# Chapter 3

# Characterizing and modeling a personal computer-based software router

## 3.1 Introduction

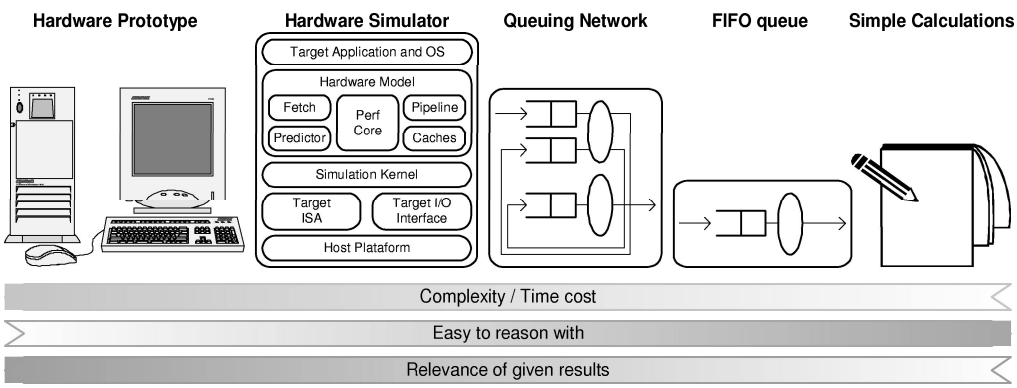
This chapter presents our experiences building and conducting the parameterization of a performance model of a personal computer-based software router; that is, a software router built upon off-the-self personal computer technology. The resulting model is an open multiclass priority network of queues that we solved by simulation. While the model is not particularly novel from the system modeling point of view, in our opinion, it is an interesting result to show that such a model can estimate, with high accuracy, not just average performance-numbers but the complete probability distribution function of packet latency, allowing performance analysis at several levels of detail. The validity and accuracy of the queuing network model has been established by contrasting its packet latency predictions in both, time and probability spaces. Moreover, we introduced into the validation analysis the predictions of a router's single first-come, first-served queue model. We did this for quantitatively assessing the advantages of the more complex queuing network model with respect to the simpler and widely used but not so accurate, as here shown, single queue model, under the considered scenario that the router's CPU is the system bottleneck and not the communications links. The single queue model was also solved by simulation. The queuing network model was successfully parameterized with respect central processing unit speed, memory technology, packet size, routing table size, and input/output bus speed. Results revel not evident and important performance trends for router design.

This chapter is organized as follows. The first two sections set the background. Section 3.2 briefly discuss about trade-offs in system modeling and puts in perspective the appropriateness of networking software's single queue models with respect to queuing network models. Section 3.3 presents personal-computer-based software routers' chief technological and performance issues. Moreover, it puts this router technology in perspective when comparing it with others. Following the background sections, section 3.4 presents the case on queuing network modeling for personal computer-based software routers and describes an example model. Furthermore, it explains how to modify or extend the example model for modeling software routers with different configurations. Then, section 3.5 describes the laborious system characterization process and the implications that this process's results had on system modeling. The model's validation is discussed in section 3.6. Section 0 argues about the model's parameterization and, as a result of this, presents interesting software routers' performance trends. Finally, section 3.8 presents example uses of the software router queuing network model for capacity planning and as an uniform experimental test bed. Then, section 3.9 summarizes the chapter.

# 3.2 System modeling

Performance models of computer systems are important for researching and developing new ideas. These performance models are commonly built from a spectrum of techniques as those shown in Figure 3.1. A researcher or engineer may use hardware prototyping, hardware simulators, queuing networks, simple queues or simple math. Evidently, there are tradeoffs to consider when choosing a technique. Complexity, ease to reason with, and obtained results' relevance are of chief concern. Clearly, hardware prototyping is the technique that can give results with greater relevance. However, it is also the most complex technique and it is difficult to reason with—it is not ease to see how a component's variation influences others or the whole system. On the other side of the spectrum, single-queue theory is a simple technique that is easy to reason with but gives results of limited relevance or scope. Networks of queues are important models of multiprogrammed and time-shared computer systems. However, these models have not been used for performance modeling of computer networking software. Instead, simple queues are generally used for modeling network nodes implementing networking software.

**Figure 3.1**—A spectrum of performance modeling techniques.



# 3.3 Personal computer-based software routers

### 3.3.1 Routers' rudiments

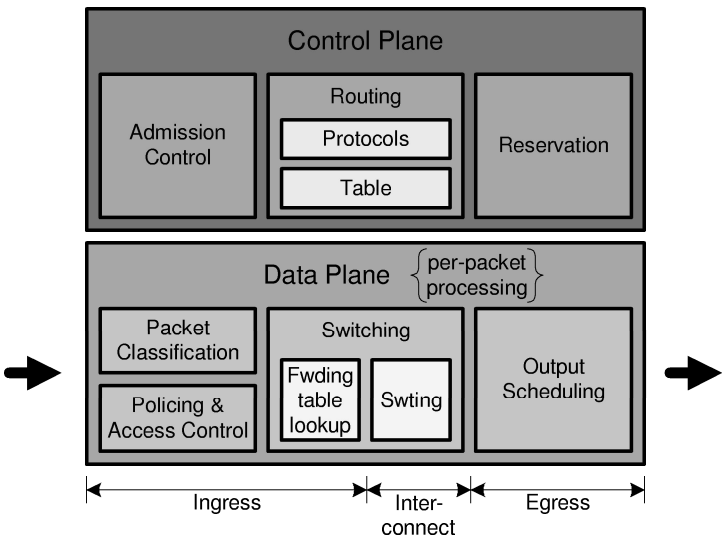
A router is a machine capable of conducting packet switching or layer 3 forwarding. Within the Internet realm, routers forward Internet Protocol (IP) datagrams and bind the subnets that form the Internet. Consequently, routers' performance directly influences packet's end-to-end throughput, latency and jitter.

In general, in order to perform its chief function, routers do several tasks [McKeown 2001]; see Figure 3.2. These tasks have different comply urgency and complexity levels, and are done at different time scales. For instance, routing is a complex task (mainly due to the size of the data it processes and because this data is stored distributively) that is done every time there is a change in network topology and may take up to few hundred seconds to comply [Labovitz et al. 2001]. On the other hand, packet classification has to be done every time a packet arrives and consequently should comply at wire speeds, where wire speed means, for example, around 5 microseconds for Fast Ethernet [Quinn and Russell 1997] and around 2 microseconds for Gigabit Ethernet (when operating half duplex and with packet bursting enabled) [Stephen 1998]. In order to cope with this operational diversity, routers have, in general, a two-plane architecture; see Figure 3.2.

### 3.3.2 The case for software router

Evidently, a router's ***data plane's*** implementation mostly determines a router's performance influence on packet throughput, latency and jitter. Naturally, different levels of router's features, like raw packet switching performance, support for extended functionality, cost and upgradeability, may be set using different implementation technologies for the *data plane*. A router's *data plane* may be implemented using the following technologies:

**Figure 3.2**—IP router architecture



- General-purpose computing hardware and software. For instance, a workstation or personal computer running some kind of Unix or Linux
- Specialized computing hardware and software. For instance, Cisco's 2500 or 7x00 hardware executing Cisco's IOS forwarding software
- Application specific integrated circuits. For instance, Juniper's M-120

For performance reasons it is interesting to classify routers in software and hardware routers.

- ***Hardware routers*** are routers whose data plane is implemented with hardware only. For instance, Juniper's M-120
- ***Software routers*** are routers whose data plane is implemented partly or completely with software. For instance, Cisco's 2500 and 7x00 product series, or a workstation or personal computer running some kind of Unix or Linux

Evidently, *hardware routers* outperform *software routers* in raw packet switching. At the Internet's core, where data links are utterly fast and expensive, *hardware routers* are deployed in order to sustain high data link utilization. At the Internet's edge, factors like multiprotocol support, packet filtering and ciphering, or above level 3 switching are more important than data link utilization, however. Consequently, at the Internet's edge *software routers* are deployed. Besides, *software routers* have other features that made them more attractive than *hardware routers* at particular scenarios; *software routers* are easier to modify and extend, have shorter times-to-market, longer life cycles, and cost less.

### 3.3.3 Personal computer-based software routers

Evidently, a software router's performance is affected by the following

- The host's hardware architecture
- The forwarding program's software architecture
- The network interface card's hardware and corresponding device driver architectures

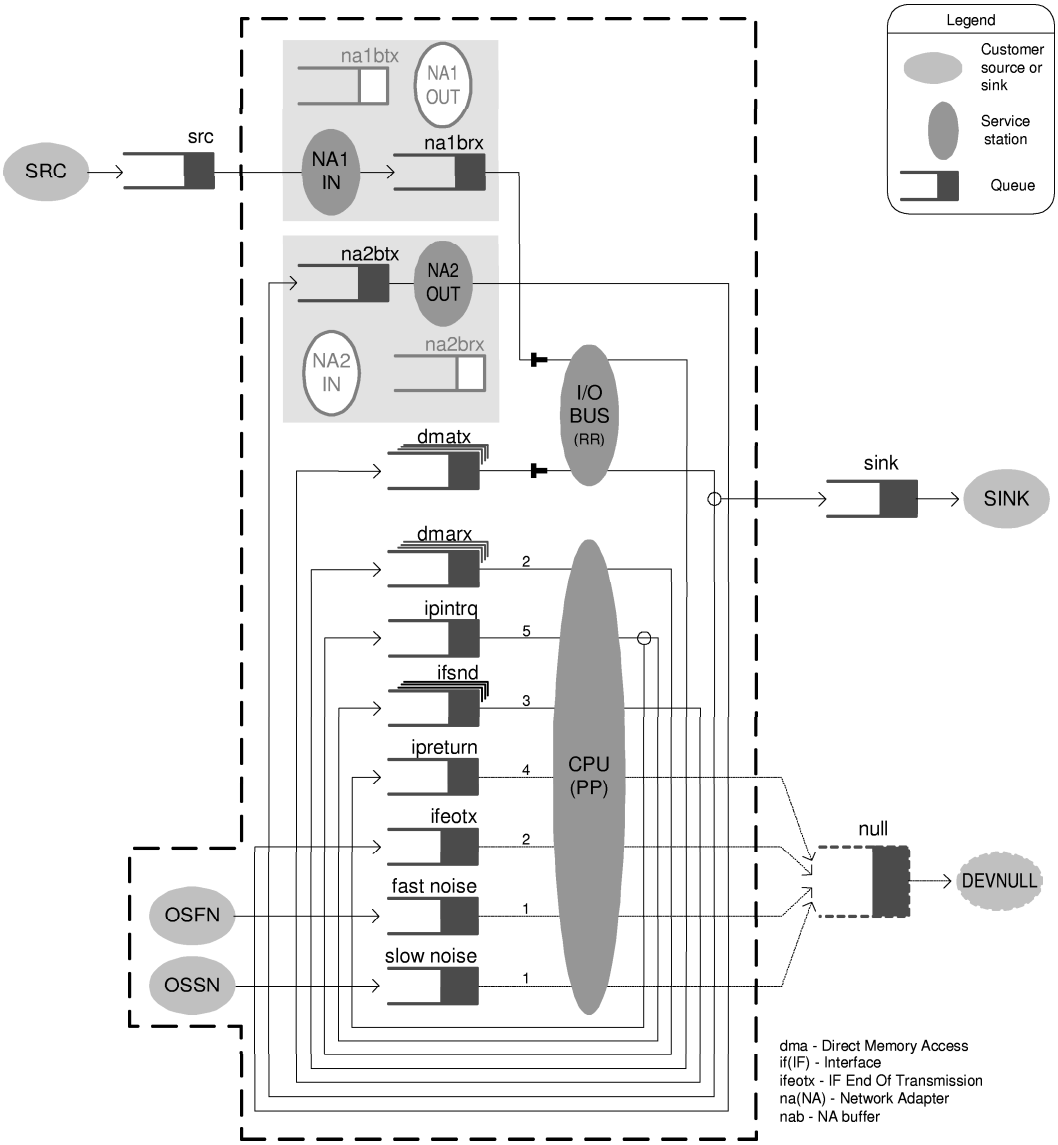
For this research we selected to work with personal computing technology. This technology provided us with an open work environment—one that is easily controllable and observable—for which there is lots of publicly available and highly detailed technical information. Moreover, other software routers have similar or at least comparable architectures so our findings may be extrapolated. For this research we used computers with the following features:

- Intel IA32 architecture
- Intel's Pentium class microprocessor as central processing unit
- Periphery Component Interconnect (PCI) input/out bus
- Direct memory access (DMA) capable network interface cards attached to the PCI input/output bus
- Networking software as a subsystem of the kernel of a BSD-derived operating system

# 3.4 A queuing network model of a personal computer-based software router

After this chapter's description of a personal computer-based software router, it should be clear that no single-queue model could capture the whole system behavior. In order to model the pipeline-like organization and priority preemptive execution of the BSD networking code, a queuing network model is at least required. Figure 3.3 shows a queuing network model of a personal computer-based software router. The model corresponds to a personal computer that executes the BSD networking code for forwarding IP datagrams between two DMA capable network interface cards. Moreover, the model is restricted to a router that is threaded by a single unidirectional packet flow. (Packets only enter the router through the number-one network interface card and only exit the

**Figure 3.3**—A queuing network model of a personal computer-based software router that has two network interface cards and that is traversed by a single packet flow. The number and meaning of the shown queues is a result of the characterization process presented in the next section



router through the number-two network interface card.) Models for routers with different number of network interface cards or different packet flow configuration may be derived after the one here shown as later explained. The shown model is comprised of:

- Four service stations, one per each router's active element: the central processing unit (CPU), the input/output bus control logic (I/O BUS), the network interface card's packet upload control logic (NA1IN) and the network interface card's packet download control logic (NA2OUT)
- Eight first-come, first-served queues (na1brx, na2btx, dmatx, dmarx, ipintrq, ifsnd, ipreturn and ifeotx) and their corresponding service time politics, which model network interface cards' local memory (na1brx, na2brx) and BSD networking mbuf queues. The number and meaning of the networking related queues, (dmarx, ipintrq, ifsnd, ipreturn and ifeotx) as well as the features of the service time politics applied by the CPU service station, are a result of the characterization process that section 3.5 describes. The service time politics applied by the I/O BUS service station (to customers at na1brx and dmatx queues) were not computed after our own measurements but after the results presented by Loeb et al. [2001]. A PCI bus working at 33 MHz, with a 32-bit data path, which data phases last 1 bus cycle, and whose data transactions were never preempted, was considered. The service time politic applied by the NA2OUT service station (to customers at the na2btx queue) corresponds to a zero-overhead synchronous data link of particular speed
- Nine customer transitions between queues representing the per-packet network processing's execution order. Cloning transitions, pictured by circles at their cloning point, result in customers' copies walking the cloning paths. Blockable transitions, having a "t" like symbol over the blocking point, result in service stations not servicing transitions' source queues, if transitions' destination queues are full
- One customer source for driving the model. It spawns one customer per packet traversing the router. An auxiliary first-come, first-served queue and its corresponding service time politic are used for modeling the communications medium's speed
- A set of two customer sources, two FCFS queues and two transitions through the central processing unit for modeling the *"noise" process*. This process is defined and its relevance explained in subsections 3.5.9 and 3.5.10.
- A set of one FCFS queue and one customer sink for computing per packet statistics
- A set of one FCFS queue and one customer sink for trashing residual customers

May this relatively simple model give useful predictions? Is this a better model of a software router than the widely used single queue model, under the considered scenario that the central processing unit is the system' s bottleneck a not the communications links? These are some questions for which we wanted an answer but couldn't find one before we started this research. But before elaborating on the quest for these an-

swers, the rest of this section is devoted to presenting modeling details. These details are important for deriving models of routers with different configurations.

### 3.4.1 The forwarding engine, the network interface cards and the packet flows

Model elements depicted at Figure 3.3 within the dotted line model the router. Model elements outside the dotted line are auxiliary elements. For modeling a router with more network interface cards some elements have to be replicated. Gray areas depict elements that conjunctively model a network interface card. The model requires one such group of elements for each network interface card that both inputs and outputs packets. Network interface cards that only inputs or output packets require only part of these elements. The model requires some other queues—and corresponding service time politics and transitions—to be replicated depending on the packet flow configuration. The figure depicts these queues with a shadow following their top right side. To further clarify this, consider the example shown in Figure 3.3. As said before, for this model, packets enter the router through one network interface card and exit it though the other. Consequently only a single `dmarx` queue and a single pair of `dmatx` and `ifsnd` queues are required—as you may already guess, one `dmarx` queue is required per each network interface card entering packets to the router and one pair of `dmatx` and `ifsnd` queues is required per each network interface card exiting them.

### 3.4.2 The service stations' scheduling politics and the mapping between networking stages and model elements

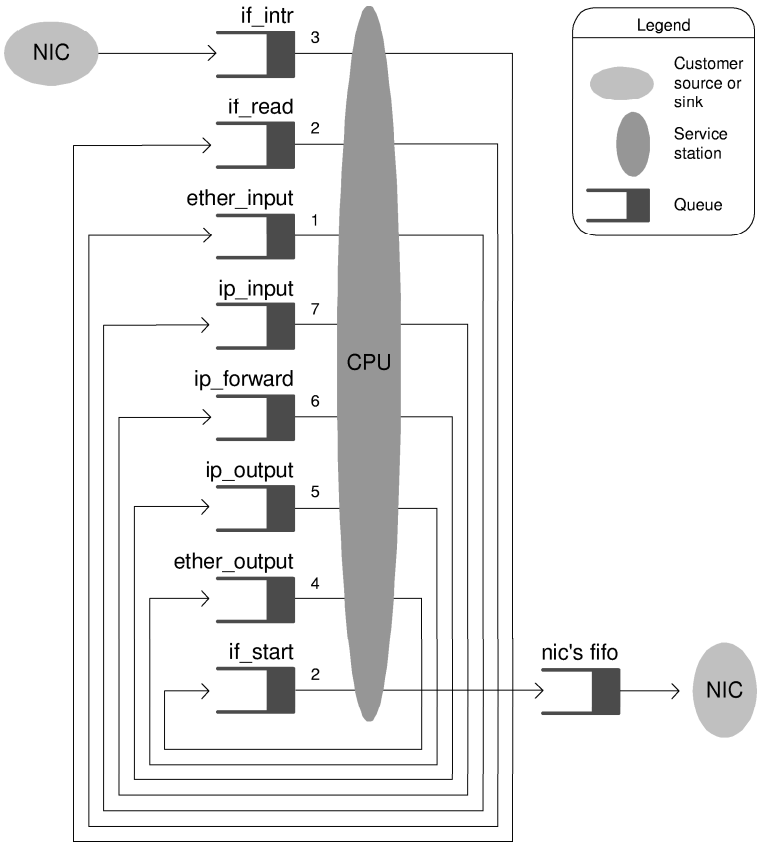
The service station acting as the input/output bus has a round-robin no-preemptive scheduling policy that mimics the basic behavior of the PCI bus' arbitration scheme. The service station acting as the central processing unit has a prioritized preemptive scheduling policy that mimics the basic behavior of the BSD's software interrupt mechanism. At any given time, this service station preemptively serves the customer up front of the queue with the smallest priority number. Figure 3.3 shows queue's priority numbers in front every queue served by the CPU service station. The priority scheme shown in Figure 3.3 models the fact that any task at the network-interfaces layer preempts any task at the protocols layer. It also models the fact that once a message enters a layer it's processing is not preempted by another message entering the same layer.

For better explaining the above consider Figure 3.4. This figure shows a router's model with a different mapping of networking stages to queues—and their corresponding service time politic. Besides, the input/output bus logic and the noise are not modeled and the network interface cards' models were simplified. This figure's model uses a one-to-one mapping between the C language functions implementing BSD networking and the model's queues. You can check this by comparing Figure 3.4 with previous chapter's Figure 2.5. Observed that the queues named `ip_input`, `ip_forward`, `ip_output` and `ether_output`, which represent tasks at the protocols layer, have all higher priority numbers than the queues named `ether_input`, `if_intr`, `if_read` and `if_start`, which are tasks at the network-interfaces layer. Moreover, the priority of `ip_input` is greater that the priority of `ip_forward`, which is greater than the priority of `ip_output`, which is greater than the priority of `ether_output`. This models the fact that a message recently arriving to `ip_input` waits processing until a message that is

being forwarding (`ip_forward`) completes the rest of the protocols layer's stages—that is, `ip_output` and `ether_output`. For the same reason the priority of `if_intr` is greater than the priority of `if_read` and `if_start`, which is greater than the priority of `ether_input`.

In order to complete this discussion let us note that while the model in Figure 3.4 is clearly a more detailed model of a software router with respect to that of Figure 3.3, it is not a better model. As will be explained in the next section, the model at Figure 3.4 gives inaccurate results due to service time correlations between some networking stages.

**Figure 3.4—**A queuing network model of a personal computer-based software router that shows a one-to-one mapping between C language functions implementing the BSD networking code and the model's queues. In order to simplify the figure, this model does not include the models for input/output bus and the noise. Moreover, it uses a simplified version of the network interface card model



# 3.5 System characterization

This section reports on the assessment of the model's service time politics, which are applied by the model's CPU service station. These service time politics model the execution times of the BSD networking code executed on behalf of each forwarded IP datagram. We used software profiling for assessing these execution times. Of the several tools and techniques that were available for profiling in-kernel software, software profiling is a good trade off between process intrusion, measurement quality and set up costs. We expected the profiled code to have stochastic behavior due to the hardware's features of cache memory and dynamic instruction scheduling—branch prediction, out-of-order and speculative execution. Thus, we strived to compute service time histograms rather than single value descriptors. We also carried out a data analysis for unveiling some data correlations and hidden processes that forced us to adapt the model's queuing network structure. The rest of this section is devoted to report the details of this process.

### 3.5.1 Tools and techniques for profiling in-kernel software

As stated before, for our research we used a networking code that is a subsystem of the kernel of a BSD-derived operating system. More specifically, we used the kernel of FreeBSD release 4.1.1. From the profiling point of view this kind of software presents a problem: the microprocessor runs the operating system kernel's code in supervisor (or similar) mode and, because of this, special tools and techniques are needed for profiling "alive" kernel's code. Where by "alive code" we mean code that is executed at full speed, in contrast to code that is executed step-by-step. Of the to-us-known tools and techniques for kernel's code profiling we used software profiling [Chen and Eustace 1995; Kay and Pasquale 1996; Mogul and Ramakrishnan 1997; Papadopoulos and Parulkar 1993]. This technique is a reasonably flexible tool that does not requires additional support. Contrarily, other tools, like programmable instrumentation toolkits [Austin, Larson and Ernst 2002] and logic analyzers, although more powerful are complex and proprietary and thus require a considerably amount of time, effort and money. For the kind of task at hand, the use of these tools was considered too timely, energetically and economically expensive.

### 3.5.2 Software profiling

A software probe may be defined as a little piece of software manually introduced at strategic places of target software for gathering performance information. The software probes, or just probes, are supplemented by data structures and routines that manipulate the recorded information. One important aspect when designing probes is minimizing its overhead.

We found two ways to use probes [Kay and Pasquale 1996; Mogul and Ramakrishnan 1997; Papadopoulos and Parulkar 1993]. Each way uses a different mechanism to gather information. One uses software mechanisms, resulting in instruments that, we can say, are self-sufficient. The other relays on special-purpose microprocessor-registers used as event counters. Both have tradeoffs. Software only probes

are portable but can incur in relatively higher overhead, depending on the complexity of the gathered information. Event-counters based probes are hardware dependant, and thus are not portable, but can gather information hidden to software, like instruction count mix, interruptions, TLB misses, etc.

As our reference systems wore Intel's Pentium class microprocessors, our probes take advantage of these microprocessors' performance monitoring event counters (PMEC). These kind of microprocessor has three such event counters, of which two are programmable [Shanley 1997]. Four model specific registers (MSR), which can be accessed through the `rdmsr` (read MSR) and `wrmsr` (write MSR) ring-0 instructions, implement the three PMECs. One MSR implements the PMEC's control and event selector register. It is a 64-bit read/write register employed for programming the type of event that each of the two programmable PMECs would count. Another MSR implements the time stamp counter (TSC), a read-only 64-bit free-running counter that increments on every clock cycle and that can be use to indirectly measure execution time. This register can also be accessed through the `rdtsc` (read TSC) ring-0 instruction. Finally, two MSRs separately implement the two programmable PMECs. Each of these is 40-bit read-only register that counts occurrences or duration of one of several dozen events, such as cache misses, instruction count, or interrupts.

The technique we employed for software profiling is as follows:

**1)** A probe is placed at the activation point and at each returning point of each profiled software object. Note that some of these objects have more than one returning point

**2)** When an experiment is exercised, each probe reads and records values from the three PMEC along with a unique tag that identifies the probe

**3)** The recorded information is placed in a single heap inside the kernel. We decided to do this, instead of using, let say, one heap-per-object, because in this way we found it is easier to observe and analyze object-activation nesting. Furthermore, a common heap facilitated calculating each object expenses in a chain of nested activations

**4)** When an experiment is finished, the recorded information is extracted from the in-kernel heap and is organized by an "ad-hoc" computer program. This program classifies the records by its source probe, computes the monitored event expenses per activation and writes the results in several text files—one per source probe. Each produced text file holds a table with one row per object activation and one column per PMEC. This kind of text files can be feed to most widely available statistical and graphical computer programs for analysis

### 3.5.3 Probe implementation

Before implementing the probes, there were some questions to answer. The first question was: which hardware events do we monitor? At first sight the answer to this question seams simple: because we were interested in measuring execution time, we had to monitor ***machine cycles***. But because we also wanted to know how those cycles were spent, and to see a proof of the concurrent nature of the software objects we instrumented, we decided to monitor as well ***instructions*** and ***hardware interrupts***. The *in-*

*struction* count will give us an idea of the path taken by the microprocessor through code. The *interrupt* count will show us the concurrent behavior of not only the instrumented objects but of the whole kernel. In turn, this will serve us to discriminate meaningless data: a high interrupt count would mean that the microprocessor spent too much time doing something else besides exercising the instrumented object.

A second question was: where do we place the profiling probes? Other way to look at this problem is to answer: what level of granularity do we use when delimiting code for profiling? A priori we choose a C language function level of granularity. This means that we flanked with profiling probes each of the C language functions implementing IP forwarding. From previous chapter' s Figure 2.5 it can be see that these functions are: `Xintr`, `Xread`, `Xstart`, `ether_input`, `ether_output`, `ipintr`, `ip_forward` and `ip_output`. However, as will be explained in the next section, we found out that this level of granularity did not result in appropriate model's service times. Therefore, a posteriori, we increased the level of granularity and flanked with profiling probes all the code executed from when an `mbuf` is taken out of any `mbuf` queue to when this `mbuf` is placed in another `mbuf` queue. This level of granularity resulted in the definition of the following five "profiling grains"; see Figure 3.3:

- The driver reception, `dmarx`; this grain spans `Xintr` (the reception part), `Xread` and `ether_input`
- The protocol processing, `ipintr`; this grain spans from the activation point of `ip_input`, through `ip_forwarding`, `ip_output` and `ether_output`, and up to the code at `ether_output` that either places the packet at the interface's transmission queue or activates `Xstart`
- The driver transmission, `ifsnd`; this grain corresponds to `Xstart`.
- The return from protocol processing, `ipreturn`; this grain spans from where the protocol processing grain left off, to the returning points of `ether_output`, `ip_output`, `ip_forwarding` and `ip_input`
- The driver end-of-transmission interrupt handler, `ifeotx`; this grain corresponds to the part in `Xintr` that attends the end-of-transmission interrupt that network interface cards signal

A third question was: how do we manipulate the PMEC? Searching the FreeBSD documentation on the web (http://www.freebsd.org) we found that this operating system implements a device driver named `/dev/perfmon`, which implements an interface to the PMECs. We also found that this device driver uses three FreeBSD kernel's C language macro definitions for manipulating the Intel's Pentium's MSR and for reading the TSC. These macro definitions, `wrmsr`, `rdmsr` and `rdtsc`, are defined in the header file `cpufunc.h` stored in the `sys/i386/include/` directory.

A fourth question was: how will we switch probes on and off? Papadopoulos and Gurudatta [1993] present one way to do it. Although their mechanism did not fit us, it did give us a hint. Like them, we decided to use a test variable for controlling probe activation. Departing from their methodology, we decided to use a spill-monitoring variable that we named `num_entries`. `Num_entries` would count the number of entries recorded in the heap and probes would be active as long as `num_entries` is less than the heap's capacity. At any given time we could deactivate the probes by writing a value bigger than the heap's capacity, and vice versa.

A last question was: how will we implement the heap of records. For answering this, we considered that heap manipulation would have to require as few instructions as possible. Thus, we decided not to employ dynamic memory and defined a static array of records. This has the inconvenient of limiting the size of the heap. (Compiler related restrictions limited the size of the profiling records heap to around 100 thousand entries.) Each record is implemented by a C language structure with the following members:

- `id` holds an integer value taken from a defined enumeration of probe identifiers.
- `tsc` holds the 64-bit number read form the TSC.
- `instruc` holds the 64-bit number read from PMEC1; which is initially configured to count executed instructions.
- `interrup` holds the 64-bit number read form PMEC2; which is initially configured to count hardware interruptions.

For the enumeration that defines the probe identifiers, we decided to employ a scheme that would allow us to readily identify both the source object and probe's location inside the object. Thus, each probe identifier has two 4-bits parts: one for identifying the source object and another for identifying the probe's location inside the object. Moreover, given that every profiled object has one activation point but many return points we decided to use the number 0x0 for identifying the activation point and numbers between 0x1 and 0xF for identifying returning points.

With all this set up, we were able to devise a sequence of instructions that would work for any probe. In order to ease the probe codification process, we defined a set of C language macro definitions that were added to the `kernel.h` header file stored in the sys/sys/ directory. The text for the macro definitions is shown in Figure 3.5.

**Figure 3.5—**Probe implementation for FreeBSD

```
#if defined(I586_CPU)
#define NAVI_RDMSR1 rdmsr(0x12)
#define NAVI_RDMSR2 rdmsr(0x13)

#elif defined(I686_CPU)
#define NAVI_RDMSR1 rdmsr(0xc1)
#define NAVI_RDMSR2 rdmsr(0xc2)

#endif

#define NAVI_REPORT(a) \
  if ( navi_entrynum < NAVI_NUMENTRIES ) { \
    disable_intr(); \
    navi_buffer[navi_entrynum].id = a; \
    navi_buffer[navi_entrynum].tsc = rdtsc(); \
    navi_buffer[navi_entrynum].instruc = NAVI_RDMSR1; \
    navi_buffer[navi_entrynum++].interrup = NAVI_RDMSR2; \
    enable_intr(); \
  }
```

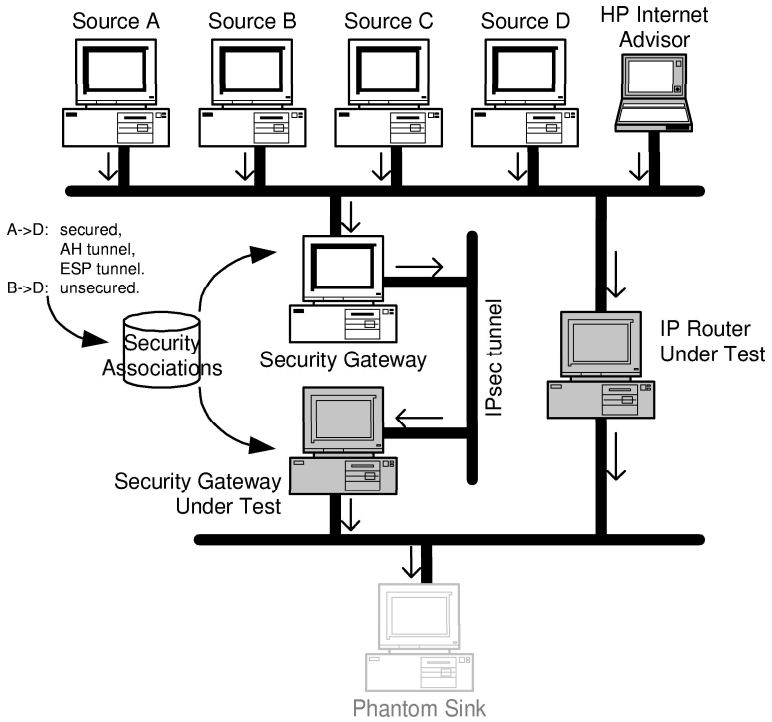
### 3.5.4 Extracting information from the kernel

Extracting special information from a "live kernel" is a well-known procedure. You just have to implement a new system call. This is trivial but cumbersome. FreeBSD has an easier way to go: the `sysctl(8)` subsystem. This subsystem defines a set of C language macro definitions for exporting kernel variables, and the system call `sysctl(2)` for reading and writing the exported variables. The scheme for identifying variables is similar to the one use for a management information base, MIB [Stallings 1997]. The header file `sysctl.h`, stored in the `sys/sys/` directory, has the definitions of the macro definitions, and the file `kern_mib.c`, stored in the `sys/kern/` directory, uses the macro definitions for initializing the subsystem. The `sysctl(8)` and `sysctl(2)` man pages explain how to use this subsystem, but they do not say how to extend it. Honing [1999] explains how to accomplish this.

### 3.5.5 Experimental setup

Figure 3.6 shows the experimental setup we used for system characterization and model validation. We characterized and modeled two systems: a plain IP router and an IP router configured as a security gateway [Kent and Atkinson 1998]. In any case, we wanted the systems under test's central processing units to be the bottlenecks. That is why the IP router under test wore as its central processing unit an 100 MHz Intel's Pentium microprocessor, while the security gateway under test wore a 600 MHz Intel's Pentium III. Indeed, it is well known that the software implementing authentication and encryption algorithms consumes more computing power than plain IP forwarding. (After our research we now know exactly how much more computing power a security gateway consumes when compared to a plain IP router.)

**Figure 3.6**—Experimental setup



Traffic generation was accomplished using a modified version of Richard Stevens' `sock` program [Stevens 1994, appendix C] and using the traffic generation facility of a Hewlett-Packard's Internet Advisor J2522B. During system characterization experiments were driven only by a single `sock` source, while during model validation we used a mix of sources. As we did not have any special requirements for the source nodes, besides being able to run the `sock` program, we used whatever we had at hand. Thus, sources A and B wore 100 MHz and 120 MHz Intel's Pentiums, respectively; source C wore a 233 MHz Intel's Pentium II and source D a 400 MHz Intel's Pentium III. During experiments with the security gateway, in order to avoid that performance limitations at the IPSEC tunnel's inbound security gateway would modify the traffic's features or interfere with the characterization of the security gateway under test, which was the outbound one, the inbound gateway wore a 650 MHz Intel's Pentium III. The "phantom sink" was implemented using an address resolution protocol (ARP) table entry for an inexistent node.

All the computers shown are personal computers with Intel IA32/PCI hardware architecture executing FreeBSD release 4.1.1 as their operating system. Main memory configuration varies between 16, 32 and 64 Megabytes. In any case, it was more than enough for storing the router's working set; that is, the most used sections of the router's software. Indeed, the FreeBSD kernel's image we used was relatively very small; it did not occupy more than 3 Mbytes. (1 Mbytes equals 2^20 bytes.) This image included, besides the normal stuff, (the kernel was configured to include only a reduced set of features [Leffer and Karels 1993]) the profiling probes and the storage for the heap of profiling records. Of course, the main memory space require for storing a FreeBSD system's data varies as a function of the software load offered to the system—number of processes, devices, open files, open sockets, etc. In our case this was not an issue because we configured each FreeBSD system so it would only execute the chief system processes—`init`, `pagedaemon`, `vmdaemon`, `syncer` and only a few `getty`. With this configuration we wanted to avoid uncontrolled process context changes that will drain computing power.

All computers had 3COM 905B-TX PCI/Fast Ethernet network interface cards attached to either 10 Megabit per second Ethernet or 100 Megabit per second Fast Ethernet channels. A 3COM Office Connect TP4/Combo Ethernet hub was used for linking all sources, the router under test and the inbound security gateway. A point-to-point Fast Ethernet link was used for the IPSEC tunnel. FreeBSD kernels were configured to use the `xl(8)` device driver for controlling the 3COM network interface cards. All computers had onboard EIDE/PCI hard disk controllers. A plain and basic FreeBSD system requires around 400 Megabytes of disk storage for holding the chief file systems. With the considered configuration, no swap space was required.

### 3.5.6 Traffic pattern

The traffic pattern for system characterization served to purposes:

- Minimizing the preemption of networking stages
- Avoiding that packet arrivals would be in synchronization with any kernel supported clock

The first feature was important for easing the system characterization process. Recall that we have set up probes at the activation and returning points of selected software tasks, and that each probe records the current value of the three PMECs. Therefore, the actual resource consumption of any profiled task is equal to the subtraction of the values recorded at a returning point minus the values recorded at its corresponding activation point. Observe that any preemption of a profiled task, which will occur between its activation and returning points, will result in a record reporting higher resource consumption.

We saw two ways to solve this problem. One was to build a computer program that would identify overblown records and would try to eliminate from them the preemption costs. The other solution was to use a traffic pattern that would assure the system under test would forward one packet at a time. The tradeoffs here were the experiment's run time and the solution's complexity. Clearly, the second solution is simpler. As for the experiment's run time it turn out not to be an issue. Indeed, given that the heap of profiling records was limited in size to 100 thousand entries and that the forwarding of a packet produced around 12 entries, experimental runs can only be as long as to produce 8333 packets. At the same time, our slowest system under test would forward one packet in less than 80 microseconds—as we found out a posteriori. Consequently, we used the second solution.

The traffic's second feature, avoiding synchronization with kernel clocks, is important for dealing with software preemptions not related to networking tasks. A FreeBSD kernel includes tasks that attend urgent events, like the system real-time clock's interrupt-handler, or a page-fault exception's or hard disk's interrupt-handler. Such tasks are executed with high priority and thus preempt any networking task. Our systems under test, which had a simplified configuration as stated before, only produced urgent events related to the real-time clock and therefore launched high priority tasks following a periodic pattern—with different periods but all being a multiple of the real-time clock's period. A posteriori we found out that if sources (which were also FreeBSD systems using the same kind of real-time clocks to time the traffic production) produced IP datagrams periodically, packets arrivals at the system under test may get in synch with one or more periodic and high priority kernel tasks. This situation would result in a number of overblown measurements larger than when the traffic is not synchronized with these kernel tasks.

Differently from the first problem, this one is unavoidable. Moreover, as we found out a posteriori, a way for characterizing this preemption process is required for producing an accurate system model. Consequently, the traffic pattern should help characterize this preemption process. The technique we used for characterizing this process, as will be explained later, required that the sequence of packets experiencing preemption would not be a periodic process. And this explains the traffic's second feature.

All the above resulted in the following traffic pattern for system characterization:

- A packet trace with random packet inter-arrival times of at least 2 times the mean system under test's packet processing time and as much between 3 and 10 time this mean value

### 3.5.7 Experimental design

We carried out a large set of experiments for assessing the influence that various operational parameters have over service times. The parameters we considered are:

- Packet's size
- Inter-packet arrival time
- Packets burst's size (number of packets in a packets burst)
- Encryption protocol
- Authentication protocol
- Traffic mix
- Routing table's size

Besides, we carried out some more experiments for assessing the behavior of various system elements. The elements we considered are:

- Code paths
- Central processing unit speed
- Main memory
- Cache memory

Finally, we carried out some more experiments for assessing the overhead and intrusion of the profiling probes.

### 3.5.8 Data presentation

Upon the data gathered from each PMEC of each profiling grain (section 3.5.3) at each experiment, we produced a set of data descriptors and corresponding charts. As stated in this section's introduction, we expected the profiled code to have stochastic behavior and thus we strived to produce histograms as data descriptors. For each experiment we produced the following six charts:

- Central processing unit cycle count trace and histogram (2 charts)
- Machine code instruction count trace and histogram (idem)
- Hardware interrupt count trace
- Correlation between cycle count and instruction count

Figure 3.7 and Figure 3.8 show some example charts sets. Appendix A shows the charts sets for all the experiments.

### 3.5.9 Data analysis

At the instruction and cycle count charts in Figure 3.7 and Figure 3.8, it can be seen that points form lanes. From the correlation cycles/instructions chart it can be seen that each lane in the instruction count chart correlates to a single lane in the cycle count chart. This was something expected as the software probes at each layer were programmed to profiled more than one routine. Thus, each data lane represents a code path through the networking software. In the case of Figure 3.7, the upper lane corresponds

to the Encapsulating Security Payload (ESP) protocol, the middle lane corresponds to the Authentication Header (AH) protocol, and the lower lane corresponds to the Internet Protocol (IP) protocol. In the case of Figure 3.8, the upper data lane corresponds to the networking message reception routine while the lower lane corresponds to the end-of-transmission interrupt handler. From the charts alone one cannot tell which data correspond to each path. To do this one has to analyze the recorded data with respect to the identification tag written by the software probes with each performance record.

It is striking although expected that the lanes at the instruction count charts are narrow—in fact these are single valued—while they are thick at the cycle count charts. This clearly proves our assumption that the networking code would have stochastic behavior and the importance of representing the model's service times with histograms.

From the shown charts it is evident that there are some kernel activities not related to networking processing that are consuming system resources. This is evident from the existence of several outlier points at the cycle and instruction charts. Moreover, these

**Figure 3.7**—Characterization charts for a security gateway's *protocols layer*

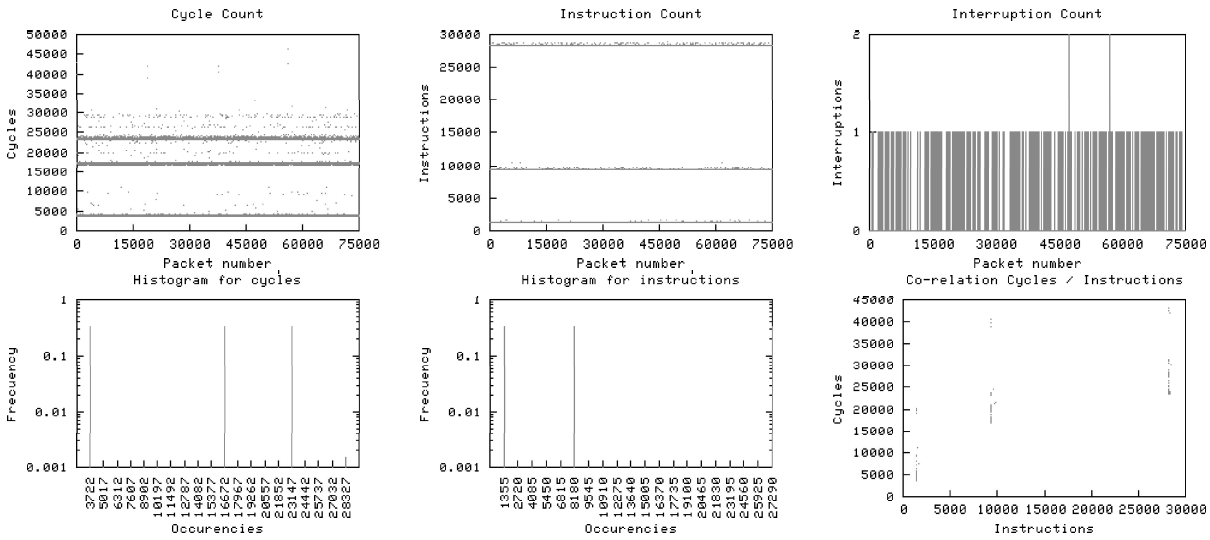
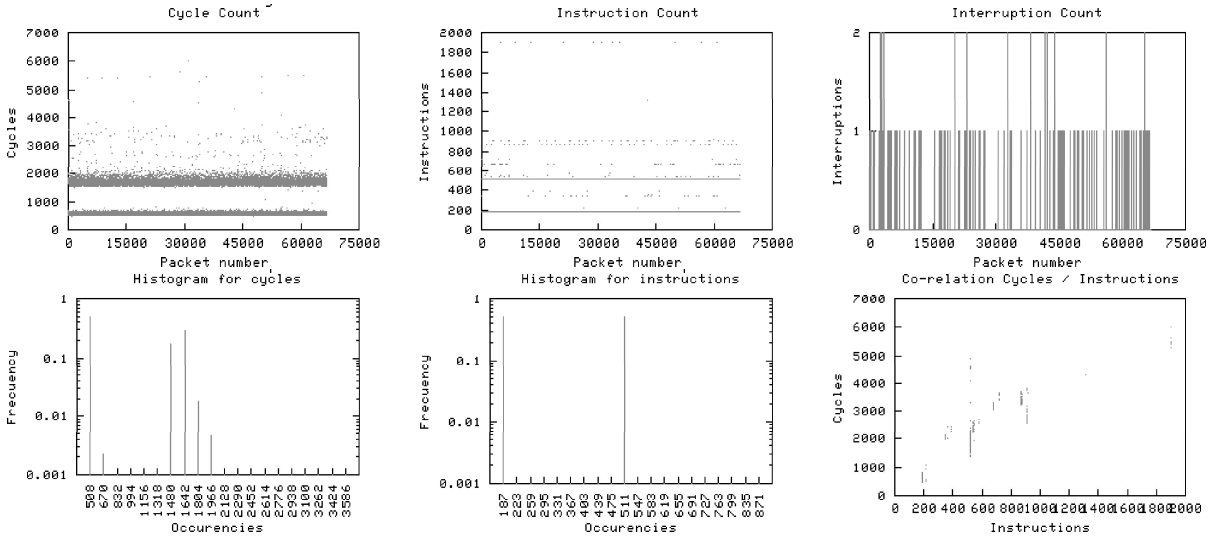



**Figure 3.8**—Characterization charts for a router's *network interfaces layer*



points correspond to probe records detecting at least one interruption—there is a one to one relationship between these points and impulses in the interruption count chart. For us this clearly implies that the networking activities were being preempted by some other not-instrumented activity. We named this activity, or collection of activities, the ***"noise" process***, as it is an ever-present process which influences system performance but over which we have no control. In order to model this "noise's" influence we added a set of elements to the router's model, as described in section 3.4. Consequently, when computing service time histograms we disregarded the outlier measurements.

### 3.5.10 "Noise" process characterization

A precise representation of the *"noise" process* would require analyzing activation sequences for several kernel tasks. A priori we decided not to do this, as it is a laborious task. Instead, we tried a simpler approach. A posteriori, the used approach turn out to be enough for producing highly accurate results, as later section 3.6 shows.

The used approach's objective was to estimate the probability density function (PDF) of two random variables: the inter-arrival time of "noise" events and the number of central processing unit cycles consume by these events. The motivation for going this way arose when doing initial model validations, as later discussed in section 3.6. Indeed, we were able to determine that the packet latency measures clustered at the second band shown in most charts were due to packets that were preempted at least once. Under the assumed load conditions, this preemption can only be due to the *"noise" process* and so finding plausibly relationships between the second band measures and this process may solve the problem.

The first relationship we found was associated to "noise" events' inter-arrival times. The inter-arrival time of "noise" events must be constant—as we have associated this process to activities tied to the system's real-time clock. Therefore, we only had to estimated its period or rate. We did this by computing the relative frequency of the second band measures. Considering no correlation between the occurrence of "noise" events and packet arrivals (something that we accomplish by using a special traffic pattern as stated before in subsection 3.5.6), we computed a mean arrival rate for "noise" events as $\lambda = p / T$, where $p$ is the probability of a record detecting a preemption and $T$ is the mean service time computed over records that did not detected preemption. The estimated period of 4.2 milliseconds is very closed to the 4 milliseconds period of some bookkeeping kernel routines reported in the literature [McKusick et al. 1996].

After settling that the *"noise" process* was a periodic process we proceed to estimate the cycle count's PDF. The fact that this process was periodic was important as this allow us to discard one random dimension and consider that any variation in the second band measures was just related to the randomness of the service times. (See section 3.6) Given this and the fact that latency times experience by preempted packets under the assumed load conditions are equal to the sum of a unpreempted service time plus a value associated to a "noise" event' service time, we decided to use the average numbers—mean and variance—that characterize the second-band packet-latency measures for characterizing, after some manipulations, the "noise" service times. Where by manipulations we mean subtracting the means and heuristically finding a well-known PDF

that best fits the observed values—given that its mean and variance are known. We tried uniform, exponential and normal PDF but the best fit was given by the normal PDF.

## 3.6 Model validation

This section presents a validation for our IP router's and security gateway's queuing network models. The validation is based on a two-dimension comparison (we elaborate on this shortly) between the queuing network models' predictions and both measurements taken from real systems and predictions from a simpler single queue model, as stated in this chapter's introduction. All the models were solved by computer simulation, using a computer program built by us.

One dimension of the comparison contrasts per-packet processing latency traces. (For simplicity, for now on we will refer to the per-packet processing latency as latency.) In this dimension we can qualitatively compare the temporal behavior of the systems. The other dimension compares the complementary cumulative probability functions, for now on CCPF, of the latency traces. Through this dimension we can quantitatively compare the performance of the systems at several levels of detail. For example, we can either do a broad comparison by comparing mean values or we can assess different service levels by watching at different percentile values.

Performance experiments were devised for stressing different system behavior. Experiments varied accordingly to the operational parameters of packet length and inter-packet time-gap. In spite of all the possible combinations of these two parameters, two main kinds of traffic are identified. On kind has fixed size packets produced with constant inter-packet time-gap, for now on IPG. The other kind has fixed size packets produced with randomly variable IPG.

The traffic of the first kind was initially intended to validate service time distributions computed after the characterization process. Later, it was also used for validating the *"noise" process's* model. It is interesting to note that the equipment we used to produce the traffic—an HP Internet Advisor—was not able to deliver a pure constant IPG traffic at all IPG values. For IPG values below 1 millisecond, the equipment produced packets bursts 1 millisecond apart. The resulted traffic had an average IPG equal to the one we selected but the IPG between packets in a packets burst was almost equal to the minimum Ethernet inter-frame gap. While this was firstly annoying, we were able to take advantage of this "equipment feature" and used the resulted one mega packets bursts per second traffic for some validation experiments. In these cases, experiments varied according to the number of packets within a packets burst, or packets burst's size.

The traffic of the second kind was devise to resemble usual traffic on a real local area network; that is, a traffic with a high ratio between its peek and average packet rate. It was produced by the superposition of four on/off sources with geometrically random state periods. During the on state, the traffic was produced with a geometrically random IPG.

When watching at the validation charts it is important to take into account that all systems, the real one and both models, were feed with exactly the same input traffic. We assured this by gathering traffic traces during the performance-measuring experiments

that we carry on with the real IP router, and later feeding these traces to both models runs.

### 3.6.1 Service time correlations

Before presenting the final model validation, allow us here discuss on the service time correlations we observed when using the one-to-one mapping between the C language functions implementing BSD networking and the model's queues. Subsections 3.4.2 and 3.5.3 presented this problem's implications on system modeling and probe implementation, respectively. Figure 3.9 presents the CCPF comparison for packet latency traces gather after some experimental runs. It clearly shows that a router's model with queues mapped as stated before is not a good model. Figure 3.10 presents an example chart we produced after a service times' correlation analysis. It clearly shows that

**Figure 3.9**—Comparison of the CCPFs computed after measured data from a software router and predicted data from a software router's queuing network model, which used a one-to-one mapping between C language networking functions and model's queue

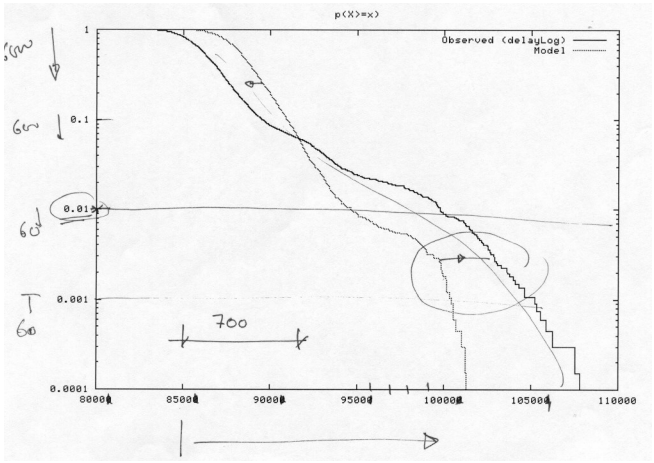
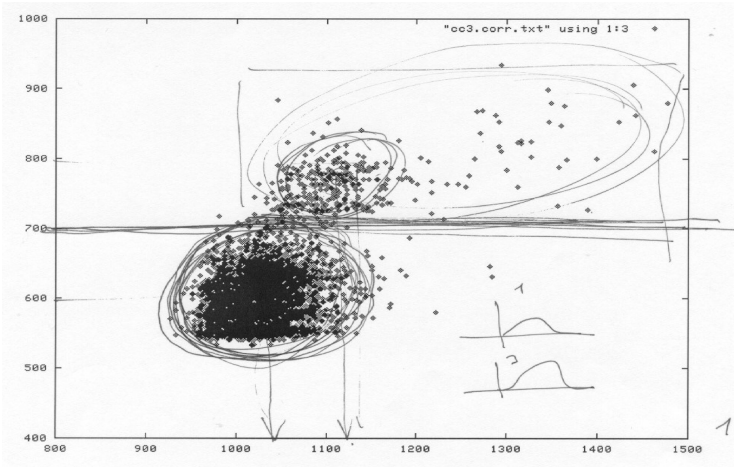



**Figure 3.10**—Example chart from the service time correlation analysis. It shows the plot of `ip_input` cycle counts versus `ip_output` cycle counts. A correlation is clearly shown



networking functions' execution times are not independent. As stated in subsections 3.4.2 and 3.5.3, these results drove changes both in the final model and the probe's implementation.

### 3.6.2 Qualitative validation

For carrying out the qualitative validation, as defined in this section's introduction, we produced latency traces charts like the ones shown along the two leftmost columns of Figure 3.11. There, each row corresponds to a particular experiment. Within each row, the left-hand chart depicts data measured at the system under test while the center chart depicts data estimated by the queuing network model. Results at a glance qualitatively validate the queuing network model. Indeed, at all rows both charts have a lot more similarities than differences. But beyond this broad comparison there are several details worth to have a look at.

The first interesting thing that shows up in the charts is the existence of two or more horizontal bands. These are more evident at the charts for the 100 MHz router but they also exist at experiments with the 600 MHz security gateway. The first band from the bottom up corresponds to packets that arrived at the system when it had empty message queues and were serviced without preemption. The variability at this band corresponds to the system's stochastic service-times. The second band (again, from the bottom up) corresponds to packets that although arriving to an empty system experienced some preemption due to the *"noise" process*.

Note that the first two bands are omnipresent in charts, and that charts corresponding to experiments with traffic of the first kind show no other bands. Differently, charts for experiments with "bursty" traffic (traffic with packets bursts) depict some other bands. These other bands correspond to packets that experienced some queuing time; they arrived to the system when this had non-empty message queues. These bands' large variance is a consequence of random message queue lengths and some additional preemption time due to the *"noise" process*. The number of these additional bands (additional to the first two) relates to the size of the driving traffic's packets bursts.

One discrepancy between the real systems' behavior and their queuing network models' is apparent when looking closely at the measures bands: measures bands' variances are a little different. The source of this discrepancy relates to the modeling of the *"noise" process*. As explained in subsection 3.5.10, our "noise" characterization trades off accuracy for simplicity. However, as next section shows, this discrepancy only results in a minor quantitative error and thus we disregarded doing a more accurate noise characterization and modeling.

---

### 3.6.3 Quantitative validation

For carrying out the quantitative validation, as defined in this section's introduction, we produced charts comparing three latency's CCPFs: the real system's, the queuing network model's and the single queue model's. Example charts are shown in the right column of Figure 3.11. Each chart depicts curves produced after the latency traces shown in the same row. Again, results at a glance quantitatively validate the queuing network model and reveal its better suitability with respect to the single queue model. Beyond this broad comparison there are some details worth to have a look at.

One interesting thing to note is the stepped nature of the real system's and queuing network model's curves. This is most evident at experiments with constant IPG. This nature relates to the central processing unit sharing between packets and the *"noise" process*. Differences in the prominence of the steps relates to differences between the *"noise" process'* service times' real density function and the estimated one. From the charts it should now be evident that the quantitative error is minor.

Another interesting thing to note is that the error introduced by the statistical estimation of the *"noise" process*' service times diminishes, as the traffic gets "bursty". This is striking although reasonable. Indeed, the average single-packet service time is significantly larger than the average "noise" service time. Thus, when packets bursts appear, the packet sojourn time is more affected by the size of message queues than by the central processing unit sharing with the *"noise" process*.

---

**Figure 3.11—**Model's validation charts. The two leftmost columns' charts depict per-packet latency traces. The right column's chart depicts latency traces' CCPFs

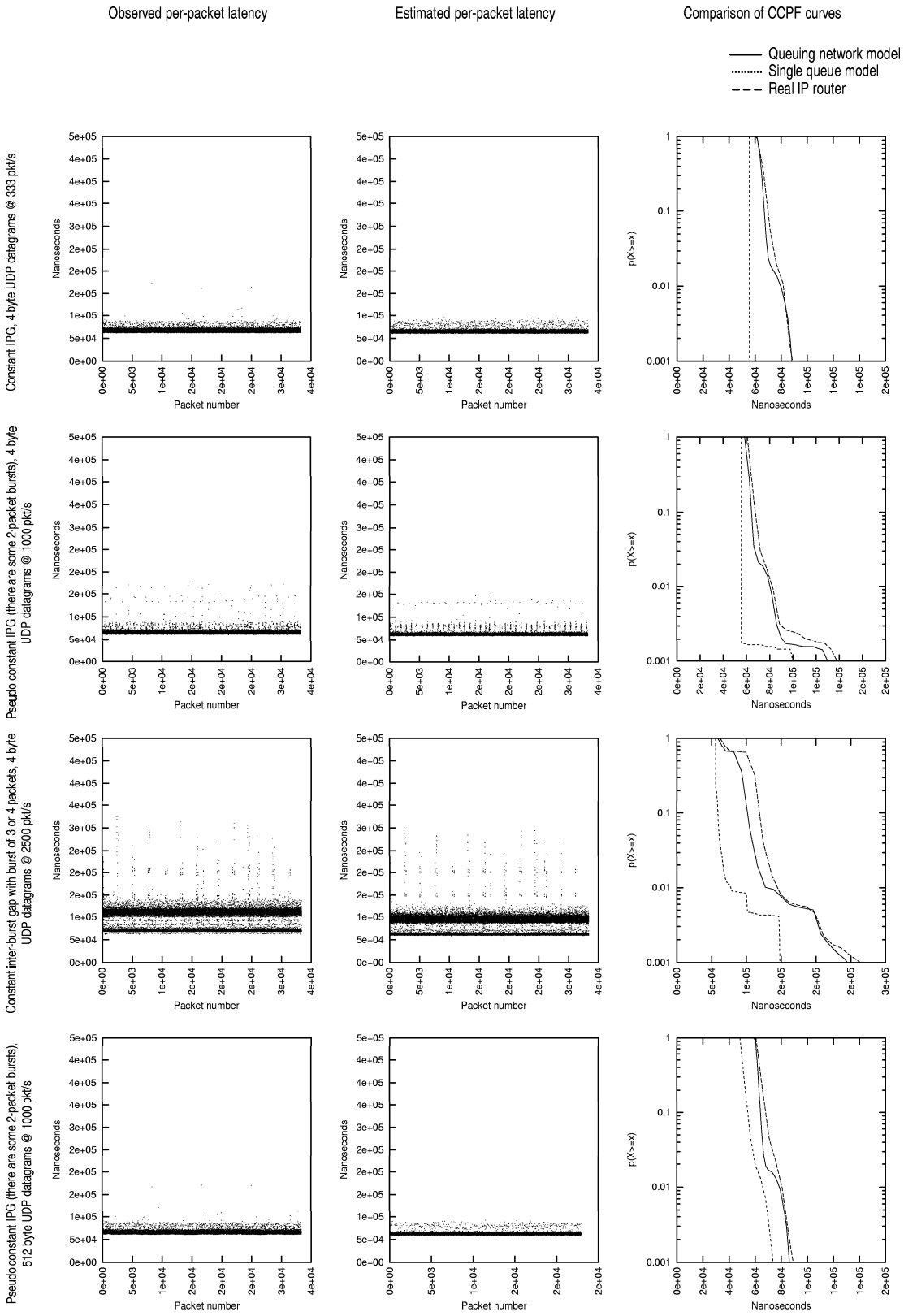
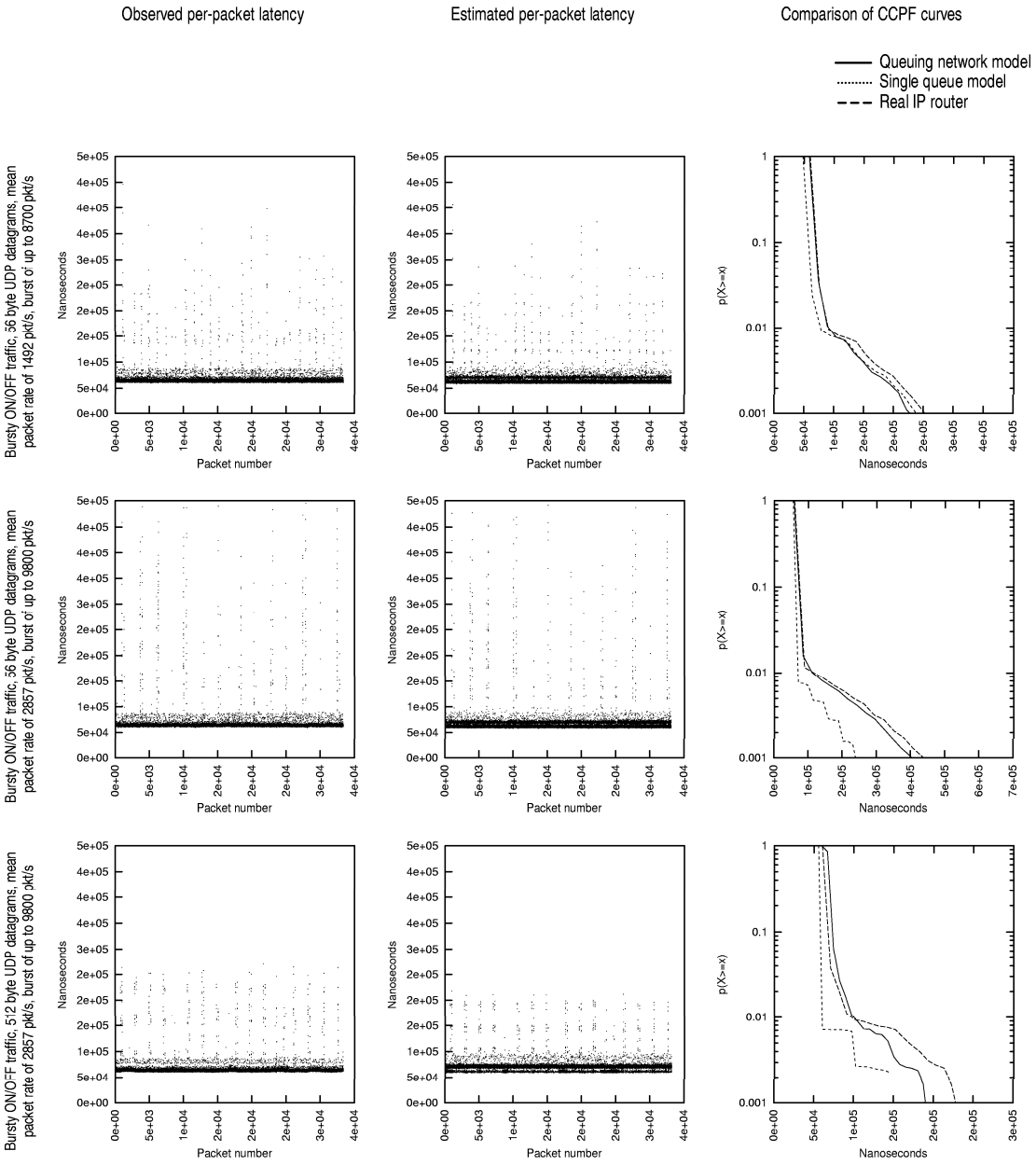


**Figure 1.11**—continuation



# 3.7 Model parameterization

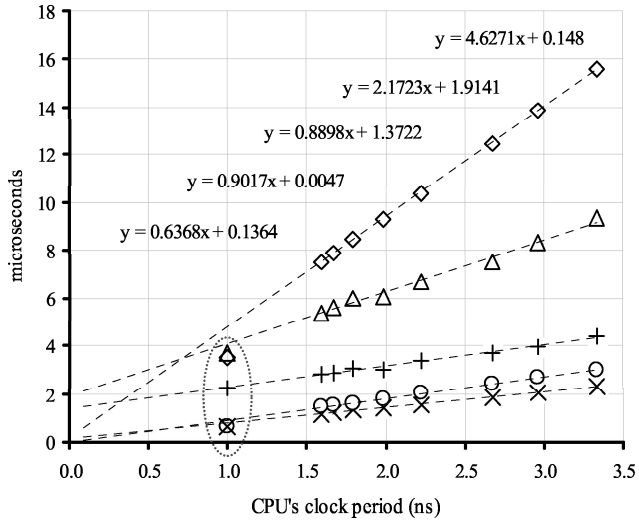
The previous section showed that it is possible to characterize a particularly slow software router and use the resulted service times for producing highly accurate predictions with a queuing network model of the same system. This section's objective is to present answers to the following questions:

- Is it possible to extrapolate our findings for performance evaluation of faster systems?
- From what point changes in hardware architecture or traffic pattern affect our model predictions?

## 3.7.1 Central processing unit speed

For answering this section's first question we produced Figure 3.12. The figure shows the execution times measured by the software probes of the plain IP router code when the operation speed of the router's central processing unit, for now on CPU, was varied between 300 MHz and 630 MHz. We were able to do this thanks to a feature of the Gigabit GA-6VXE+ motherboard and the Intel's Pentium III type SECC2 microprocessor that one our test systems had. A set of switches on this motherboard allowed us to change the operation speed of this type of microprocessor. Let us emphasize the importance of this feature. Without it, we would have need to change microprocessors, which may have different cache memory configurations, and even whole motherboards, which most certainly would have different chipsets, for experimenting at different central processing unit speeds. This would result in experiments with too many latitudes that would produce data too complex to analyze and thus useless.

**Figure 3.12**—Relationship between measured execution times and central processing unit operation speed. Observe that some measures have proportional behavior while others have linear behavior. The main text explains the reasons to these behaviors and why the circled measures do not all agree with the regression lines



Returning to Figure 3.12, it can be seen that `ipintrq`'s, `ifsnd`'s and `ipreturn`'s execution times are proportional to the CPU speed, while `dmarx`'s and `eotx`'s execution times vary linearly with respect to the same variable. (See subsection 3.5.3 for the definition of these software probes.) This has a clear explanation. The second set of probes profile code that executes some input/output (I/O) instructions for communicating through the I/O bus with the network interface cards. The operation speed of the I/O bus was constant during the experiment and therefore the time required to execute the I/O instructions did not scale with the CPU speed but remained constant. This results in the offset observed for `dmarx`'s and `eotx`'s curves. Note that the value of this offset depends not only on the I/O bus speed but also on the network interface card and corresponding device driver architectures.

In order to complete this analysis we included in Figure 3.12 a set of measurements taken from a different computer. These measurements are circled shown in the figure. The "new" computer had a 1 GHz Intel's Pentium III with different cache configuration and a different motherboard, an ASUS TUV4X with different chipset and main memory chips' technology. Both computers' I/O buses operated at identical operation speeds, however. As the figure shows, not all "new" computer's measurements agree with the regression lines of the "old" computer's measurements. In fact, only `eotx`'s new measurement agrees, the rest are lower than their corresponding regression line. Moreover, `ipintrq`'s new measurement is the farthest from its corresponding regression line.

We believe there is no single source for these discrepancies. Moreover, as the compared computers have no few differences we cannot categorically be sure what these sources are. However, after knowing that the router software's performance is not influence by the memory technology and that the router software is small enough to fit inside the CPU cache memory, see next subsection, we believe the discrepancies' main source is the different level two cache suited by each CPU. Indeed, the SECC2 600 MHz Intel Pentium III had an on-package, half speed, level-two cache while the FC-PGA2/370 1 GHz Intel Pentium III had an on-chip, full speed, advanced transfer, level-two cache. What we believe this means is that the "new" computer, when compared to the "old" one, not only executed instructions faster but also had its pipeline fuller or with less bubbles as, on average, it did not had to wait as much for the instructions and their data.

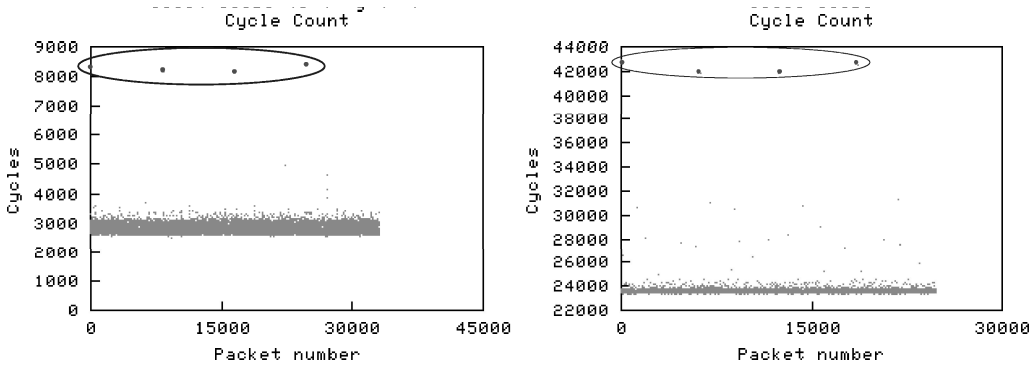
Clearly, different code segments profited differently after this cache's performance improvement. Although we have not done a detailed code analysis, we believe it is mainly due to the relative code sizes each software probe profiled and the mix of instructions. Basically, we believe it is reasonable to expect that a large piece of code without I/O instructions would profit more from a faster cache than a small piece of code with many I/O instructions. `Ipintrq` is the probe that profiled the biggest piece of code, 1261 instructions, and none of them are I/O instructions. We believe that is why its 1 GHz measurement is the farthest from its corresponding regression line. The other code sizes are: `dmarx`, 475; `eotx`, 236; `ifsnd`, 216; and `ipreturn`, 192. As stated before, `dmarx` and `eotx` profiled some I/O instructions. We believe the reason `eotx`'s 1 GHz measurement is just over the regression line is that the profiled code has a relative large number of I/O instructions. However, we have not assessed this.

### 3.7.2 Memory technology

We found that the working set of the BSD networking code for both the router and the security gateway fits inside their CPU's cache memory. Partridge et al [1998] have found similar results. Therefore, these systems' performance is not affected by main memory technology. We observe this after analyzing several cycle-count traces from different profiling probes. The analysis consisted in filtering preempted records—records whose corresponding interrupt count was not zero. Figure 3.13 shows example charts from two different profiling probes: the `ipintrq` probe and the probe profiling the DES algorithm for the Encapsulating Security Payload protocol (ESP). The figure shows four outlier measurements circled in each chart. Each of these records corresponds to the first packet being processed by the system after some non-networking-related activities have been executed. Observe that after these first packets are processed, no other packet experiences a high cycle count. For Figure 3.13 this means that only four out of almost 25 (or 35) thousand packets were affected. This suggested to us that during the processing of the so-called first packet the CPU's instruction cache should have been out of networking instructions, and thus the CPU required extra cycles for reading the instructions from main memory. But this also suggested to us that all other packets were processed by instructions read from the CPU's cache, and thus did not require reading from main memory. Therefore, this subsection's premise gets sustained.

For completing the picture, allow us here discuss why is it that we hold that some non-networking related activities trashed the CPU's instruction cache. It happened that because we could only allocate a probe-record heap big enough for storing 100 thousand records—for all the probes in the kernel—we had to break each experiment in four rounds. After each round, we have to start a shell session in the system's console and interact with the system. During this interaction we executed several shell commands for extracting the heap of profiling records from the kernel's address space and save it to a file. Then, we executed some other shell commands for preparing the system for the next round and for launching the round. Clearly, after all this, the CPU's instruction cache was filled with whatever instructions but the ones of the networking software.

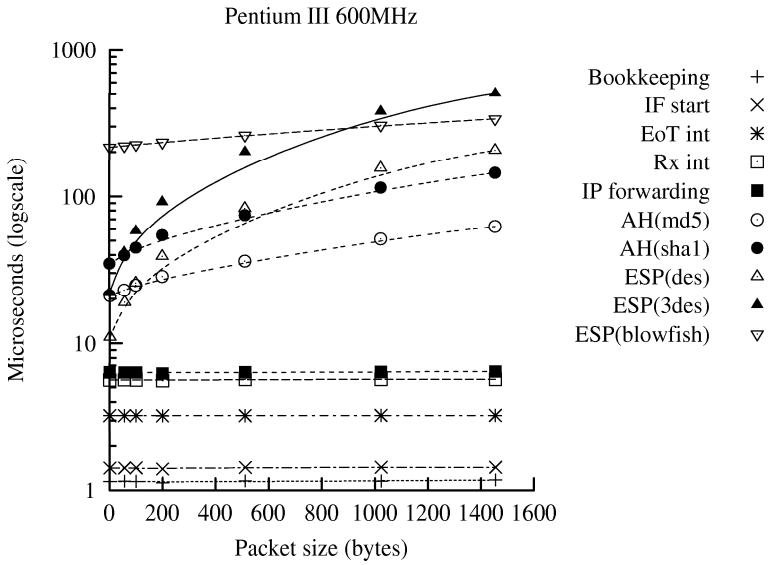
**Figure 3.13**—Outliers related to the CPU's instruction cache. The left chart was drawn after data taken from the **`ipintrq`** probe. The right chart corresponds to the ESP (DES) probe at a security gateway. Referenced outlier points are highlighted



### 3.7.3 Packet's size

Figure 3.14 shows the influence that packet size has over probe measured execution times. On one hand it shows that the execution time of the plain IP router code is insensitive to the packet size. This is most likely due to the way FreeBSD release 4.1.1 manages message buffers—it uses cluster message buffers indistinctly of packet size. On the other hand, as expected, it shows that the execution time of the code implementing authentication and encryption algorithms for IPSEC protocols augment with the size of the packet. The figure shows curves for a pair of authenticating algorithms—md5 and sha1—and for three encrypting algorithms—DES, triple DES and blowfish. Each of these curves reflects the various behaviors of these algorithms. It can be seen that blowfish has the smallest dependence to packet size, while DES and triple DES have the greatest. On the other hand, it can be seen that blowfish has the largest set up time and that triple DES is truly three times costlier than DES.

**Figure 3.14**—Relationship between measured execution times and message size



### 3.7.4 Routing table's size

Another aspect that generally influences the performance of routers is the routing table lookup algorithm. Table 3-I consigns IP forwarding's average execution times for our 600 MHz system, when its routing table had 4, 16, 64, 128, 256, 512 and 1024 entries. Traffic threading the router was randomly distributed across addresses. The times obtained (15.44, 15.98, 16.16 and 16.37 s) do not show significant variations. Consequently, we can say that for the expected routing table's sizes that a software router may face—at the Internet's edge—routing performance is not influenced by routing table's size.

TABLE 3-I

| Routing table size | IP forwarding time [μs] |
|---|---|
| 4 | 15.44 |
| 16 | 15.98 |
| 64 | 16.16 |
| 128 | 16.37 |
| 256 | 16.56 |
| 512 | 16.81 |
| 1024 | 17.41 |

### 3.7.5 Input/output bus's speed

After settling that router's networking software's execution time varies proportionally with CPU's speed, we computed the expected execution times for IP forwarding and encrypting/authenticating of 1500 bytes network messages at selected CPU speeds and compared it with transfer times through I/O buses of, one, 33 MHz operation speed and 32-bit data path, and the other, 66 MHz operation speed and 64-bit data path. The results at Table I clearly show that for the CPUs that will be available on the market in the following years, the bus is potentially a bottleneck to the system.

TABLE 3-II

| CPU speed [MHz] | Forwarding [μs] | Forwarding + 3DES [μs] | IO Bus transfer [μs] | |
|---|---|---|---|---|
| | | | 33 x 32 | 66 x 64 |
| 600 | 15.44 | 580 | 11.4 | 2.9 |
| 1200 | 7.72 | 290 | 11.4 | 2.9 |
| 4800 | 1.93 | 72.5 | 11.4 | 2.9 |

# 3.8 Model's applications

Now that we have proved the validity of the queuing network model, in this section we apply it for capacity planning and as a uniform experimental test-bed. When used as a capacity planning tool, the queuing network model may help to devise system configurations for tuning a system supporting communication quality assurance mechanisms, widely known as QoS mechanisms. Indeed, user QoS levels may be map to parameters such as overall maximum latency, sustainable throughput or packet loss rate. Then, the queuing network model's ability to estimate the complete probability density function of performance parameters may be used for identifying system's operational regions that meet these QoS levels. Subsection 3.8.1 elaborates on this.

When used as a uniform experimental test-bed, the queuing network model eases the performance study of systems that incorporate some modifications or extensions. Generally, all there is to do is to adjust the model's queuing network. Certainly, when adjusting the queuing network care must be taken to remap service times. And although this remapping may not be trivial it is clearly easier than producing correct software for a new real system. Following this rationale, subsection 3.8.2 presents performance evaluation results of two modified software routers: one that incorporates the Mogul and Ramakrishnan's [1997] receiver live-lock avoidance mechanism and another that besides this incorporates a weighted fair queuing scheduler for IP packet flows.

### 3.8.1 Capacity planning

Figure 3.15 shows two charts drawn from data estimated by the queuing network model. In each chart some router performance variable in plot against the average traffic load in packets per second (pps). We are assuming the performance bottleneck is the router and that the traffic load is Poisson. Five curves are drawn in all charts, each one corresponding to the performance of a system under particular conditions. Let us now discuss how to use these charts for capacity planning.

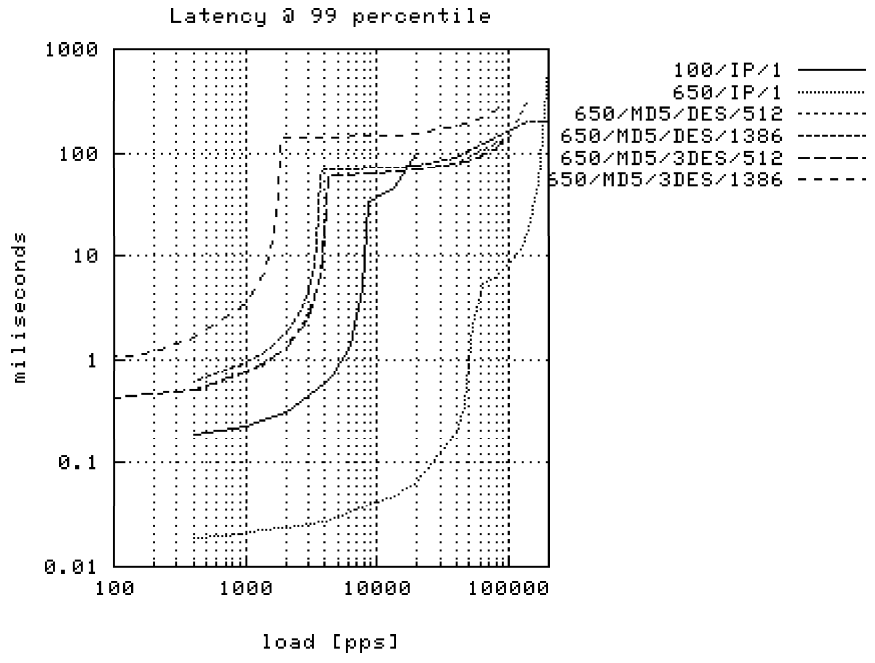
**Forwarding capacity.** Let us define forwarding capacity as the maximum load value at which an IP router can forward packets without losing any. Observe that, as noted in subsection 3.7.3, packet forwarding as implemented in FreeBSD 4.1.1 is insensible to the packet size. Thus, for assessing the forwarding capacity of a system we look at the losses chart and find the load at which the system starts to loose packets. For example, a 100 MHz system has a forwarding capacity of, more or less, 8.5 kpps. This means that it can hardly drive a single Ethernet—which may produce packets at rates exceeding 14 kpps. On the other hand, a 650 MHz system has a forwarding capacity of 55 kpps.

**Encryption capacity.** The above mapped to an IPSEC router results to the encryption capacity. That is, the maximum load value at which an IPSEC router can forward encrypted packets. Differently from packet forwarding, as also noted in subsection 3.7.3, packet encryption performance is influence by the packet size and the encryption algorithm used. In fact, an IPSEC router has a bimodal response with respect to packet size. When packets are smaller than a particular value the router performance is limited by its forwarding capacity. When packets are bigger than this value, the encryption capacity is the performance limit. To numerically show this, consider the following. Let us

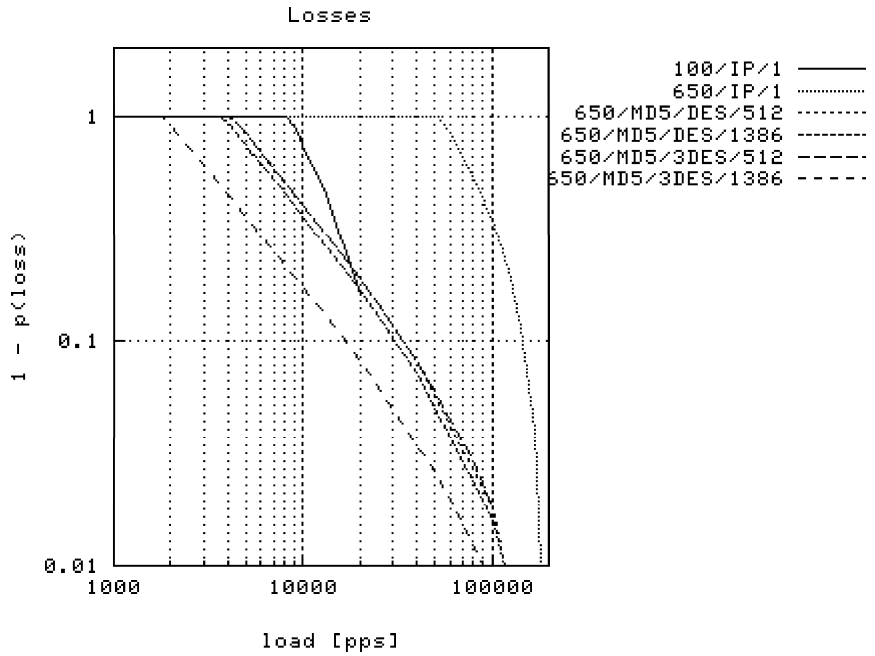
defined $t_f$ as the packet forwarding time ad $t^u_e$ as the per-byte encryption time. Thus, the maximum processing capacity of an IPSEC router expressed in packets per second is equal to $C(pps) = 1/(t_f + t^u_e * L)$, where L is the packet size in bytes. This expression measure in bytes per second maps to:

$$C(Bps) = \frac{1}{t^u_e + \frac{t_f}{L}}$$

**Figure 3.15—**Capacity planning charts




a)




b)

Clearly, when $L \to 0$, $C(Bps) \to L/t_f$ and when $L \to \infty$, $C(Bps) \to 1/t^{\mu}_{e}$

Returning to the encryption capacity example, from the losses chart, a 650 MHz system configured for authenticate and encrypt a communication session—using both AH and ESP protocols, and using DES for encrypting, has an encryption capacity of 4.2 kpps when processing 512 byte packets and 3.6 kpps when processing 1386 byte packets. If it uses 3DES for 1386 bye packets, this system's encryption capacity is around 1.5 kpps.

**Maximum latency.** If someone is interested in knowing what is the minimum clock rate (read: price) that supports packet latencies not higher than certain value with some probability, he must look at the corresponding latency percentile chart. For example, a 100MHz IP router operating at its forwarding capacity of 8.5 kpps has a 99-percentile latency around 10 ms. As another example, a 650 MHz IPSEC router configured to use 3DES, processing 1386 byte packets and operating at its encryption capacity of 1.5 kpps has a 99-percentile latency around 10 ms.

### 3.8.2 Uniform experimental test-bed

The following study's objective is to observe the performance of a personal computer-based software router supporting communication quality assurance mechanisms, or Quality-of-Service (QoS) mechanisms. Here it is shown that the considered software routers cannot sustain system wide QoS behavior solely by adding QoS aware CPU schedulers. Where by this schedulers we do not mean user processes schedulers but networking tasks schedulers; that is, schedulers regulating the marshalling of packets through the networking software. The next chapter presents a solution to this problem.

The study that concerns us here compares the performance of two routers. The one's CPU speed is 1 GHz and the other's is 3 GHz. These routers model's service times were extrapolated applying section 3.7's parameterization rules to section 3.5's measurements for the 600-MHz system. We considered a PCI I/O bus like the one described in section 3.4. The data links were considered to have a 1 Gbps throughput.

For this study the router workload consisted on the superposition of three traffic-flows with characteristics shown in Table 3-III. The offered load in bits per second for each flow is identical. As we are interested mainly in system throughput under overload we have considered Poisson input traffic.

TABLE 3-III

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

It was expected that an ideal QoS aware router would allocate one third of its resources to each flow.

**Basic system.** In order to set a comparison baseline, we firstly studied the performance of the considered routers when they use a basic BSD networking software. Figure 3.16 shows the queuing network model for a basic software router traversed by three packet flows. Figure 3.17 shows performance data computed after the models for global offered loads in the range of [0, 1400 Mbps]. As can be seen, the router with the 1 GHz CPU has a linear increase of the aggregated throughput for offered loads below 225 Mbps. At this point, the CPU utilization is 100% while the bus utilization is around 50% and the system enters into a saturation state. If we further increase the offered load, the throughput decreases until a receiver live-lock condition appears for an offered load of 810 Mbps. During the saturation state most losses occur in the IP input buffer.

For the router with the 3-GHz central processing unit, the bus saturates for an offered load of 500 Mbps. The bus utilization at this point is 100% while the CPU utilization is around 70%. The system behavior for increasing offered loads depends on which priorities are used by the bus arbiter. The results here shown correspond to the case in which reception has priority over transmission. We observe that system throughput decreases with increasing offered loads. This can be explained if we observe that CPU utilization is increasing and most losses occur in the drivers transmit queue. This indicates that the transfer from NIC to CPU increases with increasing offered loads and hence the CPU processes more work. However this work cannot be transferred to the output NIC as the bus is saturated by the NIC to CPU transfers.

If transfers NIC-CPU and CPU-NIC have the same priority the CPU never reaches saturation, and losses can either occur in the NIC's reception queue or in the device driver's output transmission queue.

Consequently, the basic system cannot provide a fair share of the resources when it is in saturation.

**Figure 3.16—**Queuing network model for a BSD based software router with two network interface cards attached to it and three packet flows traversing it

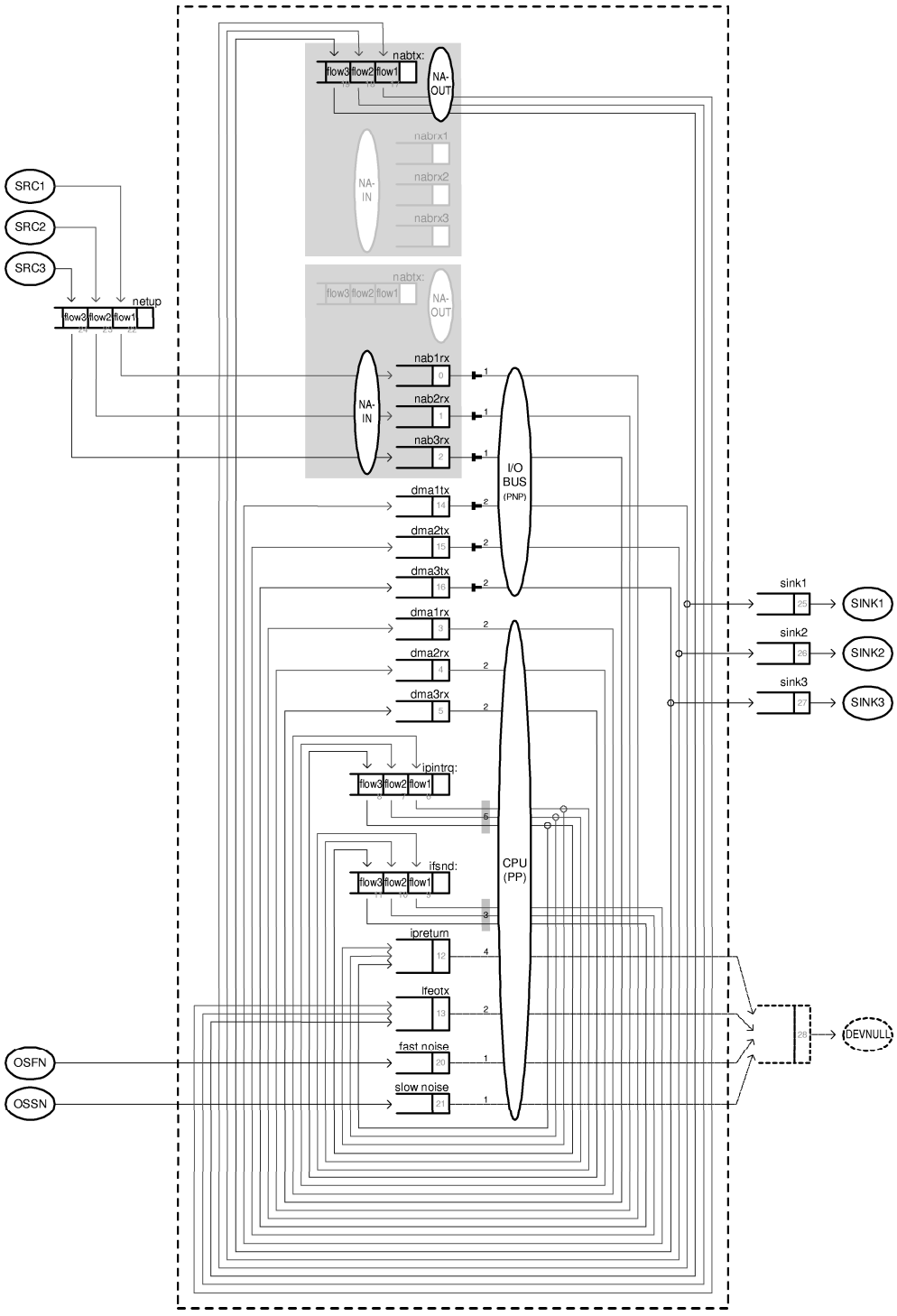
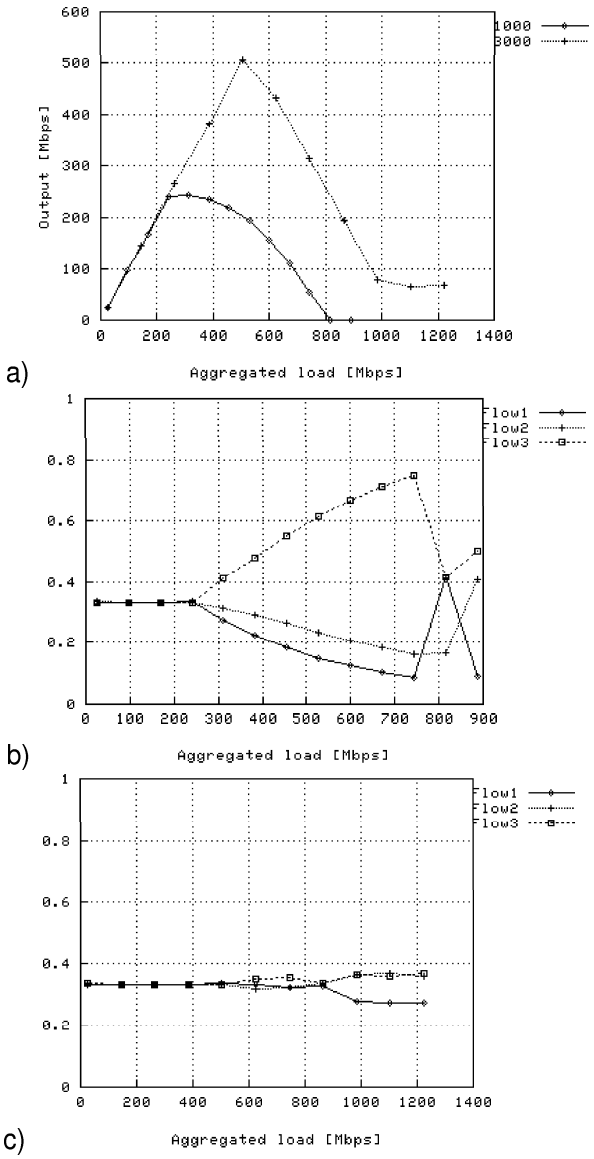


**Figure 3.17—**Basic system's performance analysis: a) system's overall throughput; b) per flow throughput share for system one; c) per flow throughput share for system two



**Mogul and Ramakrishnan [1997] based software router.** We performed similar experiments with a model of a software router that includes the modifications proposed by Mogul and Ramakrishnan. The proposed modification's objective is to eliminate the receiver live-lock pathology. Basically, this is accomplished by turning off the software interrupt mechanism and driving networking processing by a polling scheduler during a router's busy period. As a consequence, networking processing is no longer conducted through a pipeline but is done as a run-to-completion task. From the queuing network modeling point of view, this results in the aggregation of all networking related service queues into a single queue. Mogul and Ramakrishnan showed that the resulting total IP processing time is actually shorter than the sum of the times for each individual phase. They showed that the added polling scheduler's processing costs is compensated by the dropped IP queue manager's processing costs. Moreover, the run-to-completion processing gets implemented with fewer function calls. We took advantaged of these observations and considering that basic networking phases' service times are independent we computed the service time of the aggregated service as the sum of the basic networking phases' service times.

Figure 3.18 shows the queuing network model for a Mogul and Ramakrishnan based software router and Figure 3.19 shows performance data computed after this model when parameterized for the two routers described at this section's introduction. Naturally, offered load was similar to the one used for the basic system. As can be seen and as expected, the performance of the two modified routers do not change with respect to that of the basic system when operating below saturation. Moreover, they reach saturation at more or less the same offered load. But now, the throughput degradation is gone thanks to the receiver live-lock elimination mechanism. However, the system still does not achieve a fair share of router resource.

**Figure 3.18**—Queuing network model for a Mogul and Ramakrishnan [1997] based software router with two network interface cards attached to it and three packet flows traversing it

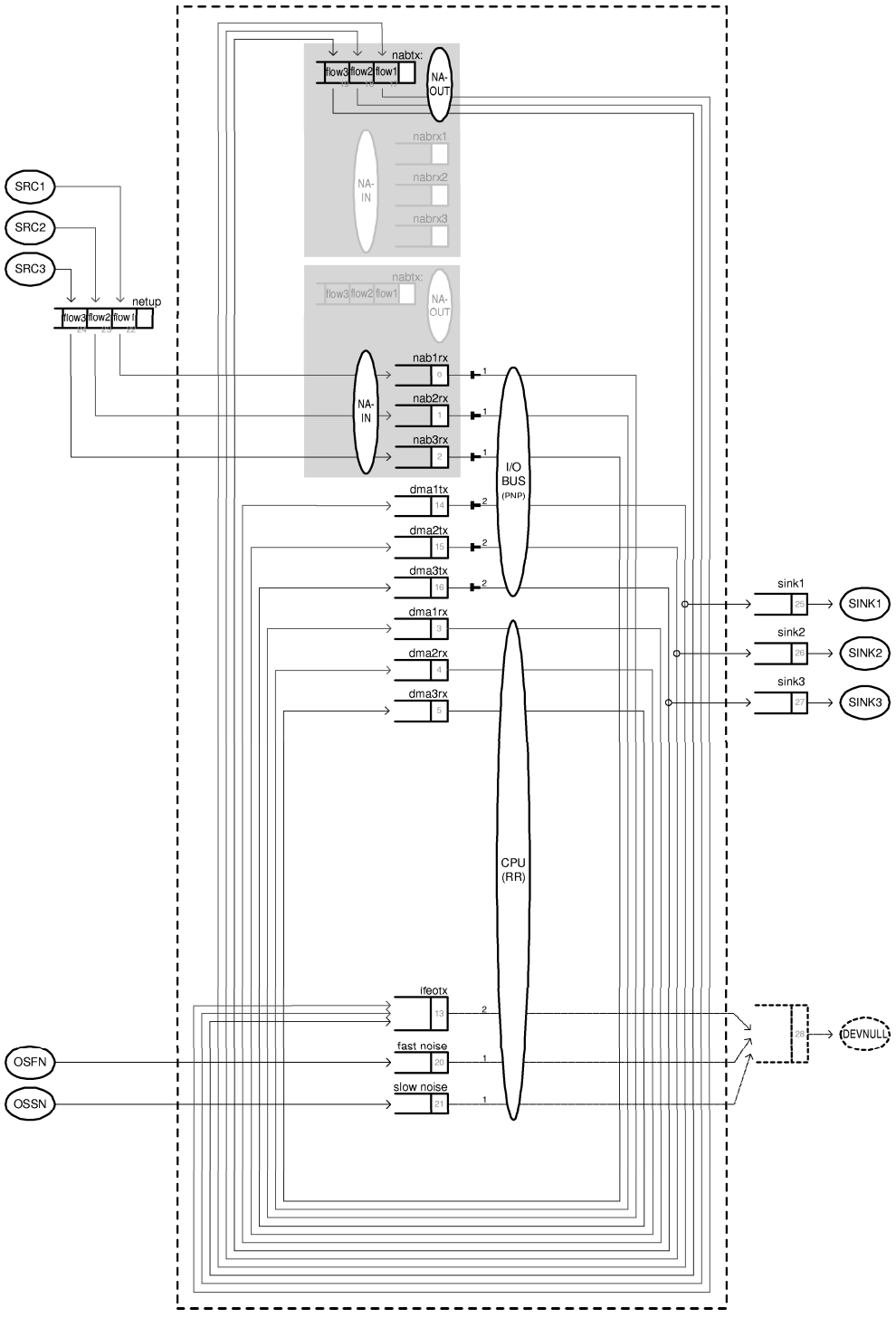
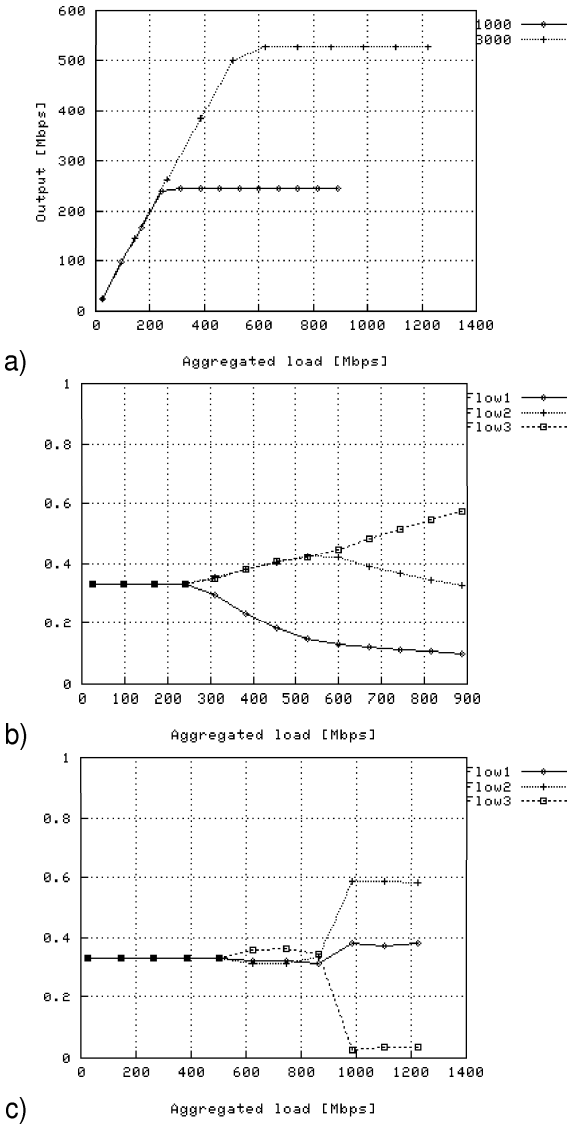


**Figure 3.19—**Mogul and Ramakrishnan [1997] based software router's performance analysis: a) system's overall throughput; b) per flow throughput share for system one; c) per flow throughput share for system two




a)

b)

c)

**Mogul and Ramakrishnan [1997] based software router with a QoS aware CPU scheduler.** When a software router includes the modifications proposed by Mogul and Ramakrishnan, to us, it seams possible to introduce a QoS aware CPU scheduler like a WFQ scheduler. Indeed, once the polling scheduler is in control, it may use whatever policy it implements to select the next packet for processing. This scheme is proposed by Qie et al. [2001], although for and operating system other than UNIX.

Figure 3.20 shows the queuing network model for this kind of software routers and Figure 3.21 shows performance data computed after this model when parameterized for the two routers described at this section's introduction. As can be seen, the considered networking architecture can only support QoS communications when the CPU is the system's bottleneck, as in the case of the router with a CPU operating at 1 GHz. (Figure 3.21.a.) However, when the bus becomes the system's bottleneck the router fair share is not achieve. (Figure 3.21.b.) By the way, the system's overall throughput chart has been omitted in Figure 3.21, as it is identical to the one presented in Figure 3.19.

**Figure 3.20**—Queuing network model for a software router including the receiver live-lock avoidance mechanism and a QoS aware CPU scheduler, similar to the one proposed by Qie et al. [2001]. The router has two network interface cards and three packet flows traverse it

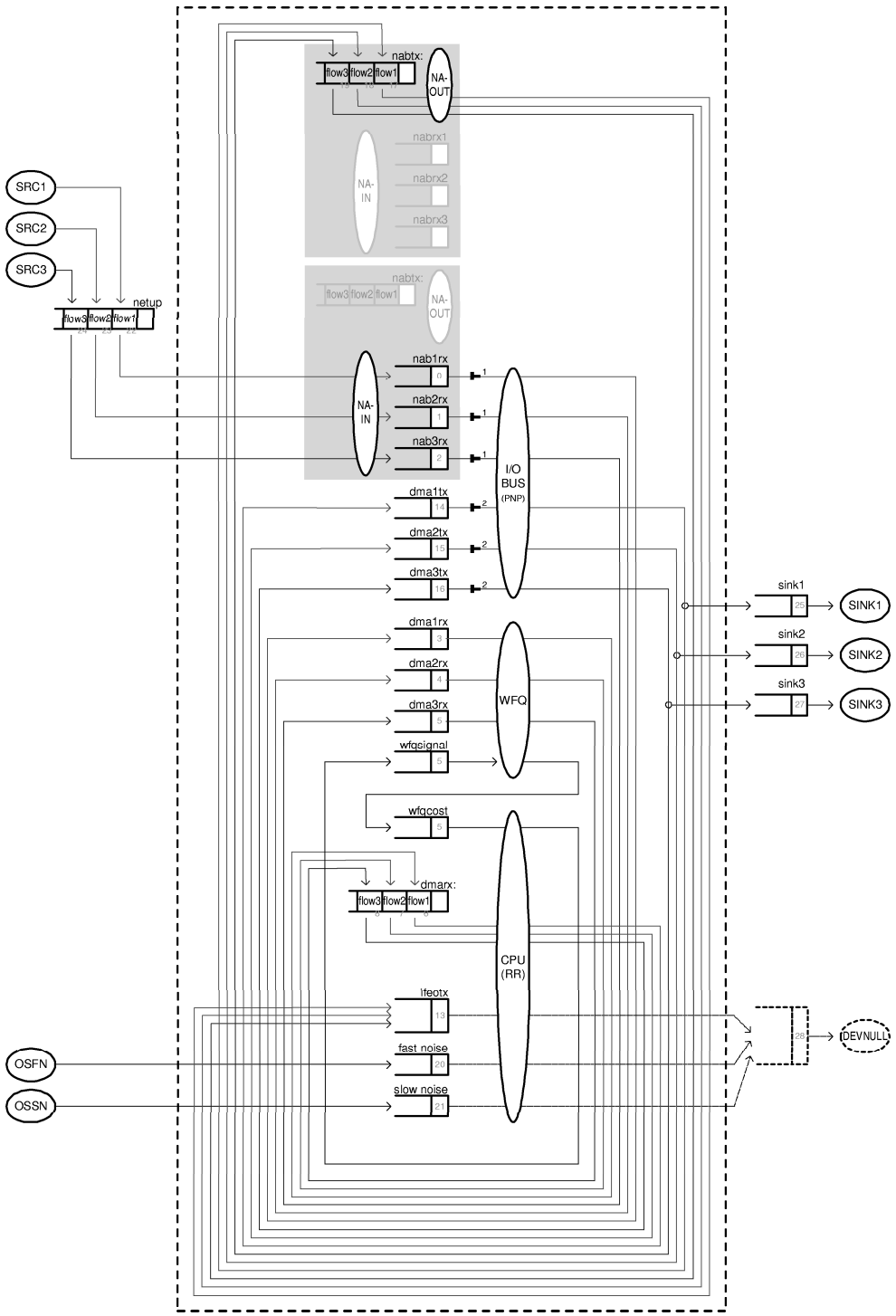
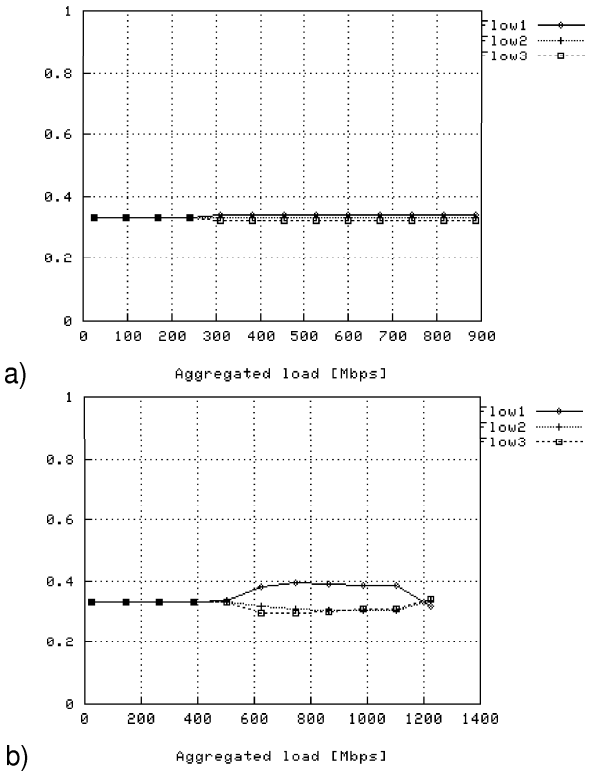


**Figure 3.21—**Qie et al. [2001] based software router's performance analysis: a) per flow throughput share for system one; b) per flow throughput share for system two



## 3.9 Summary

- A queuing network model of a software router gives accurate results
- Single queue models of software routers ignore chief system features
- Characterizing the system was hard not because of the studied system but due to the required process
- Armed with a mature process, characterization becomes straightforward
- The model's service times computed after some system may be used for predicting the performance of other systems, if scaled appropriately
- Service times scale linearly with CPU operation speed but can be considered constant with respect to message and routing table sizes
- Service times' offset varies with respect to the network interface card's and device driver's technology and the cache memory performance
- In the advent of CPU's working above 1 GHz, the I/O bus is the bottleneck