TigerSwitch: A Case Study in Embedded Computing System Design

Wayne Wolf, Andrew Wolfe, Steve Chinatti, Ravi Koshy, Gary Slater, and Spencer Sun
Departments of Electrical Engineering and Computer Science
Princeton University

Abstract

This paper describes and analyzes the design of TigerSwitch, a PC-based private branch exchange (PBX) designed at Princeton University. Building TigerSwitch required creating custom hardware and software designed to fit onto a standard IBM PC-compatible platform. Our design experience provides several lessons which we believe extend to other embedded design domains: the system architecture required to meet performance goals is often not isomorphic to the structure of the specification; system-level performance analysis is an essential part of system architecture design; architectural decisions must be made on the basis of estimates before complete implementations of the components are available; and most allocations of functions to software or custom hardware are obvious, while a few are very difficult.

1 Introduction

We designed TigerSwitch as a study in embedded computing system design. TigerSwitch is a private branch exchange (PBX), or a customer-owned telephone switching system. This paper describes the major features of the design itself and the important decisions we made while designing the system.

A PBX is a good example of embedded computing system design for two reasons. First, it implements a wide variety of functions: the actual switching of data between telephones is relatively simple; call routing and billing functions require more complex algorithms unlikely to be implemented in custom hardware; switch maintenance and initialization routines require a user interface. Second, some of the PBX's functions must be performed to **hard real-time deadlines**—for example, the customer will hear problems in the phone call if voice samples are not supplied at the required 8 kHz rate. The combination of complex function with strict performance goals is typical of many embedded applications and presents a significant challenge to embedded system tools.

Designing the architecture of an embedded computing system can be broken into four steps[7]: **partitioning** the system's function into processes; **allocating** those processes to processing elements; **scheduling** the processes in time; and **mapping** the generic processing elements into physically realizable component types. During each of these steps, design decisions for TigerSwitch were guided on the one hand by the performance requirements of telephone systems and on the other hand by the desire to minimize design time and implementation cost.

The next section surveys telephone switching systems and describes the TigerSwitch architecture. The next three sections describe three important problems we faced during the design of TigerSwitch: Section 3 describes the requirements which dictated the process structure of TigerSwitch; Section 4 describes performance analysis for TigerSwitch and how it affected function allocation. Finally, Section 5 provides some general comments on these design problems.

2 Telephone switching system design

2.1 Switching system principles

TigerSwitch implements only basic voice service, known as **plain old telephone service (POTS)**. We chose not to implement advanced ISDN features like data transmission for expediency. Telephone-quality audio is sampled at an 8 kHz rate, corresponding to a sampling interval of 125 μs, using an 8-bit sample which has been scaled using the **μ-law.** to correct for the dynamic range of the human ear.

A **switching system** makes connections required by callers as well as keeping billing records, providing switch maintenance services, providing advanced customer features like call forwarding, and other services. A **private branch exchange (PBX)** is a switching system operated by a customer rather than the service provider; PBX's are usually constructed to lower standards of reliability than are the switching systems used in the network. The **subscriber loop** is the circuitry between the switch and the customer's

0-8186-6315-4/94 $04.00

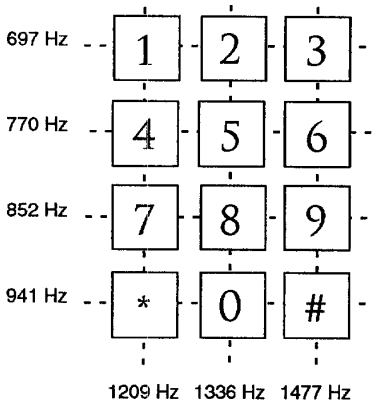

FIGURE 1. The DTMF signaling scheme.

telephone. A **line card** is the interface between the switch and the subscriber loop. A telephone goes **off-hook** when the receiver is lifted and goes **on-hook** when the phone is hung up. A **CODEC** is a analog-to-digital/digital-to-analog converter pair. **Dual-tone multi-frequency (DTMF)** signaling[1] is used to send digits to the switch from the subscriber's telephone. As shown in Figure 1, DTMF signaling assigns a distinct sinusoidal frequency to each row and column on the touch pad; signals at the proper row and column frequencies identify a digit, in the absence of significant energy components at other frequencies. The **switching fabric** is the component which routes telephone call data between subscribers.

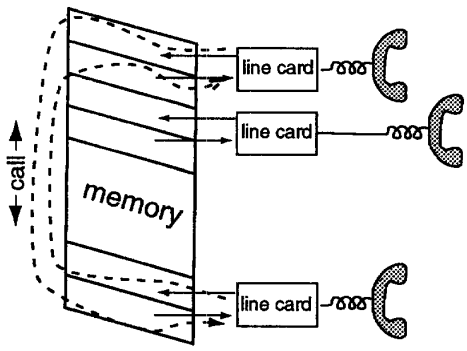

FIGURE 2. Time-division multiplexing for digital call routing.

1. Touch-tone is AT&T's trademarked name for DTMF.

A simple and common architecture for digital PBXs[1,2] is to implement the switching fabric using time-division multiplexing through a shared memory as shown in Figure 2. Each subscriber line has one writable data port connected to the telephone speaker and one readable port connected to the microphone. Each subscriber line is associated with two memory locations, either by directly mapping the line cards into memory or by transferring the CODEC data to the memory location. A call requires transferring data between two phones once every sampling period: the microphone data from one telephone is copied to the other telephone's receiver and *visa versa*. The pattern of data transfers is determined by the connections required customers' requested connections.

A line is typically specified as a state machine which describes what actions are to be taken at each in the calling process. The line-as-state machine model breaks down when features like conference calling are introduced, since a call no longer corresponds to a connection between two lines. The switch also cannot be modeled as a collection of disjoint call state machines, since operations like billing and call routing must be handled globally. However, modeling the switch activity as a collection of processes which represent calls does give us a good handle on the major operations which must be performed during call processing.

A line goes through several states in the normal processing of a call:

1. The caller's line goes off-hook, moving the line from the on-hook state to a state at which the switch is waiting for a number to be dialed.
2. The switch allocates a DTMF decoder to the line. The caller's DTMF digits are decoded.
3. The call routing system determines a destination for the call.
4. The DTMF detector is deallocated and a ring signal is sent to the destination, placing the line in the ringing state.
5. When the callee's phone goes off-hook, both lines are put in the connected state. The billing system starts the call timer.
6. When either phone goes on-hook, that line is put in the on-hook state and the billing system stops timing the call.
7. When the other line goes on-hook, it is put in the on-hook state.

A switching system has several hard real-time deadlines of varying durations:

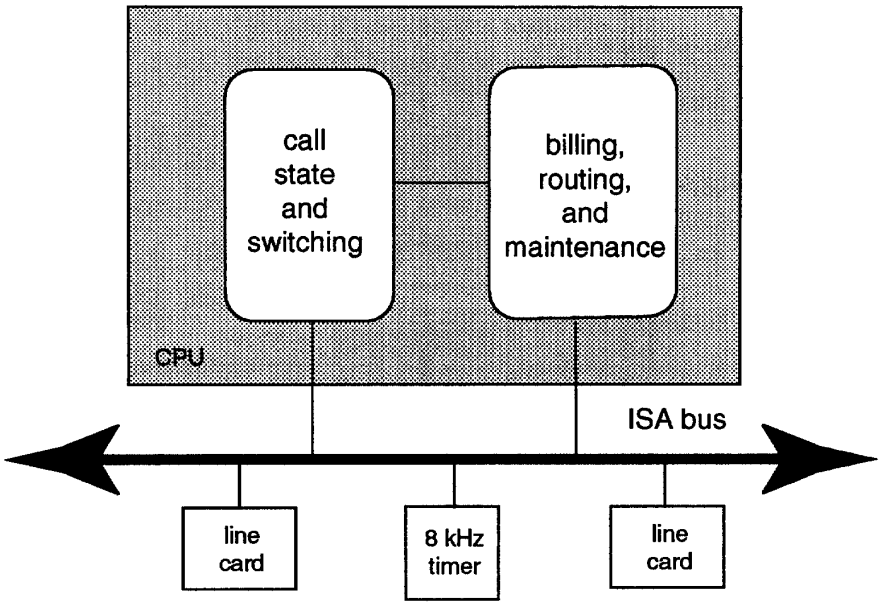

FIGURE 3. The process architecture of TigerSwitch.

- voice samples must be transferred by the switching fabric every 125 μs;
- a DTMF signal must be held for at least 0.1 sec to be valid.
- the start and termination of a call must be measured by the billing system to an accuracy of 0.5 sec;
- the average time from a phone going off-hook to generating a dial tone should be no more than 3 sec.

(A line is not given a dial tone until a DTMF detection unit is allocated to the line. Because some switching systems supply only a small number of DTMF detectors, the dial tone acquisition time is probabilistic. The performance requirement is usually specified as an average dial tone acquisition time; customers become unhappy if they pick up their telephone and do not promptly hear a dial tone.)

2.2 The TigerSwitch implementation

TigerSwitch is implemented on an IBM PC-compatible computer. A PC consists of a CPU, a system bus commonly called the ISA bus though different models have slightly different busses, some standard peripherals and timers, and some low-level software known as BIOS (Basic I/O System). We chose a PC as the platform for TigerSwitch because it combined low cost with a powerful, well-known software development environment. The ability to quickly write programs to test system function was especially important for the development of custom hardware. Panasonic sells a small office PBX which is implemented on a PC platform, demonstrating that this design choice is a practical one for the low end of the PBX market. Many other laboratory and industrial control systems are implemented on PCs as well. Our design goal for the project was to **minimize design time**. TigerSwitch was designed in one semester as a class project; because students' commitments change radically at the end of each semester, we had to be sure that our system could be complete or nearly so in one semester's time. Many commercial embedded systems are designed to strict time-to-market constraints which resemble our deadline; we have also tried to describe how the design would change under a different set of constraints.

The combined hardware/software architecture of TigerSwitch is shown in Figure 1. Each line card is a device on the PC's ISA bus; we also had to design a timer to generate interrupts at an 8 kHz rate since the PC's timers cannot be programmed to run that fast. The software is partitioned into two processes. One is executed every 125 μs; it keeps track of the basic call state and executes the data moves required to implement the switching functions. The other is executed as a background task; it keeps track of the call state pertinent to call routing (such as how many digits have been dialed by a line), maintains billing information, and provides simple maintenance functions like displaying the current set of calls.

FIGURE 4. The TigerSwitch line card and its breadboard prototype.

A first difficulty encountered in the design process was finding a succinct specification of a PBX's function. In fact, we found no complete functional description of a basic PBX, though we found some papers such as Zave's[9] which gave some helpful descriptions of switching functions. We performed several **walkthrough simulations** to determine both whether our specification was sufficiently complete and consistent and to help us make some simple architectural decisions. In a walk-through, we played the roles of various hardware and software components: the line card, the switching fabric, *etc.* We simulated data and control communication by speaking, such as "I'm going off-hook now." Our first walk-through simulated a simple call; later walkthroughs considered billing and non-standard conditions like failing to hang up. Walkthroughs made certain deficiencies immediately apparent and clarified the work required to keep track of the call state. While we felt comfortable with this informal specification, we never had a formal specification of the switching system's function.

The TigerSwitch software was implemented in C++ using the Microsoft C++ development environment. The switching system is implemented in about 1,000 lines of code. As mentioned above, we had to build our own 8 kHz timer to provide interrupts at the sampling rate; the billing/routing process was executed periodically from interrupts generated by a standard PC/DOS timer.

The lowest-level I/O routines for standard devices such as the keyboard and the disks are provided by BIOS, a collection of ROM-based code in the PC. BIOS does not support non-blocking I/O—it does not return control to the caller until the I/O operation is complete. The lack of non-blocking I/O enforced serious limitations on the real-time architecture. Most importantly, we could not access the disk at any time the switching fabric is activated. As a result, TigerSwitch cannot write billing data to disk while the system is in operation. Rather than build a non-blocking non-volatile memory system, we chose to buffer account data in memory and write it to disk during maintenance operations. However, such a design would not be acceptable for commercial-scale switching systems. The possibility of major design restrictions, which may not be discovered until sufficient experience with the platform has been obtained during implementation, is an inherent risk in using existing platforms. Any system design methodology which takes advantage of large components must either ensure that the components have been thoroughly characterized or that the time required for characterization is built into the design schedule.

The line card and its breadboard prototype are shown in Figure 4. We implemented one line per line card largely to make the printed circuit board less dense and easier to route; a more commercial design would have implemented two or more lines on a single line card. The number of line cards per switch is limited by the capacitive loading restrictions of the PC's ISA bus, which can handle 8-12 cards. The line card circuits and the ISA bus interface were debugged on the breadboard. A two-layer printed circuit board was

then designed and fabricated. The line card uses the 16-bit AT bus standard to send both sample data and line status in a single bus transaction. Each card is mapped into the I/O address space; its address can be set via DIP switches.

3 Process architecture

Our first lesson in the design of TigerSwitch was that the functional decomposition which is natural in specifying the system does not necessarily correspond to the hardware-software architecture which provides the required level of performance. In the course of the system design, we ended up with a process structure which was dramatically different from the functional decomposition natural to switching system specifications.

We started the project with the intent of using the simplest, cleanest possible architecture for the system. The most direct architecture would implement each call as a separate process. We identified severe performance problems with the call-as-process decomposition during the first walk-through. The worst-case performance in an *n*-line PBX is easy to identify: each call is a process; all lines go off-hook simultaneously, causing *n* processes to either be created (assuming each call starts with an off-hook) or to be activated (assuming that a call is rooted at a line). Activating all these processes requires substantial initial overhead for creating/activating the processes, with continual overhead for context switching between all the simultaneously active processes. When a line goes off-hook, the switch must allocate a DTMF detector and initiate the dial tone. Telephone systems specify a maximum amount of time that a customer should wait to receive a dial tone. Even though this specification is not safety-critical, failing to meet it may be sufficient reason for the customer to refuse delivery of the switch. Given that both process creation and process context switching often take milliseconds, it is unlikely that any platform would provide sufficient performance to let us implement each call as a separate process.

We quickly settled on the two process architecture of Figure 3 to reduce context-switching overhead. We split the call state between the two processes: the switching fabric process keeps the state of interest for switching, such as the call's destination; the billing/routing process keeps state of primary importance to these operations. However, each process keeps the relevant state for all lines in the system. For example, the switching fabric process keeps a table which shows, for each line, whether the line is active and any other line to which it is connected for a call.

4 Performance analysis and allocation

Embedded computing systems are built from complex components. The selection of the proper components suited to the performance and cost constraints is the major task of system implementation. We prefer to view the design of the hardware and software architectures of such a system to be an function-to-component allocation process rather than a hardware-software partitioning processes. While we may partition a function into a sequence of processes, we allocate those processes to particular components to determine their implementation—a function allocated to a CPU is implemented as software executing on that CPU while another function may be allocated to a custom hardware unit. .

When designing TigerSwitch, we had to choose the PC platform, a combination of CPU and bus, and allocate functions to that hardware engine to sustain the required performance at the lowest cost. Because the components are complex, characterizing them is difficult to do in general—a single performance number does not usually reflect the component's performance on any particular application. We used a combination of estimates and direct measurements to analyze the performance of critical sections of TigerSwitch.

Walkthroughs identified two potential bottlenecks in the switch: the switching fabric and DTMF detection. We found the analysis of switching fabric performance to be straightforward but DTMF detection analysis to be both critical and challenging.

```
for (i=0; i<N; i++) {
    /* one half of the transfer */
    temp = get_card(i);
    put_card(route[i],temp);
    /* transfer the other way */
    temp = get_card(route[i]);
    put_card(i,temp);
}
```

FIGURE 5. Structure of table-driven time-division multiplexed switching fabric code.

Figure 5 shows the structure of a table-driven implementation of the time-division multiplexed switching fabric. The array route[i] gives the index of the line j to which line i is connected. Unconnected lines can be routed to the bit bucket. (We considered but rejected a plan to generate the switching fabric code on the fly, using direct transfers to avoid the indirection through the route[] array.) Bus bandwidth rather than instruction execution time is likely to be the bottleneck of switching fabric performance.

More expensive PBX architectures, like System 75[1] and the Rolm CBX[4], switch calls over a dedicated bus. Because the

bus type	throughput (MBytes/sec)
PC AT 16-bit 0 wait state 8 MHz	8
PC AT 16-bit 0 wait state 10 MHz	10
PC AT 16-bit 0 wait state 12 MHz	12
EISA 32 bit	33

FIGURE 3. Performance levels of IBM PC busses (from Eggebrecht[3]).

bus is used for a single purpose, bus utilization is easy to compute—each call receives its own time slot, and the bus bandwidth directly determines the number of calls which can be handled at any one time. Our PC-based architecture, in contrast, uses the bus for the fetching of program instructions and data as well as switching, making performance analysis a little more difficult. We used a back-of-the-envelope analysis to first determine whether switching would require enough bandwidth to potentially conflict with the CPU's bus operations. If this simplified analysis indicated that the bus might be overutilized, a much more detailed analysis would be required to determine the proper bus bandwidth.

Assuming 10 line cards on a switch gives a maximum of five calls active at any one time. Each call requires four 16-bit transfers every 125 μs (each card-to-card movement requires two bus transactions). Therefore, switching requires a bus utilization of .64 MBytes/sec. Figure 5 gives the throughput of several different PC busses. The ISA bus on a 386 system runs at 8 MHz, so switching will use only 4% of the available bus bandwidth, leaving the rest for instruction and data transfer. Because bus utilization is so low and the switching fabric operation can easily be well-characterized, a more detailed analysis of bus utilization was not required.

Once we partitioned the functional specification into the processes shown in Figure 3, most of the allocation decisions were obvious. Basic line card functions would clearly be implemented on the custom line cards. Similarly, billing and routing have sufficiently long deadlines that they can easily be implemented as software processes executing on the central CPU. The performance analysis of Section 4 showed that the switching fabric could easily be implemented by a small process on the main CPU which passed data between the line cards.

A DTMF digit must be held for at least 0.1 sec to be a valid digit. DTMF detection may be implemented in analog circuits as a filter bank; it may be implemented on the samples using the fast Fourier transform or more efficiently by Goetzel's algorithm[6], which implements a digital filter bank. There are three locations in the switch at which DTMF digits can be detected:

- a DTMF detection filter bank on each line card;
- a process running on the main CPU;
- a process running on a co-processor attached to the ISA bus.

Process allocation and component selection are intimately related in this case—the feasibility of implementing DTMF detection on the main CPU depends on the speed of the CPU selected.

DTMF detection implemented on the main CPU competes with the switching fabric for CPU cycles. The detection process requires a substantial amount of computation because of the 0.1 sec digit duration specification. While the switching fabric deadline is tighter than the deadline for detecting DTMF digits, if the CPU is not fast enough then the switching fabric will keep DTMF detection from completing on time.

platform	digit detection time (sec)
i386DX, 20 MHz	.028
Pentium, 60 MHz	.0030

FIGURE 4. DTMF detection times for candidate platforms.

To determine whether DTMF detection could be implemented on the main CPU, we implemented Goetzel's algorithm in a C program and ran it on two candidate platforms: an i386DX running at 20 MHz and a Pentium Processor[1] running at 60 MHz. We did not test the algorithm on a 486 because we did not have one readily available. The results are summarized in Figure 6, which shows the total CPU time required to detect a digit in a 0.1 sec interval. The algorithm ran sufficiently fast on the Pentium processor that DTMF detection would not be a significant load on the processor. On the 386, however, detecting a digit required about 1/4 of the length of the detection interval itself.

1. Pentium Processor is a trademark of Intel.

Rate monotonic analysis[5] can be used to conservatively estimate CPU capacity. RMA gives an upper bound of CPU utilization of 69%; if the CPU used 28% of its capacity detecting digits, switching, billing, and routing would be able to use only 41% of the CPU capacity under RMA assumptions. Even if a more efficient schedule could be found (loosening the requirements of rate-monotonic analysis can allow schedules with higher utilizations), we believed that this margin was too tight to accommodate changes in functionality or implementation which were likely to occur during development. Equally important, designing software to meet tight utilization requirements might require more time than we had allotted to complete the project. As a result, we ruled out allocating DTMF detection to a 386 platform.

We could have chosen a Pentium PC for the system platform, but at the time the purchase decision was made, Pentium PCs were both significantly more expensive than either 386 or 486 platforms and also required long lead times. We considered building a co-processor board which would contain a TI TMS32010 DSP, which can execute DTMF detection for a single call. However, building an additional piece of custom hardware would greatly increase the design time required and add substantial design risk. We chose to add an analog DTMF detection chip to every line card; the line card presents the detected DTMF digit as status bits continuously available to the routing process. A switch designed to support a large number of line cards would probably not include DTMF detection on every line card, since the cost of line cards is a major component of the cost of large switching systems. However, given our emphasis on minimizing design time, we found this solution to give us both low design time and low cost.

5 Conclusions

We were able to complete most, but not all of the software and hardware design in the one semester allocated to the project. Most of the software was completed on time, but al full test was not possible until we had at least two functioning line cards; a test of multiple, simultaneous calls could not be made until we had at least four line cards. The line card PCB was shipped for fabrication in December, 1993. We spent most of the subsequent semester testing and debugging the PCB. One bug with the DTMF circuit still remained even after this extensive analysis. Even though we chose the PC as a platform because it allowed us to develop a great deal of code without waiting for prototype hardware to be built, our ability to meet our schedule was still compromised by a lack of line cards. A prototyping system which would allow us to run large software applications on simulations of as-yet-unconstructed devices would have let us more completely separate the software and hardware design flows. The software and hardware designs for TigerSwitch are available via anonymous FTP from ee-ftp.princeton.edu, with the sources in the pub/Embedded/Examples/TigerSwitch directory.

We close with a few general observations on TigerSwitch and hardware-software co-design:

- We take a broad definition of a process—a single task which may be implemented in either software or custom hardware. Most of the critical design decisions were made at the process level. Once we partitioned the system's functions into tasks and allocated those tasks to components, implementing the software and hardware components themselves was straightforward. The architectural analysis which led us to our partitioning and allocation, however, required a great deal of effort.
- Performance analysis drives the design of systems which have hard real-time deadlines. Processes on the critical path require the most detailed performance analysis. We had to implement the DTMF function in C code and measure its performance on candidate CPUs to determine that process's performance to sufficient accuracy. A less-critical function like switching required only a back-of-the-envelope analysis. This experience is analogous to hardware design: a few critical functions, like the ALU carry chain, may be analyzed using timing or circuit simulation, while other functions may be checked using much simpler timing models.
- Specifications are very important to the design of complex systems. Good specifications are often hard to get. While the structure of the specification may be a long way from the system's internal architecture, functional and performance bugs are very likely to be created when designers do not have a clear description of what they are trying to build.

Acknowledgments

The construction of TigerSwitch was funded in part by a grant from the AT&T Foundation and by an independent project funds grant from Princeton University.

References

[1] L. A. Baxter, P. R. Berkowitz, C. A. Buzzard, J. J. Horenkamp, and F. E. Wyatt, "System 75: communications and control architecture," *AT&T Technical Journal*, 64(1), January, 1985, pp. 153-173.

[2] John Bellamy, *Digital Telephony*, second edition, John Wiley and Sons, 1991.

[3] Lewis C. Eggebrecht, *Interfacing to the IBM Personal Computer*, second edition, Sams, 1990.

[4] James M. Kasson, "The Rolm Computerized Branch Exchange: an advanced digital PBX," *IEEE Computer*, June, 1979, pp. 24-31.

[5] C. L. Liu and James W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, 20(1), January, 1973, pp. 46-61.

[6] Amy Mar, ed., *Digital Signal Processing Applications Using the ADSP-2101 Family*, Volume 1, Prentice Hall, 1992.

[7] Wayne Wolf, "Hardware-software co-design of embedded systems," *Proceedings of the IEEE*, July, 1994.

[8] Nam S. Woo, Alfred E. Dunlop, and Wayne Wolf, "Codesign from cospecification," *IEEE Computer*, January, 1994, pp. 42-47.

[9] Pamela Zave, "A compositional approach to multiparadigm programming," *IEEE Software*, September, 1989, pp. 15-25.