced.

In the following sections, we describe three approaches for obtaining the different slowdown factors of an application.

3.2. Empirical Approach

The first approach to determining the WCSF of an application is purely empirical. In uniprocessor systems, the PCI bus is the only source for external load not generated by the processor(AGP cards can be considered as a PCI-bus extension). We obtain the WCSF by saturating the PCI bus and measuring the execution time of the application in contrast to the application's execution time under no external load. To generate such PCI-bus load, we used five identical bus-master-capable FORE PCA200e ATM network cards, each with a programmable PCI-bus interface.[2] The processor on the network card was used to initiate PCI-bus read or write transactions of variable burst length. We achieved a maximum write data transfer rate of up to 118MB/s, which is also given by the manual as the maximum value of the host bridge [58].

To obtain some representative results, we chose three different applications: The data encryption standard (DES), for an application with low internal load; a sorting algorithm (Quicksort), for an application with medium internal load; and raw-data transfers, for an application with high internal load. The classification into low, medium and high internal load is based on the number of CPU cycles required for precessing one 32-bit word as shown in Table 3.2. All applications were implemented to process data (in 32-bit words) from a 2MB source buffer and write the results to an adjacent 2MB target buffer. They were designed to touch source and target buffers exactly once per 32-bit word

We disabled any caching for the source and target-buffer region but not for any other memory area.[3] As a result, any data word of the source buffer must be individually fetched from main memory and any data word written to the target buffer is forced to main memory. This makes the access to the source and target buffer predictable. The same effect can be achieved by processing only one 32-bit word out of every cache line with a read-allocated, write-through cache. To determine the impact of external load in conjunction with caching,

[2] The ATM card uses a 50MHz embedded version of the Intel i960 processor, 2MB application memory, and a proprietary PCI-bus interface [13].

[3] Caching policies can be defined for "power of 2" multiples of 4KB memory frames ($2^{(2+n)}KB$) aligned to their size in the physical address space by use of the processor's memory-type range registers (MTRRs).

we performed the same tests with caching enabled for both source and target buffer.

The number of transactions on the memory bus was measured using the processor's internal performance counters. IA-32 processors provide performance counters for internally generated (t_{int}) and all (t_{tot}) memory-bus transactions but not for externally generated memory-bus transactions (t_{ext}). On a uniprocessor system, where only one processor and the host bridge can generate memory bus load, the number of external transaction is:

$$t_{ext} = t_{tot} - t_{int}. \tag{3.1}$$

Characterizing the type of external transaction leads to the distinction between external read transactions (t^r_{ext}) and external write transactions (t^w_{ext}).

DES Algorithm

DES [59] is a well-known symmetric cryptographic system. It uses a 56-bit key to encrypt and decrypt data. In sixteen iteration steps, the input data is permuted and written to the target buffer. The C-code implementation of this permutation uses local variables to store temporary results. Since IA-32 processors have a small number of available registers, these variables are stored on the stack frame of the procedure in memory. Recall that only source and target-buffer accesses are uncached; stack and local-data accesses are cached. Due to the large number of temporary local variables, DES has strong data locality, which leads to a high cache-hit rate. It is very likely that temporary results are held completely in the first- or second-level caches and are never actually written to main memory. We measured an $\mathcal{F}^{L_{max}} = 1.05$ for the encode and $\mathcal{F}^{L_{max}} = 1.03$ for the decode operation.

With caching enabled for the buffer, the application was about 26% faster and generated only 9% of the memory-bus load (430,000 versus 4,560,000 transactions per second). However, in relation to the 265 million external transactions generated by the PCI cards at the same time, the increase from 430,000 to 4,560,000 internal transactions (0.16% to 1.7% in relation to the number of total transactions) is negligible and the $\mathcal{F}^{L_{max}}$ values are identical.

Sorting algorithm (Quicksort)

Quicksort is a fast sorting algorithm based on divide-and-conquer. An array is repeatedly divided into two halves until two adjacent items can be directly compared against each other, and exchanged if necessary. We implemented an iterative version described by Sedgewick [60, 61] that sorts the 2MB source buffer into the 2MB target buffer. The measured worst-case slowdown factor without caching is $\mathcal{F}^{L_{max}} = 1.09$.

Raw-Data Transfer

In the two applications described above, memory accesses were only a small or medium fraction of all executed instructions. Under these circumstance, the WCSF values range between 3% and 10%.[4]

Figure 3.1 shows the maximally possible number of CPU-generated internal transactions in relation to the number of external transactions. Figure 3.2 shows the derived slowdown factors. Each graph in both diagrams represents a measurement with a different combination

[4]Earlier measurements [5] on an older machine (Pentium 90MHz) showed slightly higher slowdown factors. DES: $\mathcal{F}^{L_{max}}$=1.03; Quicksort: $\mathcal{F}^{L_{max}}$=1.1. Additionally, we measured decoding an "IPP-coded" MPEG-1 video (384x288 pixel, 24 bit color, compression factor 1:60 for 33 I-frames, 1:169 for 66 P-frames) with $\mathcal{F}^{L_{max}}$=1.36 and data transfer: read $\mathcal{F}^{L_{max}}$=2.15, write $\mathcal{F}^{L_{max}}$=1.18, copy $\mathcal{F}^{L_{max}}$=1.21.

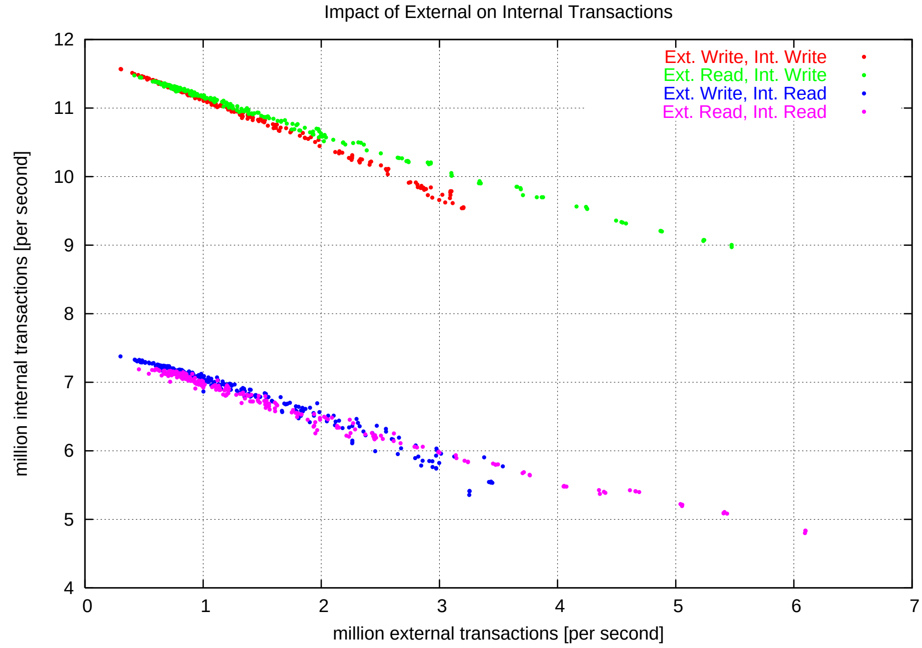


Figure 3.1.: Impact of external transactions on processor-generated (internal) memory transactions

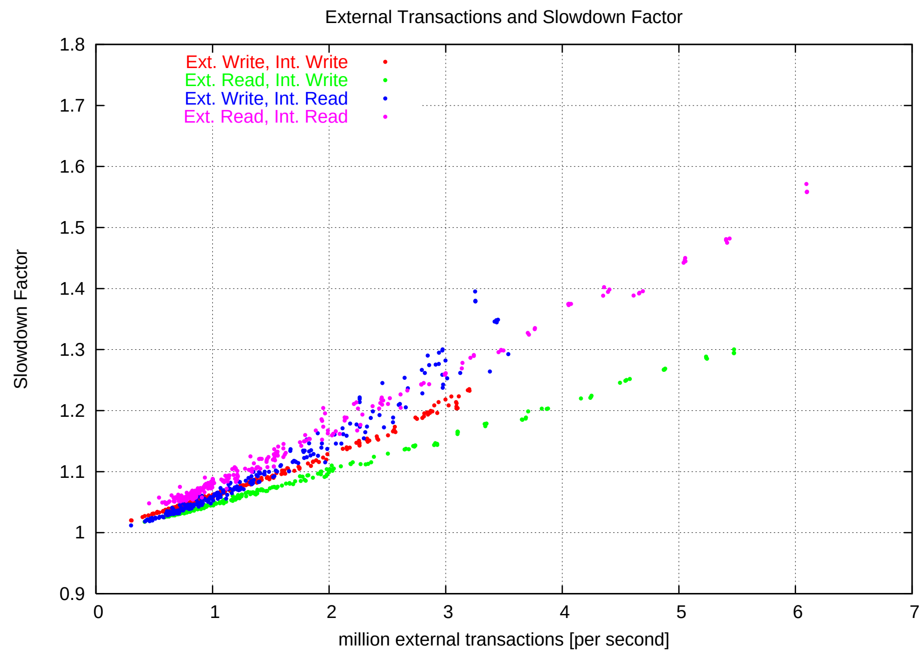


Figure 3.2.: Resulting slowdown factors, calculated from Figure 3.1

of external and internal transaction types, *e.g.*, a combination of read and write transactions.

For example, read–write means that the PCI-bus devices execute main-memory read operations while the processor executes main-memory write operations. Table 3.1 summarizes the worst-case slowdown factors.

Table 3.1.: WCSF for processor-issued memory operations, derived from Figures 3.1 and 3.2

$\mathcal{F}^{L_{max}}$	**CPU read**	**CPU write**
PCI read	1.49	1.26
PCI write	1.38	1.21

Summarizing Measurement Results

Table 3.2 gives a detailed overview of the number of processor cycles, the number of transactions on the memory bus to process one 32-bit word, and the resulting worst-case slowdown factors.

Table 3.2.: CPU cycles to process one 32-bit input word, bus transactions per 32-bit word

Operation	**CPU Cycles per 32-bit word**	**Bus transactions per 32-bit word**	$\mathcal{F}^{L_{max}}$
memory read	55.5	1	1.49
memory write	35.1	1	1.26
memory copy	99.4	2	1.35
Quicksort	457.7	2	1.09
DES encode	701.3	2	1.05
DES decode	699.2	2	1.03

Obviously, code executed on the host processor that performs only memory read operations to generate internal memory-bus load is the most sensitive to write operations as external bus load. We consider the slowdown of this application accessing as the upper-bound WCSF ($\mathcal{F}^{L_{max}}_{A_{max}}$) for this system. For our machine, we determined an upper-bound value of 1.49.

Knowledge of this value allows for a very simple test for admission that considers the impact of PCI-bus load. If an application can be scheduled even with the upper-bound WCSF, the application can be scheduled under any load possible on that machine. If it cannot be scheduled under the upper-bound WCSF but under no external load, the application-specific WCSF must be considered.

3.3. Algorithmic Approach

The advantage of an empirical approach is its simplicity—once the measurement environment has been built, only a simple test must be performed. However, the WCSF depends on many factors of the system used and individual measurements must be taken on every system.

For schemes based on the slowdown factor, not only one but a series of measurements under various external-load values must be taken. This makes the empirical approach expensive.

To overcome this disadvantage, we consider an algorithmic approach and strive to calculate the slowdown factor based on characteristic values of both the algorithm and the hardware.

3.3.1. Calculation of Application Worst-Case Slowdown Factor

In the previous sections, we showed that only memory operations are effected by external bus load, and we have described the maximally possible influence by the upper-bound WCSF on the execution time of applications. In this section, we strive to determine an application's worst-case slowdown factor as a combination of separate characteristics of the application and of the machine. The major advantage of such a separation is that an application can be described independent of the underlying machine. In combination with the characteristics for the machine, the machine-specific slowdown factor is determined. However, it will become clear later that we can achieve this separation only with certain limits.

From related work described in Section 2.6.2, we know that the execution time of an application is the linear combination of the number of times each abstract operation of the instruction mix is executed (C_i), multiplied by the time it takes to execute each operation (P_i). The execution time (T) of an application using an instruction mix with n individual abstract operation can be determined by the following:

$$T = \sum_{i=1}^{n} C_i P_i. \tag{3.2}$$

In terms of bus-induced impacts on applications, it is sufficient to describe an application by an instruction mix that divides the operations into three abstract operations: memory read-sensitive (C_r), memory write-sensitive (C_w) and non-memory-sensitive (C_o) operations. Given the total number of executed operations by $C_{tot} = C_w + C_r + C_o$, we can extend Equation 3.2 to:

$$\begin{aligned} T &= C_r P_r + C_w P_w + C_o P_o \\ &= C_{tot}\Big(\frac{C_r}{C_{tot}} P_r + \frac{C_w}{C_{tot}} P_w + \frac{C_o}{C_{tot}} P_o\Big). \end{aligned} \tag{3.3}$$

The time for an operation (P) is determined by the CPU frequency (f) and the number of CPU cycles this instruction takes (cyc). We can substitute P_x with $\frac{cyc_x}{f}$, and rewrite $\frac{C_x}{C_{tot}}$ as S_x, the share each instruction has on the amount of total instructions, and obtain:

$$T = \frac{C_{tot}}{f}\Big(S_r cyc_r + S_w cyc_w + S_o cyc_o\Big). \tag{3.4}$$

A coarse estimation of an application's WCSF is based on the assumption that read and write instructions are slowed down by the upper-bound WCSF ($\mathcal{F}_{A_{max}}^{L_{max}}$). The application under maximum external load is now executed in the time $T^{L_{max}}$:

$$T^{L_{max}} = \frac{C_{tot}}{f}\Big(S_r cyc_r \mathcal{F}_{A_{max}}^{L_{max}} + S_w cyc_w \mathcal{F}_{A_{max}}^{L_{max}} + S_o cyc_o\Big). \tag{3.5}$$

The WCSF for the application can be determined from Equations 3.4 and 3.5:

$$\mathcal{F}_A^{L_{max}} = \frac{T^{L_{max}}}{T} = \frac{S_r cyc_r \mathcal{F}_{A_{max}}^{L_{max}} + S_w cyc_w \mathcal{F}_{A_{max}}^{L_{max}} + S_o cyc_o}{S_r cyc_r + S_w cyc_w + S_o cyc_o}. \tag{3.6}$$

We can infer from Equation 3.6 that it is sufficient to describe the instruction mix as a triple IM that contains the shares of read, write, and all other operations. All other values are machine dependent and can be denoted by a machine descriptor MD:

$$\mathrm{IM} = (S_r, S_w, S_o) \qquad \mathrm{MD} = (cyc_r, cyc_w, cyc_o). \tag{3.7}$$

Ideally, the parameters of the machine descriptor depend only on the structure and capabilities of the processor. However, due to performance-improving features such as pipelining and parallel execution of instructions, which make the processor less predictable, these values can vary and are also influenced by the application.

In a more accurate formula, we consider the difference between read worst-case slowdown factor ($\mathcal{F}_r^{L_{max}}$) and write worst-case slowdown factor ($\mathcal{F}_w^{L_{max}}$). Both values can be taken from measurements of the system. For our measurement system, they are shown in Table 3.2. The modified formula can be derived from Equation 3.6:

$$\mathcal{F}_A^{L_{max}} = \frac{S_r cyc_r \mathcal{F}_r^{L_{max}} + S_w cyc_w \mathcal{F}_w^{L_{max}} + S_o cyc_o}{S_r cyc_r + S_w cyc_w + S_o cyc_o}. \tag{3.8}$$

To verify these results, we applied the measurement results of simple read and write operations ($cyc_r = 55.5, cyc_w = 35.1$) as given in Table 3.2 to calculate the worst-case slowdown factor of the memory-copy application. The source code in assembly language of the memory-copy application is shown in Figure 3.3. The integer operations of lines 2, and 4, and the branch operation of line 6 are paired by the processor with the instructions in lines 1, 3, and 5, respectively. Since the code allows pairing of all instructions, the cycle value of integer operations (cyc_o) is 0.5. To avoid influence by the number and order by which memory modules are plugged into the system, the code is written so that no two 32-bit words can be merged into one single 64-bit memory-bus transaction. The same holds for the read and the write test.

```
(1) l1: mov   eax, dword ptr [esi]
(2)     add   esi, 8
(3)     mov   dword ptr [edi], eax
(4)     add   edi, 8
(5)     dec   ecx
(6)     jnz   l1
```

Figure 3.3.: Machine code in assembly language for “copy” application

The shares for read and write are each 1/6, the share for other (non-memory-accessing) operations is 4/6. Applying these values to Equation 3.6 results in a calculated slowdown factor $\mathcal{F}^{L_{max}} = 1.392$. This compares favorably to the measured slowdown factor for the memory-copy application of 1.35.

We performed the same calculation for the DES application. In this case, instructions cannot be paired so nicely, since we have many data dependencies. We have measured an average value of 0.9 cycle for executing a non-memory operation. We have seen one memory read and one memory write bus request per 750 executed instructions. These values are identical for encode and decode operations. Applying the previously given equations results in the parameters given in 3.9.

$$\mathrm{IM} \;=\; \Big(\frac{1}{750}, \frac{1}{750}, 1 - \frac{2}{750}\Big) \qquad \mathrm{MD} = \Big(55.5, 35.1, 0.9\Big)$$

$$\mathcal{F}^{L_{max}} = 1.047. \tag{3.9}$$

This measured slowdown factor is $\mathcal{F}^{L_{max}} = 1.05$ for DES encode and $\mathcal{F}^{L_{max}} = 1.03$ for DES decode. These results show that the proposed solution is also applicable to relatively complex operations and not only to trivial operations such as memory copy.

3.3.2. Calculation of Application Slowdown Factor

All previous considerations were based on the worst-case slowdown factor for combinations of read and write operations, or in the worst case on the upper-bound WCSF of the system. As common to all worst-case-based reservation schemes, they also lead to an over-reservation and a waste of resources.

If reservations can be based on the real PCI-bus load, available resources can be utilized better. To determine the PCI-bus load in advance, all devices must announce their future bandwidth consumption at a central instance of the operating system. With cooperating resource managers, DROPS already provides an applicable scheme where PCI-bus bandwidth reservations can be made. Legacy device drivers, which cannot give exact information about the generated PCI-bus load, can give worst-case approximations based on information about the card. The PCI-bus interface chip used often provides information adequate to determine the maximum possible PCI-bus load of a card. Even more trivial, the type of the card can be considered. A 100Mbit network card is barely capable of generating more than 12.5MB/s of sustainable PCI-bus load. If the driver detects that the card is only connected to a 10Mbit network, it is sufficient to assume a maximum bandwidth of 1.25MB/s.

In the next step, we determine the slowdown factor in relation to a given bus load. All previous considerations were based on transactions. Hence, we have to convert a load given in MB/s into a load given in memory-bus transactions per second. If caching is enabled, the processor always reads and writes an entire cache line, instead of each individual memory cell and we see only one bus transaction per cache line. Hence, the number of CPU-generated memory-bus transactions does not depend only on the number of memory instructions, but also on the caching behavior of the application. The number of bus transactions caused by a certain PCI-bus load is easier to determine. The memory-bus interface of the host bridge has only a few read-buffer or write-buffer entries. Since all relevant transfers to or from main memory are bursts, the bridge combines multiple data words into one single memory-bus transaction. The result is an almost linear relation between PCI-bus bandwidth and the number of transactions on the memory bus.

The influence of external transactions on the maximal possible number of internal transactions on accessing memory has already been shown in Figure 3.1. We see one transaction on the memory bus for a 32-byte write access or a 16-byte read access. These two values are host-bridge dependent. In the test-bed with five PCI cards, the maximum achieved PCI-bus bandwidth is about 102MB/s (6.1M transactions) for read and 118MB/s (3.5M transactions) for write operations.

The calculation of the slowdown factor is based on Equation 3.6. We replace the upper-bound worst-case slowdown factor by the slowdown factors for read and write under a certain load. To determine $\mathcal{F}_r(t_{ext})$ and $\mathcal{F}_w(t_{ext})$ from the number of external transactions, we consider two methods. The simpler method to get the slowdown factor uses a lookup-table filled with slowdown-factor values taken from the tests described in the paragraph "Raw-Data Transfer." If an exact value is not available, an approximated value is used. A drawback is the required amount of data that must be available for the table.

A more elegant version approximates the slowdown factor by a monotonically increasing function. At a glance, one might expect a linear dependency between the slowdown factor of an application and the external bus load. However, analyzing Figure 3.2 shows that a polynomial function of order two (k=2) (*i.e.*, a quadratic function) approximates the relation more accurately. The reason for a non-linear dependency is that contention on the bus increases with the square of the bus utilization. If memory-bus speed is high and the length of each memory-bus transaction is short, contention reduces and the resulting almost linear relation can be described with a β_2 coefficient of zero.

The quadratic function of order two is given by $f(x) = \beta_2 x^2 + \beta_1 x + \beta_0$, with x as the number of external transactions per second. To calculate the actual value for $\mathcal{F}_A(t_{ext})$, we obtain:

$$\mathcal{F}_A(t_{ext}) = \beta_2 \Big(t_{ext}\Big)^2 + \beta_1 \Big(t_{ext}\Big) + \beta_0. \tag{3.10}$$

The β coefficients are machine dependent and must be determined for each machine individually. In order to find the best approximation, we consider the biased variance σ^2 of the pairs (x_i, y_i) of all measured samples:

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} \Big(y_i - (\beta_2 x_i^2 + \beta_1 x_i + \beta_0) \Big)^2 \tag{3.11}$$

must be minimized using the non-linear *least-squares-fitting method*. We can derive the Equation system 3.12 for the polynomial function of order two:[5]

$$\begin{aligned} \beta_0[x^0] + \beta_1[x^1] + \beta_2[x^2] &= [y] \\ \beta_0[x^1] + \beta_1[x^2] + \beta_2[x^3] &= [yx] \\ \beta_0[x^2] + \beta_1[x^3] + \beta_2[x^4] &= [yx^2]. \end{aligned} \tag{3.12}$$

Solving the Equation system 3.12 with our measured values leads to the tuples of coefficients as given in Table 3.3. Since the slowdown factor of an application running on a system without external bus load ($\mathcal{F}(0)$) is one, the β_0-coefficient must be 1 as well. This holds for all our calculated coefficients.

Table 3.3.: Coefficients for the polynomial function to calculate the slowdown factor for combinations of external and internal, read and write operations, derived by the least-square-fitting method from Figure 3.2

$(\beta_2, \beta_1, \beta_0)$	**CPU operation**	
	read	**write**
PCI read	(0.7345e-15, 88.191e-9, 1.004)	(0.9191e-15, 50.924e-9, 0.995)
PCI write	(17.737e-15, 40.461e-9, 0.969)	(7.2877e-15, 44.633e-9, 0.996)

Figure 3.4 shows measured values identical to Figure 3.2 but also shows the approximated polynomial functions.[6] One external write transaction transfers 32 bytes from the host bridge to main memory, but one external read transaction transfers only 16 bytes. Hence, the impact on applications of write transactions is increases quicker than the impact of read transactions.

[5] Notation as defined by Gauss: $[x] = \sum_{i=1}^{n}(x_i)$.

[6] For a better visibility, we have split up Figure 3.4 into four graphs in Appendix B.

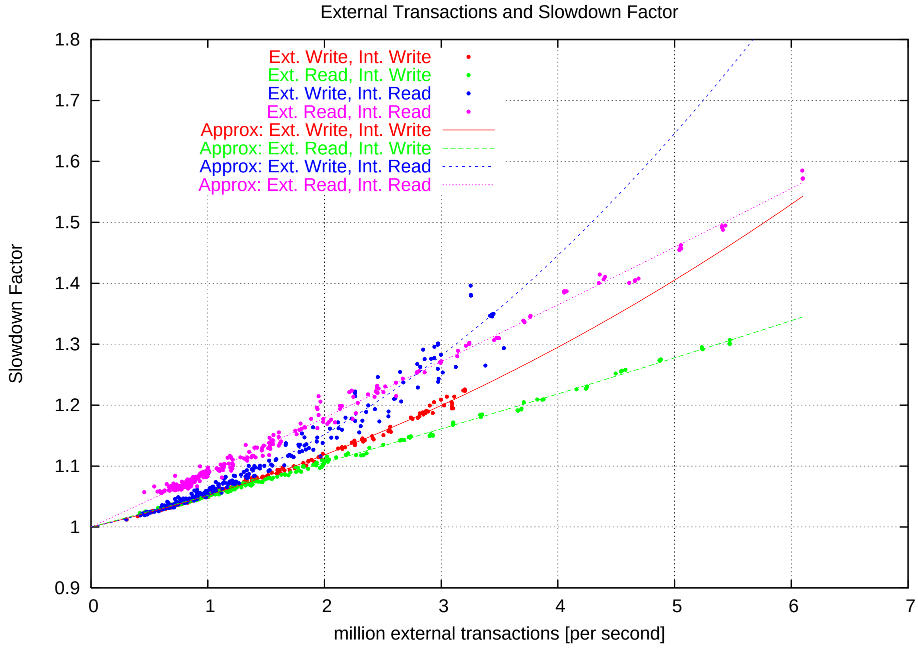


Figure 3.4.: Approximation of resulting slowdown factors using polynomial functions and coefficients from Table 3.3

Figure 3.5 shows that the weighted absolute errors e_i, as given in Equation 3.13, are less than 6%. This indicates that a quadratic function approximates the behavior of the real machine very well.

$$e_i = \left| \frac{y_i - f(x)}{f(x)} \right| . \tag{3.13}$$

Another parameter that describes the exactness of an approximated function is the mean error σ_y which can be calculated by

$$\sigma_y = \sqrt{\frac{1}{n-k-1} \sum_{i=1}^{n} \Big(y_i - f(x) \Big)^2} . \tag{3.14}$$

The smaller the mean error, the better the approximation. In our case, we calculated $\sigma_{ww} = 0.003865$, $\sigma_{wr} = 0.013028$, $\sigma_{rr} = 0.007934$, and $\sigma_{rw} = 0.002696$. This also demonstrates the quality of our results.

Analyzing the graphs of Figure 3.2, we can also see that the impact on CPU read operations is different than on CPU write operations and also depends on the type (*i.e.*, read or write) of the external transaction. To further improve the method to calculate the application's WCSF presented in the previous section, we determine a *weighted slowdown factor* for an application based on the ratio of its read and write transactions. The ratio of external read transactions on the total number of transactions is

$$\rho_r = \frac{t^r_{ext}}{t^r_{ext} + t^w_{ext}} , \tag{3.15}$$

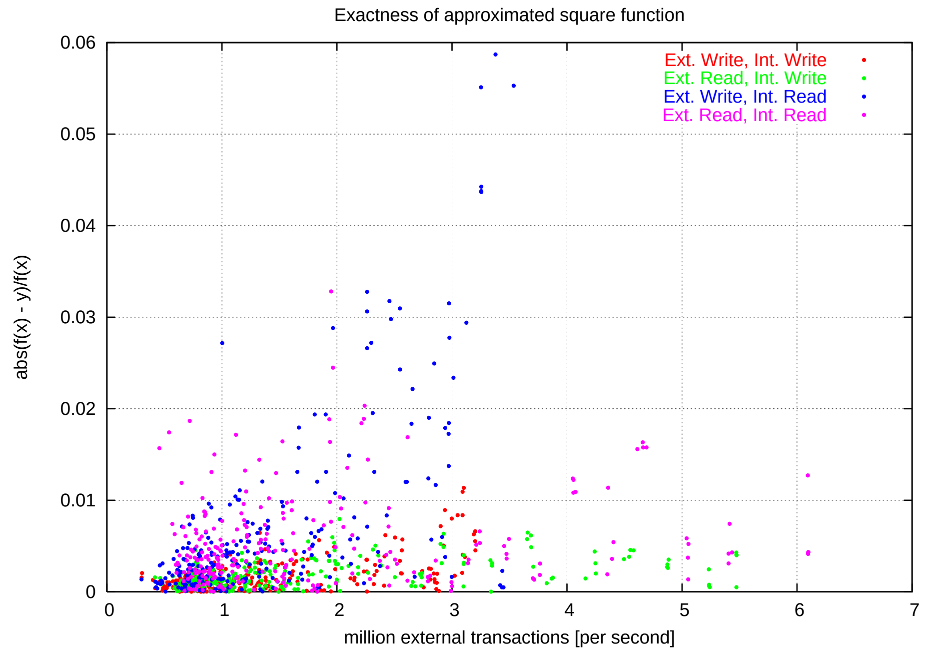


Figure 3.5.: Exactness of the approximated square functions for the calculated slowdown factors

and the ratio for write transactions is

$$\rho_w = \frac{t^w_{ext}}{t^r_{ext} + t^w_{ext}}. \tag{3.16}$$

To calculate the weighted slowdown factor for internal read transactions, we add the slowdown factor for external read transactions ($\mathcal{F}_{rr}$) weighted by the read ratio (ρ_r) and the slowdown factor for external write transactions ($\mathcal{F}_{wr}$) weighted by the write ratio (ρ_r). We obtain the following:

$$\mathcal{F}_r(t_{ext}) = \mathcal{F}_{rr}(t^r_{ext})\rho_r + \mathcal{F}_{wr}(t^w_{ext})\rho_w \tag{3.17}$$

for processor read operations. The same method is applied to obtain the weighted slowdown factor for write operations:

$$\mathcal{F}_w(t_{ext}) = \mathcal{F}_{ww}(t^w_{ext})\rho_w + \mathcal{F}_{rw}(t^r_{ext})\rho_r. \tag{3.18}$$

We can determine the weighted slowdown factor of an application by replacing $\mathcal{F}_r^{L_{max}}$ and $\mathcal{F}_w^{L_{max}}$ in Equation 3.8 with $\mathcal{F}_r(t_{ext})$ and $\mathcal{F}_w(t_{ext})$, respectively. We receive:

$$\mathcal{F}_A(t_{ext}) = \frac{S_r cyc_r \mathcal{F}_r(t_{ext}) + S_w cyc_w \mathcal{F}_w(t_{ext}) + S_o cyc_o}{S_r cyc_r + S_w cyc_w + S_o cyc_o}. \tag{3.19}$$

In the following section, we demonstrate how to apply these formulas to obtain the slowdown factor of an application under specific external load and compare it to measurement results.

3.3.3. An Example Calculation

Applying Equation 3.19 to the DES application with a very small worst-case slowdown factor gives a result almost equal to 1. This result can also be confirmed by measurements which showed that DES is virtually uninfluenced by any external load. Additionally, we performed a test of the memory-copy application with two PCI cards. While the first card generated 25MB/s PCI-bus read load ($t^r_{ext} = 1,562,500$ transactions per second), the second card generated 30MB/s PCI-bus write load ($t^w_{ext} = 937,500$ transactions per second). The memory-copy performance was about 23MB/s.[7] We measured the number of CPU read and CPU write transactions 6,029,312 for each transaction type.

We can calculate $\rho_r = 0.625$ and $\rho_w = 0.375$. Based on Equation 3.10, we can further calculate the individual slowdown factors as a combination of read and write operations by $\mathcal{F}_{ww} = 1.044$, $\mathcal{F}_{rw} = 1.081$, $\mathcal{F}_{wr} = 1.022$, and $\mathcal{F}_{rr} = 1.139$. Applying Equations 3.17 and 3.18 we obtain $\mathcal{F}_r = 1.095$ and $\mathcal{F}_w = 1.067$. Finally, the weighted slowdown factor is calculated by applying Equation 3.19: $\mathcal{F}_A(rd = 25\text{MB/s} + wr = 30\text{MB/s}) = 1.082$. This confirms our measured slowdown factor of 1.08.

Instruction fetches from main memory are handled as read transactions, and therefore are automatically considered. If the size of the executed code is small enough to fit in the L2 cache, the contribution of instruction-fetch-based read transactions is very small. This is the typical case for most of the real-time application with a short path of periodically executed code. To prevent cache flushing due to context switches, techniques such as cache coloring [62] can be used.

3.4. Summary

In this chapter we introduced the slowdown factor to express the impact of external bus load to the execution time of an application. The slowdown factor can be used to adjust a scheduler reservation to external load.

A quick, but coarse, estimation of whether an application can be scheduled on a system under any possible external load is based on the system's upper-bound worst-case slowdown factor. As is common for all worst-case estimations, the difference between actual and calculated (or estimated) value can be very high. An alternative uses the instruction mix, a description of an application, and the system's upper-bound worst-case slowdown factor. In contrast to the first method, the actual share of read and write instruction of an application is considered to determine an application-specific worst-case slowdown factor.

In the third method, the system's upper-bound worst-case slowdown factor is replaced by the worst-case slowdown factors for read and write operations. The fourth approach replaces these worst-case slowdown factors by the weighted slowdown factors under a certain load. The weighted slowdown factors are calculated by means of a polynomial approximation function. The result is an application-specific slowdown factor under a certain load.

On current PCI-based systems, the slowdown factors are very small and not very relevant. However, with the advent of new high-speed bus systems the impact of external load on the execution time of applications will become relevant again. The described technique is a promising method to handle this impact and needs further investigation.

% $Id : bussys.tex, v2.42003/05/2604 : 26 : 29 schoenbgExp$

[7] Remember that we still operate over uncached memory.

4. Peripheral Component Interconnect Bus

In this chapter, we focus on use of the Peripheral Component Interconnect (PCI) bus in real-time systems. Based on its most relevant features, we develop a model to analyze and describe the behavior of PCI devices. We derive formulas to calculate worst-case access latency and worst-case bandwidth of PCI hardware used in off-the-shelf PC systems.

The main contribution, however, is a novel PCI-bus arbiter that allows assigning variable shares of the PCI-bus bandwidth to individual devices. The arbiter combines guaranteed bandwidth reservation with the characteristics of commodity systems such as high bus utilization and low guaranteed arbitration latency. By chaining two such arbiters, the scheme also supports a combination of hard and soft reservation.

4.1. Bus Characteristics

The PCI bus is a multiplexed bus architecture; address and data signals share the same physical wires. Data transfers are always initiated by an *initiator device*, the receiver of the data is called the *target device*. Every device that can act as an initiator device is called *master*. To coordinate bus accesses among devices connected to the bus, an *arbiter* is required to grant the right of using the bus for a certain time to one device. After the bus has been granted to a device, the data exchange takes place. The basic transfer mechanism is the *burst*. A burst is composed of an address and one or more data words. The address is automatically incremented for each consecutive data word. In cases where main memory is addressed by a PCI device, the *PCI host bridge* is responsible for routing data between the PCI bus and the main memory. This allows us to consider main memory as a PCI device. Since main memory cannot initiate transfers, main memory is not a master.

A write operation transfers data from the initiator device to the target device. After the initiator device has received access to the bus, it puts the physical target address on the bus and immediately starts sending the data stream. To perform a read operation, the initiator put the source address on the bus. Hereafter, the target device handling this address responds to the request by delivering data. It is important to note that the CPU is not involved in transferring the data, neither between devices nor between a device and the main memory.

To allow interaction with PCI devices, the PCI bus provides three types of addressing targeting three different types of address spaces. The *configuration space* is a per-device area of 256 bytes which allows the configuration of the device. It contains protocol-specific information such as a *latency timer*, addresses and sizes of the other spaces, the interrupt line, but also an identifier of the vendor, the name, and the type of the device. To each vendor a unique 16-bit number is assigned by the PCI Special-Interest Group (PCISIG). The device name is also a 16-bit number which can be freely chosen by the vendor. The combination of vendor and device number forms a unique identifier for any device. The configuration space is addressed by *configuration cycles* generated by the host bridge.

The *memory space* and the *I/O space* are relevant to the exchange of data with the device. The memory space of all PCI devices and the main memory belong to one single physical address space. Hence, the physical address is sufficient to select a device. While memory space can be used by any bus master, the I/O space can be accessed only using special I/O cycles. Processors of the IA-32 architecture provide various `in` and `out` instructions to operate on the I/O space. In other systems, such as Alpha or MIPS-based machines, where the CPU does not provide such instructions, the host bridge transforms memory-read/write cycles targeting a certain area in the physical address space into I/O cycles on the PCI bus. I/O transactions are, in general, not used to transfer large amounts of data because of their poor performance in comparison to transactions with memory space. The PCI specification highly recommends preference of memory space over I/O space.

4.1.1. Important Properties and Used Terms

To build a common basis for the following sections, we use these terms: Basic measurement unit is the *PCI-bus clock cycle*. A single PCI-bus clock cycle takes 30ns on a standard 33MHz PCI bus. The bus is 32 bits wide, resulting in a theoretical bandwidth of 132MB/s.

Arbitration: Arbitration is the mechanism that selects one device among those trying to gain access to the bus and then grants ownership of the bus to the selected device. Arbitration is mandatory and takes place every time a device wants access to the bus.

Burst Transfer: The burst is the basic transfer mechanism. It is composed of an address and one or multiple data words with contiguously increasing addresses. The length of the burst is variable and can change for each transaction; the maximum burst length is limited by the PCI bus specification.

Latency: Latency is the number of bus clock cycles it takes between a master's request of the bus and when the arbiter grants the bus to that master. In general, the reason for latency is an occupied bus. To achieve a low latency, initiators and targets are limited (by the PCI-bus standard) as to the time they can occupy the bus. This considers the length of the burst as well as the number of wait states they can add to a transaction.

Latency Timer: Each master capable of bursting more than two data words in one transaction must support a latency timer, accessible by the host processor in the device's configuration space. This programmable timer limits the time a device can occupy the bus during one transaction. When a transaction starts, the latency timer value is automatically loaded by the device into an internal register which is decremented with each PCI-bus cycle. If the latency timer of the device expires, the device must terminate the transaction within three cycles.

Stalling Cycle: If during a transaction the initiator is not able to send or the target is not able to accept data, a stalling cycle is inserted. No data is transferred, but the bus is still occupied. A device can insert up to three contiguous stalling cycles. If more than three stalling cycles are required, the device must terminate the transaction.

4.1.2. PCI-Bus Arbitration

The PCI bus uses a central arbitration scheme. By granting the bus to a master, control is handed over and the master can initiate the transaction. Figure 4.1 illustrates the central arbitration scheme. Each device has an individual pair of request/grant lines connected to the arbiter. To reduce arbitration costs, arbitration can be performed while another transaction is still in progress (*hidden arbitration*). The selected master must wait until the active transaction has been completed before it can start its transaction.

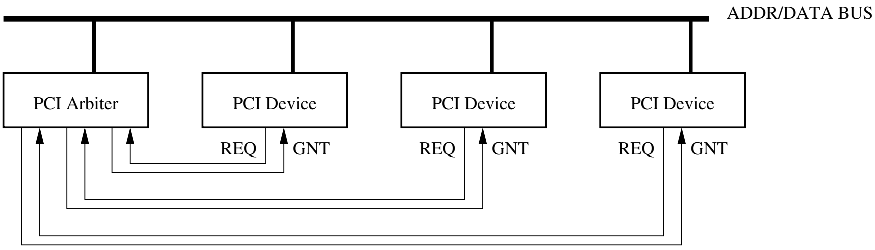


Figure 4.1.: PCI-bus central arbitration scheme

In order to minimize the average access latency, the arbitration scheme is access based, rather than time-slot based. While fixed time slots allow high predictability, an access-based scheme increases performance, but makes prediction more difficult. The arbitration algorithm is not part of the specification and can be adapted to the platform requirements.

> "However, since the arbitration algorithm is fundamentally not part of the bus specification, system designers may elect to modify it. [...] An arbiter can implement any scheme as long as it is fair [...]. The arbiter is required to implement a fairness algorithm [...]" (PCI Local Bus Specification [6], p. 55)

Fairness is defined so that each potential master must be granted access to the bus independent of other requests. However, this does not mean that all devices must have an equal share of the bus. The fairness argument is important for time-sharing systems to assure progress of all components, but conflicts with the requirements of real-time systems with hard priorities, where a high-priority device must be able to prevent a low-priority device from accessing the bus. This means that an accepted reservation must be enforced independent of "fairness" regarding to other devices. The ideal algorithm for a mixed real-time/time-sharing system such as Drops supports reservation for transfers of real-time data as well as fairness among time-sharing devices.

4.1.3. Bandwidth and Latency Considerations

As stated in previous sections, the PCI specification defines only the electrical and physical properties of the bus. The standard PCI bus is 32 bits wide and operates at a frequency of 33MHz, which allows a theoretical maximum bandwidth (bw_{pci}) of 132MB/s if data could be sent in one infinite burst. However, the size of every burst is limited, which leads to a lower available bandwidth. The length of the burst also determines the latency of other devices. The longer a burst, the higher the resulting bandwidth for a device, and the higher the latency for other devices.

To estimate the bandwidth available to devices, we assume that the overhead $ov(d)$ is proportional to the transaction's burst size d. We can make this assumption, since the number of wait-state cycles depend directly on the size of the burst. The maximum bandwidth of one device in relation to the burst length, without considering the behavior of other devices, can be calculated by

$$bw_{max} = \frac{d}{d + ov(d)} * bw_{pci}. \tag{4.1}$$

As we show later, round-robin arbitration is common in standard host bridges. To calculate the worst-case bus-access latency with such an arbitration scheme, we initially assume all other devices receive the bus. The worst-case latency τ_{lat} for n devices can be given by

$$\tau_{lat} = (n - 1)(d + ov(d)). \tag{4.2}$$

Figure 4.2 illustrates the relation between burst size and maximum bandwidth available to transfer data. This bandwidth is shared among all devices. It also shows the relation between burst size and worst-case latency to access the bus when five devices are connected.

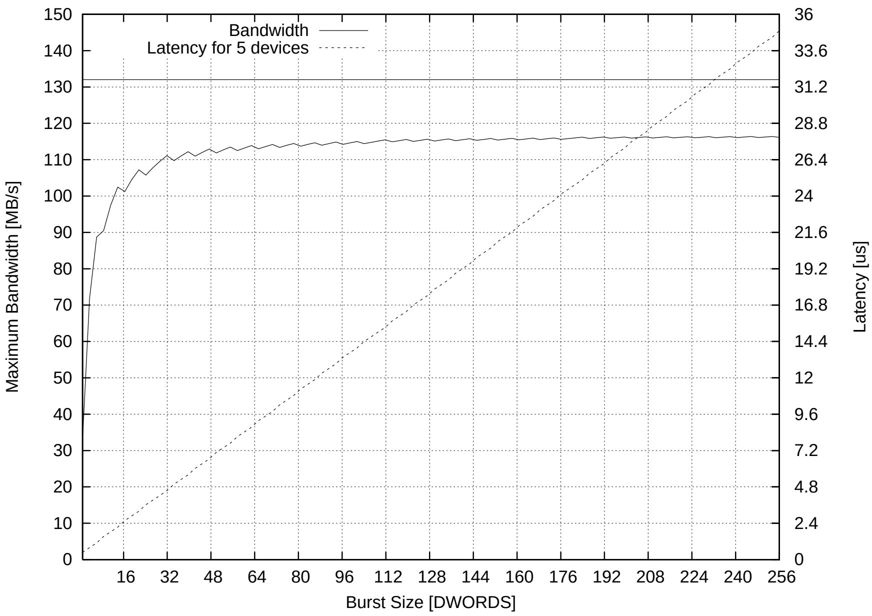


Figure 4.2.: Relation between burst size and both maximum bandwidth and latency.

To verify Equation 4.1, we measured the maximum bandwidth on our test system[1]. The result of 118MB/s is also confirmed by the specification of the host bridge [63], where a maximum memory-access bandwidth of 120MB/s by use of read-prefetch and write-posting buffers is given.

4.1.4. PCI-Bus Conclusion

Neither the specification nor the basic functionality of the PCI bus prevent its application for real-time systems. However, access latencies and bandwidth can only be guaranteed if the arbitration algorithm used supports bandwidth reservation.

Various publications claim the possibility of using the standard PCI bus in real-time environments [64, 65]. However, they all lack substantial proof and do not describe existing

[1]For a detailed description of the test system see also Section 1.2.1.

limitations of hardware such as numbers for worst-case latency and bandwidth. To determine these limitations, we investigate the behavior of commercially available standard PCI hardware in the Section 4.3.

4.2. Simplified PCI-Bus Model

To analyze the behavior of the PCI bus and to perform worst-case calculations of bandwidth and latency, we need a model of the bus. A model that includes all features of the PCI-bus protocol would allow us to describe the exact behavior. It would also contain many features which are not necessary to determine bandwidth and latency bounds, making it unnecessarily complex and not suitable for a fast online admission test. However, such a test is required to estimate the real behavior in combination with the probabilistic techniques described in Section 2.5.2.

The simplified model is based on the characteristics of PCI-bus transactions and subsumes the original state machines given by the specification. These state machines describe the transitions from one bus state to another and give information about the possible states of the bus. Regardless of the characteristics and type of a transaction, the bus is always in one of the following bus states:

Idle: The bus is not used by any device and no data is being transferred. A device that requests the bus in the idle phase is immediately granted access to the bus. A bus arbitration cannot be hidden behind any other transaction and at least two cycles are required before the address can be transmitted.

Busy: The bus is currently granted to one device, either addresses or data may be transferred. If the initiator or the target cannot deliver or accept data, wait-state cycles are added. The busy state indicates only that the bus is in use, but not how the bus is used.

Busy/Data: The bus is in the busy state and it is known that data is transferred.

Busy/Non-data: The bus is in the busy state and it is known that no data is transferred. The reason can be sending the address or a stalling cycle.

Based on the two types of busy phases, our model consists of two phases to describe the behavior of devices: a *non-data phase* (s)[2] and a *data phase* (d). The two-phase model is sufficient to describe all devices that can start their next transaction immediately after finishing one transaction.

However, many devices are not capable of such a behavior and require a certain time between two transactions. To consider this behavior, we introduce the *recovery phase* (r). The recovery phase is not a bus phase and does not specify whether the bus is idle or busy. It is guaranteed only that devices do not access the bus during their recovery phase. In our model, a device is henceforth described by a triple called device descriptor $\mathcal{D}$:

$$\mathcal{D} = (s, d, r). \tag{4.3}$$

The distinction between s and d phase must be made since the device does not transfer data during the entire time it has occupied the bus.

[2]Initially named to indicate the stalling phase, but has been extended later to comprise all non-data cycles

In addition to the device descriptor and the bus states, we introduce the term *contention*. Contention is a situation in which the bus has been granted to one device (bus busy) and at least one other device is requesting the bus. The higher the bus contention rate, the more often devices must wait to receive access to the bus, which automatically increases the latency of devices.

Our model does not consider special types of transactions such as I/O transactions, delayed transactions, or interrupt-acknowledgment cycles. These transactions are either infrequent or can be mapped to our model. For instance, delayed transactions can be seen as two independent transactions. Interrupt-acknowledgment cycles are short (usually 4 cycles) and can be mapped to our model by a device descriptor $\mathcal{D}_{\mathrm{IRQ}} = (3, 1, \delta)$, where δ is the number of cycles between two interrupts.

4.2.1. Consistency with State Machines

To check consistency of our model with the state machines given by the PCI-bus specification, we reduce the original target and master state machine as illustrated in Figure 4.3 and combine the states of the state machine according to the bus states they represent.

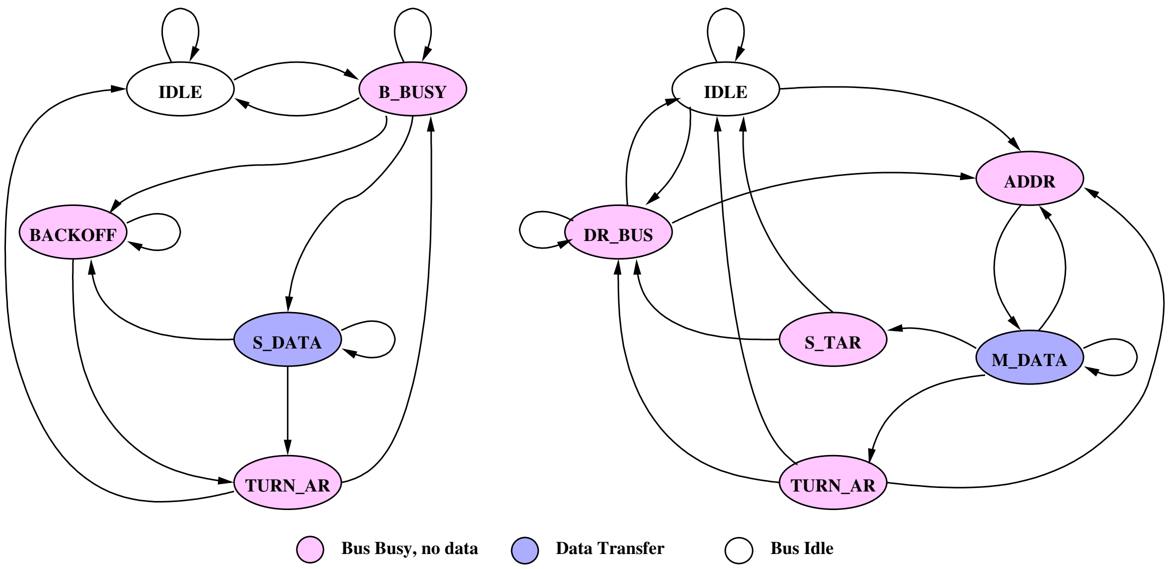


Figure 4.3.: State machines for PCI target (left) and PCI master (right) (A detailed explanation of the states can be found in [6])

The resulting simplified state machine is shown in Figure 4.4. The components of the device descriptor $\mathcal{D}$ describe, how many cycle the bus is in each state during a transaction.

Since the recovery phase of a device is not part of the specification and cannot be seen on the bus, we cannot derive the recovery phase from the state machine.

4.2.2. Consistency with Timing Diagrams

Additionally, we check the consistency of our model with timing diagrams. Timing diagrams show the state of individual bus lines for every bus cycle during a transaction. In the following

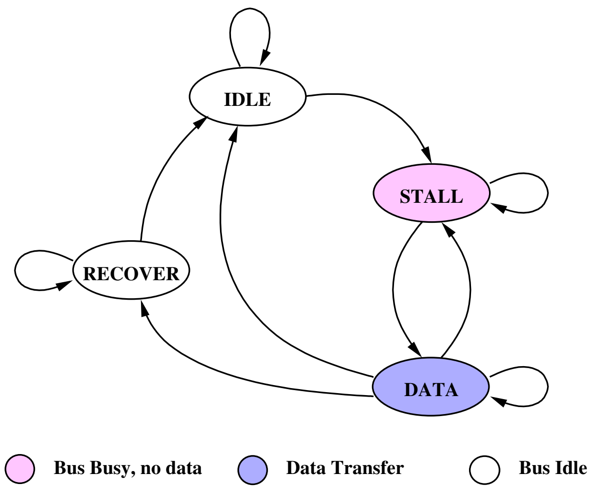


Figure 4.4.: Simplified PCI state machine

example, we explain such a timing diagram for a PCI-bus read transaction where a master reads a three-word data burst from main memory. The control flow is as follows: After the bus has been granted to the master (*i.e.*, now the initiator), the master sends the address to the target and expects one or more data words as result. Since the target is the main memory and not a physical PCI device, the host bridge acts on behalf of the memory and provides the data.

Figure 4.5 illustrates the most important electrical signals on the PCI bus. The GNT line indicates that the arbiter has granted bus access to the master. In the first bus cycle, the $\overline{\text{FRAME}}$ line is driven by the initiator and indicates the beginning and duration of a transaction. The initiator drives the start address and the transaction type onto the address/data bus (AD bus). In cycle two, the initiator ceases driving the AD bus. An additional so-called turn-around cycle is required to avoid collision of signals which can be driven by more than one device. The turn-around cycle is used to transfer ownership of the bus from the initiator to the addressed target—in our case the host bridge—which takes control over the AD bus. Additionally, the initiator asserts $\overline{\text{IRDY}}$ to indicate that it is ready to receive the first data word from the target. In cycle three, the target deasserts $\overline{\text{DEVSEL}}$ to indicate that it has recognized its address and will be the target of the transaction. In the following cycles, $\overline{\text{TRDY}}$ indicates that data is available on the AD bus. If $\overline{\text{TRDY}}$ is deasserted, as illustrated in cycle five, the target indicates a wait-state cycle: In our example, the main memory cannot send data fast enough. In case of a hidden arbitration, the arbiter would grant the bus to the next device immediately in cycle seven, otherwise the bus returns to the idle state as shown in cycle eight. A very detailed description of various transaction types and their diagrams can be found in [6, 66]. Every PCI-bus transaction consists of the parts listed in Table 4.1.

While $c_{address}$ appears exactly once within a transaction, c_{arb} and c_{turn} can occur at most once. The data transaction is described by the three cycles c_{data}, $c_{wait,i}$, and $c_{wait,t}$. We can assign each cycle either to s or to d by:

$$s = c_{arb} + c_{address} + c_{turn} + c_{wait,i} + c_{wait,t} \quad (4.4)$$

$$d = c_{data} \quad (4.5)$$

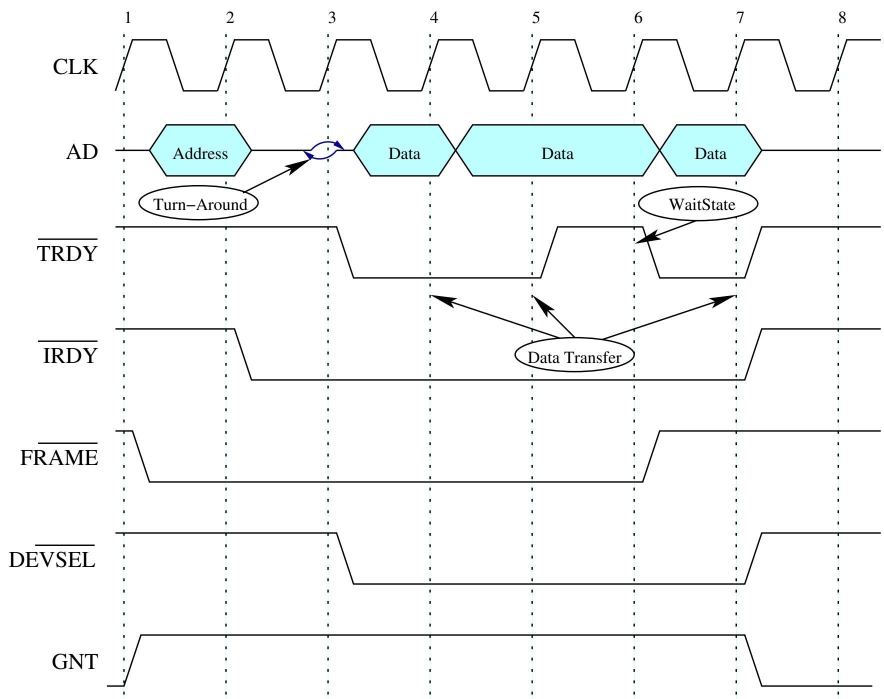


Figure 4.5.: Timing diagram of a read transaction with a three-word data burst and one wait-state cycle

Table 4.1.: Parts of a PCI-bus transaction

Cycle	Phase	Name	Number
c_{arb}	non-data	Arbitration cycle	0 or 1
$c_{address}$	non-data	Address cycle	1
c_{turn}	non-data	Turn around cycle	0 or 1
c_{data}	data	(Burst) data cycle	1 – 255
$c_{wait,i}$	non-data	Wait states due to initiator	w_i
$c_{wait,t}$	non-data	Wait states due to target	w_t

The value of the latency timer (L) is considered as follows: upon expiration of the master's timer, a data transaction must be terminated by the master to free the bus within three cycles ($s + d \leq L + 3$).

Since the recovery phase of a device cannot be seen on the bus, we cannot derive the value for the recovery phase r from timing diagrams.

4.2.3. Model Verification

To verify our assumptions and the model, we performed measurements with the Pamette board and a software performance tool for the Pamette described in [17]. The software tool programs the board such that it records approximately 32,000 PCI-bus cycles. The DMA engine of the Pamette board is programmed to generate bursts of variable length with variable gaps in between. As shown in Appendix C on page 83, we can match the measurements taken by the Pamette board and our model. In the given case, the non-data phase (s) is 3, the burst length (d) is 16. The tool allows us to freely define the number of PCI cycles between two bus requests of the card. In our case, the recovery phase (r) has been set to 31; the resulting device descriptor is $\mathcal{D} = (3, 16, 31)$. Sometimes, we have seen slight jitter of the burst length and the number of stalls before a transaction starts. Some devices behave differently on reading or writing data. This problem can be solved by considering one physical device as two logical devices. The first logical device is responsible for read operations, the second device for write operations. This results in two device descriptors, one for read and another for write operations. Since no device can read and write data within the same transaction, this is an acceptable and correct solution.

In older systems, where main memory is relatively slow, we have seen typical burst lengths of only eight 32-bit words, independent of the value specified in the device's latency timer. This behavior is caused by the host bridge which buffers exactly eight words in its internal buffers. Thereafter, the host bridge must either force these data words out of the buffer and write them into main memory for a write operation, or must fetch the next data from main memory for a read operation. Due to the maximum allowed three contiguous stalling cycles, this leads often to a termination of the transaction. As a result, the d and s values of device descriptors in such systems are mainly determined by the host bridge and not by the latency timer or the characteristics of a device. This fact must be considered when device parameters are determined.

4.3. Analysis of Current Standard Hardware

We begin by focusing on standard hardware as used by the DROPS system. To analyze the behavior of commodity PCI hardware, we looked into the following commercial host bridges: Intel 430TX, 440FX, 440BX, and i810 [58, 67, 63, 68]. For analysis of the impact of the PCI bus on real-time systems, the arbitration scheme is of special interest. We observed that all these host bridges use only a simple, non-prioritized round-robin scheme. The device selected next is based on the currently active device. Figure 4.6 illustrates the used PCI-bus arbitration scheme and how access to main memory is shared between the processor and the host bridge.

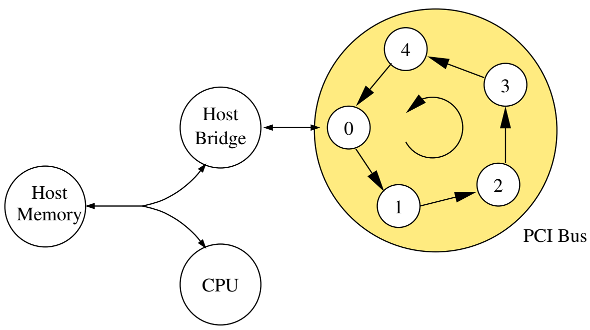


Figure 4.6.: Round-robin PCI-bus arbitration scheme and access to main memory of commodity bridges

To estimate bandwidth and latency of a PCI device, the formulas given in Section 4.1.3 can be used. However, these simple formulas are not sufficient to determine upper bounds for bandwidth and latency resulting from the interaction of arbitrary devices.

4.3.1. Model for Identical Devices

The first step towards a description of the behavior of arbitrary PCI devices assumes n identical devices. "Identical devices" means that all devices can be described by the same device descriptor $\mathcal{D}$ and therefore generate the same bus-access pattern. The result can be used to describe the behavior of PCI devices in a PC-based router or firewall, or in a fast file caching server such as the "hit-and-miss server" [69] with multiple identical network cards.

The worst-case latency τ_{lat} that one device must wait until it is granted the bus occurs when all other devices obtain access to the bus before that device. The result is similar to that given by Formula 4.2 and can be calculated by:

$$\tau_{lat} = (n-1)(s+d). \tag{4.6}$$

In the worst case, every access is postponed by this latency. Hence, we receive a worst-case bandwidth that can be calculated by:

$$bw = bw_{pci}\frac{d}{(s+d+r)+((n-1)(s+d)-1)}. \tag{4.7}$$

However, this requires gaps between the end of a recovery phase and the bus request of all devices. Otherwise, the bus access can be interleaved. If we assume that all devices operate at their maximum possible bandwidth, *i.e.*, we have no gaps between two requests, the following two cases must be considered:

1. If it holds that $r \geq (n-1)(s+d)$, a device can start immediately after it has finished its recovery phase, since each of the $n-1$ remaining devices has completed its transaction before the end of the recovery phase of the first device. Figure 4.7 gives an example for two devices.

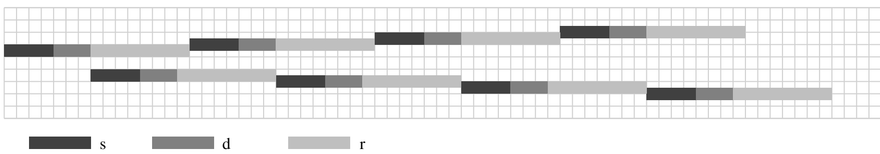


Figure 4.7.: Two identical devices with $\mathcal{D} = (4, 3, 8)$

The shortest time between the end of a transaction and the start of the next transaction is determined only by the length of a device's recovery phase. Since all devices have a recovery phase of the same length, it is not possible for one device to perform multiple transactions while another device recovers.

2. We consider $r < (n-1)(s+d)$. Here, the device is ready before all other devices have finished their transaction. In our case, where arbitration is based on a simple round-robin scheme, the device scheduled next depends on which device is completing a transaction. Figure 4.8 illustrates the behavior of such two devices.

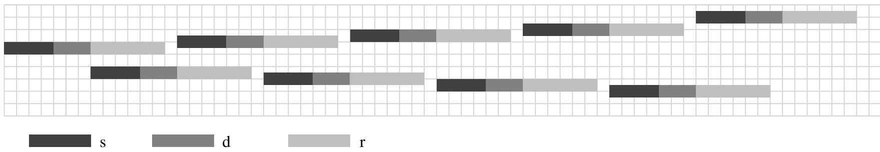


Figure 4.8.: Two identical devices with $\mathcal{D} = (4, 3, 6)$

From a combination of both case, we can calculate the bandwidth a device can achieve by:

$$bw = bw_{pci} \frac{d}{max((s+d)n, (s+d+r))} \tag{4.8}$$

This value is not of direct interest for hard real-time systems since it is not the worst-case. However, it can be used to determine how far the achieved bandwidth differs from the worst-case in the case that all devices generate their maximum bandwidth. The result can be used to decide whether other reservation schemes, for instance schemes based on probabilistic parameters, can provide better results [1].

Based on these two cases, we can calculate the bus utilization $U(n)$ for n active devices. It is the ratio between the number of cycles where the bus is occupied and the number of elapsed cycles:

$$U(n) = \frac{(s+d)n}{\max((s+d)n, (s+d+r))}. \tag{4.9}$$

The denominator is the maximum of the number of cycles n devices can occupy the bus, and the recovery time of one device. The bandwidth n devices can generate when each device tries to send data with its maximum rate is given by:

$$bw(n) = bw_{pci} \frac{dn}{\max((s+d)n, (s+d+r))}. \tag{4.10}$$

The bandwidth of a system with only one device is determined by $bw(1)$. Setting r to zero and considering s as the overhead with $s = ov(d)$ leads to the simplified maximum bandwidth formula already given by Equation 4.1 on page 42. Under high bus load, the arbitration cycle does not require an extra cycle (hidden arbitration). The minimum value of s for a write operation is 1 (address cycle), and for a read operation is 2 (address and turn-around cycle).

The bus-access pattern of five network cards can be simulated by our "PCItrace" bus simulator. Figure 4.9 illustrates the interleaving of the devices and the contention- and idle-state of the bus.

The maximum number of devices we can connect to the bus, where each device can still send with its maximum bandwidth, even though all other devices also request the bus and send data at their maximum rate is n_{max}. We can derive this value from the first case, where all $s+d$ phases of all other $(n-1)$ devices fit exactly in the recovery phase of a device, *i.e.*, $r = (n-1)(s+d)$:

$$n_{max} = \left\lfloor \frac{r}{s+d} \right\rfloor + 1. \tag{4.11}$$

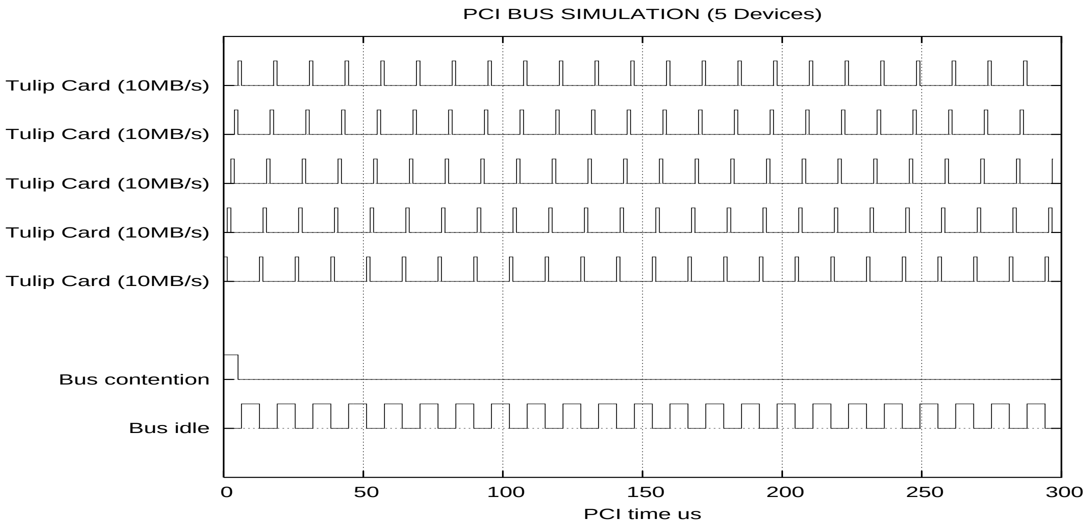


Figure 4.9.: Simulated bus-access pattern for five identical devices

In the next section, we extend the model to cover the interaction of devices with arbitrary parameters.

4.3.2. Model for Arbitrary Devices

A typical system deploys PCI devices with various functionalities and different properties. In contrast to the behavior of "identical devices" as described in the previous section, the values for s, d and r may vary for each device and bus access cannot be interleaved as neatly as for devices with an identical access pattern. This results in a lower bus utilization than theoretically possible since the bus utilization is less than the sum of each device's utilization. This problem is identical to that described by Liedtke *et al.* [20] and could be solved by the technique proposed in the paper. An example of the bus-access pattern of two devices is shown in Figure 4.10.

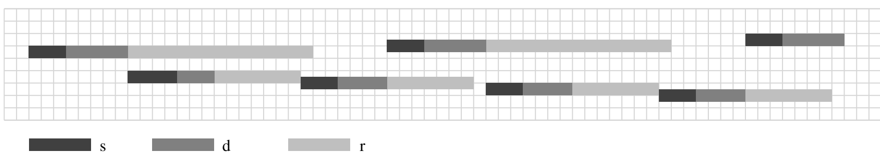


Figure 4.10.: Two sample devices with $\mathcal{D}_1 = (3, 5, 15)$ and $\mathcal{D}_2 = (3, 4, 7)$

As apparent from the figure, a new situation may arise with arbitrary devices: While one device recovers, another device can perform multiple transactions. In the following paragraphs, we develop the model for two arbitrary devices and attempt to generalize the model for multiple devices.

4.3.2.1. Two Devices

In this section, we extend the model to comprehend two arbitrary devices, including the case in which one device performs multiple transactions while the other device recovers. A device $\mathcal{D}_2$ must meet the following condition to perform multiple transaction cycles while another device $\mathcal{D}_1$ recovers:

$$s_2 + d_2 + r_2 < r_1. \tag{4.12}$$

This is sufficient, since $\mathcal{D}_2$ can start at least a second transaction while $\mathcal{D}_1$ has not completed its recovery phase. Without loss of generality, we assume $s_1 + d_1 + r_1 \geq s_2 + d_2 + r_2$; the *displacement* of $s_2 + d_2$ in relation to $s_1 + d_1$ is denoted by a:

$$a \in [0, r_2 \dot{-} (s_1 + d_1)] \tag{4.13}$$

where $\dot{-}$ is the proper subtraction which returns zero in case of $s_1 + d_1 > r_2$.

We define *block time* (b) as the distance between two starts of $\mathcal{D}_1$ at $s_1 + d_1$ depending on the displacement (a) between the end of the transaction of device 1 and the start of the transaction of device 2 by $b = b(a)$. The number of transactions of length $s_2 + d_2$ that $\mathcal{D}_2$ can perform during the interval $[s_1 + d_1 + a, s_1 + d_1 + r_1)$ is denoted by $\nu = \nu(a)$.

Figures 4.11 and 4.12 illustrate two devices where the second device $\mathcal{D}_2$ can perform multiple (at least two) transactions during the recovery phase of the first device $\mathcal{D}_1$. The time where a device cannot transfer data since the bus is used by the other device is indicated by $\frown$.

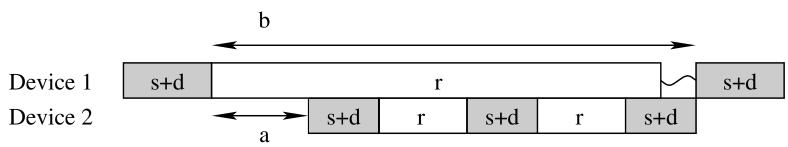


Figure 4.11.: Two devices, multiple transaction cycles, with delay

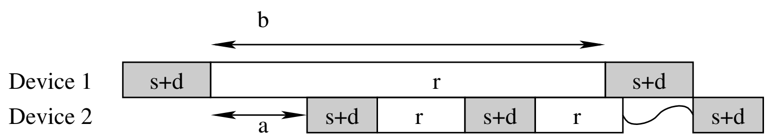


Figure 4.12.: Two devices, multiple transaction cycles, without delay

As the figures illustrate, two possible cases have to be considered:

1. Figure 4.11 illustrates the case where the last transaction of $\mathcal{D}_2$ has been started during but exceeds the recovery phase of $\mathcal{D}_1$. Hence, $\mathcal{D}_1$ cannot start immediately after its recovery phase and must wait until $\mathcal{D}_2$ has completed its transaction.

2. Figure 4.12 is easier to handle. The last work phase of $\mathcal{D}_2$ is shorter than the recovery phase of $\mathcal{D}_1$. At this time, the first device is able access the bus, and is immediately granted the bus since the second device is in the recovery phase.

The number (ν) of transactions device $\mathcal{D}_2$ performs during the recovery phase of $\mathcal{D}_1$ can be calculated by:

$$\begin{aligned} & \max\{\nu \mid r_1 > (\nu-1)(s_2+d_2+r_2)+a_i\} \\ \nu - 1 &< \frac{r_1 - a_i}{s_2+d_2+r_2} \text{ with } r_1 > a_i \geq r_2 \\ \nu &= \left\lceil \frac{r_1}{s_2+d_2+r_2} \right\rceil . \end{aligned} \quad (4.14)$$

Starting with $a_0 = 0$, the block time b_i and the next displacement a_i can be calculated by:

$$\begin{aligned} b_i &= s_1 + d_1 + max(r_1, (\nu-1)(s_2+d_2+r_2) + a_{i-1} + (s_2+d_2)) \\ a_i &= a_{i-1} + \nu * (s_2+d_2+r_2) \dot{-} b_i . \end{aligned} \quad (4.15)$$

Equation 4.15 allows the iterative calculation of when and how often a device is scheduled. However, even for two devices, we cannot give one closed formula to determine the exact bandwidth and latency. The worst-case latency for device $\mathcal{D}_1$ is $\tau_{lat,1} = s_2 + d_2$ and for device $\mathcal{D}_2$ is $\tau_{lat,2} = s_1 + d_1$.

4.3.2.2. Multiple Devices

While the number of access combinations of two devices is limited and can be described by the formulas given in the previous section, the number of possible access patterns increases polynomially for higher numbers of devices, since multiple devices can do multiple transactions while other devices recover. As was the case with two devices, we cannot give a closed formula to exactly determine which device accesses the bus at which time. A complex iterative scheme can be developed similar to that of two devices; however this is identical to a full simulation on a per-cycle basis. Figure 4.13 illustrates the simulated bus-access pattern of three different devices.

In deterministic real-time systems it is sufficient to know worst-case bandwidth and worst-case latency. To obtain average values or statistical data to perform admission in a probabilistic system, the "PCItrace" simulation tool can be used.

To determine the worst-case latency, we must assume that exactly one cycle before the recovery cycle of device x ends, all other devices have requested the bus, and the round-robin scheduler picks all other devices first. All n devices receive the bus for $s_i + d_i$ cycles. The worst-case latency for device x can be given by

$$\tau_{lat,x} = \sum_{i=1, i\neq x}^{n} (s_i + d_i) - 1. \quad (4.16)$$

The bandwidth a device can achieve can be derived from Equation 4.1 and is calculated in Equation 4.17

$$bw_x = \frac{d_x bw_{pci}}{s_x + d_x + r_x + \tau_{lat,x}}. \quad (4.17)$$

To determine the worst-case bandwidth of device x we combine Equation 4.17 and Equation 4.16:

$$bw_x = \frac{d_x bw_{pci}}{s_x + d_x + r_x + \left(\sum_{i=1, i\neq x}^{n} (s_i + d_i) - 1 \right)}. \quad (4.18)$$

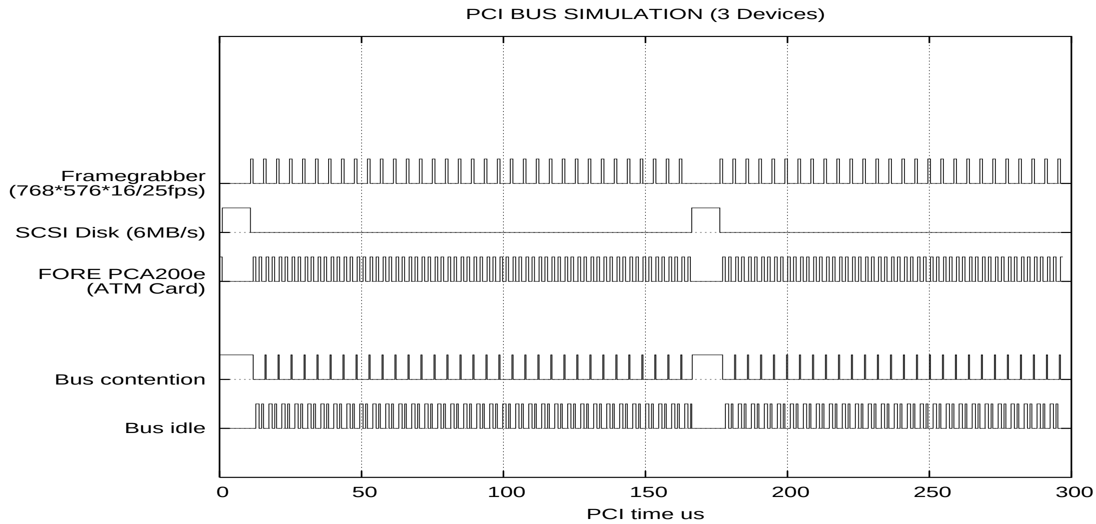


Figure 4.13.: Simulated bus-access pattern of three different devices

The maximum possible bandwidth a device can generate is determined by setting the latency to zero, *i.e.*, $\tau_{lat,x} = 0$; the device will immediately receive the bus at every request.

We give the following example to illustrate the results of these equations. We assume five identical devices on the bus. A device with a descriptor $\mathcal{D} = (8, 6, 12)$ can generate a maximum bandwidth of $132 * 8/(8+6+12) = 40.6$MB/s. In combination with the other four devices sending data, the calculated worst-case bandwidth of this device is only $132 * 8/(5 * (8+6) + (12-1)) = 13.03$MB/s. This makes the major drawback of a bus arbitration based on a simple round-robin scheme obvious: guaranteed worst-case bandwidth per device is only a fraction of the maximum bandwidth of the device acting alone. The the worst-case latency is $((5-1) * (8+6) - 1) * 30ns = 1.65\mu s$.

4.3.3. Influence of Latency Timer

On the PCI system we investigated, the latency timer is the only available method by which the behavior of individual devices can be influenced. As long as all devices have similar parameters, bandwidth is also distributed evenly among devices. However, if some devices can generate a much higher bandwidth than other devices, the difference between average and worst-case bandwidth of these devices is also high. For example, we consider the following case: A Gigabit Ethernet adapter is able to transfer a sustained bandwidth of 80MB/s from the network into main memory. However, the total bandwidth of all accepted network connections by the system is only 10MB/s. If the network card does not provide a mechanism to drop unsolicited packets already on the card instead of transferring all packets into main memory and letting the processor to decide on them, another host can disrupt the correct behavior of the system by generating network traffic. Since the host bridge does not provide a mechanism to explicitly reserve and also enforce the bandwidth to 10MB/s, at a glance, setting the latency timer seems an acceptable solution.

The specification recommends zero-wiring the lower three bits of the latency timer, allowing it to be changed by multiples of eight. The smallest value of eight restricts the device to perform exactly three initial cycles (address, turn-around, and one stall cycle) plus eight data cycles. Applying this fact to Equation 4.16 with $s = 3$ and $d = 8$, we cannot guarantee a restriction of the bandwidth to less than 96MB/s. As a result, the latency timer cannot be used to restrict bandwidth.

4.3.4. Conclusions on Current Standard Hardware

The model and formulas we gave in this section allow us to determine worst-case latency and worst-case bandwidth for PCI devices in a system using one of the current host bridges. Since the model considers cases in which devices send with their maximum possible bandwidth and does not consider any jitter in the input data, it is designed for the overload situation, *i.e.*, the sum of the bandwidth all devices request is higher than or equal to the maximum PCI-bus bandwidth.

Assuming that bus utilization is less than 100% and that all devices behave well and cooperate (*i.e.*, devices do use more than their reserved bandwidth), the PCI bus can be used in real-time environments. Here, our calculated worst-case values are very pessimistic, and the model should be extended by parameters of the input stream as proposed in [70].

If devices can generate higher bus bandwidth and the resulting utilization would exceed 100%, regulation is mandatory to guarantee real-time capabilities and to enforce resource reservation. With the current implementation, the latency timer is the only way to control a device; however, as shown in Section 4.3.3 it cannot be used to effectively restricted the bandwidth a device can generate. Hence, the problem can be solved only by an arbiter that provides reservation of individual amounts of bandwidth for each device.

4.4. Alternative PCI-Bus Real-Time Solutions

Existing alternatives used by telecommunication applications deploy the cost-effective PCI bus or CompactPCI bus only as a control bus. Real-time voice or video data is transferred over a deterministic side bus, such as a time-division multiplexing bus [71].

The problem of any shared-medium bus is that contention can occur when multiple devices try to access the bus at the same time. The switched implementation of the PCI interface as proposed by Wilson [46] overcomes this problem. The major drawback of this solution is the additional space required on the motherboard. Instead of sharing almost all wires among all devices, a complete set of all wires must be connected to each device, which physically limits the number of devices that can be used.

Another alternative to prevent contention on the bus is the following: Since almost all data transfers take place between devices and main memory and transfers between PCI devices are uncommon, the host bridge—on behalf of the main memory—is the only master-capable device. To make this scheme efficient, the host bridge must provide an interface to transfer larger quantities of data between main memory and devices without additional interaction of the processor. The disadvantage is higher demand on the processor since all devices must announce data to be transferred at the processor using an interrupt. The processor responds to that interrupt and initiates the transfer at the host bridge. Since an exact coordination of transfers between main memory, PCI devices, and the processor is possible and also actively initiated by the processor, problems caused by PCI-bus load as described in Chapter 3 can be minimized or avoided.

The major drawback of all solution is that they either lead to major changes in the operating system, or require a complete redesign of the system's hardware. Additionally, no support for a combination of soft and hard reservation is provided. Hence, we concentrate on a solution based on a novel arbiter with support for bandwidth reservation. The changes required to implement the algorithm in a current host bridge are low compared to the other hardware solutions.

As described in Section 2.4.4, a combination of an arbiter with hard priorities and support in the operating system can be used to give bandwidth guarantees. This encourages us that, merely by implementing an arbiter with soft and hard reservations, the PCI bus can be well deployed in the wide area of soft real-time systems.

4.5. Arbiter with Support for Bandwidth Reservation

The results presented in the previous sections can be used to calculate worst-case bandwidth and latency of commercial off-the-shelf systems. However, these calculated worst-case results are pessimistic and may diverge widely from the values achieved in a real-life situation. The major drawback of the arbitration algorithm in commodity host bridges is that it does not provide a method for reserving bandwidth or for limiting the bus access of a device. This is essential for real-time systems where overload may occur. Since this is also the case in the Drops system, we focus on designing an arbiter with support for bandwidth reservations. The new arbiter must provide sharing of the PCI-bus bandwidth in arbitrary proportions among devices. It must also maintain the advantages of the typically implemented round-robin arbitration such as low latency and high bandwidth.

We must consider the following requirements to find an appropriate algorithm:

- throttling of high-bandwidth devices to prevent such devices from monopolizing the bus and disrupting reservations of other devices;
- reserved bus bandwidth of one device that remains unused must be available to other devices, for instance to devices providing best-effort services;
- low and guaranteed arbitration latency;
- low complexity; and
- finally, the algorithm must allow a simple and fast implementation in hardware.

These requirements lead us to the decision to focus on a proportional-share algorithm. All the proportional-share algorithms presented in Section 2.2 are suitable of reservation of the processor as well as for other resources. However, they have been designed to be executed by a general purpose processor and require operations such as floating-point division, maintaining multiple queues, or sorting of requests by their parameters.

In search of an adequate algorithm, we analyze the typical properties of a proportional-share algorithm: Resources are assigned to clients in a certain ratio. For two clients, this means that by knowing the ratio a and the amount x of a resource that one client has received, it is possible to calculate the amount y the other client has received by $y = ax$. Geometrically seen, this formula is exactly the description of a line, however with discrete steps, since a resource can be given out only in discrete quanta. Hence, we chose a line-drawing algorithm as basis for the new arbitration algorithm.

The remainder of this chapter is organized as follows. At first, we describe the Bresenham line-drawing algorithm and how to apply this algorithm to schedule an arbitrary number of PCI devices. Then, we discuss how time-sharing devices can be handled and which extensions are necessary to support a combination of hard and soft reservations. We propose an implementation that can be easily implemented in hardware. Finally, we give a set of formulas for calculating bandwidth and latency values of our arbitration scheme and verify these values by simulation.

4.5.1. Bresenham Line-Drawing Algorithm

The *Bresenham line-drawing algorithm* [72] is a fast algorithm to determine the points of a line between two given points. It was invented by Bresenham to draw lines with a two-dimensional, digital plotting machine, where the pen can be moved by stepping motors in discrete steps only. Its main advantage is the simplicity of the calculation, since neither floating-point operations nor any complex operations such as multiplication or division are required. To implement the algorithm, simple add, subtract, and compare operations are sufficient. All these operations can be easily implemented in hardware with simple logic units. Many modern graphic cards with hardware acceleration implement Bresenham-based line drawing. For all operations, it is sufficient to consider only lines between $(0, 0)$ and (x_e, y_e) that have a positive slope (δ) less than or equal to 1, *i.e.*, $0 < y_e \leq x_e$. Lines with a negative slope, a slope greater than 1, or an origin other than $(0, 0)$ can be implemented by changing the drawing direction, swapping x-axis and y-axis, or adding an offset if the line does not start at $(0, 0)$. Figure 4.14 illustrates the principle of the Bresenham algorithm.

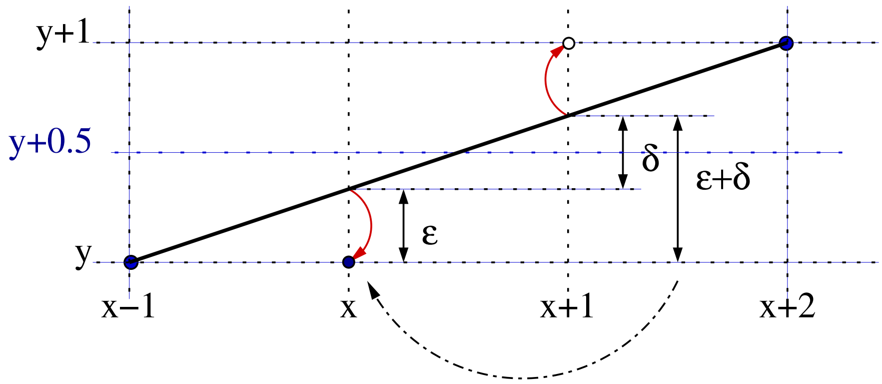


Figure 4.14.: Bresenham line-drawing algorithm

The algorithm works as following: For each integer value in x-direction, an exactly corresponding value on the y-axis is calculated. If, due to the discrete resolution, it is not possible to draw a point at the calculated (x, y) location, the y-value is rounded off to the nearest drawable location. An error ε with the right-side open interval of $[-0.5, 0.5)$ describes the error between the calculated and the drawn point. The line requires discrete x and y-values for the origin. Therefore, it is always possible to draw a point at the location (x, y) and the initial value for ε is 0.

On incrementing the x-value by 1, the exact value for the new y-value is calculated from the previous y-value by adding the line's slope δ. The next point is drawn at the nearest possible place and the error value ε is updated appropriately.

$$\begin{aligned} x_{new} &= x_{old} + 1 \\ y_{new} &= \delta x_{new} = \delta x_{old} + \delta = y_{old} + \delta \\ \varepsilon_{new} &= \varepsilon_{old} + \delta \end{aligned} \tag{4.19}$$

The y-value of the plotted point is either identical to the y-value of the previous point or this value plus 1. The central decision whether to extend in y-direction is based on the new error value ε_{new}. If it exceeds its valid interval, *i.e.*, $\varepsilon_{new} > 0.5$, the y-value is incremented by 1 and the error variable ε_{new} is decreased by 1.

$$\begin{aligned} y_{new} &= \begin{cases} y_{old} & \text{if } (\varepsilon_{old} + \delta < 0.5) \\ y_{old} + 1 & \text{if } (\varepsilon_{old} + \delta \geq 0.5). \end{cases} \\ \varepsilon_{new} &= \begin{cases} (\varepsilon_{old} + \delta) & \text{if } (\varepsilon_{old} + \delta < 0.5) \\ (\varepsilon_{old} + \delta) - 1 & \text{if } (\varepsilon_{old} + \delta \geq 0.5) \end{cases} \end{aligned} \tag{4.20}$$

To remove floating-point operations from Equation 4.20, we rewrite the equation and replace the slope δ with y_e/x_e. Additionally, we multiply both sides of the equation by $2x_e$ and substitute $2\varepsilon x_e$ by ε'. It can be shown easily that ε' is always an integer. As a result of the substitution, we obtain the following:

$$\begin{aligned} y_{new} &= \begin{cases} y_{old} & \text{if } (\varepsilon'_{old} + 2y_e < x_e) \\ y_{old} + 1 & \text{if } (\varepsilon'_{old} + 2y_e \geq x_e) \end{cases} \\ \varepsilon'_{new} &= \begin{cases} \varepsilon'_{old} + 2y_e & \text{if } (\varepsilon'_{old} + 2y_e < x_e) \\ \varepsilon'_{old} + 2y_e - 2x_e & \text{if } (\varepsilon'_{old} + 2y_e \geq x_e). \end{cases} \end{aligned} \tag{4.21}$$

All expressions in Equation 4.21 are integer operations. These formulas can be transformed into the algorithm shown in Figure 4.15. The error variable ε' is represented by *eps*. We additionally optimize the code by introducing $dy = 2 * y_e - x_e$. We can now reduce the if-statement expression from $(eps + 2 * ye < xe)$ via $(eps + 2 * ye - xe < 0)$, $(eps + dy < 0)$ to $(eps < -dy$. Finally, we initialize $eps = dy$ instead of zero which further reduces the comparison to $(eps < 0)$.

4.5.2. Application for Bus Scheduling

The algorithm described in Section 4.5.1 is used to draw a line with a given slope. In this section we describe the transition from the basic line-drawing algorithm to an algorithm that can be used for bus scheduling.

4.5.2.1. Scheduling of two Devices

For bus scheduling of two devices, A and B, we assign the x-axis to device A, and the y-axis to device B. The arbitration is based on the following four rules:

Rule 1: The bus is granted to device B, if the y-value is incremented and device B has requested the bus.

```
x = y = 0;
eps = 2*ye - xe; // eps = dy
while (x < xe) {
    plot (x, y);
    if (eps < 0) {
        // chose right
        eps = eps + 2 * ye;
    } else {
        // chose up and right
        y = y + 1;
        eps = eps + 2 * ye - 2 * xe;
    }
    x = x + 1
}
```

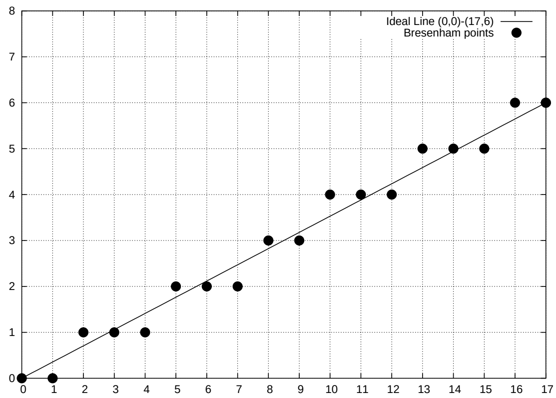
Figure 4.15.: Code for an integer-only Bresenham algorithm to draw a line between (0,0) and (xe, ye).

Rule 2: The bus is granted to device B, if the y-value is not incremented and device A has not requested the bus, but device B has requested the bus.

Rule 3: The bus is granted to device A, if the y-value is not incremented and device A has requested the bus.

Rule 4: The bus is granted to device A, if the y-value is incremented and device B has not requested the bus, but device A has requested the bus.

Figure 4.16 illustrates scheduling of two devices with a ratio of 11 : 6 (the slope is 6/17), where both devices are permanently requesting the bus. The left picture shows the result from the line drawing algorithm. The right picture is a simplified representation, where each device is represented by a direction. Here, an extension into a direction means granting bus access to the device assigned to this direction.



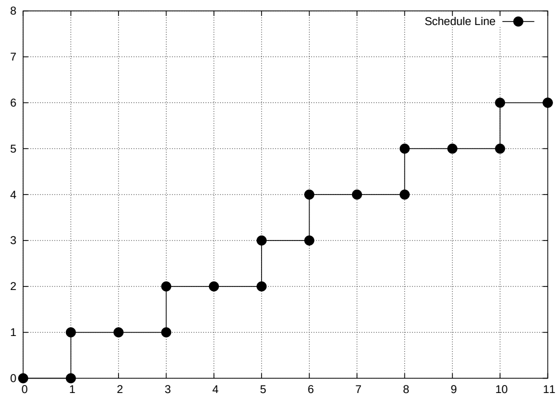


Figure 4.16.: Proportional PCI-bus arbitration based on the Bresenham algorithm. Share ratio between devices A and B is 11:6. Scheduling order is ABAABAABABAABAABA

Contrary to drawing a line, where x and y-value can be modified at every time, granting the bus to a device is only possible if the device has also requested the bus. This is similar to an artificial barrier that prevents the line from extending into one direction. In the case where we cannot extend into the calculated direction, the absolute error value ε increases and may leave the initially given valid interval of $[-0.5, 0.5)$. As soon as we can extend into the desired direction, the algorithm corrects the error as quickly as possible to reach the ideal position. In other words, if the device later issues frequent requests for the bus, it will receive the bus more often to adjust its actually received share to its reserved proportion. During the time a device receives the bus more frequently than actually planned, it affects the guaranteed bandwidth of other devices. We discuss solutions to control this behavior in Section 4.5.3.

For the following, we assume that the bus is given to two devices with a ratio of $n : m$, *i.e.*, assigning n shares to device A and m shares to device B. Since the x-value is incremented by the algorithm on every iteration, we draw the line between $(0, 0)$ and $(n+m, m)$, instead of a line between $(0, 0)$ and (n, m). Since n and m are both non-negative, it holds that $\frac{m}{n+m} \leq 1$. Hence, considering a line with an origin of $(0, 0)$ and a slope of less or equal to 1 is sufficient.

To show the operation of the algorithm, we consider a device ratio of 2 : 1. According to the previous paragraph, the slope is $\delta = \frac{1}{3}$. We consider the following three cases:

1. Both devices request the bus as often as possible. This is identical to the simple line drawing algorithm.

2. Device A is always requesting the bus. Additionally, device B requests the bus in step 4 and 6.

3. Device B is always requesting the bus. Additionally, device A requests the bus in step 4 and 6.

Table 4.2 lists the arbitration order and the intermediate results of ε_{old} and ε_{new} for the three cases.[3]

Table 4.2.: Example of PCI-bus arbitration with the new arbiter

	Case 1				**Case 2**				**Case 3**			
Step	REQ	ε_{old}	ε_{new}	Device	REQ	ε_{old}	ε_{new}	Device	REQ	ε_{old}	ε_{new}	Device
1	AB	0	1/3	A	A	0	1/3	A	B	0	-2/3	B
2	AB	1/3	-1/3	B	A	1/3	2/3	A	B	-2/3	-4/3	B
3	AB	-1/3	1/3	A	A	2/3	1	A	B	-4/3	-6/3	B
4	AB	1/3	-1/3	A	AB	1	1/3	B	AB	-6/3	-5/3	A
5	AB	-1/3	1/3	B	A	1/3	2/3	A	B	-5/3	-7/3	B
6	AB	1/3	-1/3	A	AB	2/3	0	B	AB	-7/3	-3/3	A

Scheduling of two devices is simple and does not require changes of the algorithm. In the next section, we describe how to adapt the algorithm to support more than two devices.

[3]For simplicity and better understanding, we show the intermediate values as a fraction and not as the integer value used by the optimized algorithm.

4.5.2.2. Scheduling of multiple Devices

To schedule more than two devices the algorithm must be extended to draw a line in multi-dimensional space. For more than three devices this seems complex since figures in a dimension higher than three are (usually) beyond human imagination. However, the step from an n-dimensional to an (n+1)-dimensional space is identical to the step from the two to the three-dimensional space. According to the four rules how devices are assigned in the two-dimensional space (see page 57), a decision is made between the first and the second device. However, if the first device is not scheduled, we cannot automatically pick the second device but have to decide, in the next step, between the second and the third device. This means that, with exception of the last device, we always have to decide for one device and cannot assume that a decision against a device automatically means a decision to pick the next device.

Adapted to the multidimensional-space case this generalizes to a decision either being made for one device, or left open and decided on the next *decision level*; the result is a *decision tree*. Figure 4.17 shows an example of such a decision tree for five devices A–E, where the decision is made on the third level granting the bus to device C.

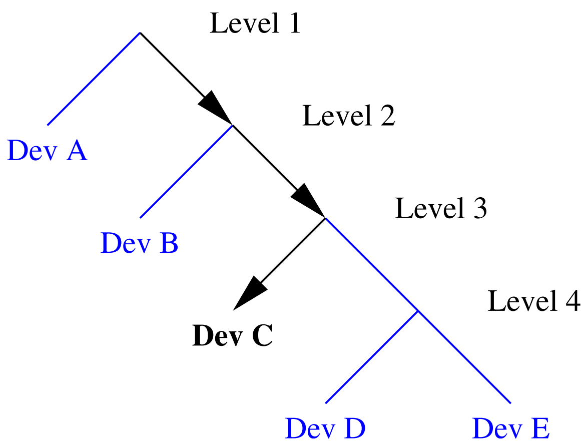


Figure 4.17.: Decision tree for five devices A–E

To obtain a bandwidth ratio of $m_1 : m_2 : ... : m_n$, the line must be drawn between $(0, 0, ..., 0)$ and $(m_1, m_1 + m_2, ..., m_1 + m_2 + \cdots + m_n)$. The single slope value δ of the two-dimensional case now becomes an array too, where each δ_x for n devices can be determined by:

$$\delta_x = \frac{m_x}{\sum\limits_{i=1}^{n} m_i}. \tag{4.22}$$

As a result, the algorithm generates a periodic schedule. The period P is equal to the sum of all shares:

$$P = \sum_{i=1}^{n} m_i. \tag{4.23}$$

Similar to the two-dimensional case, we show the line and the arbitration for three devices in Figure 4.18. The left figure shows the line generated by the Bresenham algorithm, the right figure shows which device gets the bus granted.

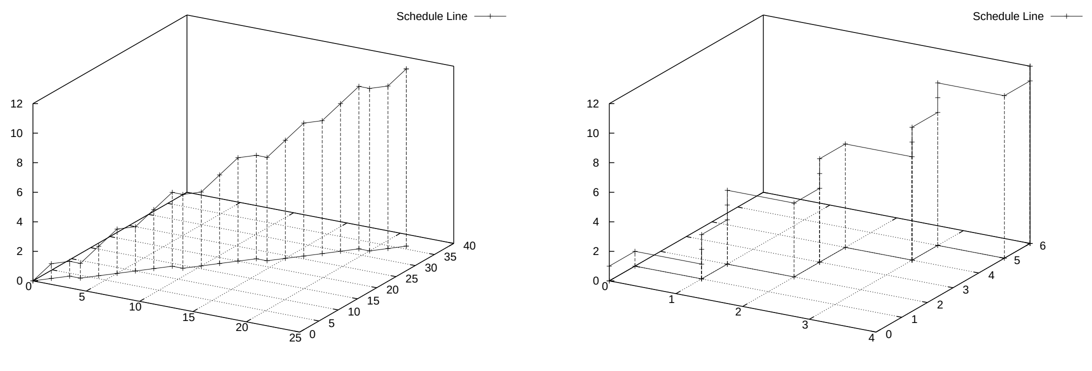


Figure 4.18.: PCI-bus arbitration algorithm with modified Bresenham algorithm, sharing ratio of the bus between three devices A (width), B (depth), and C (height) is 4:6:12

A line between $(0, 0, ..., 0)$ and $(m, 2m, ..., nm)$ provides a similar behavior as the actual implementation of a simple round-robin arbitration scheme, since all devices receive the same share of the bus. This allows us to keep all legacy systems relying on the current arbitration scheme of a non-prioritized round-robin arbitration.

4.5.3. The Memory Effect

In the previous section, we have described that a device that does not request the bus as often as allowed by its requested share will accumulate the assigned but unused share and eventually receive the bus more frequently to adjust the received share to the requested share. This behavior is caused by the absolute value of the error variable ε, which increases every time the device would have been scheduled but did not request the bus. According to the decision tree (Figure 4.17), a device will cut off all devices on a lower lever in the tree. For example, device B has not requested the bus for a while and is now permanently requesting the bus. Device A will still receive its correct share, but all remaining shares are given to device B until it has adjusted its share. As a result, devices C–E are completely cut off the bus. We call this behavior *memory effect* of a *memorized device* and the number of accumulated but unused bus grants *credit*. There are two possibilities to control the impact of such a memorized device: (1) by a *window*, which limits the time of the memory effect, and (2) by a *credit limit* that describes the maximum value of the credit.

In the first case, the credit is limited only by the maximum value a device can build up during the window, but will be reduced after the time determined by the window. If the credit is always retained for the same time, we have a *fixed-size window*; otherwise, we have a *variable-size window*. The simplest method of implementation is to completely cancel the credit after the time of the window has gone by. However, other alternatives are possible such as a timed reduction of the credit. In this case it is important that a credit cannot be built up quicker than it is reduced. Otherwise, the credit must be limited or other devices can be blocked for an unbounded time.

In the second case, the credit limit determines exactly how long a device can cut off other devices from the bus. Since the time during which the credit is valid is not time-bound, an established credit will never be canceled. Since it is very likely that the device will request access to the bus less often than the bus is reserved, the device will always keep a credit and the credit must be considered for all other reservations.

Both cases allow devices to adjust their bus usage to a jitter of an incoming data stream. The major advantage of the first case is that it also limits the resulting increased latency of other devices. The advantage of the second case is the easy implementation. In this thesis, we focus only on the second case.

4.5.4. Time-Sharing Devices

Until now, we have considered only devices with real-time requirements that have reserved a certain share of the bus. One of the initial requirements was also to support time-sharing devices that do not make reservations based on bandwidth requirements.

In the described technique, the bus is by default of the algorithm granted to the requesting device with the highest number, if the scheduler would actually pick a device that has not requested the bus. Therefore, devices should be sorted according to their requested share in descending order. As a result, devices with a larger share that are not ready when the scheduler would pick them give their shares to devices with smaller shares. Consequently, all time-sharing devices must set their shares to small, but non-zero values, which means that time-sharing devices with small shares will receive all bandwidth not used by real-time devices with larger shares.

If a certain share should always be distributed among time-sharing devices, we need to add an 'artificial device,' as placeholder between real-time and time-sharing devices that requests the total fraction of all time-sharing devices but never requests the bus. Every time the scheduler would select the artificial device, one of the time-sharing devices with lower share is selected instead.

For example suppose, we have three real-time devices that should share the bus by 50%, 30%, and 20%; unused bandwidth should be distributed evenly among the time-sharing devices. We set the values for the real-time devices to $m_1 = 500$, $m_2 = 300$, and $m_3 = 200$. All time-sharing devices receive values of 1. Strictly speaking, this assigns each time-sharing device a (negligible) bandwidth of 1 out of 1003 requests. The arbiter schedules the bandwidth in the requested ratio among the real-time devices as long as they request the bus.

However, if none of them requests the bus, but the time-sharing devices do so, the bandwidth is evenly distributed among the time-sharing devices. In contrast, assigning a share of zero to the time-sharing devices would not result in such an even distribution. Instead, the first time-sharing device would get always access to the bus when no real-time device has requested the bus. The second time-sharing device would only get access to the bus, if the first time-sharing device does not request the bus. Finally, the third time-sharing device would only receive bus access when no other device tries to access the bus.

4.5.5. Arbiter Implementation

In the following sections we describe an implementation of our proposed real-time arbiter in software as well as in hardware. The software solution is also used in the "PCItrace" simulation environment.

4.5.5.1. Software Implementation

An implementation of the algorithm for the n-dimensional space is shown in Figure 4.19. The values for the line endpoint that describes the ratio among the devices are held in the `points[n]` array. The `eps` error value, which was a scalar in the two-dimensional version is now an (n-1)-dimensional array `eps[n-1]`. Its initial values are calculated from the points of two adjacent dimensions, *i.e.*, `eps[i]` is calculated by `2*points[i+1] - points[i]`. This calculation is done in the *initialization* part.

We have split the simple line drawing algorithm into two parts, a *calculation* and an *update* part.

In the calculation part, new `eps` values for all devices are determined. If a device's `eps` value becomes negative, the device is due to be scheduled and the appropriate bit in the `arb` variable is set. Since we scan through all devices, multiple bits in the `arb` variable may be set.

In the update part, again we scan through all devices, beginning from the device with the lowest number. We keep the device with the highest number that has requested the bus in the `grant` variable, independent of the choice of the scheduler. This allows us to give the bus to devices that have requested the bus, but that were not selected by the scheduler. The first device that was selected by the scheduler and has also requested the bus is scheduled. A device that was selected by the scheduler but has not requested the bus cannot be scheduled. The `eps` variables of all devices with a higher number than the scheduled device are not updated. If no device has been found, the bus is idle.

4.5.5.2. Implementation in Hardware

One of our design goals was to find an algorithm that can easily be implemented in hardware. The complexity of the presented algorithm is $O(n)$, since for a full calculation cycle the values for n devices must be calculated. This is practicable but not efficient. To reduce the complexity we must parallelize the algorithm.

This leads to the following design illustrated by the code in Figure 4.20. The function within the loop modifies the `eps` value for each device, depending on the sign of the current `eps` value; a positive value is represented when the highest bit is zero (Line 1). In this case, temp1 is the value of `points[cur]`, otherwise it is zero (Line 2). In the next step, we subtract this value from `points[cur+1]` (Line 3). To multiply the result by two, we only need to add an always-zero wire as the lowest bit (Line 4). Finally, we have to look at the break condition of the algorithm. The loop is left when one `eps` has a negative value. In our parallelized version, `eps` must not be updated if the value of all previous devices is negative (Line 5).

The logical-AND operation of step 2 means that every bit of `points[cur]` is logically ANDed with the single `upd[cur]` bit.

The logic block is designed such that it only updates the `eps[cur]` values if a write-enable line is asserted. The calculation can be done within one cycle that is needed to delay the update of the original `eps` value. For the initial block (cur=0), the write-enable line is always asserted, since the algorithm always updates the first `eps` value. The final decision of which device is scheduled is based on the highest active write-enable line. Figure 4.21 shows the block diagram of a single logic block of the new proportional-share arbitration algorithm. This implementation has a complexity of $O(1)$.

The initialization of the `points` array is either done by software during setup

```
INITIALIZATION:
    // Initialize point values from given ratio
    sum = 0;
    for (cur = DEVS - 1; cur >= 0; cur--) {
        sum = sum + ratio[cur];
        points[cur] = sum;
    }
    // Initialize eps values from points
    for (cur = 0; cur < DEVS - 1; cur++) {
        eps[cur] = (2 * points[cur + 1]) - points[cur];
    }

ALGORITHM:
    // Always grant the bus to the last device
    arb = (1 << (DEVS-1));
    for (cur = 0; cur < DEVS - 1; cur++) {
        if  (eps[cur] < 0) {
            // device shall get the bus
            eps_new[cur] = eps[cur] + 2 * points[cur + 1];
            arb = arb | (1 << cur);
        } else {
            // device shall not get the bus
            eps_new[cur] = eps[cur] + 2 * points[cur + 1] - 2 * points[cur];
        }
    }
UPDATE:
    grant = BUSIDLE; // Assume the bus will be idle
    for (cur = 0; cur < DEVS; cur++) {
        eps[cur] = eps_new[cur];
        if (req & (1 << cur)) {
            // If device has requested the bus, keep it
            grant = cur;
            if (arb & (1 << cur)) {
                // If the device shall receive the bus, grant the bus
                break;
            }
        }
    }
    return grant;
```

Figure 4.19.: C-code for proportional-share PCI-bus arbiter for multiple (DEVS) devices

```
(1) upd[cur] = MSB (eps[cur])
(2) temp1 = upd[cur] ∧ points[cur]
(3) temp2 = points[cur + 1] - temp1;
(4) temp3 = temp2 SHIFT-LEFT 1;
(5) eps[cur] = WRITE-IF[!(upd[cur - 1] ∨ ··· ∨ upd[0])] temp3;
```

Figure 4.20.: Pseudo-code to describe each logic block of the new arbiter

or due to changes in the bandwidth reservation of the schedule. The `eps` values are set after changes in the points array either by software or by hardware to `eps[n] = 2*points[n+1] - points[n]`.

The implementation of the fixed-size window to reduce the memory effect requires only

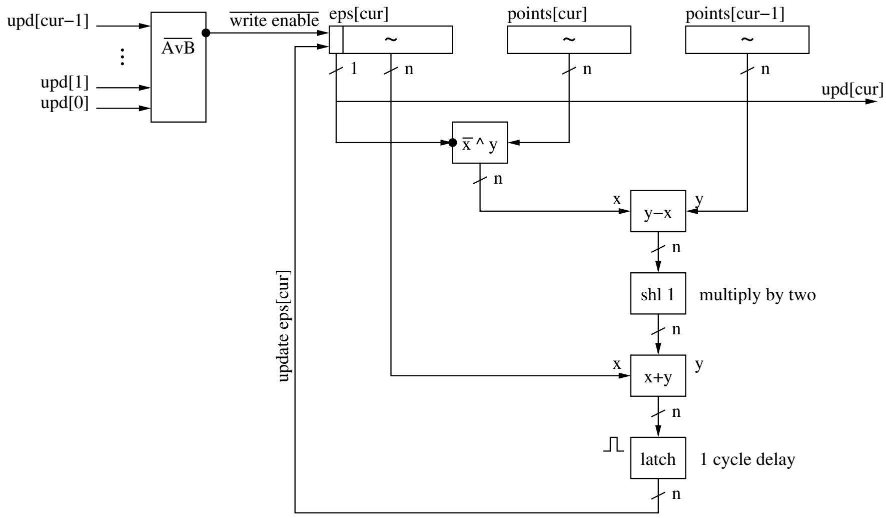


Figure 4.21.: Block diagram for proportional-share arbiter

few changes. The `eps` updates are not on the critical scheduling path and can be performed in the next PCI-bus cycle after the arbitration, while the bus transaction takes place. If the values in the `eps` registers exceed or fall below the acceptable deviation limit, the `eps` values are updated to stay in the requested range, *i.e.*, they are set to the maximum allowed values. The compare-and-set operations for lower and upper limit can be done in parallel since only one result is used. Which result is written back is determined by the compare operation. Figure 4.22 illustrates the logic block of one `eps`-update circuit.

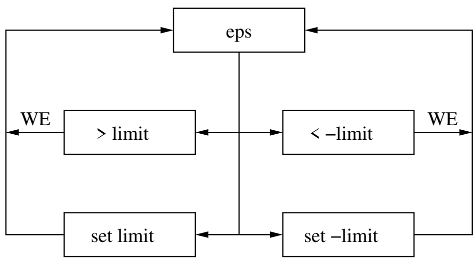


Figure 4.22.: Block diagram for `eps`-update logic

To allow the system to sort devices in arbitrary order, we need an additional small logic block. By means of an $n \times n$ matrix, the order of devices, physically determined by their location on the bus, can be translated into a virtual order used for scheduling and vice versa. The same matrix is used to translate the virtual order back into the physical bus order.

4.5.6. Support for Soft Reservations

The proposed scheme can easily be extended to support a combination of soft and hard reservations. We only need to double the proposed hardware, resulting in two separate arbiters and two request/grant-line pairs for each device. A device uses the first pair of request/grant lines to acquire bandwidth for its hard reservation and the second pair to get soft-reserved bandwidth. To guarantee a certain share of the bus bandwidth to be given to soft reservations, a proxy device P is added to the arbiter that handles hard reservations. This bandwidth is distributed among the devices with soft reservations according to their soft shares. Figure 4.23 illustrates this scheme.

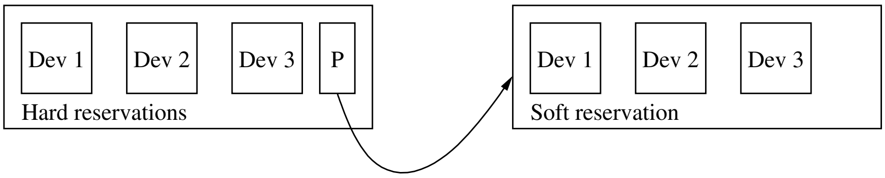


Figure 4.23.: Block diagram supporting hard and soft reservations

For example, a hard reservation tuple of (5 : 2 : 1 : [2]) and a soft reservation tuple of (6 : 3 : 1) distributes 80% of the total bus bandwidth among hard reservations and reserves 20% of the bus bandwidth for soft reservations. In addition to the reserved 20%, all bandwidth which remains unused by hard reservations is distributed among devices with soft reservations in a 6 : 3 : 1 ratio.

4.5.7. Bandwidth and Latency Considerations

In this section we describe how the share parameters must be calculated such that a device obtains a certain bandwidth and how the worst-case latency of devices can be determined

The described proportional-share algorithm has no knowledge about s and d properties of a device. It considers only time where the bus is occupied, *i.e.* $s + d$ as a whole. Since data is only transferred during time d but not during s, the achieved bandwidth is lower. Therefore, we must adapt the requested share if a certain bandwidth must be achieved. It must hold that the total bus utilization is less than or equal to 1:

$$U = \frac{1}{bw_{pci}} \sum_{i=1}^{n} bw_i \frac{s_i + d_i}{d_i} \leq 1. \tag{4.24}$$

Additionally, it must hold that device x is able to generate the required bandwidth bw_x:

$$bw_x \leq bw_{pci} \frac{d_x}{s_x + d_x + r_x}. \tag{4.25}$$

To calculate the shares of each device, we assume that a device generates as much bandwidth as it can and that we have to throttle its access to the bus. If all devices would generate only exactly the requested amount of bandwidth and Equation 4.24 holds, all devices would receive their requested amount of bandwidth, even under a round-robin algorithm. The δ_x value of a device determines its share of the bus. The proportional-share algorithm is designed to share the entire bandwidth among all devices. However, sharing the entire bus bandwidth (bw_{pci}) is often not the intention and devices want to reserve only a certain amount of bandwidth. Hence, we introduce a dummy device with $\mathcal{D}_\zeta = (1, 0, 0)$. This dummy device has only one non-data cycle to absorb all remaining bandwidth. In a real system, where devices request the bus less frequently and generate less bandwidth, the remaining bandwidth can be given to time-sharing devices that have not made a bandwidth reservation. The dummy device is merely a help to calculate the correct shares. Since no bandwidth can left unused, following must hold:

$$\sum_{i=1}^{n} \delta_i = 1 - \zeta. \tag{4.26}$$

The bandwidth each device receives is:

$$bw_x = bw_{pci} \frac{\delta_x d_x}{\zeta + \sum\limits_{i=1}^{n} \delta_i (s_i + d_i)}. \tag{4.27}$$

4.5.8. Results and Simulation

To verify these results, we give the following example. Three devices $\mathcal{D}_1$=(5,8,3), $\mathcal{D}_2$=(3,16,4), and $\mathcal{D}_3$=(10,12,4) require the bus for 6MB/s, 16MB/s, and 8MB/s respectively.

We derive the following equation system from Equation 4.26 and 4.27:

$$\begin{array}{rcrcrcrcl} 6\zeta & - & 978\delta_1 & + & 114\delta_2 & + & 132\delta_3 & = & 0 \\ 16\zeta & + & 208\delta_1 & - & 1808\delta_2 & + & 352\delta_3 & = & 0 \\ 8\zeta & + & 104\delta_1 & + & 152\delta_2 & - & 1408\delta_3 & = & 0 \\ \zeta & + & \delta_1 & + & \delta_2 & + & \delta_3 & = & 1 \end{array} \tag{4.28}$$

Solving this equation system leads to the following values: $\delta_1 = 0.00824$, $\delta_2 = 0.01099$, $\delta_3 = 0.00733$, and $\zeta = 0.97344$.

To verify these values, we perform a simulation with our "PCItrace" bus simulation tool. We set the share values of the devices to $m_1 = 824$, $m_2 = 1099$, and $m_3 = 733$. The share value of the dummy device is $m_4 = 97344$. The simulation run of 1,000,000 cycles—this is about 30ms on the PCI bus—takes about 2.3s on a 600MHz Pentium III system. Figure 4.24 shows the results generated by the "PCItrace" tool.

The bus-access pattern for the three devices is shown in Figure 4.25.

Applying Equation 4.25 to the descriptors of each device shows that the device's achievable bandwidth is much higher than the requested bandwidth. The smaller the recovery value, the smaller the probability that due to contention a device does not deliver the requested bandwidth. To explore this relation, we calculate the recovery values under which the devices would still be able to generate the required bandwidth if no contention occurs at all. We determine $r_{x,max}$ by the following

$$r_{x,max} = \frac{bw_{pci} d_x}{bw_x} - s_x - d_x, \tag{4.29}$$

resulting in $r_{1,max} = 163$, $r_{2,max} = 113$, and $r_{3,max} = 176$. To verify this result, we perform the same simulation, but set the recovery values to these calculated maximum values. Since now the total bus bandwidth is completely consumed by the three devices, the dummy device is not needed anymore.

The results of the listing in Figure 4.26 shows that even under these maximum recovery values, the requested bandwidth can be guaranteed.

Figure 4.27 shows the simulated bus-access pattern for the devices with the maximum recovery values under which devices achieve the requested bandwidth.

```
Arbitration scheme                    : Proportional Share
------------------------------------------------------------
Statistics of Device 1 [Device 1]
Burst max 32 bytes, stall 5, recovery 3

Maximum single card bandwidth         :  66.00MB/s
Simulated Bandwidth                   :   6.03MB/s

------------------------------------------------------------
Statistics of Device 2 [Device 2]
Burst max 64 bytes, stall 3, recovery 4

Maximum single card bandwidth         :  91.82MB/s
Simulated Bandwidth                   :  15.94MB/s

------------------------------------------------------------
Statistics of Device 3 [Device 3]
Burst max 48 bytes, stall 10, recovery 4

Maximum single card bandwidth         :  60.92MB/s
Simulated Bandwidth                   :   8.08MB/s

------------------------------------------------------------
Statistics of Device 4 [Dummy device]
Burst max 0 bytes, stall 1, recovery 0

Simulated Bandwidth                   :   0.00MB/s

============================================================
Simulated Total Bandwidth             :  30.05MB/s
Bus Idle                              :   0.00%
Bus Contention                        : 100.00%
```

Figure 4.24.: “PCItrace” simulation run for example

In contrast to the previous simulation, where achieved and requested bandwidth are identical, setting the recovery values to the maximum possible values achieved bandwidth that is slightly lower than the requested bandwidth. The reason for this behavior is that all devices now achieve the requested bandwidth only when they operate under their maximum theoretical bandwidth without any bus contention. However, as soon as the access of multiple devices must be coordinated and contention may occur, the resulting bandwidth of each device is slightly lower. Hence, to achieve the requested bandwidth devices must not operate at their theoretically possible bandwidth limit.

The worst-case access latency is determined by the maximum time other devices occupy the bus. We can estimate the worst-case latency as follows: a device x that receives m_x-times during the period P receives the bus m_x-times exactly at the beginning of the period. Hence, the maximum distance between two requests Δ is the following

$$\Delta_x = \sum_{i \neq x} m_i \tag{4.30}$$

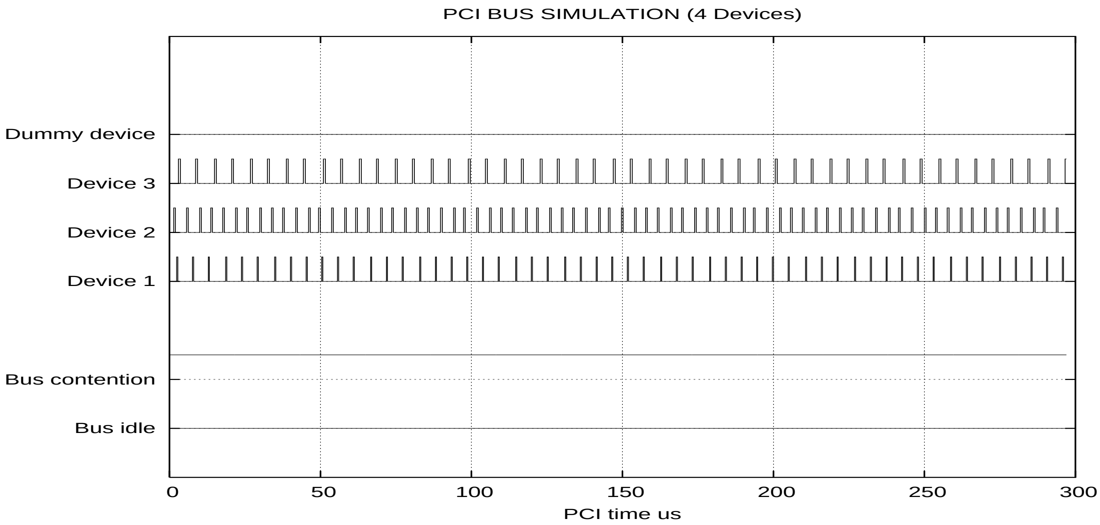


Figure 4.25.: Simulated bus-access pattern for the example devices

```
Arbitration scheme                     : Proportional Share
------------------------------------------------------------
Statistics of Device 1 [Device 1 (r-max)]
Burst max 32 bytes, stall 5, recovery 163

Maximum single card bandwidth          :   6.00MB/s
Simulated Bandwidth                    :   5.81MB/s

------------------------------------------------------------
Statistics of Device 2 [Device 2 (r-max)]
Burst max 64 bytes, stall 3, recovery 113

Maximum single card bandwidth          :  16.00MB/s
Simulated Bandwidth                    :  15.94MB/s

------------------------------------------------------------
Statistics of Device 3 [Device 3 (r-max)]
Burst max 48 bytes, stall 10, recovery 176

Maximum single card bandwidth          :   8.00MB/s
Simulated Bandwidth                    :   8.08MB/s

============================================================
Simulated Total Bandwidth              :  29.83MB/s
Bus Idle                               :  67.34%
Bus Contention                         :   4.36%
```

Figure 4.26.: “PCItrace” simulation run with calculated maximum recovery values

The worst-case time each device x must wait is:

$$\tau_{lat,x} = \sum_{\neq x} \Big(m_i (s_i + d_i) \Big) \tag{4.31}$$

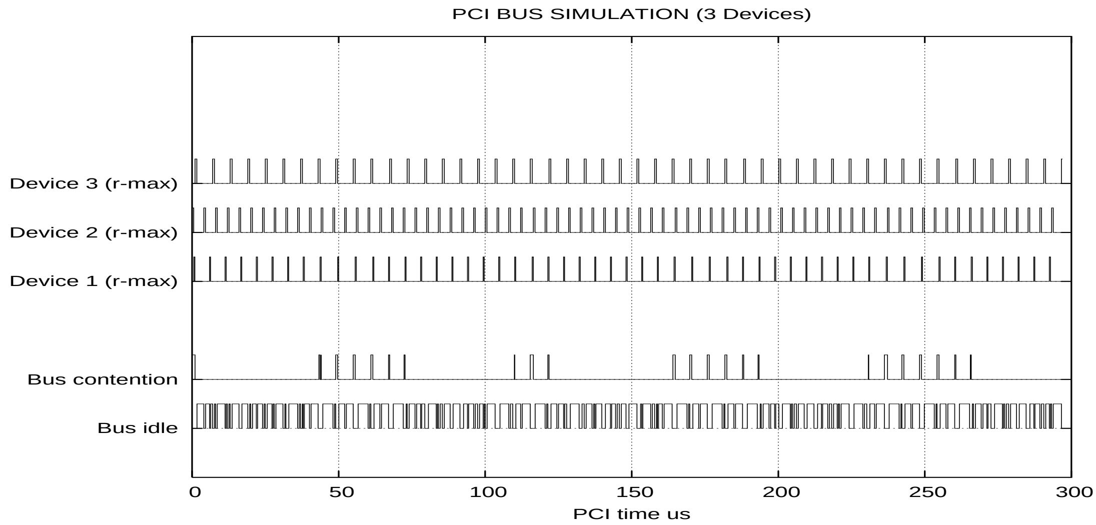


Figure 4.27.: Simulated bus-access pattern for the example devices

This worst-case estimation is relatively pessimistic, especially for evenly-sized shares with larger numbers. Based on the properties of the algorithm, exact worst-case-latency values for two devices can be calculated; for multiple devices, simulation must be used.

For the two-devices case, the maximum time during which the one device will not receive the bus occurs when the error variable ϵ of the original algorithm is initially set to -0.5. The earliest time at which the device will receive the bus is when the error value reaches 0.5 (see also Figure 4.14). With each arbitration cycle, the error variable is incremented by the share δ (see also Equation 4.19). Let Δ now be the number of arbitration cycles during which the algorithm does not assign the bus to a device. To obtain the maximum number of arbitration cycles, the following must hold

$$-0.5 + \Delta\delta \geq 0.5. \tag{4.32}$$

Since the number of arbitration cycles is discrete, the value for Δ_x for device x can be calculated by

$$\Delta_x = \left\lceil \frac{1}{\delta_x} \right\rceil - 1. \tag{4.33}$$

The resulting worst-case latency is

$$\begin{aligned}\tau_{lat,x} &= \Delta_x\Big(\sum_{i\neq x}^{2}(s_i + d_i)\Big) \\ &= \Delta_x\Big(s_{3-x} + d_{3-x}\Big)\end{aligned}$$

4.6. Conclusions

In this chapter we presented a model to describe the behavior of PCI-bus devices. By splitting a single PCI transaction into three phases, non-data (s), data (d) and recovery (r), we can calculate worst-case bandwidth as well as worst-case latency of each device. Additionally, we can determine the maximum bandwidth a device is able to deliver.

As shown in Section 4.3.3, the latency timer that each master must implement cannot be used effectively to restrict the bandwidth a device can generate. Since an effective restriction is a fundamental requirement for giving bandwidth guarantees, only a new arbiter can effectively solve this problem.

Based on the Bresenham line-drawing algorithm, we developed a novel arbiter to assign arbitrary shares of the bus bandwidth to individual devices. The proposed algorithm has a complexity of $O(n)$, which has been reduced by parallelizing to $O(1)$. The functionality required for each logic block is simple enough that arbitration can be made within one PCI-bus clock cycle. By chaining two arbiters, a mixed hard/soft-reservation scheme, as already used for other resources by Drops, can be implemented. While the first arbiter handles all hard reservations, the second arbiter is responsible for all soft reservations. A proxy device in the first arbiter ensures that a certain fraction of the bus bandwidth is always given to soft reservations.

The new arbitration algorithm not only distributes bandwidth among devices but guarantees also a worst-case latency. The new arbitration algorithm achieved good results in our simulation environment.

5. Results — Conclusions — Future Work

This chapter summarizes the research results of this thesis, draws conclusions and discusses open points for future work.

5.1. Thesis Results

Chapter 3 discusses the influence of memory-bus load on an application accessing main memory executed on the host processor. This load can be generated either by other processors or by devices on peripheral buses. Since we have restricted our research to uniprocessor, PCI-bus-based systems, the memory-bus load is generated only by devices on the PCI bus. As a result, the execution time of the application increases under PCI load. To quantify this impact, we introduced the slowdown factor in Sections 3.1. The slowdown factor describes the ratio between the execution time of an application under external load and under no external load. In an empirical approach, measurements under various external loads are taken to determine the slowdown factor. For reservation schemes that consider worst-case execution times—as in hard real-time systems—the worst-case impact is of interest. By increasing the external load to the maximum possible bandwidth, the worst-case slowdown factor of an application can be measured. The drawback of the empirical approach is that many measurements on a platform must be taken. Two factors determine the slowdown factor of an application: (a) the characteristics of the application, and (b) the influence of external load on memory-sensitive processor instructions.

The characteristics of an application can be described by the instruction mix introduced in Section 2.6.2. The worst-case impact on memory-sensitive processor instruction is described by the upper-bound worst-case slowdown factor (WCSF). This upper bound can be determined by measuring the impact of maximum external load on an application accessing only memory. A combination of the instruction mix and the upper-bound WCSF is used to determine the application's worst-case slowdown factor.

To determine the slowdown factor under a certain load, we replace the upper-bound WCSF by the slowdown factors of read and write operations under the given external load. These two slowdown factors can be either taken from measurements or approximated by a polynomial function of order two. The results of the approximation by a polynomial function are very precise.

In Chapter 4, we focus on the PCI bus, especially its use in soft real-time systems such as Drops. Many publications in the embedded real-time community claim that the PCI bus is an alternative to current bus systems for use in real-time environments. However, they do not substantiate this claim by any means. Therefore, we analyzed standard off-the-shelf PCI-bus host bridges to determine their behavior in terms of arbitration, latency and bandwidth.

Starting from the models given by the PCI-bus specification, we derived a simplified model that allows us to describe the behavior of devices on the bus. Each device can be described by a triple $\mathcal{D} = (s, d, r)$: s describes the number of non-data cycles per transaction, d gives the number of data cycles, and finally, r describes the minimum time a device requires between two transaction. The model is powerful enough to calculate latencies and bandwidths of individual devices and can also be used to perform PCI-bus simulations.

All analyzed bridges provide only a simple round-robin arbitration scheme; none of the analyzed bridges support any kind of bandwidth reservation to guarantee a certain bandwidth to a device. Based on our model, we can determine the minimum bandwidth a device can achieve under worst-case, and the maximum bus-access latency a device can experience. This is sufficient to use current PCI-bus systems, especially in low-bandwidth real-time environments, but it does not efficiently handle the requirements of high-bandwidth systems where bandwidth reservation is important. Under the conditions that bus utilization is less than 100% and that all devices behave well and cooperate, the PCI bus can be used in real-time environments. Here, the calculated worst-case values are very pessimistic

If devices can generate higher bus bandwidth and the resulting utilization would exceed 100%, regulation is mandatory to guarantee real-time capabilities and enforce resource reservation. Hence, we designed a new arbiter that provides such bandwidth guarantees, but which retains the advantages of low latency, by a design that allows an easy implementation in hardware. This arbiter also allows us to keep the behavior of current round-robin-based PCI host bridges.

One design goal was to support both hard and soft reservations. Doubling the hardware leads to an arbiter that allows each device to make two reservations, one for mandatory and another for optional bandwidth. This allows us to provide the required support for systems such as those based on imprecise computations or on quality-assuring scheduling.

To test and verify our results under current hardware as well as with the new arbitration scheme we developed the simulation tool "PCItrace."

5.2. Conclusion

We can conclude from Chapter 3 that any external load on the memory bus delays the execution of an application. The factor by which the execution is slowed down depends on the ratio of bandwidth available to the processor, the main-memory interface, and the bandwidth of the external source, in our case the PCI bus. Compared to earlier measurements on slower systems, such as Pentium machines with 90MHz, the impact on modern systems is much lower since memory and local bus became much faster but the PCI bus still operates at 33MHz. However, with new technologies such as PCI Express or InfiniBand providing higher bandwidth to devices, the impact will increase again. Additionally, on a multiprocessor systems, not only the PCI host bridge but each individual processor is a source of memory-bus bandwidth. In this case, we expect a much higher impact than in our case.

The conclusion of Chapter 4 is that the design of the PCI bus is flexible enough to be deployed in real-time environments. The advantages of the round-robin arbitration scheme used by current host bridges are simplicity and low arbitration costs. However, it is too simple to provide adequate features for using the PCI bus in real-time systems where overload can occur. Our simple yet powerful design of a new arbiter provides support for soft and hard bandwidth reservation and keeps the costs for arbitration low.

5.3. Future Work

This thesis opens up many opportunities for future work. Describing the memory behavior of an application based on its instruction mix has been verified on modern processors. Extensions were made to cover unpredictabilities such as multi-level caches, translation look-aside buffers, branch predictions, and so on. It remains open how future architectures with speculative execution, processors with explicit instruction-level parallelism such as Intel's Itanium, or systems with simultaneous multi-threading (SMT) can be described by this technique. Additionally, we have to consider multiprocessor systems, where multiple processors can generate memory-bus load. In such systems, we expect a much higher impact on applications.

We have designed a hardware solution for the new arbiter. However, we could not implement this arbiter in real silicon. Future work needs to be done to integrate the new arbiter into a host bridge, especially if other buses such as AGP are present. Additionally, further research is also required to integrate into current processor scheduling schemes PCI-bus bandwidth reservation and its impact on applications executed by the host processor.

A. Glossary and Abbreviations

Admission control: Test to allow or deny addition of new applications to the system. The result is a yes-no decision and not a set of scheduling parameters.

Arbiter: Bus unit that manages all requests for the bus and grants bus access to one device.

Arbitration: Process of selecting one from a set of bus-requesting devices and assigning bus access to it.

Alpha, AXP: 64-bit architecture designed by Digital Equipment Corporation (DEC) based on the 21x64 Alpha processor.

Backside bus: Connects the processor core and the second-level cache. Operates with the full processor speed, *i.e.*, the core frequency (determined by the ≻*frontside bus*) multiplied by the CPU's internal multiplier.

Bresenham algorithm: A fast, integer-only algorithm to draw a line between two points. Originally invented for (two-dimensional) plotting devices.

Bridge: Couples two (or more) identical or different buses. Examples are PCI-to-PCI bridge, or PCI-to-ISA bridge.

Bus master: PCI-bus device that can initiate a transfer.

Bus scheduling: Combines the ≻*arbitration* and the calculation of scheduling parameters to achieve a requested goal such as assured bandwidth or maximum latency.

Contention: The state where the bus is occupied by one device and one or more devices try to acquire the bus.

Deadline: Point in time by which an operation must be successfully completed to achieve a valid result.

Delayed transactions: Transactions introduced to reduce the number of cycles the PCI bus is held by a device, without transferring data. They allow starting a transaction and completing it later. Other bus masters are allowed to use the bus bandwidth that would be wasted if the originating bus master were to hold the bus in wait states. Delayed transactions were designed for slow devices and are only used by bridges or I/O controllers.

External load: Specific to a processor. All load on the ≻*frontside bus* that is not generated by this processor. In the uniprocessor case, all external load is generated by I/O devices.

Frontside bus: Also known as local bus or memory bus. Connects the processors and the ≻*host bridges* with main memory. Data read from or written to physical main memory passes the frontside bus. The frontside-bus speed determines the basic core frequency of the CPU. The memory bus is often proprietary and designed in conjunction with the processor.

Form factor: Standardizes the physical dimension of a device.

Host bridge: Connects the I/O bus, such as PCI or VMEbus to the ≻*frontside bus*.

IA-32: The 32-bit Intel Architecture, formerly known as x86. IA-32 CPU models are Pentium 4, Pentium III, Xeon, Celeron, but also Pentium II, Pentium, as well as i486 and i386.

Internal load: Specific to a processor. It describes the load on the ≻*frontside bus* that is generated by this processor.

Latency: The time between the triggering of an event and the response to that event. For buses, latency describes the time between requesting the bus and granting the bus to the device.

Latency timer: A programmable timer on each PCI card that limits the maximum tenure on the PCI bus during heavy bus traffic for this card.

PCI command: The PCI bus allows 16 possible commands to drive the bus, four of which are reserved. The defined commands are: Memory Read, Memory Read Line, Memory Read Multiple, Memory Write, Memory Write and Invalidate, I/O Read, I/O Write, Configuration Read, Configuration Write, Interrupt Acknowledge, and Special Cycle.

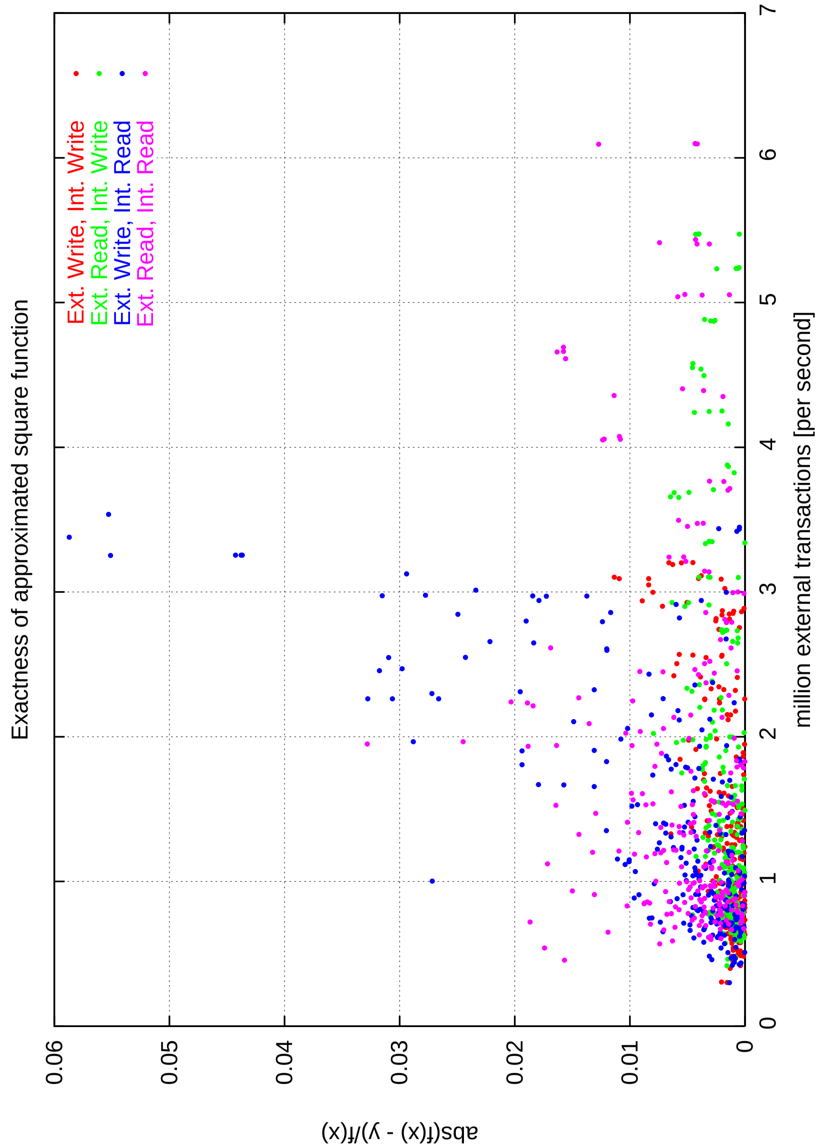
Proportional-share algorithm: An algorithm that assigns resources in a fixed ratio. The entire resource is shared among all clients.

Slowdown factor: Describes the ratio of the execution time of an application on the host processor between an idle system and a system under I/O load.

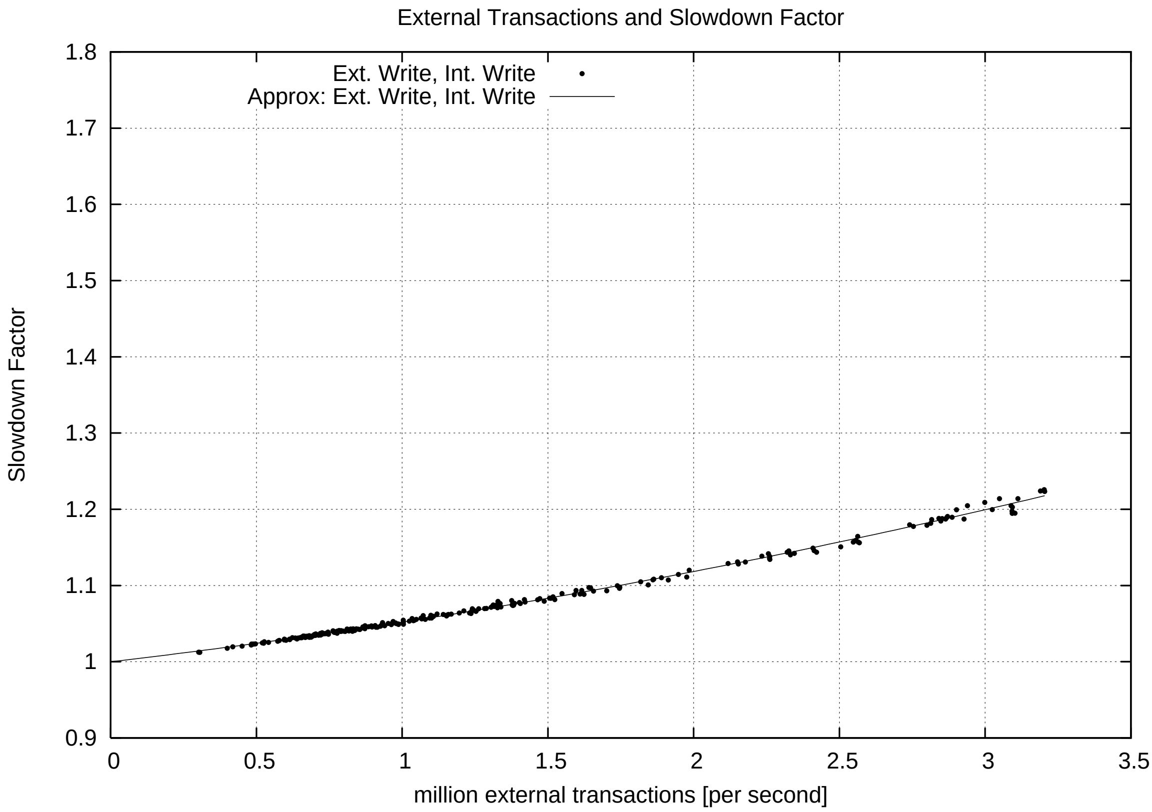
Upper-bound worst-case slowdown factor: The ≻*worst-case slowdown factor* of an application that is most sensitive to ≻*external load*, *i.e.*, performs only memory-sensitive instructions.

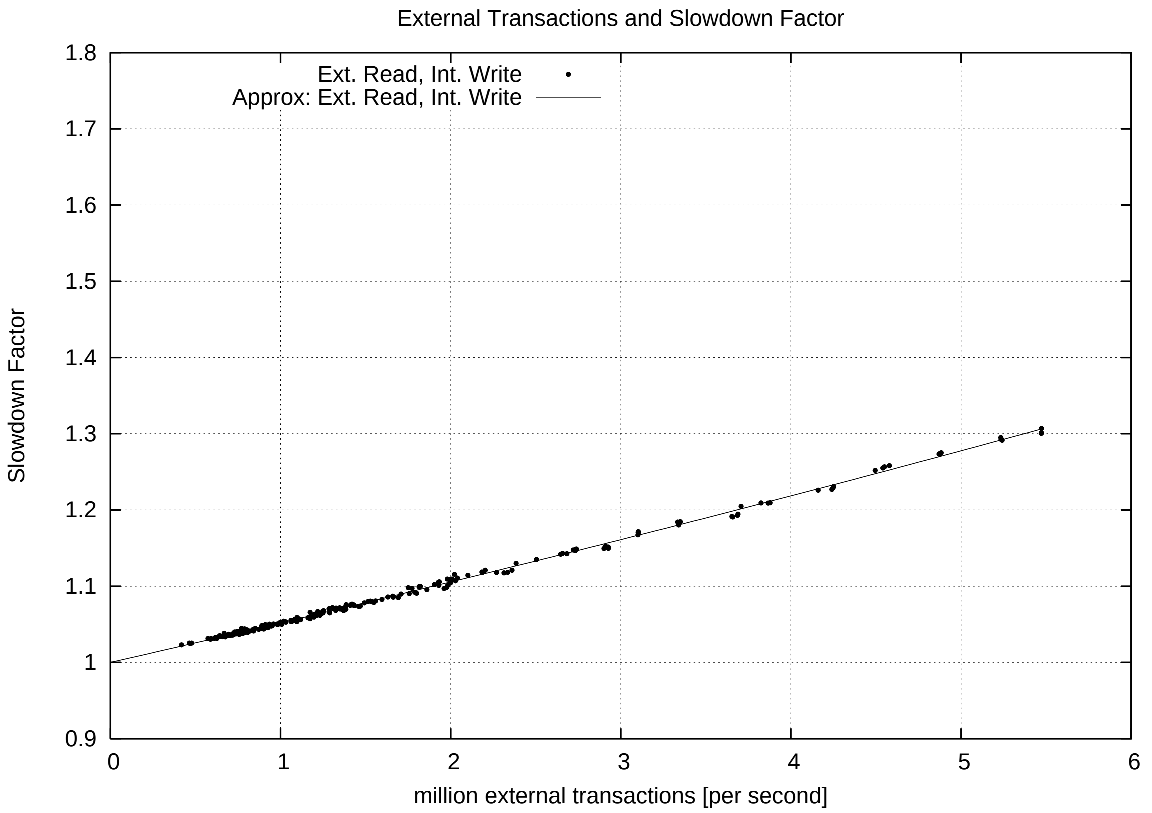
Worst-case slowdown factor: An application-specific value that describes the maximum possible impact of ≻*external load* on an application.

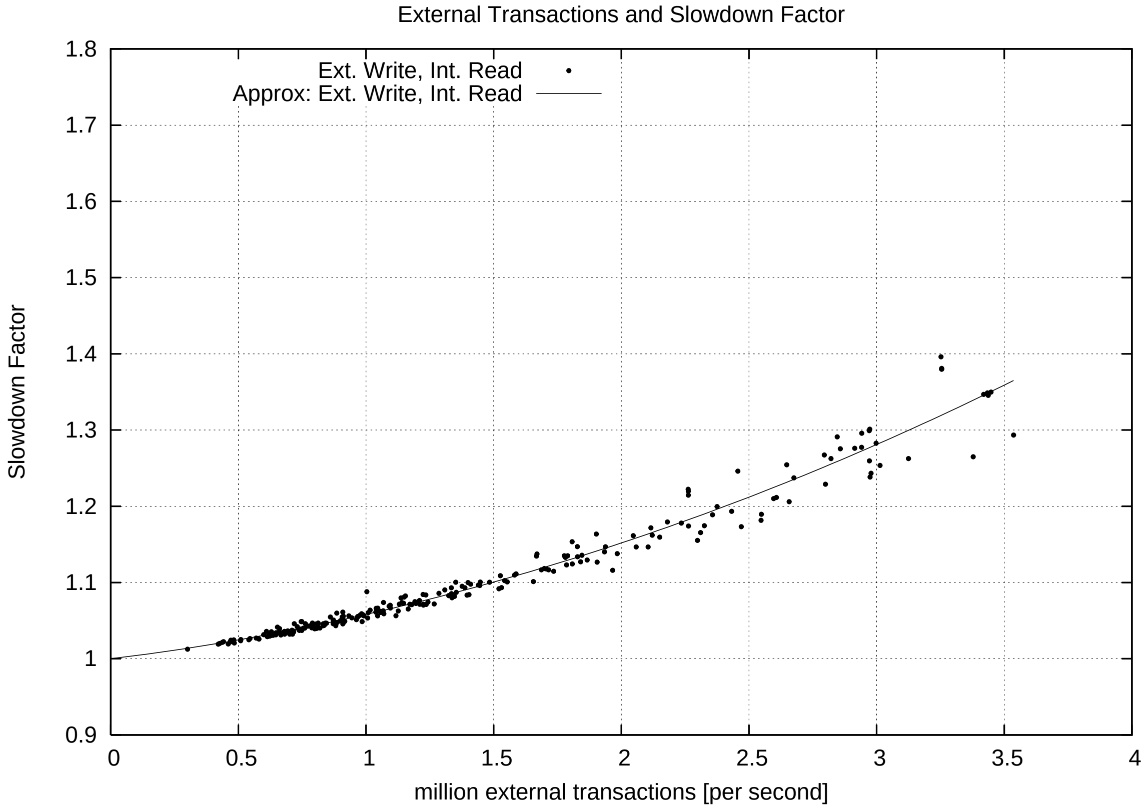
B. Polynomial Approximation Functions

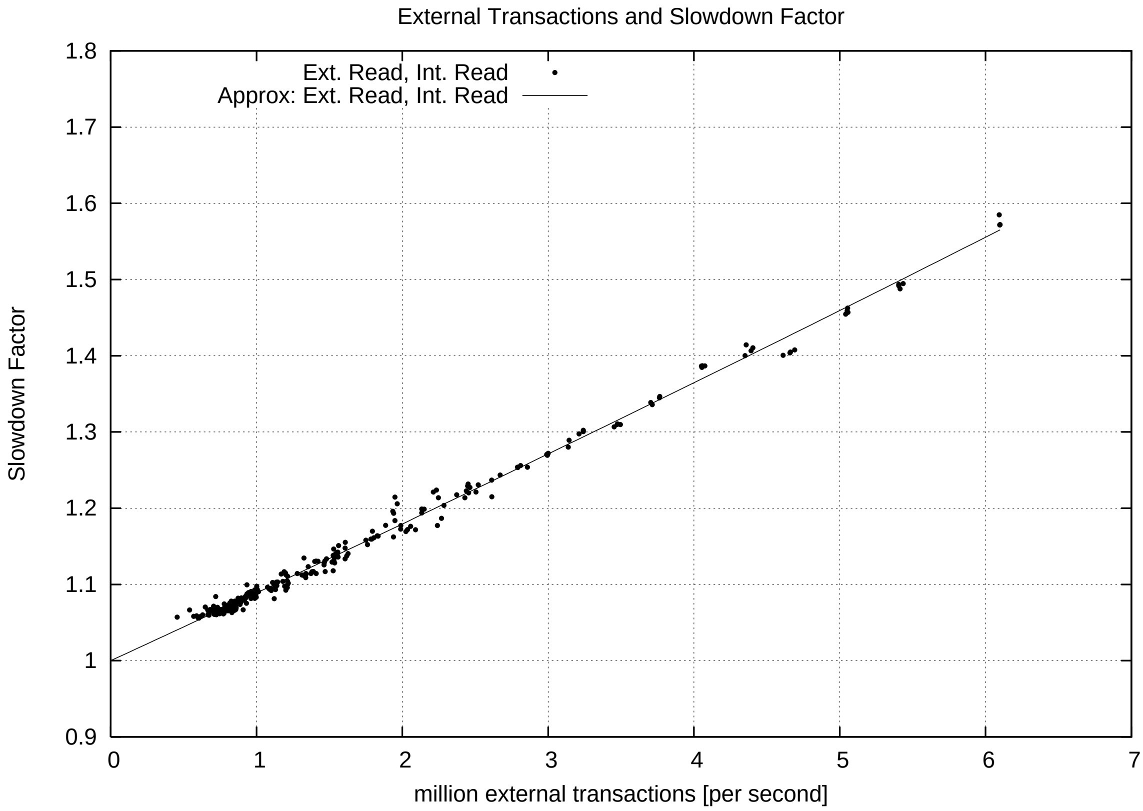


Exactness of the approximated square functions and the slowdown factors determined from measured transaction numbers.


External Transactions and Slowdown Factor
Ext. Write, Int. Write
Approx: Ext. Write, Int. Write
Slowdown Factor
million external transactions [per second]
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0
0.5
1
1.5
2
2.5
3
3.5


External Transactions and Slowdown Factor
Ext. Read, Int. Write
Approx: Ext. Read, Int. Write
Slowdown Factor
million external transactions [per second]
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0
1
2
3
4
5
6


External Transactions and Slowdown Factor
Ext. Write, Int. Read
Approx: Ext. Write, Int. Read
Slowdown Factor
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0
0.5
1
1.5
2
2.5
3
3.5
4
million external transactions [per second]


External Transactions and Slowdown Factor
Ext. Read, Int. Read
Approx: Ext. Read, Int. Read
Slowdown Factor
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0
1
2
3
4
5
6
7
million external transactions [per second]

C. Pamette PCI-Bus Logfile

Figure C shows an extract of a logfile, generated by the PCI-performance-measurement program and the Digital Pamette FPGA board. The first column is the captured cycle number, the last column shows the state according to our PCI bus model. The Pamette reads 128 bytes from main-memory address `0x00BDA300`, the data is `0x1234FEDC`.

```
0206 -- F0 I1 T1 D1 Cmd:7 Addr:00BDA300      s
0207 -- F0 I0 T1 D1 :0 --------      s
0208 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0209 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
020A -- F0 I0 T0 D0 be:0 data:1234FEDC      d
020B -- F0 I0 T0 D0 be:0 data:1234FEDC      d
020C -- F0 I0 T0 D0 be:0 data:1234FEDC      d
020D -- F0 I0 T0 D0 be:0 data:1234FEDC      d
020E -- F0 I0 T0 D0 be:0 data:1234FEDC      d
020F -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0210 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0211 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0212 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0213 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0214 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0215 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0216 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0217 -- F1 I0 T0 D0 be:0 data:1234FEDC      d
0218 -- F1 I1 T1 D1 :0 --------      s
0219 to    0238      31*r
0239 -- F0 I1 T1 D1 Cmd:7 Addr:00BDA340      s
023A -- F0 I0 T1 D1 :0 --------      s
023B -- F0 I0 T0 D0 be:0 data:1234FEDC      d
023C -- F0 I0 T0 D0 be:0 data:1234FEDC      d
023D -- F0 I0 T0 D0 be:0 data:1234FEDC      d
023E -- F0 I0 T0 D0 be:0 data:1234FEDC      d
023F -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0240 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0241 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0242 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0243 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0244 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0245 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0246 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0247 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0248 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
0249 -- F0 I0 T0 D0 be:0 data:1234FEDC      d
024A -- F1 I0 T0 D0 be:0 data:1234FEDC      d
```

Log of a Pamette-generated PCI-bus trace. Symbols: F: Frame, I: IRDY, T: TRDY, D: Data, be: bus-enable lines.

D. Bibliography

[1] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig, "Quality Assuring Scheduling — Using Stochastic Behavior to Improve Resource Utilization," in *IEEE Real-Time Systems Symposium*, Dresden University of Technology, Dec. 2001.

[2] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *16th ACM Symposium on Operating System Principles (SOSP)*, (Saint-Malo, France), pp. 66–77, Oct. 1997.

[3] H. Härtig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter, "DROPS: OS support for distributed multimedia applications," in *Proceedings of the Eighth ACM SIGOPS European Workshop*, (Sintra, Portugal), Sept. 1998.

[4] R. Baumgartl, M. Borriss, H. Härtig, C.-J. Hamann, M. Hohmuth, L. Reuther, S. Schönberg, and J. Wolter, "Dresden Realtime Operating System," in *Proceedings of the First Workshop on System Design Automation (SDA'98)*, (Dresden), pp. 205–212, Mar. 1998.

[5] S. Schönberg, F. Mehnert, C.-J. Hamann, L. Reuther, and H. Härtig, "Performance and bus transfer influences," in *First Workshop on PC-based System Performance and Analysis*, (San Jose, CA), ACM, Oct. 1998.

[6] PCI Special Interest Group, Portland, OR 97214, *PCI Local Bus Specification*, 2.1 ed., June 1995.

[7] M. Scottis, M. Krunz, and M. Liu, "Enhancing the PCI bus to support real-time streams," in *IEEE Performance, Computing abd Communications Conference (IPCCC'99)*, (Phoenix/Scottsdale, Arizona, USA), pp. 303–309, IEEE, Feb. 1999.

[8] PCI Special Interest Group, Portland, OR 97214, *PCI-X Addendum to the PCI Local Bus Specification*, 1.0a ed., July 2000.

[9] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design.* San Francisco: Morgan Kaufmann, 2 ed., 1998.

[10] Intel Corp., *IA-32 Intel Architecture Software Developers Manual, Volume 3: System Programming Guide*, 2002. Order # 245472-006.

[11] Digital Equipment Corp., Maynard, Massachusetts, *Alpha 21164 Microprocessor Hardware Reference Manual*, July 96.

[12] FORE Systems Inc., *FORE PCA-200e Reference Manual.*

[13] M. Benson, *PBI - PCI Bus Interface.* FORE Systems Inc., Nov. 1995.

[14] Intel Corp., Santa Clara, *i960 CA/CF Microprocessor Reference Manual*, Mar. 1994. Order # 270710-003.

[15] U. Dannowski, "Portierung des Linux ATM-Treibers für FORE PCA-200E auf den Mikrokern L4," term paper, TU Dresden, 1997. In German. Available from URL: `http://os.inf.tu-dresden.de/project/atm/`.

[16] M. Shand, *PCI Pamette user-area interface for firmware v1.9.* Digital Equipment Corporation, Systems Research Center, Jan. 1998.

[17] L. Moll and M. Shand, "Systems performance measurement on PCI pamette," in *IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 125–133, IEEE Computer Society Press, Apr. 1997.

[18] J. Liedtke, "L4 reference manual (486, Pentium, PPro)," Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, Sept. 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY.

[19] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul, "Cooperating resource managers," in *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, (Vancouver, Canada), June 1999.

[20] J. Liedtke, M. Völp, and K. Elphinstone, "Preliminary thoughts on memory-bus scheduling," in *9th SIGOPS European Workshop*, (Kolding, Denmark), Sept. 2000.

[21] K. A. Kettler, J. P. Lehoczky, and J. Strosnider, "Modeling Bus Scheduling Policies for Real-time Systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, (Carnegie Mellon University), pp. 242–255, 1995.

[22] W. Fischer, "IEEE P1014 — a standard for the high-performance VME bus," *IEEE Micro*, vol. 5, pp. 31–41, Feb. 1985.

[23] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 46–61, Jan. 1973.

[24] B. P. Aichinger, "Futurebus+ as an I/O Bus: Profile B," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (Gold Coast, Australia), pp. 300–307, ACM SIGARCH and IEEE Computer Society TCCA, May 19–21, 1992.

[25] *IEEE Standard for Futurebus - Logical Protocol Specification*, std 896.1-1991 ed., Mar. 1992.

[26] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Real-Time Scheduling Support in Futurebus+," in *Proceedings of the Real-Time Systems Symposium* (I. C. S. Press, ed.), (Lake Buena Vista, Florida, USA), p. 331, IEEE Computer Society Press, Dec. 1990.

[27] IEEE, "New York, NY, USA", *IEEE Standard 896.3-1993, IEEE recommended practice for FutureBus+*, July 1994.

[28] Infiniband Trade Association, *Infiniband Architecture*, Mar. 2000.

[29] International Organization for Standardization (ISO), *CAN, International Standard 11898*, 1993.

[30] M. D. Natale, "Scheduling the CAN bus with earliest deadline techniques," in *Proceedings of the 21st Symposium on Real-Time Systems (RSS-00)*, (Los Alamitos, CA), pp. 27–27, IEEE Computer Society, Nov. 27–30, 2000.

[31] P. Pedreiras and L. Almeida, "A Practical Approach to EDF Scheduling on Controller Area Network," in *IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, (London, UK), IEEE/IEE, Dec. 2001.

[32] K. Zuberi and K. Shin, "Scheduling Messages on Controller Area Network for Real-Time CIM Applications," *IEEE Transactions On Robotics And Automation*, vol. 13, pp. 310–314, Apr. 1997.

[33] M. Hayter and D. McAuley, "The desk area network," Tech. Rep. CS-TR-228, University of Cambridge Computer Laboratory, May 1991.

[34] *Universal Serial Bus 2 Specification*, Apr. 2000.

[35] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, *Universal Serial Bus Specification 2.0*, 2.0 ed., Apr. 2000.

[36] Intel Corp., *Accelerated Graphics Port Interface Specification.* Santa Clara, May 1998. Revision 2.0.

[37] A. Atlas and A. Bestavros, "Statistical Rate Monotonic Scheduling," Technical Report 98-010, Boston University, May 2, 1998.

[38] C. A. Waldspurger and W. E. Weihl, "Stride scheduling: Deterministic proportional-share resource management," Tech. Rep. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

[39] I. Stoica, Hussein Abdel Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems," in *Proc. of 17th IEEE Real-Time Systems Symposium*, pp. 288–299, IEEE, Dec. 1996.

[40] U. Maheshwari, "Charge-based proportional scheduling," Tech. Rep. MIT/LCS/TM-529, MIT Laboratory for CS, July 1995.

[41] G. J. Henry, "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal*, vol. 63, pp. 1845–1857, Oct. 1984.

[42] J. Kay and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM*, vol. 31, pp. 44–55, Jan. 1988.

[43] I. Stoica, Hussein Abdel Wahab, and K. Jeffay, "On the Duality between Resource Reservation and Proportional Share Resource Allocation," in *Multimedia Computing and Networking*, vol. 3020 of *SPIE Proceedings*, pp. 207–214, Feb. 1997.

[44] K. J. Lin, S. Natarajan, and J. W. S. Liu, "Imprecise results: Utilizing partial computations in real-time systems," in *Proc. IEEE Real-Time System Symp.*, 1987.

[45] J. P. Lehoczky and L. Sha, "Performance of real-time bus scheduling algorithms," *ACM Performance Evaluation Review*, vol. 14, 1986.

[46] T. Wilson, "Switched PCI and Next Generation Embedded PowerPC."

[47] I. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proceedings of the Real-Time Systems Symposium - 1989* (I. C. S. Press, ed.), (Santa Monica, California, USA), pp. 166–171, IEEE Computer Society Press, Dec. 1989.

[48] F. Bellosa, "Process cruise control: Throtteling memory access in a soft real-time environment," Tech. Rep. TR-I4-97-02, IMMDV IV, University of Erlangen, July 1997.

[49] T.-Y. Huang, J. W. Liu, and J.-Y. Chung, "Allow Cycle-Stealing Direct Memory Access I/O Concurrent with Hard-Real-Time Programs," in *Int. Conf on Parallel and Distributed Systems (ICSPAD)*, (Tokyo), June 1996.

[50] "AIM Multiuser Benchmarks." `http://www.caldera.com/developers/community/contrib/aim.html`. Suite VII and IX of what used to be AIM Technology's benchmarks, GPL-ed.

[51] A. B. Brown and M. I. Seltzer, "Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture," in *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Seattle, WA), pp. 214–224, June 1997.

[52] "The SPEC benchmark suite." `http://www.specbench.org`.

[53] B. Peuto and L. Shustek, "An instruction timing model of CPU performance," in *International Conference on Computer Architecture, 25 years of the international symposia on computer architecture (selected papers)*, (Barcelona, Spain), pp. 152–165, ACM Press, New York, NY, USA, 1998.

[54] D. Ofelt and J. L. Hennessy, "Efficient performance prediction for modern microprocessors," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ACM SIGMETRICS Performance Evaluation Review, (New York), pp. 229–239, ACM Press, June 2000.

[55] R. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Transactions on Computer Systems*, vol. 14, pp. 344–384, Nov. 1996.

[56] R. H. Saavedra and A. J. Smith, "Measuring cache and TLB performance and their effect on benchmark run times," *IEEE Trans. on Computers*, vol. C-44, pp. 1223–1235, Oct. 1995.

[57] S. Edgar and A. Burns, "Statistical Analysis of WCET for Scheduling," in *Proceedings IEEE RTSS 2001*, (London, UK), IEEE/IEE, Dec. 2001.

[58] Intel Corp., Santa Clara, *Intel 430TX PCISET: 82439TX System Controller (MTXC) Reference Manual*, Feb. 1997. Order #290559-001.

[59] United States. National Bureau of Standards, *Data Encryption Standard*, vol. 46 of *Federal Information Processing Standards publication.* Gaithersburg, MD, USA: U.S. National Bureau of Standards, 1977.

[60] R. Sedgewick, "Implementing Quicksort programs," *Communications of the ACM*, vol. 21, pp. 847–857, Oct. 1978. See corrigendum [61].

[61] R. Sedgewick, "Corrigendum: "Implementing Quicksort Programs"," *Communications of the ACM*, vol. 22, pp. 368–368, June 1979. See [60].

[62] J. Liedtke, H. Härtig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, (Montreal, Canada), pp. 213–223, June 1997.

[63] Intel Corp., Santa Clara, *Intel 440BX AGPset: 82443BX Host Bridge/Controller*, Apr. 1998. Order #290633-001.

[64] R. David, "Why CompactPCI will replace VME," *Real-Time Magazine*, vol. Q4, pp. 31–36, 1998.

[65] D. Somes, "Is CompactPCI a Replacement for VME," *Real-Time Magazine*, vol. Q4, pp. 37–42, 1998.

[66] T. Shanley and D. Anderson, *PCI System Architecutre.* New York: Addison-Wesley, 4 ed., 1999.

[67] Intel Corp., Santa Clara, *Intel 440FX PCISET: 82441FX PCI and Memory Controller (PMC) and 82442FX Data Bus Accelerator (DBX) Reference Manual*, May 1996. Order #290549-001.

[68] Intel Corp., Santa Clara, *Intel 810 Chipset Reference Manual.*

[69] J. Liedtke *et al.*, "Hit and Miss Server, a scalable network server," in *Panelsection at the Symposion on Operating System Principles*, ACM, Oct. 1997.

[70] C.-J. Hamann, "On the quantitative specification of jitter constrained periodic streams," in *MASCOTS*, (Haifa, Israel), Jan. 1997.

[71] T. Wynia, "CompactPCI Poised To Meet High Performance Telecom Demands," *Real-Time Magazine*, vol. Q3, pp. 48–49, 2000.

[72] J. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[73] D. E. Knuth, *The Art of Computer Programming.* Reading: Addison Wesley, 3 ed., 1998.