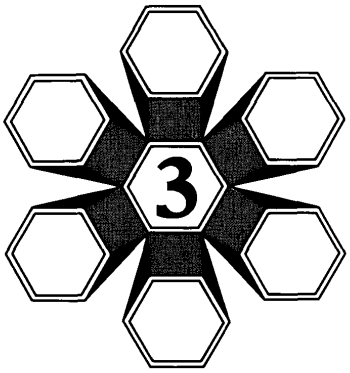
Homayoun

Reference 33



Pipeline Design Techniques

We learn in Chapter 2 that memory is a major bottleneck in high-speed computers, and that the bottleneck can be relieved somewhat by taking advantage of the characteristics of typical programs. The objective has been to store the most-frequently referenced items in fast memory and less-frequently referenced items in slower memory. It is not necessary to make all memory equally fast; we need use only as much fast memory as necessary to hold the active regions of a program and data.

This chapter concerns a different approach to relieving bottlenecks. The idea is to use parallelism at the point of the bottleneck to improve performance globally. If the design techniques are successful, then the extra hardware devoted to performance enhancement is present in only a small portion of a computer system, yet its effect is to increase performance as if the full computer system were replicated.

To contrast the approaches of the last chapter and the present one, in one case the speed differential is due to faster hardware, whereas in the second

case the speed increase is obtained by replicating slower hardware. In both cases, clever architecture is required to create efficient computer systems in which enhancements of a relatively small cost have a global impact on performance.

Pipeline computer techniques described in this chapter are by far the most popular means for enhancing performance through parallel hardware. The basic ideas from which pipeline techniques developed are apparent in von Neumann's proposal to build the first stored-program computer. In Burks *et al.* [1946], a discussion on input/output techniques describes a buffer arrangement that would permit computation to be overlapped with input/output operations and thereby provide a crude form of pipeline processing that is used widely in today's machines.

Although von Neumann did not build the input/output capability into his first machine, the basic ideas for pipelined computer design evolved rapidly after the first appearance of magnetic-core memory as the primary storage medium for main memory. This storage was roughly a factor of 10 or more slower per cycle than the transistor technology used in high-speed registers and control logic. Designers quickly conceived of a variety of techniques to initiate one or more concurrent accesses to memory while executing instructions in the central processor. This body of techniques eventually evolved and is exemplified in the pipeline-processing structures described in this chapter.

In the 1960s, when hardware costs were relatively high, pipelined computers were the supercomputers. IBM's STRETCH and CDC's 6600 were two such designs from the early 1960s that made extensive use of pipelining, and these designs strongly influenced the structure of supercomputers that followed. By the 1980s, hardware costs had diminished to the extent that pipeline techniques could be implemented across the entire range of performance, and indeed, even the Intel 8086 microprocessor that costs just a few dollars uses pipeline accesses to memory while performing on-chip computation.

Our approach is to develop a basic understanding of the principles of pipeline design in the next section. In subsequent sections we observe where it can be used and how to design effective pipelines.

3.1 Principles of Pipeline Design

The basic idea behind pipeline design is quite natural; it is not specific to computer technology. In fact the name *pipeline* stems from the analogy with petroleum pipelines in which a sequence of hydrocarbon products is pumped through a pipeline. The last product might well be entering the pipeline before the first product has been removed from the terminus. In the remainder of this section we treat pipeline design first in abstract terms, and then follow with concrete examples.

The key contribution of pipelining is that it provides a way to start a new task before an old one has been completed. Hence the completion rate is not a function of the total processing time, but rather of how soon a new process can be introduced.

Consider Fig. 3.1, which shows a sequential process being done step-by-step over a period of time. The total time required to complete this process is N units, assuming that each step takes one time unit. In the figure, each box denotes the execution of a single step, and the label in the box gives the number of the step being executed.

To perform this same process using pipeline techniques, consider Fig. 3.2, which shows a continuous stream of jobs going through the N sequential steps of the process. In this case each horizontal row of boxes represents the time history of one job. Each vertical column of boxes represents the activity at a specific time. Note that up to N different jobs may be active at any time in this example, assuming that we have N independent stations to perform the sequence of steps in the process.

The pipeline timing of Fig. 3.2 is characteristic of assembly lines and maintenance depots as well as oil pipelines. The total time to perform one process does not change between Fig. 3.1 and Fig. 3.2, and it may actually be longer in Fig. 3.2 if the pipeline structure forces some processing overhead in moving from station to station. But the completion rate of tasks in Fig. 3.2 is one per cycle as opposed to one task every N cycles in Fig. 3.1.

Figure 3.3(a) shows a box that represents a server that can perform any of the N processing steps in a single unit of time. If the job stream is processed by this one server, then the rate of completion is one job every N steps, and the time behavior of the job stream is as described in Fig. 3.1.

Compare Fig. 3.3(a) with Fig. 3.3(b), which shows N servers concatenated in a sequence. A task flows through the collection of servers by visiting Server 1, then Server 2, and so on, and finally emerging from Server N after N steps. The time behavior of this system is described by Fig. 3.2. Figure 3.3(b) is an ideal model of a constant-speed assembly line, such as an automobile assembly plant.

Now let's relate the general ideas presented in Figs. 3.1–3.3 to computer design. Where can we find an N-step task that can conveniently be broken up, as shown in Fig. 3.2? Consider the steps required to execute a single in-

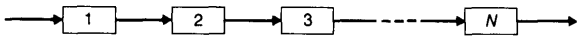


Fig. 3.1 An N-step sequential process.

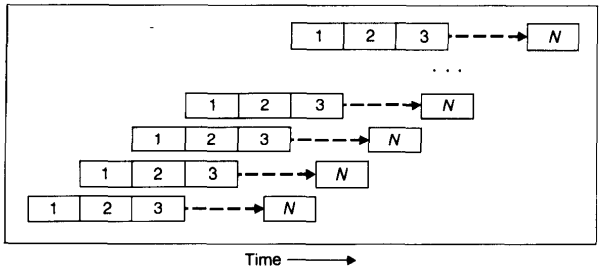


Fig. 3.2 Pipelined execution of an N-step process.

struction. This sequence has traditionally been implemented using pipeline design. A typical instruction-execution sequence might be:

1. *Instruction fetch*: obtain a copy of the instruction from memory.
2. *Instruction decode*: examine the instruction and prepare to initialize the control signals required to execute the instruction in subsequent steps.
3. *Address generation*: compute the effective address of the operands by performing indexing or indirection as specified by the instruction.
4. *Operand fetch*: for READ operations, obtain the operand from central memory.

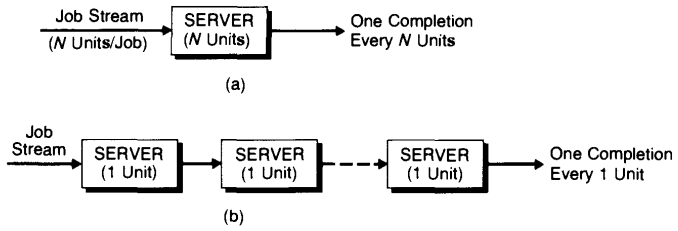


Fig. 3.3 Two ways to execute N-unit jobs in a stream:
(a) Sequential execution with a 1-unit server; and
(b) Pipelined execution with 1-unit servers.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.