G. S. Rao
P. L. Rosenfeld

Integration of Machine Organization and Control Program Design—Review and Direction

This paper discusses the relationship between machine organization and control program design in high-end commercial computer systems. The criterion is cost/performance, subject to achieving an acceptable performance level. A brief discussion of the environment expected for the design and operation of high-end commercial computer systems is outlined, followed by a discussion of machine organization techniques which are classified and reviewed to permit a qualitative evaluation of the degree to which control program intent is exploited in machine organization. The thesis is developed next, using a hierarchical model which illustrates the contention that architecture has acted as a barrier to communication between the control program and machine organization. Examples of techniques that exploit knowledge of the intent of the control program and comments on the methodology that might be used to investigate such techniques follow. Directions for further research are then proposed.

1. Introduction

The purpose of this paper is to discuss the relationship between machine organization and control program design in high-end commercial computer systems. Our criterion is cost/performance, and in our discussion we adopt the point of view that a computer system can be modeled as a hierarchical series of abstractions. We concentrate on the control program because it is the source of a large and growing proportion of the instructions executed on commercial high-end systems.

Our thesis is that the intent of the control program should be exploited at the machine organization level. Both resource requirements and dependencies between activities can be inferred from "intent," thereby allowing the efficient allocation of resources. Thus, recognizing the purpose of control program activity at the machine organization level has the potential for allowing greater use of performance improvement techniques. Although existing machine organization techniques exploit intent to a certain degree, we argue that the full potential of this approach has not yet been realized.

Our concern is with good design in machine organization and control programs. Although we do not deal with specific measurements, we use cost/performance as a measurement criterion. We view cost in terms of the number of circuits used at the machine organization level, and our concept of performance is the performance seen by an application. Throughput and transactions per second are examples of application-level measurements, which have the advantage of including the effects of differences in control programs and in machine languages. Conceptually, we assume that the same application runs on the machines to be compared and that the compilers for the machines we are comparing are equally efficient. Since we are concerned with high-end computer systems, we do not insist on minimizing cost/performance absolutely. Rather, we try to minimize it subject to achieving a specified performance level.

Most of our examples are drawn from either the IBM System/370 architecture [1] or the OS/VS2 MVS operating system [2].

2. Environment

In this section we briefly outline the environment we expect for the design and operation of high-end commercial computer systems. We believe that integration of control pro-

gram design and machine organization has the potential to help satisfy the need for increased cost/performance in this environment [3].

• *Hardware environment*

The demands placed on high-end commercial systems will continue to increase, requiring higher performance. In addition, reliability and availability requirements will become even more stringent than they are today.

We expect that chip densities will continue to increase, resulting in a lower cost per circuit. Thus, the use of a greater number of circuits in a system will become an attractive path to increased performance. While our discussion centers on machine organization as a way to make machines faster, improvements in the raw speed of circuits or in packaging (resulting in less distance between circuits) will also lead to improved performance.

• *Software environment*

High-end commercial control programs are characterized by a richness of function that gives rise to a high degree of complexity. These programs comprise thousands of modules, and with their compilers can reach ten million lines of code. High-end supervisors support a wide variety of devices and applications, and characteristically manage large numbers of users concurrently. Data on these systems are shared by multiple applications, and a wide selection of services is provided for the user. The proliferation of professional workstations raises the possibility that the primary tasks performed by high-end systems will be communication and the management of shared data and shared devices.

In providing a rich set of services to the user, high-end commercial control programs necessarily undergo a high rate of change. This is due to the addition of function over time and to the maintenance of function required by a changing environment (e.g., changes in device operation). Keeping control program overhead low in such a complex and fast-changing environment can result in a certain lack of structure and discipline in control program code.

A given high-end commercial control program may be found in over a thousand installations. Since all of these installations rely on information processing for the basic operation of their businesses, there is a severe compatibility requirement on such control programs. The investment in user-written software built on top of the control program is so great that customers cannot tolerate significant recoding, nor recompiling programs written in a high-level language, to accommodate a control program change. Many high-end control programs have therefore followed an evolutionary strategy, preserving the application interface while adding new features. The evolutionary strategy leads to additional complexity, as previous design decisions often limit a designer's flexibility in adding new functions. Changes to the control program must therefore be made only after careful consideration of their impact.

We observe that the control program comprises a great proportion of the code executed by a high-end processor in the commercial environment. There are many reasons why the proportion of system code could continue to grow. These reasons can be summarized as increased user reliance on vendor-supplied code brought about by limited manpower available to fill a large demand for application programming. In particular, an interactive, shared data base using increasingly sophisticated devices (and a greater number of device types) requires a great amount of resource management. As the number of processors in a single installation increases, control programs have to do more work to allow distinct processors to share data while preserving an image of a single system.

3. Machine organization

This section identifies and categorizes the major characteristics utilized in machine organization. We use this categorization to measure qualitatively the degree to which control program intent is exploited in machine organization.

• *Definitions*

Here, we define machine architecture and machine organization. We also comment on our use of cost/performance as a measure of good design.

To define machine organization we must first define the architecture of a machine. The architecture of a machine consists of the entities visible to the machine language programmer and the various rules under which those entities can be manipulated. Registers and memory are common visible entities. Other entities may include condition codes, status words, and storage keys. The instruction set of a machine is the primary set of rules for manipulating the entities. Certain other rules also apply. For example, there are rules that govern the points at which an interrupt can occur, the results of an interrupt, and, for multiple processor configurations, simultaneous updates.

Machine organization is the way in which the architecture is implemented. A given architecture may be implemented in a variety of ways, depending on cost and performance goals and the state of technology at a given time. Given the limitations on the introduction of new architecture discussed in the previous section, an architecture is likely to see several implementations.

It is understandable that most architectures today contain entities that were included in the earliest machine organiza-

tions. Architectural entities are often implemented directly: hardware registers to implement "registers," memory cells to implement "memory," status latches to implement "status words." Often, however, the hardware that is built consists of both more than and less than the entities specified by the architecture. Rich instruction sets are implemented by a simpler set of microcode to achieve speed and simplicity. "Memory" is implemented by a hierarchy that includes a cache. Multiple instructions are simultaneously decoded while the architecture insists that the results produced are equivalent to having executed only one instruction at a time. On lower-speed machines, "registers" (which have an implied performance advantage over memory) are implemented by memory.

Machine organization leads to improved performance through parallelism at different system levels, through the re-use of information (retained in a fast and accessible medium), through various look-ahead schemes, and through design shortcuts for path optimization [4]. While we concentrate on cost/performance issues in terms of number of circuits, we also note that machine organization can influence cost/performance in other ways, such as hardware cost reduction through regular silicon structures and multiple-use parts, and cycle time reduction through the placement of functional units (such as instruction and execution units) on single packages. In addition, we note that machine organization must take account of reliability, diagnosability, and serviceability requirements.

- *Parallelism*

Parallelism is the most important machine organization technique used. It is employed at several levels within a computing system, e.g., the functional unit level, the vector level, the software function level, and the processor level. At each level, parallelism can be limited by the work that is visible to that level, and by resource dependencies.

At the cycle and subcycle level, parallelism is expressed as data width: width of arithmetic units, registers, and paths to memory. At these levels, parallelism is limited by the maximum rate at which instructions can use data. For example, if the architecture were to specify an eight-bit word, an addition could not proceed more than eight bits at a time.

At the functional unit level (e.g., instruction unit, execution unit, etc.), parallelism manifests itself in different forms. Pipelining is one major form. In pipelining, the objective is to reduce the impact of the latency of an operation by structuring the operation in stages and by initiating other operations before the first is finished. Here, the stages typically do dissimilar work, and parallelism is limited by dependencies between stages. An example of such a dependency is the address generation interlock in System/370 pipelined processors: the stage that loads a general-purpose register for one instruction holds up a stage that does address computation, using that register for another instruction, when the two instructions are successive in the program sequence and the two stages are nonsuccessive in the pipeline sequence.

A different type of parallelism occurs at a slightly more global level in processors with multiple execution units which can process work simultaneously, e.g., concurrent operation of a floating-point unit with a fixed-point unit. Here the dependency is at the computation level. In addition, parallelism is limited by the availability of work that is to be done concurrently. For example, the architecture may require that interrupts be precise, thus constraining certain operations to be done sequentially.

On the other hand, the architecture may allow greater parallelism. For example, vector architectures allow several different data to undergo the same operation, which offers several efficiencies. Pipelining techniques can be used with nearly optimal efficiency, since several identical operations are carried out on independent data. The single instruction needed for a vector operation can be expressed in a structured way, eliminating loop control and uncertain branching. Furthermore, the amount of instruction fetching and decoding is reduced, compared to non-vector operations. The parallelism at this level is limited by the amount of "structure" one can find in the application.

Typically, high-performance vector-machine organizations allow chaining, in addition to pipelining of functional operations. Chaining can be thought of as "pipelining" at the vector or aggregate data level: before one vector operation is complete (e.g., Vector Load), the next vector operation (e.g., Vector Multiply) is started.

At the next level, we find various offload schemes that seek parallelism in the software. The objective here is to find high-level functions in a single job that can be performed independently. These functions are generally performed on different but not necessarily dissimilar processors. This type of parallellism generally occurs above the architectural border. While the processor has to detect the conflicts that might limit parallelism at the less global levels, it is generally the control program that is responsible for scheduling offload activity. The use of channels (which are just specialized processors) to perform I/O operations is a common example of offloading.

The final level at which parallelism is employed is multiprocessing. The supervisor plays the key role at this level. A major difference from other levels occurs in the way in which multiprocessing improves performance. While the other techniques employ parallelism to decrease the time needed

to process a single job, multiprocessing only improves throughput in a multijob environment. A result of this is reduced dependency because dependencies occur only when different jobs require the same global resources.

We observe that a dependency that limits parallelism results in more wasted resource at the levels farther from the processor. An empty pipeline may result in several wasted cycles, but the lack of a second job for a multiprocessor wastes the whole processor.

- *Re-use of information*

A second major characteristic effectively used to meet performance needs is *re-use of information.* The objective here is to learn from program execution history. Typically, computing systems repeat several actions and regenerate data. To reduce repetitive operation, frequently used data are saved and reused. A memory hierarchy tries to achieve a suitable balance between cost and performance by keeping data at several levels with different fetch and access times. Such a hierarchy includes the registers, which require fastest access (typically subcycle, because results of the register read-out are used in the same cycle), the cache, which requires access time comparable to one cycle, and various other levels. The hierarchy is required by the different rates of progress in processor logic technology and memory technology. It works because among all the data available to the processor, only a small fraction is needed at any given point in time. The portion needed can be kept "close" to the processor, somewhat like an inventory management system. The data that are needed are frequently reused and change slowly.

Translation look-aside buffers (TLBs) provide another example of how the re-use of information strategy can improve efficiency. A TLB makes use of a table to keep the results of recent translations of virtual addresses to real addresses. Yet another example, though infrequent, is the use of history to predict branch action (see Holgate and Ibbett [5]).

It is a common characteristic of history-based mechanisms that they perform better with more (longer) memory. However, these mechanisms are never perfect, hence they must be built to detect those instances when the history-based action is wrong, and also to recover from the error in prediction.

- *Look-ahead*

Hedging, or look-ahead, is another category of techniques that have been used to improve performance. The basic idea here is that alternatives are evaluated ahead of time and a principal course of action is chosen. In addition, alternative actions are performed with lower priority. As with history-based techniques, there is a possibility of a wrong choice of action as well as the requirement for additional actions to be performed. An example of this strategy is the branching mechanism of the IBM 3033 [6]. While it guesses most conditional branches not to be taken and decodes the next sequential instruction, it also initiates the fetch of the target stream in case the guess is wrong. Until the branch is resolved, the sequential stream and the target stream are both prefetched. A similar strategy is used in prefetching lines into the cache to avoid cache misses. Since the cache is demand-managed, an incorrect action requires an additional operation to correct.

- *Path optimization*

The objective of path optimization is taking short cuts where possible to avoid redundancy. Additional circuits are justified by the frequent saving of operations.

Below the architectural boundary, path optimization leads to various kinds of bypasses. An example is the high-speed buffer bypass: on a cache miss, the missing information is brought directly into the processor instead of having the cache loaded first and then the processor. Another example is the *Load Bypass* on the IBM 3033: data can be supplied to the address generation mechanism at the same time they are supplied to the execution unit, thus short-circuiting register loading and access. The objective here is to approximate the ideas of data-driven computation machines (data-flow machines), where the computation takes place as soon as the data become available.

Above the architectural boundary, path optimization takes the form of machine assists, or "vertical migration" [7]. Common sequences of instructions are combined into one to save instruction accesses and to reduce the number of microcode operations needed to perform the original sequence. An example of an assist is the System/370 instruction, *Obtain CMS Lock*, which obtains an MVS lock under certain conditions [8].

- *Limitations*

Technological restrictions can limit the performance enhancement achievable with the above techniques. For example, cooling restrictions may limit the number of circuits that may be packaged in close proximity, leading to longer signal propagation times. Thus some enhancements may be self-defeating: too many additional circuits may increase the cycle time. Conversely, a machine organization that is structured with packaging capacities in mind has the potential of getting the most benefit out of an available technology.

Various machine organizations can be vastly different in their effectiveness against varying workloads. Richer machine organizations have to choose which of the opportunities

offered by the workload are to be optimized. On the other hand, simple organizations, with heavier reliance on technology, are more robust. (We use the term *robust* here to mean a strategy that performs well in a variety of environments.) Some examples of the workload-related optimization opportunities are branching characteristics, storage-access activity, I/O activity, data dependencies, complex instructions, and code structure. A machine organization geared toward an engineering/scientific environment, with its simpler branching activity, simpler instructions, and heavy floating-point computation activity, is not as effective in a commercial environment characterized by more frequent branching, little floating-point activity, more complex instructions and more decimal activity.

4. Integrating machine organization and control

We now discuss hierarchical structures and the occurrence of such structures in computer systems and the use of hierarchy to reduce complexity. We first outline a conceptual hierarchy in computer systems and then discuss ways in which additional levels of a hierarchy are generated. We argue that in computer systems, hierarchical structuring is uniquely flexible, reduces complexity, and can lead to improved cost/performance. It is our thesis that greater integration of machine organization and control program design can lead to improved cost/performance. We note the role of architecture as the starting point for machine organization and control program design, and assert that the architecture can thus hide useful information from the machine organization level. We conclude with a discussion of the role of naming.

Hierarchy in computer systems

Simon [9] gives an eloquent account of the ways in which complexity, in general, is managed. He argues that hierarchy "is one of the central structural schemes that the architect of complexity uses," and supplies examples from social, natural, and symbolic systems. Simon's definition of a complex system is one "made up of a large number of parts that interact in a non-simple way." We note the emphasis on *interaction.* Systems with large numbers of parts are not necessarily complex. Given that commercial high-end computer systems are among the most complex systems created by man, it is not surprising to observe hierarchical structures in such systems.

A review of the several levels of hierarchy associated with a typical data processing activity begins with the overall goal of the activity: processing an application. The application is broken into several functions, each of which may be coded as a separate module in a high-level language. A compiler, drawing on the services provided by the control program, produces a collection of machine language instructions. Additional hierarchical structure is found in the hardware. The processing of an instruction is broken into several "pipelined" steps: decoding, address generation, data fetch, and execution. Execution is further broken down into microprogram orders. At the lowest level, we have the actual cycle-by-cycle logic operations. Figure 1 illustrates the levels we have discussed.

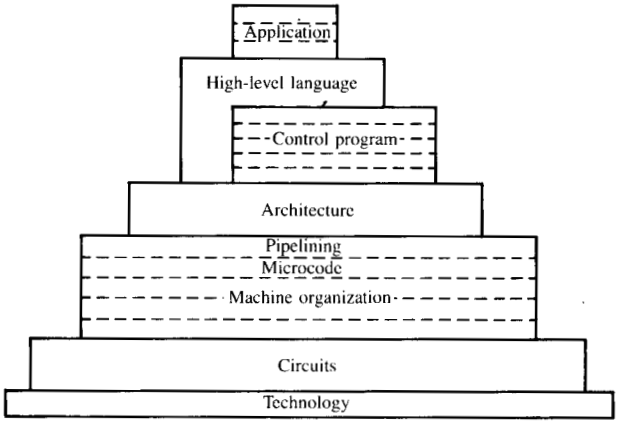

Figure 1 Hierarchical levels associated with data processing.

Given a single level of a conceptual hierarchy, adjacent levels can be generated through abstraction or analysis. The process of recognizing higher-level activities is called *abstraction.* Abstraction begins with a set of components and leads to a single, general activity. This reduces complexity by allowing reason and resources to be concentrated on the overall task. A *DO* loop in a high-level language, for example, hides the machine-language-level details from the programmer and allows him to concentrate on his application. The process of creating a series of components from a general activity is called *analysis.* Analysis can improve cost/performance by allowing several functions to use the same common element. As Simon points out, analysis also allows the separation of interacting portions of an activity from independent portions. Parallel operation of the independent portions is then possible. Hierarchical structuring thus has two complementary goals: reducing complexity and improving cost/performance through the removal of interaction.

We note that computer systems are unique in the degree of flexibility allowed in structuring hardware and software into levels of the hierarchy. In part, this is because the processing of information is not constrained by physical laws to the same extent that most other activities are. It is particularly easy to set up an additional level of abstraction in software. While there can be an arbitrary number of levels of abstraction, and an arbitrary set of activities at each level, we know intuitively that the resulting structure should be tailored to the overall

process. We feel that a "successful" model of an activity should be parsimonious: each level should serve to reduce cost, or improve performance, or reduce complexity. It is in this sense that programming and computer architecture remain crafts rather than acts of engineering.

The hierarchical structure of computer systems reduces complexity in several ways. Hierarchical structuring is used as a means for reducing complexity within the high-level-language level. Iliffe [10] discusses abstraction in software. The complexity of machine language programming is hidden from the programmer by the high-level language. Furthermore, the complexity of managing resources directly is hidden from the compiler by the control program. Dijkstra [11] and Lister and Sayer [12] discuss the value of additional levels within the control program. In Voldman et al. [13] intriguing evidence is presented that Simon's notion of hierarchical interaction applied to software manifests itself as statistical similarity in the pattern of references to memory. Leaving the software levels, we observe that the architecture hides the complexity of micro-orders from the control program and the compiler. At each level, a "virtualization" occurs. Each successive level presents a higher-order machine image. Agnew and Kellerman [14] offer an interesting discussion of the ways in which multiple levels of machine organization can be used to implement a complex architecture.

Hierarchical structure improves cost/performance in computer systems primarily at the machine organization level. The notion of sharing, or centralization of resources, is a major way of reducing cost. When the cost of a particular component is high, we attempt to share it among a group of "users." For example, an adder in a CPU is "shared" among all memory locations. An extravagant alternative would be to have one adder and associated data paths for each pair of memory locations. Multiple additions could then take place independently. At present this alternative is not cost-effective, and thus all data must first be moved to a central adder before the operation can take place. Note that this movement of data to a central resource is handled at the machine-organization level and may be hidden from the programmer by the architecture. The System/370 *Add Decimal* instruction, for example, gives the programmer the illusion that he is directly adding together two operands at arbitrary memory locations. The key parameters in determining whether or not a functional unit is to be replicated or shared are the circuit cost and the frequency of function requests. As the cost of hardware comes down, units that are now shared will be replicated and customized for particular "users." When the request frequency is high, replication becomes more desirable. The implications of replication are being investigated in several data-flow projects (see-Agerwala and Arvind [15]).

Integration of machine organization and control program design

Historically, the levels shown in Figure 1 did not arise in sequence from top to bottom, or vice versa. Machine language programs existed before high-level languages and were executed by direct logic before pipelining was introduced. Computer system design today generally echoes the historical pattern: the architecture for a machine is determined first, with machine organization and software for the machine following. Design of an architecture thus has two major goals. The architecture must allow systems and application programmers to write general programs with reasonable effort, and it must allow an implementation at the desired cost/performance. The control program completes the task of allowing programmers to write efficiently, and machine organization completes the task of allowing an implementation at the desired cost/performance.

We assert that generally, in computer system design, machine architecture has been the starting point from which additional levels of the hierarchy are derived. In particular, the control program level is built on top of the architecture, and the machine organization level is built below it. The architecture has been the reference level for both the control program and the machine organization. Thus, machine organization and control program design have had minimal interaction. An essentially sequential von Neumann architecture, such as System/370, results in both benefits (in terms of reduced complexity) and limitations (in terms of performance). We believe that there is an opportunity for improved cost/performance in the greater integration of control program design and machine organization.

This belief is based on an examination of the techniques described in Section 3 and the degree to which they are exploited on behalf of the control program. For example, most machine organization techniques are based on the execution of a single instruction and make no use of any higher-level information. At the supervisor level, however, processing is hardly a one-instruction-at-a-time activity:

- Instructions are grouped to form higher-level operations.
- Sequentially performed activities are logically independent.
- The same operation is performed repetitively on multiple pieces of data.

We believe that, in the future, machine organization techniques should exploit the characteristics of control program behavior. These characteristics are currently hidden from the machine organization level because of abstraction through the architecture. Significant performance gains may result since information can be used to minimize the cost of performing a function, and control program functions comprise a large fraction of executed code. In MVS running

under System/370, for example, we may recognize logically independent activities (e.g., real memory management) but have no way to carry them out in parallel, short of creating a task that requires a full processor for some fraction of time. Alternatives that could be offered at the machine organization level (for example, a low-cost slow processor tailored to memory management) cannot be entertained with reference only to the architectural level. Considering the machine organization level and the control program level together raises the possibility of greater parallelism, better re-use of information, more accurate hedging, and additional path optimization. We discuss further possibilities in Section 5.

Greater interaction between machine organization and the control program is feasible in the current high-end environment because system "users" (including compilers and other system software) interact almost entirely with levels at or above the control program. Joint optimization of lower levels is thus possible without disruption to most users. Furthermore, the control program is supplied by the hardware vendor. Radin's approach [16] involves integration of levels in a different way. The 801 architecture, compared to System/370, does not have as many complex instructions and offers greater resources, such as registers, at the architectural level. In Radin's case, complexity is hidden from the user because the user interacts with the system at the programming language level. The (system) compiler optimizes the use of resources available at the machine organization level for all code: control program code as well as application code. The compiler must also compensate for architectural limitations.

- *Naming*

In essence, abstraction is recognizing and naming the overall result of multiple activities, and analysis is recognizing and naming the components of a more general activity. Naming is important because objects that are indistinguishable from one another cannot be treated differently. In particular, an activity must be recognized (named) in the level at which we wish to manipulate it. For example, if we wish to exploit the sequentiality of memory references in certain specific cases, but not in all cases, we must make sure that the *yes*-cases can be distinguished from the *no*-cases by the memory hierarchy. If accurate detection is only possible at a higher level, the detected information must be communicated across levels for the technique to be effective. In our case, communicating information from the control program level to the machine organization level requires additions at the architectural level.

We observe that, currently, entities at the machine organization level generally recognize, or name, a limited number of cases. For example, the pipeline treats every instruction as having the same stages, regardless of the instruction type. A limited number of names at a level result from the sharing of high-cost items, discussed above. Although the importance of naming is well known in software [17], it has not been adequately recognized in hardware. As the cost of circuits decreases, it will become feasible to recognize a greater number of names at the machine organization level. We note that addresses are names for memory locations that are extremely easy to manipulate.

5. Directions for research

In previous sections we have outlined the types of machine organization techniques that lead to improved cost/performance. We have argued that the opportunity exists for greater exploitation of control program intent at the machine organization level. In this section, we indicate several distinct areas in which machine organization and supervisor design can interact. We discuss primitive operations, management of the memory hierarchy, relationships to scheduling, and co-processing opportunities. We conclude with comments on the methodology for investigating these areas.

Recently, several research projects have been aimed at defining an architecture based on the code that one expects to execute. Although these projects have tried to integrate the design of several levels of our conceptual hierarchy, they have primarily been directed toward the execution of high-level languages [18, 19]. In the high-end commercial environment, the control program constitutes the bulk of the executed code, and we thus believe that such efforts should be based on typical control program function rather than typical high-level language function.

- *Primitive operations*

The simplest way in which the supervisor can have an impact on machine organization is to architect "primitive" (from the control program point of view) operations. This exploits the technique of path optimization. In several instances, control program requirements have resulted in additions to the System/370 architecture, e.g., added instructions for multiprocessing. System/370 has also had vertical migration of code into the architecture. The dual-address-space facility of the System/370 architecture is a more sophisticated addition based on the need to communicate across address spaces. Additional, more complex primitives could also be added to high-end architectures.

Typically suggested operations include those for queue management (e.g., add, delete), process management (e.g., create, suspend, resume, destroy), and communication (e.g., wait, signal) [20]. The IBM System/38 [21] incorporates many of these functions and has multiple machine organization levels. As discussed previously, these "machine assists" potentially improve performance by avoiding certain steps in instruction processing. Performance improvements result

from not having to store back or re-fetch operands between instructions. Assists also simplify building a control program by standardizing common operations, reducing the possibilities for error, and simplifying the system programmer's task. Repetitive execution of a sequence of instructions might also be detected in the machine organization without specifically adding an instruction to the architecture. Striding through an array by columns is an example.

The key parameters that determine the value of an assist are the number of steps saved by the assist and the frequency with which it is used. Assists formed from a set of instructions in which the same data items are re-used offer the opportunity to save more steps than simpler assists. To increase the chance of re-using the same data, larger sets of instructions can be considered. The drawback of such complex assists is that a complex operation is not used as frequently as a more fundamental assist. In addition, more circuits may be needed to implement a complex assist. A well-structured control program is necessary to achieve the highest frequency of use for an assist. Otherwise, some potential uses of the assist may go unrecognized or require significant recoding. Note that determining which instruction sequences should be combined into a single unit is nearly impossible without an understanding of what the software is trying to accomplish. This is a major area for further research.

Management of the memory hierarchy

Management of the memory hierarchy is the second area in which the supervisor and machine organization can interact. The techniques we describe here fall into the "look-ahead" category. Currently, most high-end processors use a cache, or high-speed buffer, to keep the processor supplied with data. (The large memories on these processors cannot supply data at the required speed.) In most cases the cache is transparent to the programmer and is not included as part of the architecture. This simplifies the task of the programmer and is in contrast to the direct management of main memory by software. Caches are typically managed according to a demand-block fetch/least-recently-used (LRU) replacement scheme. That is, a block of data (e.g., 128 bytes) containing the requested operand is brought from memory into the cache, replacing the least recently used item in the cache. Caches exploit a property of program behavior known as *locality*. Basically, there are two types of locality: (a) the processor is likely, in a "close" interval of time, to need data located physically near the data previously requested; and (b) the processor is likely to re-use data that it has already referenced in a "close" interval of time [22]). The LRU scheme is very robust, compensating for the lack of programmer control.

Interestingly, much control program activity does not exhibit locality to the degree that application code does. In searching a linked list, for example, the operating system neither references data physically close to the last referenced data nor does it re-use the list in a "close" interval of time. At the expense of greatly increased programming complexity, one could increase flexibility in managing the cache by making it visible to the programmer. However, limited communication between the control program and the cache offers the possibility of performance improvements without burdening the programmer, particularly the application programmer. For example, the control program might signal that it was going down a linked list, and the cache would respond by fetching the entire list. Since the linked list is not likely to be reused in the near future, the data fetched would be subject to early replacement.

Further performance improvements might be possible if the memory hierarchy prefetched data on the basis of what the operating system was doing. Such prefetching could be triggered by a history of past activity or by a signal from the operating system that a particular function was being entered. For example, the cache might respond to the linked list signal by prefetching entries on the list. Prefetching could be extended to move data across all levels of the hierarchy simultaneously.

Scheduling of work

This leads us to the third area in which the control program might interact with machine organization: the scheduling, or dispatching, of work. This area falls into the "re-use of information" category. Wherever a "setup cost" (such as a memory fetch or a state swap) is incurred, organizing the work can lead to increased performance. Some setup costs, such as the cost of bringing pages into memory, are recognized by the control program, and scheduling decisions take it into account. Other setup costs, such as a cache miss, are hidden from the control program and thus cannot be taken into account. The control program views the processor as a single entity, letting priorities and interrupts determine the next unit of work to be dispatched. Using the cache as an example, one might be able to achieve better performance by dispatching work based on current cache contents. On multiple processors, one might attempt to move those units of work that use the same data to a particular processor. A user, for example, might be dispatched only on one processor, so that his data would not move from cache to cache. Functional separation might also provide good separation of data. However, the control program would have to distinguish one function from another explicitly. While control programs have control blocks for users, they are not as likely to have internal representations of functions.

Co-processing

The final area in which machine organization and control programs might interact is co-processing. Co-processors

cooperate with the main processor by performing particular tasks. They share some memory with the main processor and are not necessarily as fast as the main processor.

At one extreme, co-processors could be highly optimized special-purpose processors that efficiently execute particular instruction sequences "in-line" while the main processor waits. In a sense, a cache is such a co-processor. The cache is dedicated to moving data between levels of the memory hierarchy. Prefetching schemes put additional intelligence in such a co-processor. At the other extreme, co-processors could "offload" work from the main processor in the sense of I/O channels. For example, much of the resource management in control programs might be done outside the main processor. (Research is necessary to determine the degree to which resource management can be separated from the main flow of the control program.) Co-processing has the greatest potential performance improvement when the co-processed activity is totally overlapped with the main processor. We concentrate on co-processing as a technique that increases parallelism.

Recognizing appropriate work for co-processors in the control program is analogous to breaking instruction execution into pipeline stages. Data movement lends itself naturally to parallel processing, either between levels of a memory hierarchy or within a level. Co-processors could thus be used in conjunction with prefetching. Co-processing can be thought of as the limit of path optimization: sets of instructions are completely removed from the main processor.

In addition to the splitting and overlapping of the work currently performed, co-processing also suggests the possibility of hedging on control program activities, just as is done currently on branches at the instruction level. For example, queue searches, a major control program activity, could be performed on a co-processor. The co-processor would find not just one matching item but as many as are available at the given time. This batching of logical activity is analogous to blocking for data movement. Search results would then be available for the main processor when needed. Control programs often must test a series of conditions to determine an appropriate action. A co-processor could keep track of the final outcome of the tests for use by the main processor when needed. Research is needed to determine whether or not this information can be provided accurately enough to avoid incorrect actions.

- *Methodology*

We have outlined several areas in which machine organization and control program design can have greater interaction. Each of these areas requires careful research to determine if it can improve cost/performance of high-end processors. Accurate performance evaluation requires control program prototypes and hardware simulations. Care must be taken not to introduce an unmanageable degree of complexity by integrating the design of several levels.

A good understanding of control program activity is necessary in order to determine, for example, which functions should be done on co-processors. We note that it is extremely difficult to determine the sequence of execution of instructions from an examination of static control program code. Such systems are typically multiprogrammed and interrupt-driven. Furthermore, a given application may invoke a complex sequence of control-program functions. Detailed traces of a running system are necessary to understand the functional flow of the control program. Gathering such traces requires a representative workload or series of workloads, since the type and frequency of control program activity varies with workload.

It is easier to assemble the tools necessary for such research for evolutionary architectures. By evolutionary, we mean machine architectures that persist across more than one product cycle. Control programs and typical workloads are more readily available for such architectures.

We note that co-processing requires good, fast process synchronization techniques, and that control programs with good structure lend themselves more readily to enhancement in machine organization.

Acknowledgments

The authors thank R. Baum, D. Gilbert, J. Knight, L. Liu, M. Kienzle, J. Pomerene, R. Rechtschaffen, K. So, D. Stein, and J. Voldman for their insights and comments that contributed to this paper. We appreciate the comments and criticisms of the referees, whose efforts enabled us to make this paper clearer and more readable.

Reference and notes

1. *IBM System/370 Principles of Operation*, Order No. GA22-7000, available through IBM branch offices.
2. A good introduction to MVS is provided by *OS/VS2 MVS Overview*, Order No. GC28-0984, available through IBM branch offices.
3. A more detailed perspective on the environment for the design of a high-end computer system may be found in E. Bloch and D. J. Galage, "Component Progress: Its Effect on High Speed Computer Architecture and Machine Organization," *High Speed Computer and Algorithm Organization*, D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Eds., Academic Press Inc., New York, 1977, pp. 13–39.
4. More details on machine organization are provided by P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Publishing Co., Inc., New York, 1981.
5. R. W. Holgate and R. N. Ibbett, "An Analysis of Instruction Fetching Strategies in Pipelined Computers," *IEEE Trans. Computers* **C-29**, 325–329 (1980).
6. *3033 Processor Complex, Theory of Operations/Diagrams Manuals*, IBM Maintenance Library, Order Nos. SR23-5296 through SR23-5302 (1978), available through IBM branch offices.

7. A discussion of vertical migration may be found in J. Stockenberg and A. van Dam, "Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/Software Systems," *Computer* **11,** 5, 35–50 (1978).
8. *IBM System/370 Extended Facility and ECPS:MVS*, Order No. GA22-7072, available through IBM branch offices.
9. H. A. Simon, "The Architecture of Complexity," *Proc. Amer. Philosoph. Soc.* **106,** 467–482 (1962).
10. J. K. Iliffe, *Advanced Computer Design*, Prentice-Hall International, London, 1982.
11. E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott, Eds., Academic Press Ltd., London, 1972, pp. 72–93.
12. A. M. Lister and P. J. Sayer, "Hierarchical Monitors," *Software—Practice and Experience* **7,** 613–623 (1977).
13. J. Voldman, B. Mandelbrot, L. W. Hoevel, J. Knight, and P. Rosenfeld, "Fractal Nature of Software-Cache Interaction," *IBM J. Res. Develop.* **27,** 164–170 (1983).
14. P. W. Agnew and A. S. Kellerman, "Microprocessor Implementation of Mainframe Processors by Means of Architecture Partitioning," *IBM J. Res. Develop.* **26,** 401–412 (1982).
15. T. Agerwala and Arvind, "Data Flow Systems," *Computer* **15,** 10–13 (1982).
16. G. Radin, "The 801 Minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* ACM, Palo Alto, CA, 1982, pp. 39–47, reprinted in this issue with permission.
17. A discussion of naming in software may be found in J. H. Saltzer, "Naming and Binding of Objects," *Operating Systems, An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmueller, Eds., Springer-Verlag, Berlin, 1979, pp. 99–208.
18. M. J. Flynn and L. W. Hoevel, "Execution Architecture: The DELtran Experiment," *IEEE Trans. Computers*, Vol. **C-32,** No 2, 156–175 (1983).
19. J. Sansonnet, M. Castan, C. Percebois, D. Botella, and J. Perez, "Direct Execution of LISP on a List Directed Architecture," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* ACM, Palo Alto, CA, 1982, pp. 132–148.
20. See for example N. Kamibayashi, H. Ogawana, K. Nagayama and H. Aiso, "Heart: An Operating System Nucleus Machine Implemented by Firmware," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* ACM, Palo Alto, CA, 1982, pp. 195–204.
21. *IBM System/38 Technical Developments,* Order No. G580-0237, available through IBM branch offices.
22. A discussion of locality may be found in E. G. Coffman and P. J. Denning, *Operating Systems Theory*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1973.

Received November 2, 1982; revised December 28, 1982

Gururaj S. Rao *IBM Data Systems Division, 44 South Broadway, White Plains, New York 10601.* Dr. Rao joined IBM at the Thomas J. Watson Research Center, Yorktown Heights, New York, in 1978. He has worked on various problems in high-performance processor organizations. Since 1982 he has been on special assignment as technical staff to the director of planning at the Data Systems Division headquarters. His interests include high-performance processor organization, architecture, operating systems, performance methodologies, and design. Prior to joining IBM, he was an assistant professor in the Department of Electrical Engineering at Rice University, Houston, Texas. Dr. Rao received his Bachelor of Engineering degree in 1971 from the University of Mysore, India, the Master of Engineering degree in 1973 from the Indian Institute of Science, Bangalore, India, and the Ph.D. degree in 1976 from Stanford University, California. All of his degrees were in electrical engineering. In 1982, Dr. Rao received an IBM Outstanding Technical Achievement Award.

Philip L. Rosenfeld *IBM Research Division, Yorktown Heights, New York 10598.* Dr. Rosenfeld joined IBM in 1978. He is currently manager of multiprocessor cache studies. He is studying cache behavior and the interaction between supervisor design and hardware performance. His interests include operating systems, multiprocessors, and memory hierarchy behavior. Dr. Rosenfeld received his B.S. in engineering from Cornell University, Ithaca, New York, in 1973 and the M.S. and Ph.D. in operations research from Cornell University in 1975 and 1980. Dr. Rosenfeld is a member of the Association for Computing Machinery, the Operations Research Society of America, and Tau Beta Pi.