Chapter 4

Input/output bus usage control in personal computer-based software routers

4.1 Introduction

In this chapter we address the problem of resource sharing in personal computer-based software routers when supporting communication quality assurance mechanisms, widely known as QoS mechanism. Others have put forward solutions that are focused on suitably distributing the workload of the central processing unit. However, the increase in central processing unit speed in relation to that of the input/output (I/O) bus means attention must be paid to the effect the limitations imposed by this bus on the system's overall performance.

Here we propose a mechanism that jointly controls both the I/O bus and the central processing unit operation. This mechanism involves changes to the operating system kernel code and assumes the existence of certain network interface card's functions, although it does not require changes to the personal computer hardware. Here we also present a performance study that provides insight into the problem and helps to evaluate both the effectiveness of our approach, and several software router design trade-offs.

The chapter is organized as follows. Section 4.2 presents the problem, section 4.3 discusses our proposed solution and section 4.4 presents a performance study of the proposed mechanism in isolation, carried out by simulation. Then, section 4.5 considers the performance of a software router incorporating the proposed mechanism, carried out

also by simulation. Section 4.6 discuses on a particular implementation of the proposed mechanism and presents measured performance data. Finally, section 4.7 summarizes.

4.2 The problem

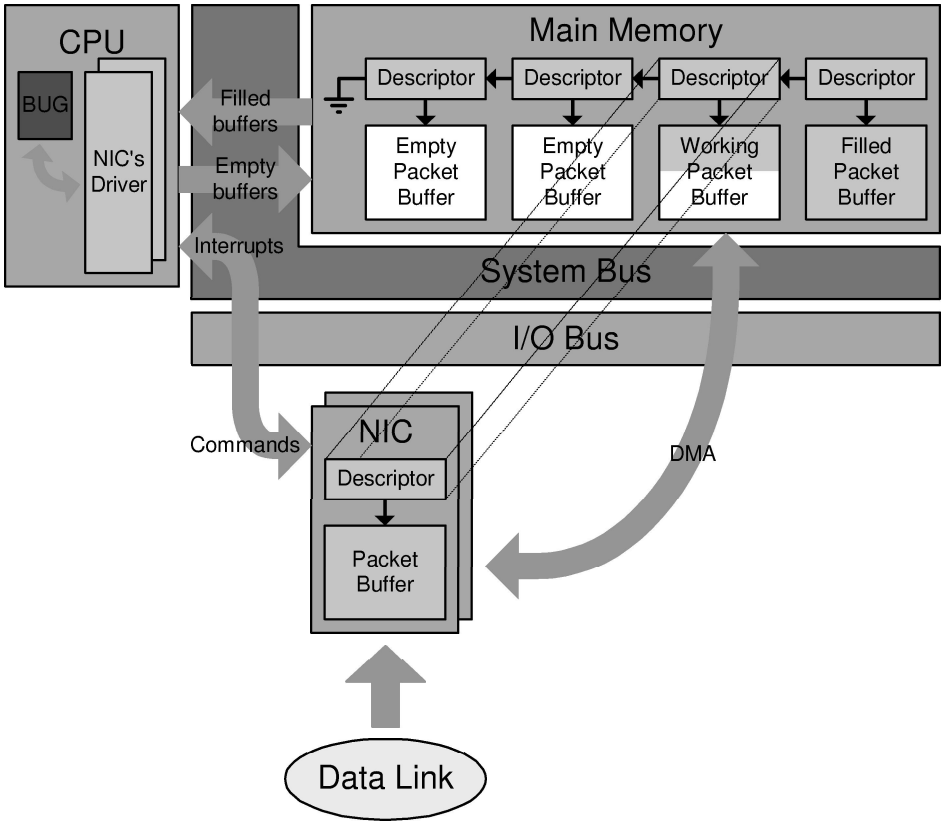
Given that there are inherent performance limitations in the architecture of a software router, but also that there are reasons that make its use attractive, (see previous chapter's section 3.3) the question of how to optimize its performance arises. In addition, if we want to support QoS mechanisms, we must find suitable ways of sharing resources and, as Nagle said, providing protection, so ill behaved sources can only have limited negative impact on well-behaved ones [Nagle 1987]. There are two different aspects of the problem of resource sharing: the fair share of the communications link for each output network interface card and the fair share of router resources, mainly central processing unit and input/output (I/O) bus. The first aspect affects output packets flows that share a single network interface card, while the second aspect affects all packets flows that go through the router. We will focus our work on the second aspect of this problem.

In other works the problem of fairly sharing software router resources is tackled in terms of protecting [Indiresan, Mehra and Shin 1997; Mogul and Ramakrishnan 1997] or sharing [Druschel and Banga 1996; Qie et al. 2001] the use of the central processing unit amongst different packets flows in an efficient way. However, the increase in central processing unit speed in relation to that of the I/O bus makes it easy for this bus to constitute a bottleneck. This is why we address this problem in this article, considering not only the sharing of the software router's central processing unit, but also its I/O bus.

4.3 Our solution

The mechanism we propose for implementing input/output (I/O) bus sharing between differentiated packets flows manipulates the vacancy space of each direct memory access receive channel (see chapter 2's subsection 2.6.6 for a description on direct memory access channels) in such a way that the overall I/O bus' activity follows a schedule similar to one produced by a general processor sharing (GPS) server [Demers, Keshav and Shenker 1989]. The mechanism, that we named Bus Utilization Guard (BUG), acts as a flow scheduler that indirectly regulates the I/O bus utilization. Packets flows wanting to traverse the router have to get registered before the BUG. This may be accomplished either manually or by means of any resource reservation protocol like RSVP. In any case, a packets flow is required to submit target resource utilization for registering and the system is required to manage packets flow identifications for packets flows it admits. Note that the system has to be able to map each differentiated packets flow to a direct memory access channel, DMA channel for short. Figure 4.1 shows the BUG's system architecture.

Figure 4.1—The BUG's system architecture. The BUG is a piece of software embedded in the operating system's kernel that shares information with the network interface card's device drivers and manipulates the vacancy of each DMA receive channel



4.3.1 BUG's specifications and network interface card's requirements

The mission of the BUG is to assist supporting QoS system behavior by indirectly controlling input/output bus usage following a GPS like policy. (For now we regard to the input/output bus simply as the bus.) We thought of the BUG as either a piece of software within the operating system's kernel or as a hardware add-on attached to the AGP connector. We wanted the BUG not to require any change to the host computer's hardware architecture. However, the BUG still requires the network interface cards to keep a running sum of packets and bytes received per differentiated packets flow. Moreover, the BUG can only differentiate packets flows when they are mapped to separate DMA channels. Therefore, if it is wanted to differentiate packets flows entering the router through a network interface card, the latter must support several DMA channels—one channel per differentiable packets flow.

Because the BUG protects a rather fast resource and due to its software only implementation requirement, it was important that the BUG would have low overhead. Moreover, because the BUG uses the bus it thrives to protect, (as will be explained later) for polling information from the network interface cards and configuring them, it was important the BUG would be as low intrusive as possible. Table 4-I summarizes these specifications and requirements.

TABLE 4-I

BUG's specifications
Avoid host hardware modifications
Low overhead
Avoid intrusion
Network interface card's requirements
Per packets flow DMA channels
To be able to keep some packets flow state information

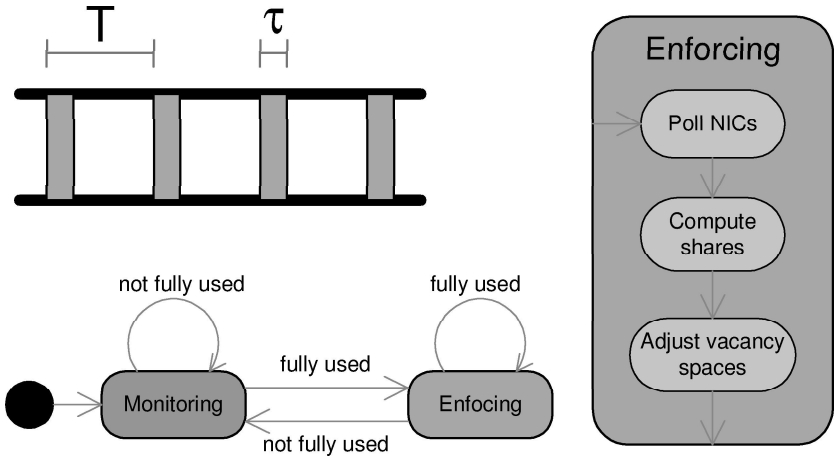
4.3.2 Low overhead and intrusion

In order to minimize overhead and intrusion, we devised the BUG to get activated only every T seconds, where $T >> \tau$ the time required in the worst case for executing BUG's activities when it gets activated. Figure 4.2 illustrates this behavior. As will be shown, τ influences packet latency because during the time the BUG is running no packet may traverse the protected bus.

May T be arbitrarily large? As will be shown, the BUG is a reactive mechanism that from a summary of what have happened at the bus during the last T seconds it adjusts DMA receive channel's vacancy spaces so that the bus' behavior during a busy period resembles that of a GPS server. Consequently, a priori, T cannot be arbitrarily large. Intuitively, there must be a T_{max} beyond which either the BUG cannot react fast enough or the required adjustments are unfeasible to perform. Besides, as will be shown, there is a proportional relationship between T and BUG's main memory storage requirement. Indeed, a priori, the mean number of packets that the BUG has to be aware of increases with T and more packets imply more *mbuf* descriptors, which require more main memory. Actually, the DMA channels are the ones requiring the added memory and not the BUG.

To further reduce intrusion, we devised the BUG as a bistate mechanism. At every activation, the BUG firstly determines how much of the bus resources have been used during the last T seconds. If the bus has not been one hundred percent utilized, the BUG enters monitoring state and no further action is taken. Otherwise, the BUG enters enforcing state and executes all its tasks. Figure 4.2 also illustrates this behavior.

Figure 4.2—The BUG's periodic and bistate operation for reducing overhead and intrusion



4.3.3 Algorithm

The BUG emulates a packet-by-packet GPS server with batched arrivals. The GPS server's inputs are computed after data polled from network interface cards and network interface card's device drivers. Figure 4.3 shows and example scenario. Appendix B lists the C++ language code of the algorithm's implementation used for the simulation experiments described in section 4.4 and 4.5.

Assume that the mechanism is in monitoring state at cycle $k \bullet T$. Then, the mechanism gathers $D_{i,k}$—the number of bytes transferred through the bus during period $((k-1) \bullet T, k \bullet T)$ by *DMA receive channel-i*, *channel-i* for short. If $sum(D_{i,k}) < T/\beta_{BUS}$, where β_{BUS} is the cost per bit of bus transfer, the mechanism remains at monitoring state and no further actions are taken. On the contrary, the mechanism detects the start of a busy period and enters enforcing state. When at this state, the mechanism polls each network interface card to gather $N_{i,k}$—the number of bytes stored at the network interface card associated with *channel-i*—and computes the amount of bus utilization granted to each DMA receive channel, or γ_{ik}. This is done after the outputs of the emulated GPS server, or $G_{i,k}$. GPS server's inputs are the $N_{i,k}$ at the start of a busy period. Afterwards, the inputs are the amount of arrived traffic during the last period or $A_{i,k}=N_{i,k}-N_{i,k-1}+D_{i,k}$. BUG is work-conservative and thus:

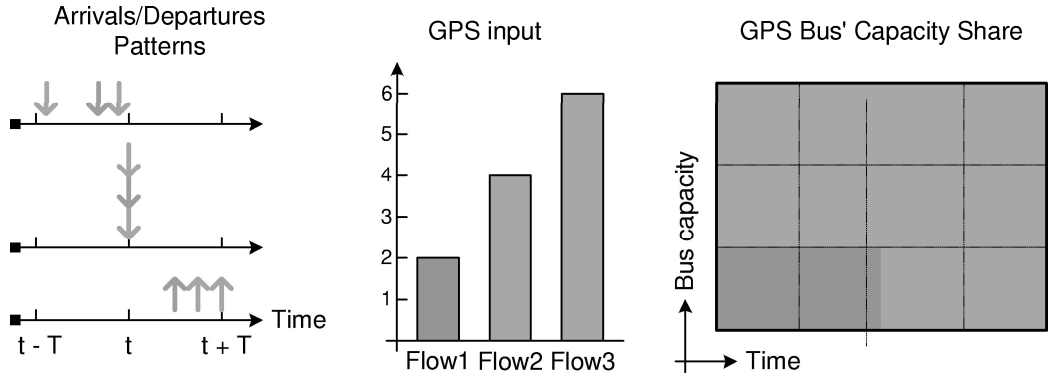
$$\gamma_{i,k} = G_{i,k} + (T/\beta_{BUS} - (G_{1,k}+...+G_{N,k})) \quad (1)$$

Observe that $sum(\gamma_{i,k}) \geq T/\beta_{BUS}$, a situation that can lead to an unfair share. Consequently, the BUG is prepared with an unfairness-counterbalancing algorithm. This algorithm computes an unfairness level per DMA receive channel, $u_{i,k}$, and if it detects at least one deprived packets flow then it reduces every depriver packets flow's bus utilization grant, $\gamma_{i,k}$, by the corresponding unfairness value. In this case we have:

$$\gamma^*_{i,k} = \lceil \gamma_{i,k} - u_{i,k} \rceil \quad (2)$$

One problem with this approach is that if unfairness is detected then $sum(\gamma^*_{i,k}) \leq T/\beta_{BUS}$; that is, the unfairness-counterbalancing algorithm may artificially produce some bus idle time. This problem also arises when packetzing bus utilization

Figure 4.3—The BUG's packet-by-packet GPS server emulation with batch arrivals



grants, as shortly explained. Happily, a single mechanism, one that allows the BUG to vary the length of its activation period, solves both problems.

The BUG will switch from enforcing state to monitoring state at the start of any activation instant that $sum(D_{i,k}) < T/\beta_{BUS}$. When this switch occurs, the BUG resets the emulated GPS server and stops regulating the bus usage by giving away:

$$\gamma_{i,k} = (T/\beta_{BUS}) \quad (3)$$

4.3.4 Algorithm's details

The BUG requires detecting when the bus is fully utilized. One approach to do this is asking the bus for its accumulated idle time at each activation period. If there is a difference between the current value and the read previously then the bus was not fully utilized during the last T time units. The problem with this approach is that the PCI bus specification does not defines such a thing as *bus accumulated idle time* [Shanley and Anderson 2000]. Upon this absence, the BUG computes the bus utilization level after the received byte-count it collects at each activation-instant from network interface cards' device drivers.

The BUG is a work-conservative mechanism in which any idle time detected at the emulated GPS server is given away for all packets flows to use. Moreover, when at monitoring state it sets all use-grants to a value proportional to a complete T period, so the bus can get busy as soon as possible. Figure 4.4 shows an example scenario. This has two evident implications. On the good side, packets do not experience any initial waiting. On the bad side, some unfairness may arise because some packets flow may use more than its solicited share in detriment of other packets flows. Happily, unfairness may only occur during a bus busy period. Therefore, the BUG only needs to look for and correct unfairness when at enforcing state. Observe that the emulated GPS server cannot deal with any unfairness because the latter is produced outside its scope. Thus, any counterbalancing mechanism has to be implemented as part of the BUG's control-logic.

Of course, unfairness may be avoided altogether by implementing a non-work-conservative mechanism, in which packet arrivals occurred during an activation period get artificially batched at the next activation instant. (By appropriately manipulating the DMA receive channel's vacancy spaces.) But then packets would experience added latency. We also have implemented a non-work-conservative BUG and it works well throughput-wise. However, here we only present results from the work-conservative version because we believe it is a better overall solution.

When at enforcing state, the BUG has to check for unfairness. To do this, upon entering this state the BUG starts comparing two running sums per packets flow: $sum_i(G_{i,k})$ and $sum_i(D_{i,k})$. Then, unfairness may be computed as:

$$u_{i,k} = sum_i(D_{i,k}) - sum_i(G_{i,k}) \quad (4)$$

A positive difference denotes a depriver packets flow, while a negative value denotes a deprived one. If the BUG detects at least one deprived packets flow, then all de-

priver packets flows will have their bus utilization grant, $\gamma_{i,k}$, reduced by $u_{i,k}$ and all deprived packets flows will get theirs augmented by $u_{i,k}$. Any $\gamma_{i,k}$ that gets negative is topped to zero. Figure 4.5 shows an example scenario.

The BUG is required to packetize the computed bus utilization grants before adjusting vacancy spaces at DMA receive channels. This is required because while the GPS algorithm's information unit is the byte, the information unit for DMA channels is the packet. Figure 4.6 shows an example scenario. When packetizing utilization grants it may happen that $modulus(\gamma_{i,k}, L_i) \neq 0$, where L_i is the mean packet length for *channel-i*. Hence, some rounding off is required. We have tested rounding off both down and up and both produce particular problems. However, the former gave us a more stable mechanism. Of course, if we let the BUG to round off its bus utilization grants, then its emulated GPS server will get out of synch with respect to what is happening at the bus. Therefore, we also programmed the BUG to accordingly adjust the state of its emulated GPS server. If nothing else is done, some bus idle time is artificially produced and the overall share assigned to that packets flow would be much less of what it should be. This problem, and the induced by the unfairness counterbalancing mechanism, can be solved if we let the BUG reduced its next activation period length by some *dt* time value. Evidently, this increases the BUG's overhead. But as long as *dt* is a small fraction of *T*, the overhead's increase will remain at acceptable levels.

The BUG normally works periodically with period *T*, but the implementation of the unfairness counterbalancing mechanism and the bus utilization grants packetization requirement may induce some bus idle time if nothing else is done. For circumventing this problem we allow the BUG to adjust its activation period and set it to the length of the expected busy period after an enforcing state activation during which the emulated GPS server ran at 100% utilization. The busy period's length is just $sum(\gamma^*_{i,k}/\beta_{BUS})$, where $\gamma^*_{i,k}$ is the packetized bus utilization grant of packets flow i at activation instant k after adjusted for unfairness if necessary. Observe that this value is equal to $T - dt$, where it is expected that $dt >> T$ thanks to the BUG implementation's properties. Note that only at the considered activation instant is the activation period adjusting required.

4.3.5 Algorithm's a priori estimated costs

When at monitoring state, the BUG only needs "to keep an eye" on the bus utilization at every activation instant, and thus we can say this task's costs are low. However, this task still requires *O(n)* work, where *n* is the number of DMA receive-channels associated to network interface cards.

When at enforcing state, the BUG performs a sequence of tasks involving loops: assessing bus utilization, computing GPS server's inputs, executing the GPS server itself and adjusting packets flows' bus grants. Of all these tasks, the GPS server requires the largest number of instructions. Shreedhar and Barghese [1995] state that a naive implementation of a GPS server requires *O(log(m))* work per packet, where *m* is the number of packets in the router. However, Keshav [1991] shows that good implementation requires *O(log(n))*, where *n* is the number of active packets flows.

Section 4.6 reports the a posteriori BUG's costs measurements.

Figure 4.4—The BUG is work conservative

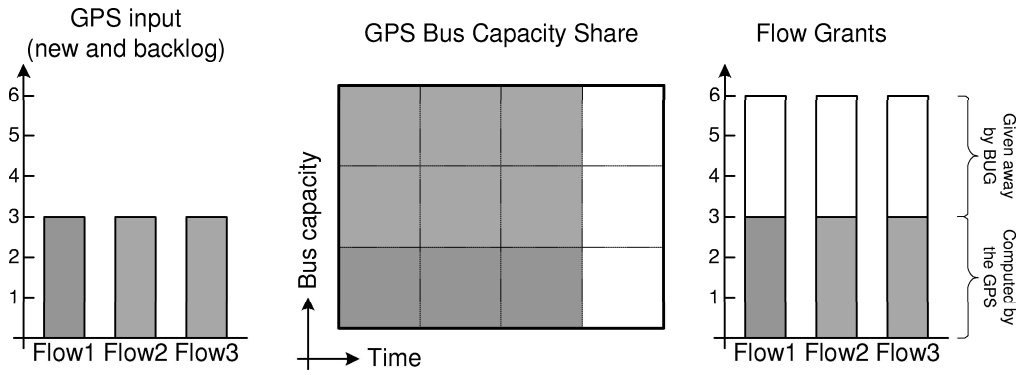


Figure 4.5—The BUG's unfairness counterbalancing mechanism.

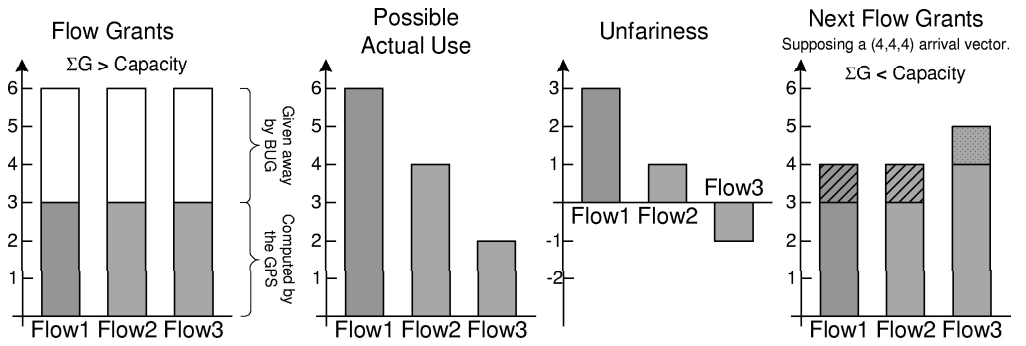
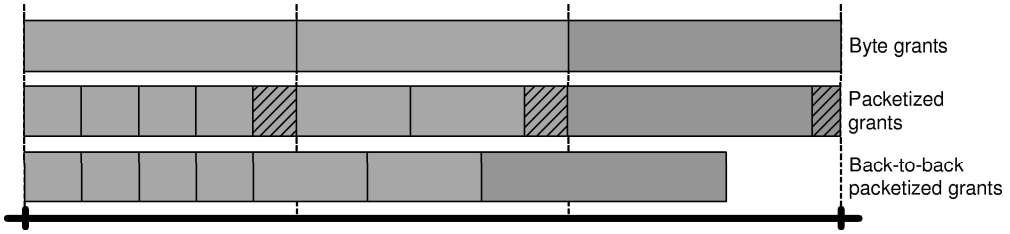


Figure 4.6—The BUG's bus utilization grant packetization policy. In the considered scenario, three packets flows with different packet sizes traverse the router and the BUG has granted each an equal number of bus utilization bytes. Packet sizes are small, medium and large respectively for the orange, green and blue packets flows. After packetization, some idle time gets induced.



4.3.6 An example scenario

In order to see how all this works allow us to present a step-by-step description of an example operation scenario for the BUG. The scenario considers the operation of the BUG protected bus in isolation. Three packets flows load the system. Each packets flow solicits one third of the bus capacity. All packets are of the same size and six packets fully occupy the bus. Figure 4.7 shows a picture of the description that follows. In the picture time runs downwards. Marks at the time axis denote BUG's nominal activation instants and thus are spaced by *T* seconds. The picture shows the BUG's operation variables as vectors. For each vector, dimension k corresponds to packets flow k. Table 4-II defines the used vectors.

TABLE 4-II

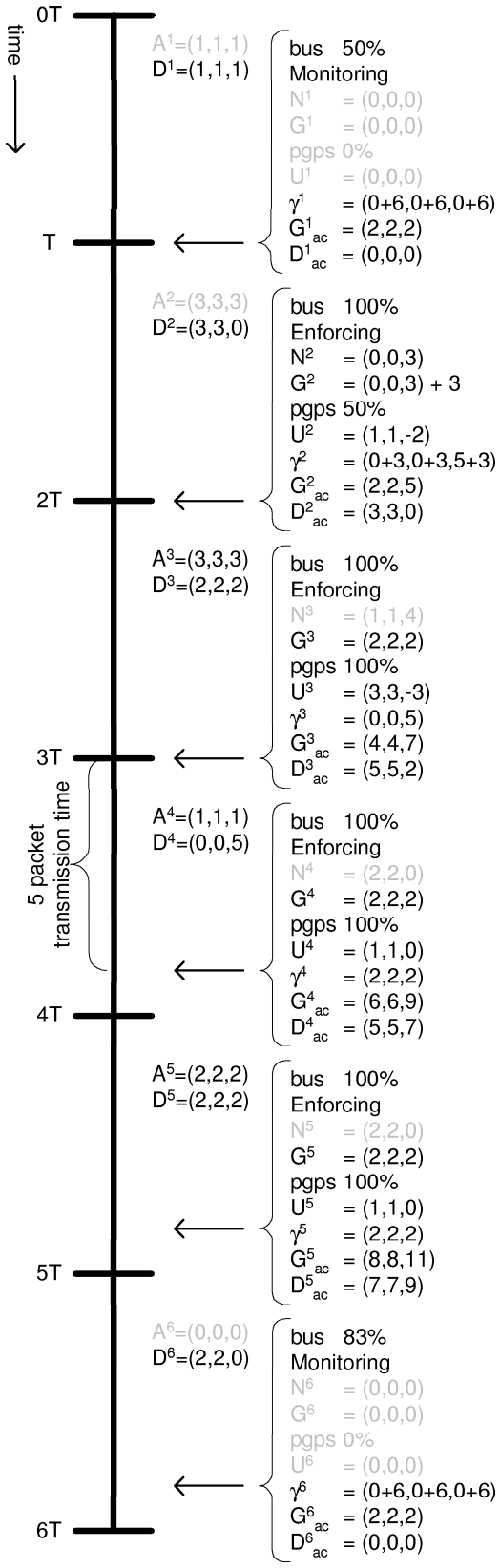
Vector	DEFINITION
A	NUMBER OF PACKET ARRIVALS DURING THE LAST ACTIVATION PERIOD
D	Number of packet departures during the last activation period
N	Number of packets waiting at the buffers of some NIC
G	Outputs produced by the emulated GPS
G_{ac}	Running sum of GPS outputs
D_{ac}	Running sum of departures
U	Unfairness (D_{ac} + D - G_{ac})
γ	USE GRANT GIVEN BY BUG

Because vectors A and B denote events occurred anytime during the previous activation period, we show them between activation instants. The rest are computed, or considered, at every activation-instant and therefore we show them grouped and pointed at the corresponding instant. Observe that D values are arbitrary and that to simplify things, all units are given in packets, not bytes.

At the first activation instant, the BUG measures a bus utilization level of 50% and thus it stays at monitoring state, sets use-grants per packets flow to six, and resets the GPS server's state. The BUG does not consider state variables printed in gray at this point time.

At the second activation instant, the BUG measures a bus utilization level of 100% and thus enters enforcing state, runs the GPS server, checks for unfairness, sets use-grants after the GPS's outputs and the unfairness levels, and allows the GPS server to keep its state. Because this is the first time into enforcing state, GPS server's inputs are taken after NIC occupation levels. In this case, inputs do not saturate the GPS server, which reports some idle time. At the same time, the BUG detects a packets flow is being deprived of some solicited bus time. Therefore, depriver packets flows have their use-grants reduced and deprived ones augmented. At the same time, all packets flows get additional use-grants proportional to the GPS server's idle time.

Figure 4.7—We show an example of the behavior of the BUG mechanism. Vectors A, D, N, G and g are defined as: A = (A_1, A_2, A_3), etc. We assume that the system serves three packets flows with the same shares and with the same packet lengths. In a period *T* up to six packets can be transferred through the bus



At the third activation instant, the BUG again measures a bus utilization level of 100% and remains at enforcing state. This time, GPS server's inputs are taken after the arrivals. The GPS server now gets 100% used and thus no idle-time related use-grants will be added. However, the BUG still detects a packets flow being deprived of some solicited bus time and thus the GPS server's outputs get adjusted appropriately for computing the use-grants. Observe that at this point, the sum of all use-grants is not six and thus the BUG reduces the length of its next activation period.

At the fourth activation instant, the BUG remains at enforcing state. At this time the GPS server gets 100% used, again, and no packets flow is detected as being deprived. Therefore, use-grants are directly taken from the GPS server's outputs. The fifth activation period is pretty much the same as the previous, apart from vector D.

At the sixth activation period, the BUG enters monitoring state and thus sets use-grants per packets flow to six, and does a GPS server mind reset.

4.4 BUG performance study

The BUG has several operational latitudes, as previous section shows, like the activation period value and its variation, the effectiveness of the rounding off policy, or the influence that a highly variant traffic has over the dual-mode operation. In order to assess how and how much this latitudes influence overall performance, we devised a series of simulation experiments. At the same time, this experiments allowed us to see how well a PCI bus controlled by a BUG approximates a bus ideally supporting QoS, like a weighted fair queuing bus, or WFQ bus.

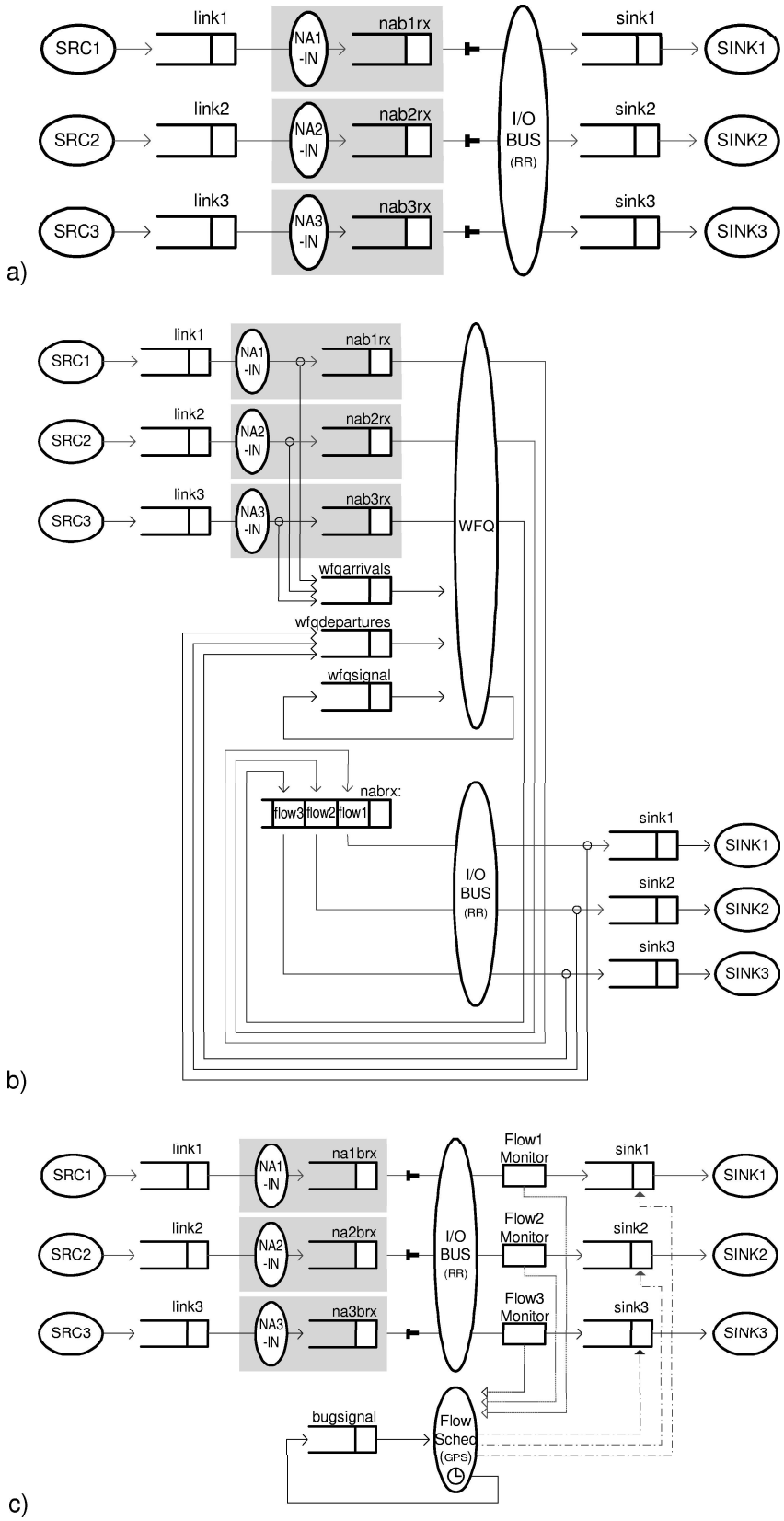
4.4.1 Experimental setup

We study the performance of a BUG regulated PCI bus, to which three network interface cards were attached. For the comparison studies we similarly configured a hypothetical, ideal WFQ bus and a plain PCI bus. Each bus was traversed by three packet-flows, each coming from a single network interface card. Buses were modeled with queuing networks. Figure 4.8 shows these models. We approximated the PCI bus operation by a server using a round robin scheduler. Operational parameters for all busses where computed after a 33 MHz, 32 bits PCI bus. Data links are assumed to sustain one gigabit per second throughput. We used a simple yet meaningful QoS differentiation: the packet size. Indeed, as reported elsewhere [Shreedhar and Varghese 1996], round robin scheduling is particularly unfair upon packets flows with different packet sizes. The packets flows used had features shown in Table 4-III. Different experiments used different inter-arrival processes to show particular behavior.

TABLE 4-III

	Packet length (bytes)	Solicited share
Flow 1	172	1/3
Flow 2	558	1/3
Flow 3	1432	1/3

Figure 4.8—Queuing network models for: a) PCI bus, b) WFQ bus, and c) BUG protected PCI bus; all with three network interface cards attached to it and three packets flows traversing them

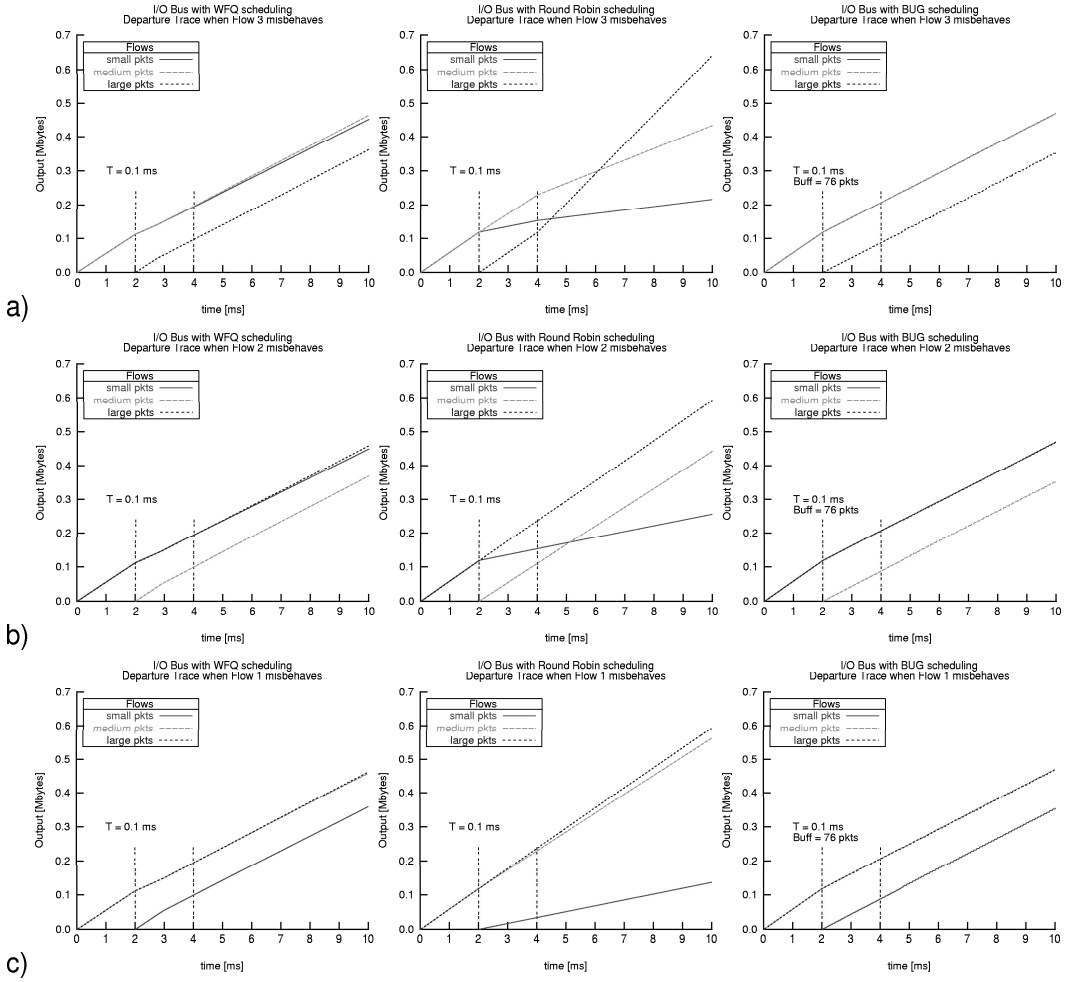


4.4.2 Response to unbalanced constant packet rate traffic

Figure 4.9 shows responses to unbalanced constant bit rate traffic. Each line at every chart denotes the running sum of output bytes over time for one of the three considered packets flows. Each chart in a row corresponds to a particular bus. Left to right, first the WFQ bus, then the plain PCI, and last but not least the BUG equipped PCI bus. Each row corresponds to a particular traffic pattern. The traffic pattern for row (a) was as follows. At time zero, flow1 and flow2 start loading the system with a load level equivalent to 45% of a PCI bus capacity each; that is, 475.2 Mbps. Two milliseconds later (first arrow) or 20 times BUG's activation period flow3 starts loading the system also at 475.2 Mbps. Then, two milliseconds later (second arrow) flow3 augments its load to 1 Gbps. Traffic patterns for row (b) and (c) are similar but changing packets flows' roles.

For these experiments we set BUG's nominal activation period to 0.1 milliseconds. At this value, and under the considered bus' speed, the worst-case DMA receive channel size is 76, which corresponds to flow1. Putting this on implementation perspec-

Figure 4.9—BUG performance study: response comparison to unbalanced constant packet rate traffic between a WFQ bus, a PCI bus and a BUG protected PCI bus; first, middle and left columns respectively. At row (a) flow3 is the misbehaving flow while flow2 and flow1 are for (b) and (c), respectively



tive we may say that 76 mbufs corresponds to a little more than half the nominal DMA channel size for FreeBSD, which is 128. At the same time, a 0.1 period is only 10 times smaller than the nominal FreeeBSD's real-time clock period and thus feasible to implement. On the other hand, this implies that the BUG should take no more than 10 microseconds to execute if we want the overhead premise of $T >> \tau$ to hold. For a software router wearing a 1 GHz central processing unit this means 10 thousand cycles. A priori that should be enough.

From Figure 4.9's left-most column we can see that during the first two milliseconds the ideal bus allows a 50% bus share between the two active packets flows. Then, after the third packets flow gets active, the bus allows a 33% bus share irrespectively of the load level of the so-called misbehaving packets flow. (The small share differences are due to WFQ's well-known misbehavior upon packet bursts. Zhang and Keshav [1991] explain this.)

From Figure 4.9's middle column we can see that a plain PCI bus only adequately follows the ideal behavior during the first two milliseconds. A that point in time, (first arrow) the round robin scheduling deprives flow1 from having enough bus time in favor of both flow2 and flow3. Moreover, flow3 is never deprived and flow2 is when flow3 gets greedy after the second arrow at row (a).

From Figure 4.9's right-most columns we can see that the BUG equipped PCI bus behaves very much like the ideal bus does.

4.4.3 Study on the influence of the activation period

We repeated the experiments of the previous subsection but only for the BUG protected PCI bus and augmenting BUG's activation period T. We wanted to see if we could find any macroscopic problems related to this operational parameter. Per BUG's algorithm description, as *T* becomes relatively larger small-scale injustices appear; we wanted to see if these microscopic injustices might reflect macroscopically and how.

We incremented *T* until the BUG regulated DMA channel's size became unreasonable large. Indeed, as stated in subsection 4.3.2, there is a proportional relationship between *T* and BUG regulated DMA channels' sizes. As stated in the previous section, for the considered bus' speed and with *T* equal to 0.1 milliseconds the BUG requires 76 mbufs on the worst case, or a little more than half the nominal DMA channel size for FreeBSD. With *T* equal to 0.5 milliseconds the required DMA channel's size is 383, already more than double the normal size. And with *T* equal to 10 milliseconds each DMA channel requires 7674 or 14.9 Mbytes of non-swappable or wired memory. (1 Mbytes equals 2^20 bytes.) Certainly, recall that FreeBSD's release 4.1.1 only uses mbuf clusters and thus each DMA channel's slot requires at least 2048 bytes.

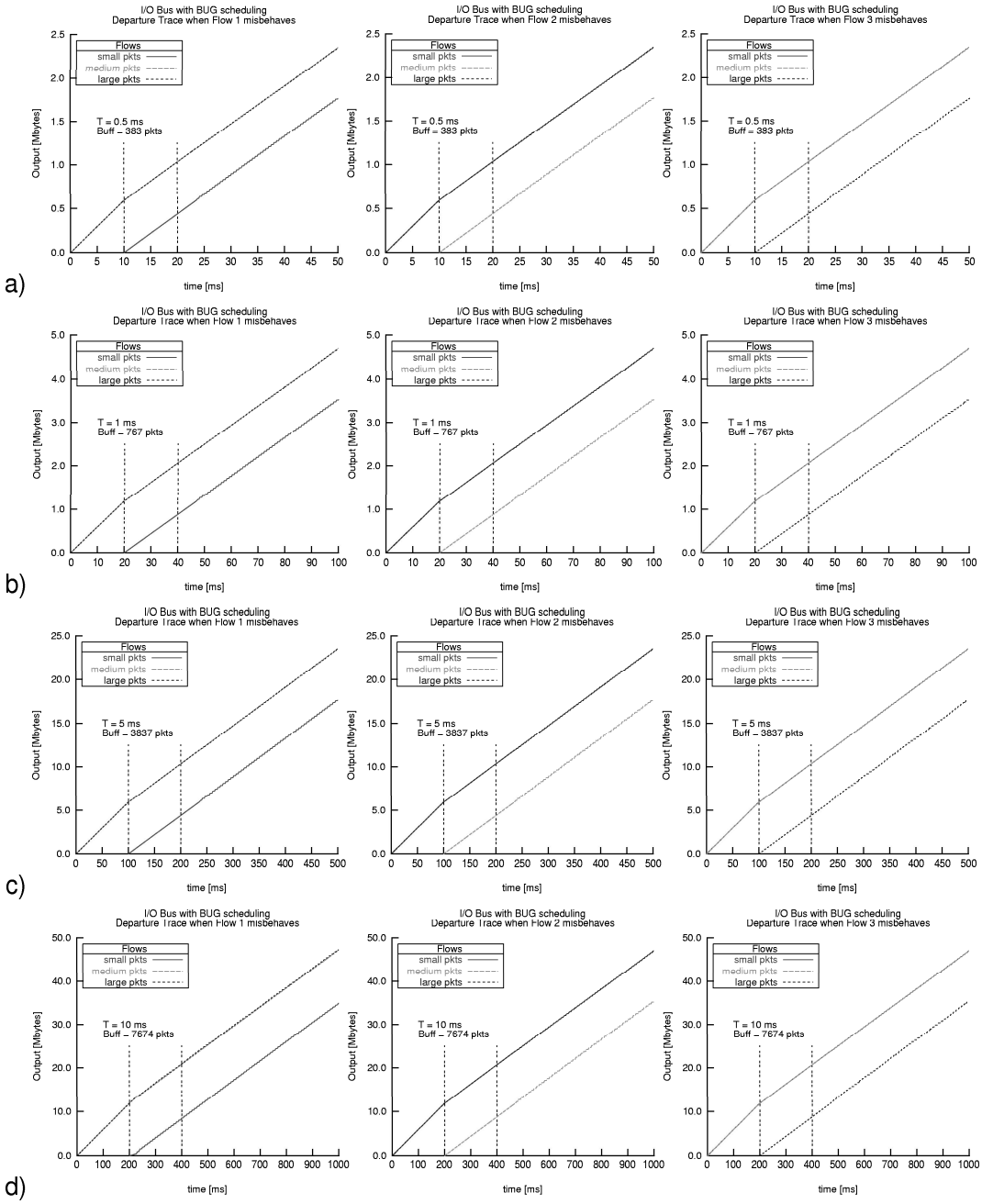
Figure 4.10 shows the study's results and as seen there the BUG protected PCI bus maintains its excellent macroscopic behavior. As in the previous figure, each line at every chart denotes the running sum of output bytes over time for one of the three considered packets flows. Departing from previous figure, each row now corresponds to a particular experiment using a particular BUG activation period *T*. Moreover, each column now corresponds to a particular traffic pattern with respect to the misbehaving

flow: flow1 for the left column, flow2 for the center column, and flow3 for the right column.

4.4.4 Response to on-off traffic

Figure 4.11 shows responses to on-off traffic. Each line at every chart but one denotes the running sum of output bytes over time for one of the three considered packets flows. The one line denotes the running sum of input bytes. Each chart compares one particular packets flow's output process as produced by each of the three considered buses. Left to right, the first chart is for flow1; the second one is for flow2; and the last

Figure 4.10—BUG performance study: on the influence of the activation period



one is for flow3. Sources' on-state period-lengths were set to a constant value. Packet inter-arrival processes were Poisson with mean bit rate equal to 3520 Mbps, or 300% of the PCI bus capacity. Sources' off-state period-lengths were drawn after an exponential random process with mean value equal to nine times the on-state period-length. Consequently, all packets flows overall mean bit rates were equal to 30% of the PCI bus capacity or 352 Mbps.

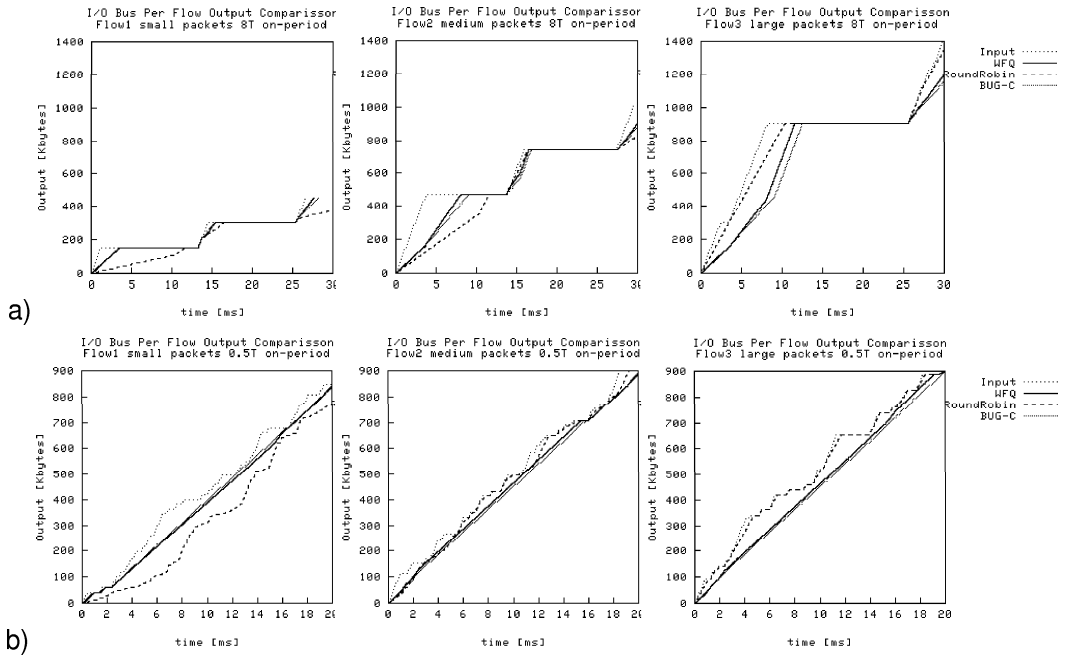
Besides observing the system response to this kind of traffic, with these experiments we wanted to see if we could find any BUG pathology related to operating-mode cycles, where the continuous but random path into and out of enforcing mode may produce some wrong behavior. Consequently, we ran several experiments with different on-off cycle lengths. Here we present results for two different on-state period-lengths.

Figure 4.11.a presents results when the on-state period is equal to eight times the BUG activation period, T, and Figure 4.11.b presents results when the on-state period is equal to 0.5T. From both figures we can see that despite the traffic's fluctuations, the BUG quite well follows the ideal WFQ policy, while the PCI like Round Robin policy again favors the largest-packets flow and affects the most to the smallest-packets flow. Furthermore, it seams that the BUG is not macroscopically sensitive to a traffic pattern that repeatedly takes it in and out of enforcing mode.

4.4.5 Response to self-similar traffic

In order to evaluate the long-range behavior of a BUG protected PCI bus we perform an experiment feeding synthetic self-similar traffic to the simulators of the compared buses. This traffic trace was composed of 6.5 million packets, classified in the three packets flows described in Table 4-III, and had an average throughput of 125 Mbytes per second (1 Mbytes equals 10^6 bytes), or 100% the maximum theoretical

Figure 4.11— BUG performance study: response comparison to on-off traffic between an ideal WFQ bus, a PCI bus, and a BUG protected PCI bus



throughput of the PCI bus. The simulation run spanned 18.874 real-time seconds. Table 4-IV lists variance values of output-byte traces' correlation functions between the ideal WFQ bus and both, the plain PCI bus (round robin) and the BUG protected PCI bus, when different observation periods where applied. In this table we can see that correlation's variance values for the plain PCI bus are higher than the values for the BUG protected PCI bus. Moreover, the values' differences get relatively larger with the observation period, although the increase is not proportional to the size of the observation period. For instance, Flow1's relative difference at an observation period of 0.1 milliseconds is around 1.845 (14.36 / 7.78), at 1 millisecond is 3.02, and at 2.5 milliseconds is 3.342.

Evidently, these results are a good news, bad news case, where the good news are, of course, that the BUG protected PCI bus better follows the long-range behavior of the ideal WFQ bus when compared to a plain PCI bus even under a very high variability operation scenario. However, the variance values for the BUG protected PCI bus are somewhat higher than we expected. We recognize that it is interesting to dig further into this issue. However, we are leaving this as future work.

TABLE 4-IV

Period [ms]	Plain PCI bus			BUG protected PCI bus		
	Flow1	Flow2	Flow3	Flow1	Flow2	Flow3
0.1	14.36	2.70	1.62	7.78	1.58	0.95
0.5	49.24	7.51	4.73	19.47	3.29	2.09
1	81.39	11.68	7.39	26.90	4.67	3.01
1.5	109.88	15.36	9.76	33.35	5.81	3.68
2	133.63	18.21	11.73	39.42	7.13	3.89
2.5	156.78	21.11	13.63	46.91	8.04	4.66

Before passing to another issue, here we consign details on how we produced the self-similar traffic trace. We used Christian Schuler's program (Christian Schuler, Research Institute for Open Communication Systems, GMD FOKUS, Hardenbergplatz 2, D-10623 Berlin, Germany), which implements Vern Paxson's self-similar traffic fast approximation method [Paxson 1997]. Upon given a Hurst parameter, a mean value, and a variance, Schuler's program produces a list of numbers. Each number represents a count of packet arrivals within an arbitrary period. Schuler's program does not give any meaning to this period, leaving to the program's user its definition. Paxson's method uses a fractional gaussian noise process and consequently the output of Schuler's program may contain negative numbers. The given mean and variance values influence the relative count of these negative numbers. It is up to the program's user the use of proper program inputs and the negative number's interpretation. Schuler's program output corresponds to an aggregated traffic trace. Given that we wanted this aggregated traffic to be composed of the three packets flows described in Table 4-III, we filtered Schuler's program output through an ad hoc program to adequately produce three traffic traces corresponding each to a required packets flow.

We employed a Hurst parameter of 0.8, a mean value of 25 packets per observation period, and a variance for times the mean value. We determined this set of parameters after what Lucas et al [1997] have reported. Their paper reports statistical characteristics of traffic threading the University of Virginia's campus network, which hosts approximately 10 thousand computers. The traffic analyzed focuses on three 90-minute intervals starting at 2:15 AM, 2:00 PM, and 9:00 PM. Regardless of the utilization levels exhibited during these periods, the paper reports the traffic adjusting to a Hurst parameter of 0.8 and having a variance for times its mean value.

We came to the value of 25 packets per observation period empirically. We ran Schuler's program with several mean values (and before the stated Hurst parameter and variance) and counted the number of negative numbers contained in the output trace. Schuler's program ran pretty fast on UPC's SGI Power Challenge so, a priori, we did not invest any time selecting the initial trial value. We randomly choose 10 packets per observation period and the output resulted with 4.9% of negative numbers. Entering 20 packets per observation we got a trace with 1.2% of those numbers. With 30 we got 0.3% and with 25 we got 0.5%.

We conveniently attribute a 72 microseconds value to Schuler's program observation period. This value, conjunctively with the given packets flows' features and the selected mean value of 25 packets-per-observation, results in a traffic trace of 125 Mbytes per second average throughput.

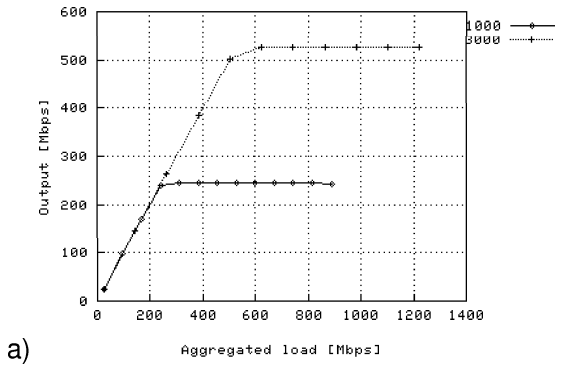
4.5 A performance study of a software router incorporating the BUG

Naturally, after proving that a BUG protected bus works fine in isolation, at least at the queuing network modeling level of detail, we wanted to see if such a bus may improve the operation of a software router bore up to provide differentiated services. In order to do this we basically extended the set of experiments presented in previous chapter's subsection 3.8.2. Following the rationale presented there, we extended the queuing network model that previous chapter's Figure 3.20 shows and introduced the flow monitors and flow scheduler shown in Figure 4.8.c. The resulted model's operational parameters and the features of the workload were set as in previous chapter's mentioned section. Once again, we have performed the simulation for routers configured with two different central processing units. The one's CPU speed is 1 GHz and the other's is 3 GHz. As before, note that for the considered traffic the CPU is the system's bottleneck for the 1 GHz router while the bus is the system's bottleneck for the 3 GHz router.

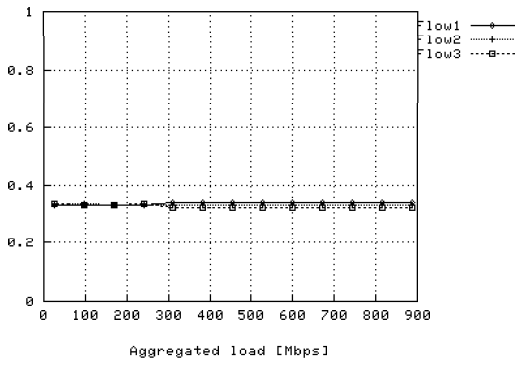
4.5.1 Results

Figure 4.12 shows results for the system with a WFQ scheduling for the CPU and the BUG mechanism for controlling The bus usage. We see that the obtained results correspond to almost an ideal behavior, as under saturation throughput does not decrease with increasing offered loads and the system achieves a fair share of both router resources: CPU and The bus.

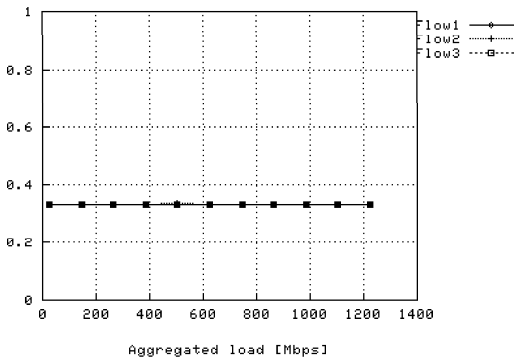
Figure 4.12—QoS aware system's performance analysis: a) system's overall throughput; b) per packets flow throughput share for system one; c) per packets flow throughput share for system two



a)



b)



c)

4.6 An implementation

Currently we are working on a BUG implementation for a FreeBSD powered PC-based software router, using 3COM's 3C905B network interface cards based on the "hurricane" PCI bus-master chips and controlled by the FreeBSD `xl` driver.

4.7 Summary

- We presented a mechanism for improving the resource sharing of the input/output bus of personal computer-based software routers
- The mechanism that we proposed and called BUG, for bus utilization guard, does not imply any changes in the host computer's hardware, although some special features are required for network interface cards—they should have different direct memory access channels for each differentiated packets flow and they should be able to give information about the number of bytes and packets stored for each of these channels
- The BUG mechanism can be run by the central processing unit or by a suitable coprocessor attached at the AGP connector.
- Using a queuing model solved by simulation, we studied BUG's performance. The results show that BUG is effective in controlling the bus share between different packets flows
- When we use this mechanism in combination with the known techniques for central processing unit usage control, we obtain a nearly ideal behavior of the share of the software router resources for a broad range of workloads