SOME ISSUES IN PROGRAMMING MULTI-MINI-PROCESSORS

A. Newell and G. Robertson
January, 1975

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by Air Force Office of Scientific Research. Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 15213.

SOME ISSUES IN PROGRAMMING MULTI-MINI-PROCESSORS*

A. Newell and G. Robertson

INTRODUCTION

Large computer systems can be constructed by joining together many minicomputers -- creating what can be called multi-mini-processors. The first such systems are just reaching the point where problems of programming and use dominate problems of design and construction. This paper attempts to share some of our early perceptions about what these problems of programming and use are. It also allows us to capture a historical record of our current viewpoint.

We are not the architects of the multiprocessors we will describe. We are not even the primary systems programmers, who create the operating system and operating environment within which the user operates. We are users of the system. But we are not arms-length users, as are the users of a typical university computation center. For to use such a system one must indeed create a special programming system on it. Thus we are, shall we say, systems exploiters. We are just coming deeply into contact with our multiprocessor. We find ourselves facing many issues of how to exploit the system and to program it -- of how to make it yield to our will.

First we will sketch the multiprocessors that we are concerned with. There are only two of them, and we, the authors, are actually working on only one. With this as background, we will discuss seven programming issues.

The role of minicomputers as components of multiprocessor systems is quite different from their classical role as laboratory computers. Though some of these seven issues will seem quite familiar to those whose world is the on-line laboratory use of computers, some of them will seem quite foreign. Hopefully, however, they will paint an interesting picture of a use of minicomputers that will become increasingly common.

---

* This paper was given as an invited talk at the 1974 Conference on the On-Line Use of Computers in Psychology.

MULTIPROCESSORS

There are only two genuine multi-mini-processors, as far as we know, though there may be others in design. A multiprocessor is characterized not only by the existence of many processors, but by the sharing of primary memory, i.e., the processors address common memory. This sets them apart from networks of computers, which have many computers, but where the intercommunication is essentially from secondary memory to primary memory, i.e., each computer sees all the other computers as peripheral devices. Multiprocessors permit a degree of computational intimacy not available with networks.

C.MMP: THE CMU MULTI-MINI-PROCESSOR

C.mmp is the multiprocessor at the Computer Science Department of Carnegie-Mellon University [Wulf, 1972]. As shown in Figure 1, C.mmp consists of 16 PDP11 computers connected through a crosspoint switch to 16 primary-memory ports. Each primary memory is 2↑16 words, for a total memory of a million words. Each processor still looks like a PDP11 with a 16 bit word and an address space of 2↑15 words (actually, 2↑16 bytes). In fact, a modest modification must be made to a processor to operate within the system.

Each of the processors can lay its address space anywhere in the million words of the primary memory. It does so through an address relocation box (Dmap in the figure), which breaks the address space of the processor into eight 4,096 word pages. Thus the system has a small number of large pages, each of which may be independently relocated through Dmap.

Each Pc has its own Unibus, the standard bus structure of the PDP11. On this hangs 4K of local memory as well as all the peripheral gear of disks, drums, printers, and connections to the external world. The last includes a connection to the PDP10, which is the large general-purpose time-shared computer system in the Computer Science Department. As the figure shows, there is also a large (60-bit 1-microsecond) clock (K.clock), which provides a common reference frame, and an interrupt (K.interrupt) which connects all processors. There is at the moment no switching between secondary devices and the various processors. A disk, for example, is permanently located with one Pc.

Not shown in the figures is the ability to partition the system, either dynamically or statically (manually), so that it consists of independent subsystems. Thus it is possible, for example, to have hardware maintenance going on at the same time that a user system is operating with other Pc's and Mp's.

The system, though made up of minicomputers, constitutes a large computer. Taking processors to be 11/40s yields about .3 to .4 million instructions per second (mips) per processor, for a total of 5 to 7 mips. This compares approximately with an IBM 360/158. There must be some contention for memory as the number of processors increase, but this is not expected to be large (the Dmap's for the

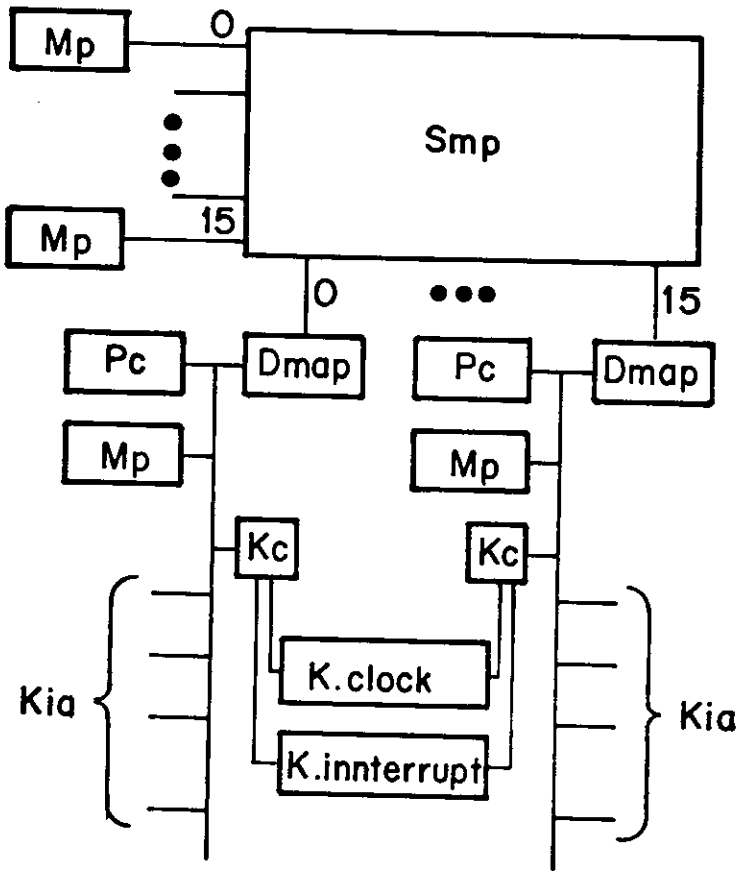


Figure 1: C.mmp Architecture

11/40s contain a cache).

The system has been operational in parts for some time. The 16x16 switch has been running since March, 1974. We currently have five 11/20 Pc's operational with 500K of memory. We do not have any modified 11/40s.

PLURIBUS IMP: THE BBN MULTI-MINI-PROCESSOR

The second multi-mini-processor system, the PLURIBUS IMP [Heart, 1973], has been developed by Bolt, Beranek and Newman to serve as a high speed modular IMP (interface message processor) for the ARPA computer network. Figure 2 shows its structure. The processors are Lockheed SUE minicomputers, which are 16-bit machines with a 15-bit word address, and which are about the speed of the 11/20. They have a bus structure which is similar to the DEC Unibus. As shown in the figure, two processors are located on each bus, each with 4K of local memory. Thus the figure illustrates a 14-processor system, the maximum size for which the PLURIBUS IMP was designed. (The number was determined by the application, not by hardware limits.) The primary memories come in 8K units with two units on each memory bus. The switch is distibuted, unlike the C.mmp which is a monolithic device. Thus, links run between the buses of the processors and the buses of the memories; any pattern of access can be obtained. As the figure shows, there are also I/O buses which are linked in similar fashion.

An initial system has been running at BBN since mid-74. It has operated with a range of configurations (up to the 14 shown). A basic design objective was to create a modular series of IMPs, which could be tailored to the processing load of the network node. Though deliberately designed for a specific application, the hardware structure is quite general. It poses many of the same basic issues we face on C.mmp, and the approaches taken on the PLURIBUS IMP offer interesting contrast points with those taken on C.mmp.

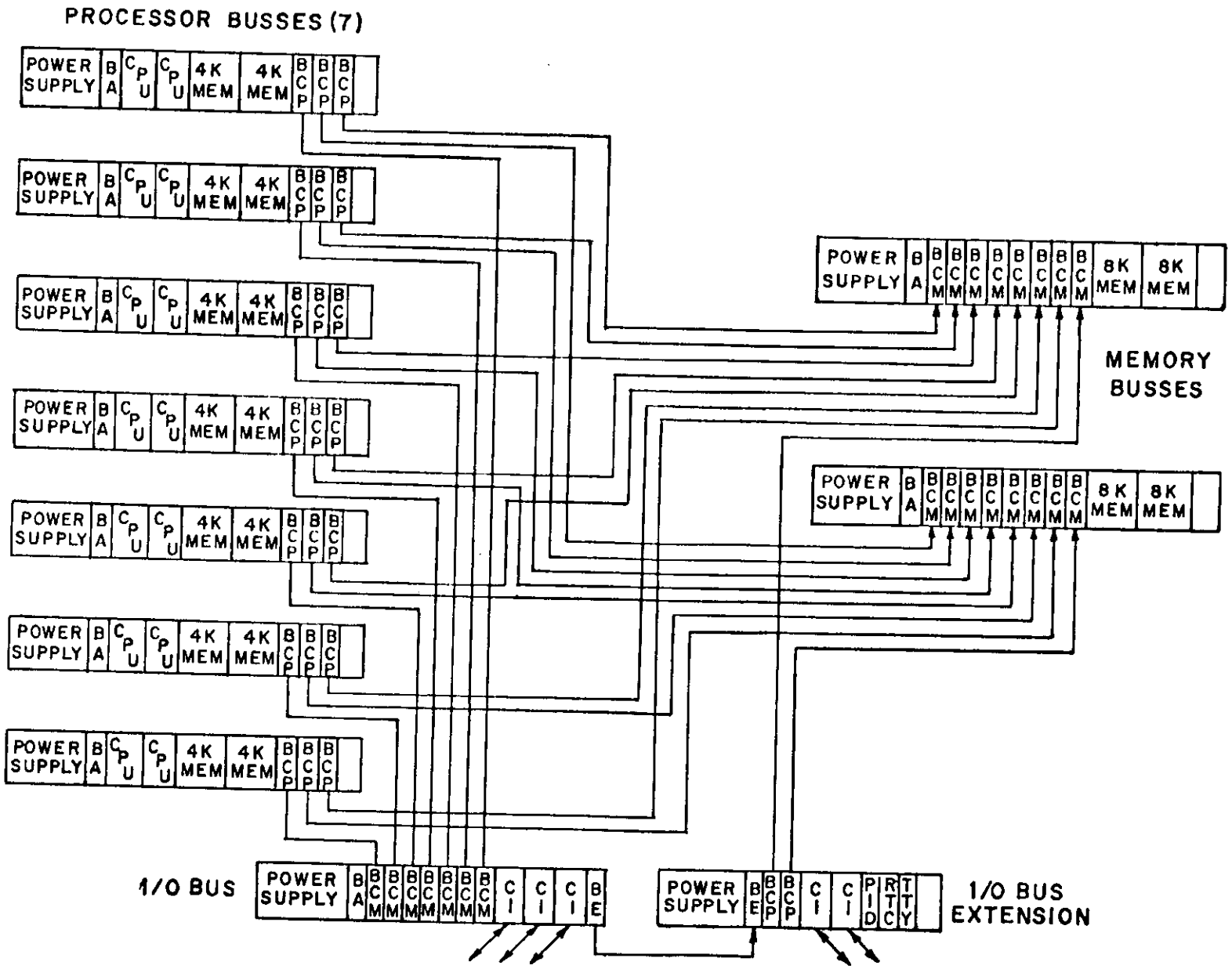


Figure 2: PLURIBUS IMP

(Adapted from Heart, 1973)

ISSUE 1: HOW TO EXPLOIT A MULTI-MINI-PROCESSOR

The first issue is simply how to exploit a multiprocessor, since it is a large system in terms of power, memory and bandwidth. It has special structural characteristics, which are easy enough to state, but not so easy to translate into performance consequences.

One might say there is no issue -- simply use the machine. But the question is not laid to rest so easily. Different strategies of exploitation require that effort be spent in different ways, thus precluding following alternative paths with any efficiency. Indeed, the issue as posed makes it sound like the multiprocessor arrived sui generis with the question of use fully open. That is not the case. The exploitation strategy is chosen before the design even begins and effects many of the structural features of the system. The actual situation is more like making a movie. Constructing the hardware system is like filming. Using it is like producing the movie in the editing room. The final editor is free to make any kind of movie he wants, but he must work with the film given him by the director.

There are three main strategies for exploiting multi-mini-processors. We take up each in turn.

PROGRAM IT FOR A SPECIAL TASK

The first strategy is to view the multiprocessor as a specialized device created to do a specialized task. Hardware and software are to be combined optimally to perform that specialized task.

This in essence is the stategy followed by the BBN group in designing the PLURIBUS IMP. The task existed ahead of time in a well defined form -- the ARPA Net is a functioning system with a minicomputer (the Honeywell 516 and 316) as IMP, and much experience, both statistical and qualitiative, has been gained with the requirements for an IMP. What was needed was an efficient and highly reliable implementation that could be scaled to the task. All this information existed prior to design time, and the software and hardware were designed together in apparent total harmony.

The effects of this can be illustrated by what is surely a striking feature of the PLURIBUS IMP -- it has no interrupt! There is no way in which an arbitarily occuring external signal can cause the system to attend to another task. Since the interrupt was introduced in the late Fifties, it has been considered as manadatory as I/O channels. Abandoning the interrupt is an important design decision.

The underlying rationale is very simple. The algorithm to be programmed was well understood and existed in code form before the hardware design began. Detailed analysis of the program revealed that it could be partitioned into segments that never take longer than 300 microseconds. Since the responsiveness of the system

fits the grain of 300 usec, all processes can run to completion without interruption.

Co-equal with the short program segments is the necessity of getting new tasks assigned to a processor. If this takes any appreciable fraction of 300 usec, then the overhead defeats the scheme. The BBN group developed a device called a PID (Pseudo Interrupt Device). This hardware device holds a set of numbers, corresponding to tasks, which have been given it at arbitrary moments. The device instantly delivers (and deletes) the highest number, corresponding to the highest priority task.

Interrupts take appreciable time (e.g., for changing processing contexts), which is avoided by the PLURIBUS IMP, along with a fair amount of operating system code. In fact, the system does not have an operating system in any general sense of the word. The necessary functions are distributed carefully, such as by the PID.

This seems highly specialized. Indeed, that is the point. If viewed as a device to achieve a narrow, well-defined total task, such specialization is possible. Furthermore, though we know of no estimates of the gains made to the PLURIBUS IMP by such specialization, we estimate that they are impressive.

As a final footnote, the motive of specialization does not condemn the results to be equally specialized, though that must be the fate of most specializations. But the PID and the associated concept of presegmenting the code into run-to-completion steps may not be of such limited generality -- though it does pose an interesting compiler problem.

STANDARD USER ENVIRONMENT

The second strategy is to view the multiprocessor as providing a standard user environment, much as any other computer does. Thus, when completed with operating system and user facilities such as file systems and language processors, the system will look no different to the user than your local computation center. Only down in the boiler room, so to speak, will the multiprocessor design become apparent.

Indeed the two specialized flavors of multiprocessors that do exist in quantity are used in exactly this way. One is the use of I/O processors; the other is the use of dual central processors. In both cases, they simply handle more efficiently the total set of tasks that has to be done for a general user shop. An interesting example of this is the CDC 6600 which has a large central processor surrounded by ten miniprocessors. With few exceptions that we know of, it shows up simply looking like a very powerful general computing system.

With this view the real questions are the economics of multiple smaller processors versus the single larger processor for

obtaining a given number of mips per dollar. That is of course always the question in computation, but here no specialization comes from specific applications.

MULTIPLE SPECIALIZED APPLICATION SYSTEMS

The final exploitation strategy is to view the multiprocessor as a system in which a number of specialized application systems will be realized, both simultaneously and over time. Each of the application systems will be adapted to the structure of a multiprocesser in order to take as much advantage of it as possible. This is the view taken with C.mmp, and our discussion reflects essentially the considerations that have arisen with respect to C.mmp.

First of all, this strategy leads to a general operating system, since several applications will be running simultaneously. Even if we envision some "production mode" where one application might dominate the system for a period of time, throughout most of the life of an application system one is coding, debugging, modifying, developing and exploring. For this one neither needs nor wants the entire multiprocessor. The operating system of C.mmp is called HYDRA [Wulf, 1974]; we will describe some of its features after introducing another important consideraion.

If small address spaces (those of the Pc's) are to move around in large memory spaces (that of the million-word Mp), then there must be a memory mapping. D.map (see Figure 1) accomplishes this for C.mmp and BCP (see Figure 2) does so for the PLURIBUS IMP. The important design question is the nature of that mapping. An attempt to build a general user system leads to making that mapping a general demand-paging scheme. Thus all addresses go through a dynamic process of discovering whether the page is in memory and if not bringing it into memory, translating the processor address into the physical address in the memory. Thus has evolved the general virtual machine concept in modern computing.

C.mmp does not have a demand-paging scheme (nor does the PLURIBUS IMP). One reason arises from the strategic view under discussion. Paging schemes are expensive (in time), more so than simple relocation schemes. Thus, to put a paging scheme in the hardware is to agree to make every user pay this cost. At this point one has eliminated some of the important possibilities for adaptation. For not only the cost goes up, but everyone will be subject to the same paging system with its particular cost profile, whether it fits the needs of their particular system or not.

Thus what we find in C.mmp is a so-called "large page" scheme, in which there are only eight pages, each large enough to hold a substantial subsystem (4K words). The use of this page system is then left to the user. There is a cost, for insofar as paging is necessary, it must be detected and executed by software. But in return, we obtain the possibility of fitting the paging system to the application system. That is, insofar as pages can be left in place we

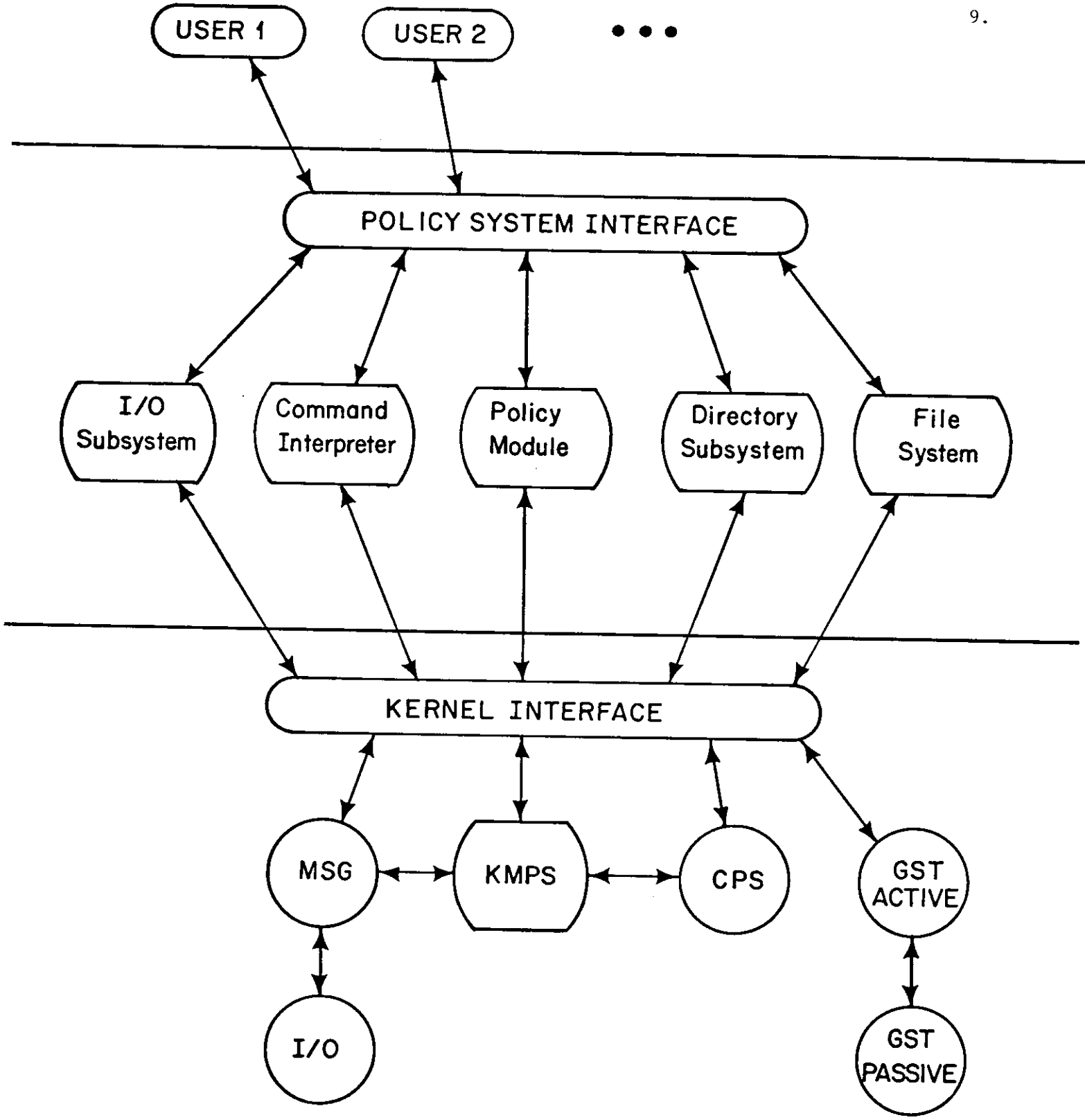


Figure 3: HYDRA System Organization

pay minimal addressing overhead.

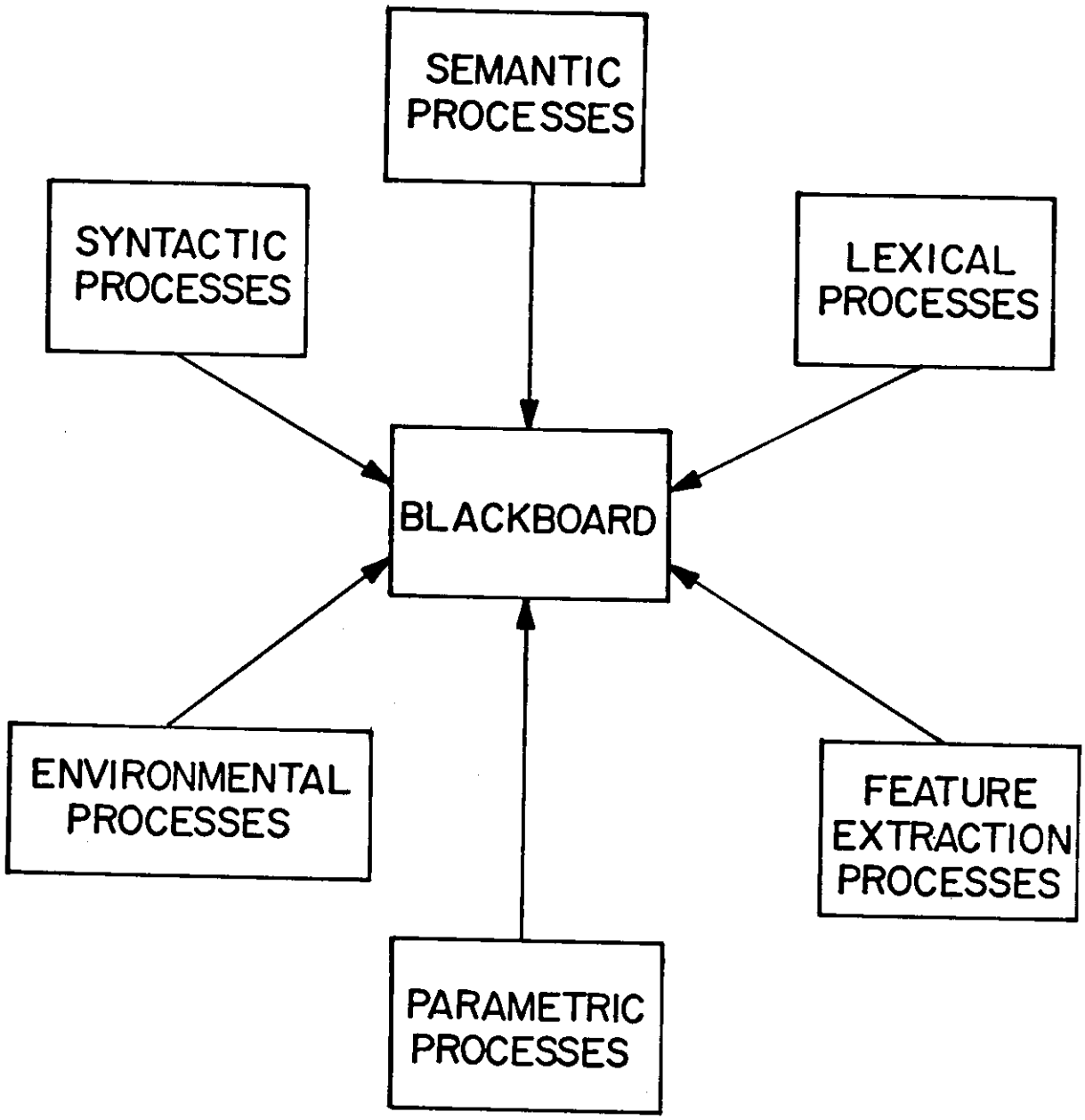
HYDRA: The C.mmp Operating System. Let us now return to describe the operating system, which has been designed in part with the strategic choice of making multiple application systems possible. There are several aspects to HYDRA that strongly reflect this strategy. One can be seen in its overall system organization, shown in Figure 3. There is a Kernel to the system, shown in the bottom part of the figure. This performs two essential functions that cannot be delegated to subsystems. One is the protection and security of the resources in the system. This is indicated in the figure by the Global Symbol Table (GST Active and GST Passive, the latter being the collection of those resource entities not currently in use). The other is a basic multiprocessing system that divides up the resources in a simple way. This is indicated by Kernel Multiprocessing System (KMPS), and by the processes that manage the set of pages in core (CPS), a primitive I/O system (I/O), and an interprocess communication system (MSG).

Thus HYDRA allows different high-level operating systems, called policy-systems, to co-exist. A policy-system has associated with it a policy module which makes decisions for KMPS about scheduling and paging, and a set of user facilities (e.g., command interpreter, file system, terminal handling system). When a user of HYDRA first logs into the system, he may request a particular policy-system. In this way, operating systems can be tailored to the specifications of individual applications systems.

An application system: Speech Understanding. An illustration of an application system will show the reason for an entire co-operating system, and also what it might mean to fit an application to a multiprocessor organization.

A speech understanding system is a system that takes in an utterance and determines the meaning of the utterance in the light of a specific task context. Such systems are being constructed at several places [Newell, 1973]. One of these efforts is at CMU and we are pursuing it independently of any interest in multiprocessors.

But we are also attempting to construct a multiprocessor version for C.mmp, and it forms the initial application system. Figure 4 shows the conceptual structure of this system, called HEARSAY-2 [Lesser, 1974]. The basic structure of the system revolves around a global data structure (the blackboard) and a set of cooperating parallel processes (the knowledge sources). Each knowledge source has expertise in dealing with some particular aspect of speech understanding. A knowledge-source process is invoked when some particular pattern in the blackboard is noticed (a precondition). The knowledge source then deals with data in the blackboard and usually makes weighted guesses about what it has seen. There is a controlling process that watches the activity of the



INDUCTION MODEL

Data Directed
Information Gathering
Hypothesize and Test
Parallel and Independent
Deactivation Simple

Figure 4: HEARSAY-2 Conceptual Structure

various knowledge sources and uses the collectively weighted guesses to eventually understand an utterance.

The HEARSAY structure lends itself to a direct decomposition into parallel processes to take advantage of a multiprocessor architecture. Each knowledge source can be a seperate process and in many cases multiple copies of a given knowledge source can be used. Having several sources of knowledge simultaneously working yields significant improvements in the time it takes to recognize an utterance. This is particularly important since the ultimate goal of such systems is to recognize speech in real time. The HEARSAY structure thus allows for effective use of a closely coupled multiprocessor where a large common data base can be easily accessed by a large collection of processes.

In a large system like HEARSAY, it may be necessary for special scheduling algorithms or paging strategies to be employed. For example, the initial operating system built on top of HYDRA does not provide for priority classes in scheduling. The large collection of processes the make up the HEARSAY system may very well need to be priority ordered to obtain the desired effects. The important point is that HYDRA does allow for another operating system to co-exist with the initial one that most users will use. Thus, we expect HEARSAY to develop and use its own operating system built on top of HYDRA.

CONCLUSION - STRATEGIES

It is important to realize that of these choices, no particular one is "right". Each is an attempt to maximize the payoff for specific, but different, goals. The first choice, that taken by the PLURIBUS IMP, attempts to maximize the efficiency and reliability for a specific task. With the second choice, that of building a general computing environment, one is trying to find the most efficient way to construct a certain environment. If multiprocessors can compete for that environment, they can be an implementation of choice and multiprocessers should be designed to meet that demand. The third choice, that taken by C.mmp, attempts to gain the advantages of specialization but over an unknown range of application systems. It must necessarily trade off some possibilities of specialization against a system that can handle several such applications simultaneously. Similarly, it must trade off the best scheme for general computing in order to permit adaptations to occur.

Nor are the choices mutually exclusive in the sense that if you choose X you are precluded from the same applications that choice Y permits. Speech systems will be brought up on general purpose systems. (We are creating a version of HEARSAY-2 on our PDP10.) We will be creating a general user environment on C.mmp, which will run simultaneously with our work in speech. And we would certainly not be surprised to see the PLURIBUS IMP used for other applications quite remote from the message processing task.

The choice of strategy does bias the application potential of a given system. It gives reasons for making design choices consistently so that the system adds up to something, at least along some dimensions.

ISSUE 2: HOW TO GET ALL THE SOFTWARE

The second major issue is how to obtain all the software that is needed for such a system. By now we are all aware that it takes an immense amount of software to make a computer system livable. In practice such software only arises with the development of a large and active user community, plus the continued efforts of the manufacturer over several years. No general preaching on this fact should be necessary in a minicomputer user community where new systems arrive from the manufacturer rather bare, despite advertizing claims.

The multi-mini-processors are composed from existing minis (C.mmp from an extensively used system, the PDP11; PLURIBUS IMP from a new machine, the Lockheed SUE) and programs exist for these minis as stand-alone systems (many for the PDP11, fewer for the SUE). Yet these do not go very far toward satisfying the need. First, all such systems must be reconditioned to work in a multiprocessor environment. To do this in a way that exploits the multiprocessing is a genuine system-programming problem. But further, these multiprocessors are big systems with big memories and they can use software systems commensurate with that power. All this adds up to a major problem.

There are several approaches to obtaining the software. Even more than with the strategy of exploitation, these are not mutually exclusive. In fact they form an armatorium and all should be used (and pretty much are on C.mmp).

CODE IT IN ASSEMBLY LANGUAGE

The first approach is to use the minimal tools provided by the manufacturer. BBN has implemented the PLURIBUS IMP in this way. A simple assembler was used for all programming. A straight-forward loader was used to transfer code to the machine. And finally, a relatively simple debugging system was used. The debugging system had no multiple-process capabilities. If the amount of software that must be produced is small, this is certainly the quickest approach.

CODE IT IN A HIGH LEVEL LANGUAGE

The main approach used by the implementers of HYDRA for C.mmp was the use of a high level language -- BLISS [Wulf, 1971]. BLISS is an ALGOL-like system implementation language which is available for both the PDP10 and the PDP11. It has an optimizing compiler that produces object code which in some cases is better (more efficient) than code produced by a system programmer using assembly language. For larger software systems, it is desirable to use a high level language. Such a language usually allows for much greater programmer productivity and for systems that are much easier to assimilate and maintain. In conjunction with the high level language, one generally finds more elaborate relocating loaders and debugging packages. The

BLISS debugging package, called SIX12, allows for symbolic debugging of multiple processes, and for source language routine level debugging.

There is little doubt about the success of this approach. Not only is the operating system running, but we have some measurements which show the productivity of the HYDRA software team to be very good, both in terms of number of debugged machine instructions per man-month and in terms of the number of BLISS statements.

COUPLE TO LARGE COMPLETE MACHINE

If a large computer is connected to the multi-mini, much of the software development can be done using the convenient user facilities of the large computer. Cross-assemblers, cross-compilers, elaborate linking loaders, a file system, and simulation packages can all be implemented and used more easily on a large computer than on a mini. The HYDRA development made use of all of these capabilities with a PDP10 that is connected to C.mmp. Thus a large computer greatly simplifies the problem of software development. This is especially true in the early period when the hardware system is still under development and somewhat unstable.

But the key to making use of a coupled large computer is that some of the software can be delayed or avoided altogether by using existing systems on the large computer. For instance, one can take advantage of the file system on the large computer in lieu of one on the multi-mini. If the multi-mini also has a file system, files could be transferred between the two, and the facilities used in the large computer become more directly accessible to the multi-mini. As another example, there are no immediate plans to create a BLISS compiler on C.mmp. Use of a BLISS compiler on the PDP10 appears to be stable and efficient. Ultimately, we wish to explore how to adapt compilers to a multiprocessing environment; but the important point is that such a project is completely off the critical software path.

GOOD-KERNEL HYPOTHESIS

There is an (as yet untested) hypothesis that if one builds the kernel of the operating system correctly, the higher level operating system facilities and application programs become much easier to implement. To some extent, HYDRA is the first test for this hypothesis. If the HYDRA kernel has the right set of basic facilities, building operating systems on top of it should be very simple compared to building the same operating systems on the bare machine.

The implementation of a file system is an example of this phenomonon. HYDRA's Global Symbol Table in conjunction with a simple directory structure allows one to maintain permanant storage of simple objects. In order to build a file system on top of this, one simply needs to define a new object, called "file", which contains the representation of the file. The directed graph structure of the

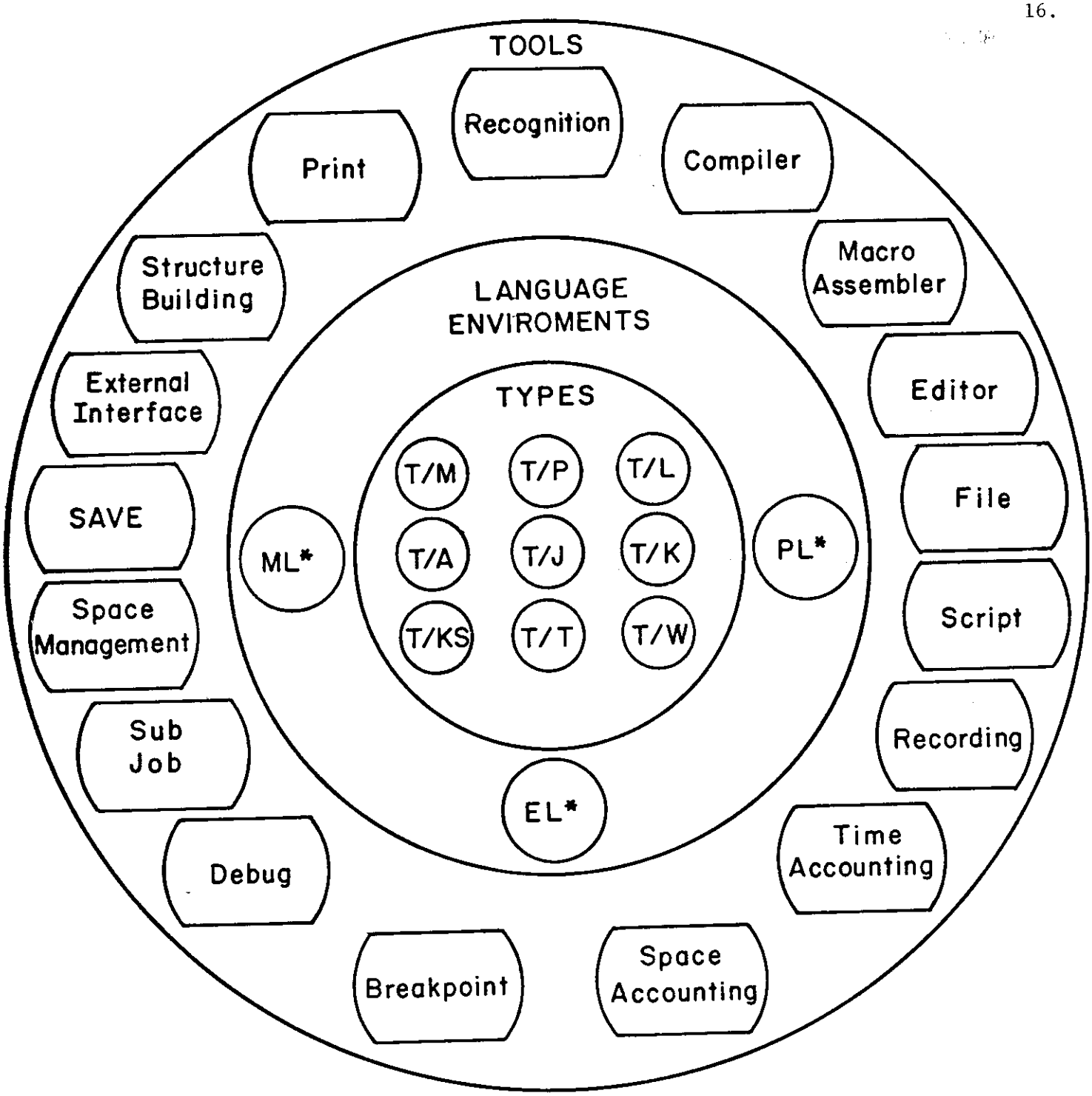


Figure 5: L* Facilites

GST allows for easy construction of hierarchical file structures. To build the equivalent hierarchical file structure on the bare machine is a very time consuming project.

INTERACTIVE SYMBOLIC IMPLEMENTATION SYSTEM

Another approach is to use an implementation system that provides a total operating environment within which one can build the application system interactively. We have been experimenting with such a system, called L☆ [Newell, 1971], for several years. Figure 5 illustrates the kinds of facilities that are available to an L☆ user. The system has a kernel which provides an initial set of types and facilities for manipulating each type. It also provides three initial languages which are used to construct a large set of user facilities (e.g., editors, debuggers, compiler, assembler). The major features of L☆ that make it attractive are:

1) L☆ is a total environment so that one need not rely on other systems to produce the required software.
2) L☆ is completely accessible to the user to modify and adapt to his own needs.
3) The surface syntax of the language is easily modified by the user.
4) The system is interactive and built around a symbol manipulation language.

One has much the same feeling as in a system like interactive LISP, even though one is working on implementing extremely basic systems programming features.

We performed an experiment with L☆ on C.mmp during April, 1974 [Robertson, 1974]. A small team of programmers (varying between one and three) was given the task of building a speech pre-processor that would take input from a real-time audio spectrum analyzer, sample the data, segment it into phonemes, and label the segments; the result to be displayed in real-time on a graphics display processor (see Figure 6). The demonstration was intended to graphically illustrate the speed-up when more processors were added to the multiprocessor system.

The experiment lasted for one month. We initially wanted to run the L☆ system under HYDRA, but were unable to because HYDRA was not yet ready for end users. We constructed a stand-alone L☆ system for C.mmp and within that system build the speech demonstation. We designed an interface between L☆ and BLISS so that the sampler, segmenter, and labeler could be written in BLISS. We also built a specialized multiprocessing system which had the unusual characteristic that it was not also a multiprogramming system. The end result of the experiment was a demonstratable multiprocess speech pre-processor. The important point is that a large amount of software was produced in a short period of time by capitalizing on the advantages of a good interactive symbolic implementation system.

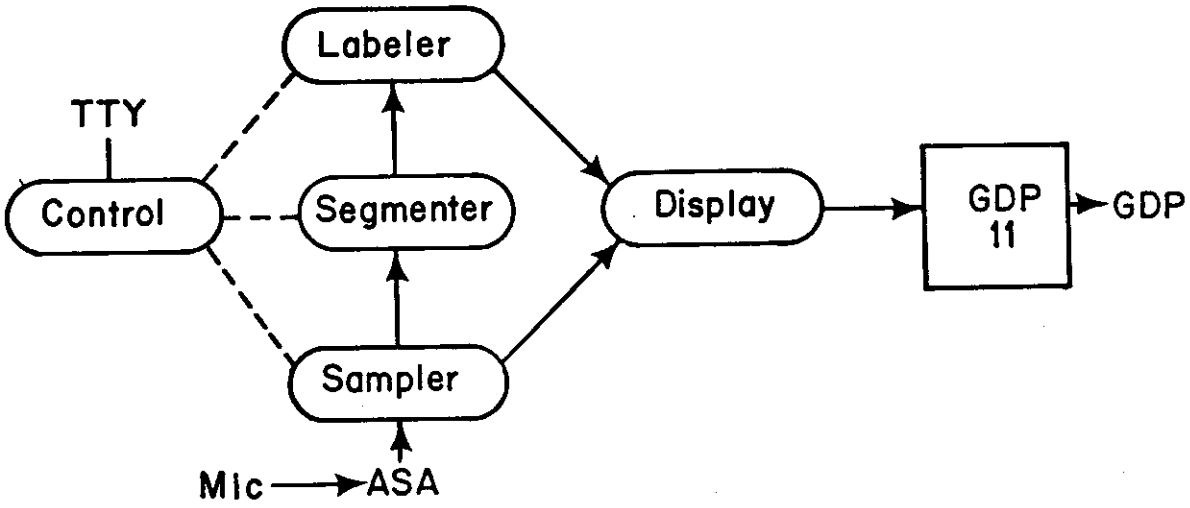


Figure 6: April-74 Software Experiment

CONCLUSION - SOFTWARE PRODUCTION

All the various approaches to producing requisite software on a multi-mini-processor are important. For each approach there is a set of software tasks where it is better than any of the other approaches. It is thus important that the multi-mini environment be rich enough to allow the use of any or all of these approaches.

ISSUE 3: SMALL-ADDRESS PROBLEM

The third major issue is how to take advantage of the large memory in a multi-mini with the small physical address space of a mini. On C.mmp, for instance, each processor can directly address only 32K words while the primary memory capacity of the system is one million words. Anyone who wants to grow programs larger than 32K is forced to make serious design decisions about his programming effort. Furthermore, since the large memory is available, large applications will be attracted to the multi-mini and the problem will occur frequently. A large number of solutions have been suggested and attempted. For small or well-behaved systems, there are solutions that work well. However, for a large, dynamic programming system, all the solutions considered so far tend to be either restrictive or costly in overhead.

STATIC OVERLAYS

The most successful solution to this problem has been to overlay pages using relocation registers (Dmap on C.mmp). HYDRA and the latest version of L* both use this solution. The overlays must always be in the same place to avoid address translation. If the overlays contain only data, as in large arrays, there is no address translation problem. However, most large programming systems have data which contains many address references, and these must either be overlayed statically or incur address translation costs. In any case, code pages always face the problem. This solution seems to work well, particularly for small or well-behaved systems.

However, there are cases in L* and in some large programming systems where code in an overlay attempts to access data or code in another overlay which must reside in the same physical addresses. Static overlays prevent this from working correctly. The other problem with this solution is the overhead incurred in changing relocation registers. Large systems are likely to want to change relocation registers frequently, thus making the overhead critical.

POSITION INDEPENDENT OVERLAYS

Another solution similar to the previous one is to use overlays that contain position independent code and data. This frees you from the restriction of always bringing the overlay into the same physical location. The primary problem with this solution is that position independent code and data tend to take more space, execute more slowly, and be harder to produce. The solution assumes that the underlying processor in the multi-mini allows for relatively low cost position independent code (as does the PDP11).

SOFTWARE DEMAND-PAGING

Another kind of solution is to provide software demand-paging. This involves accessing data with "fat addresses",

e.g., 32 bits of effective address that determine a page and an offset within that page. It also involves making any code that crosses page boundaries go through both demand-paging and address translation. We tried this solution in an early version of L☆. This solution takes more space and has some position independent code which is harder to produce. It also suffers from severe overheads for the demand-paging and address translation. Its great virtue is that it provides a truely general system; one where the entire large memory is directly addressable.

MIXED SOLUTIONS

The most promising approach appears to involve mixing several of the previous solutions. There are two basic ways to do this. One method would start with a software demand-paging system (fat addresses) and allow for some pages to be specialized (i.e., have small addresses for efficiency). The other method would start with a static overlay structure and build up mechanisms to allow for arbitrary virtual addresses (pseudo-fat addresses). We are currently working with the latter of these methods. We have constructed the latest L☆ system with a simple static overlay structure. We are now in the process of designing the mechanisms necessary for certain facilities (e.g., compiler) to be able to access arbitrary virtual addresses. The main problem now is to determine whether a large application system like HEARSAY-2 can be decomposed into small overlays that don't need direct access to each other except on rare occasions. It remains to be seen whether this solution is adequate for large systems.

ISSUE 4: PROTECTION

The forth major issue is protection. The style and amount of protection in a multi-mini operating system depends greatly on which strategy is chosen for exploiting the multi-mini.

NO PROTECTION

The PLURIBUS IMP represents one end of the spectrum. The decision to build a specialized system for a specialized multi-mini led BBN to provide no protection at all. The assumption here is that there is only one application system running and it was built by a small, closely-knit programming group. Every module in the system must be aware of other modules and their conventions so that they don't get in each others way. There is also an assumption that the system is small enough to be easily debugged. A module with a bug could accidentally destroy another module. If there are too many modules (or they are too complex), it may be very difficult to find the incorrect module. This solution generally works well only for small and well understood applications.

AUTHORITY-BASED PROTECTION

At the other end of the spectrum, the decision to allow many (unknown) applications to run, with program development occurring simultaneously, leads to great concern with protection, both between users and between various processes being run by one user. Protection can be viewed as a central issue of operating systems; i.e., the control of resources, the distribution of the rights to use these resources to various processes on a moment-to-moment basis, and the guaranteeing of these rights. Most of the first and second generation operating systems, such as the existing DEC systems (TOPSTEN and TENEX) and the IBM OS360, are so-called authority-based systems. In these systems, protection is associated with the data and not with the processes accessing the data. This tends toward crude categorization of protection (e.g., the familiar read/write/read-write distinction). There are currently no multi-mini systems that use authority-based protection, although it is clearly the alternative that would have been used a few years ago.

CAPABILITY-BASED PROTECTION

HYDRA is a capability-based system, which means that it associates rights to use resources with individual processes and not with individual data. Capability systems permit much finer distribution of rights, essentially on arbitrary processes. We are only just beginning to see capability-based operating systems, and this aspect of HYDRA represents an independent research effort.

HYDRA has an abstract view of the entities that can be protected and the rights to manipulate these entities. Because of this, it is possible to build higher level protection into

specialized subsystems. This is another aspect of the HYDRA design that reflects the basic exploitation strategy of multiple specialized application systems.

There is a subtle disadvantage with capability-based systems that we are learning the hard way. You generally must do much planning in a session to insure that you will have all of the capabilities you need. If your program has a strange bug and you don't have the proper rights or capabilities, you may not be able to explore the bug. At this point, we lack the experience with HYDRA to know whether the advantages of such a protection system outweigh the disadvantages or not.

Generally, there is a computing cost associated with protection and the more protection, the higher the cost. This leads the user of an over-protected system to find ways of avoiding the protection mechanisms. However, with an under-protected system, the user tends to lose much work when something that belongs to him is destroyed by someone else. He also tends to lose time trying to debug complex systems when the various parts of the systems are not protected from each other. Finding the correct balance of protection is both important and difficult, and we expect this issue to become more visible as C.mmp gets more and more use.

ISSUE 5: TIME-CONSTANT PROBLEM

The fifth major issue has to do with the time constants for basic functions that must be performed on any multi-mini-processor. This is actually a class of problems, one for each application system against the pattern of basic time constants. For this reason we cannot enumerate general alternatives, but must select illustrative issues that arise for particular examples. The important point is that the time constants have an immense influence on programming style and system design.

Consider two basic ways of building a large programming system: 1) have one process that has many overlays and does a great deal of relocation register changing; or 2) have many small processes that communicate with inter-process communication and don't ever change relocation registers. In HYDRA, relocation register changes are about an order of magnitude faster than inter-process communication, so the correct choice is the first way. Functionally, many intercommunicating processes may be the preferred way to organize the system, but the time constants preclude it. The time constants may have more impact on design decisions than the functional characteristics of the operating system.

Another example of this problem has to do with prevention of deadlocks, a pervasive problem in all multiprocessor systems. The HEARSAY system wishes to have a large data base shared between many processes. In order to prevent a process from having the data it is working with changed by another process, semaphores are used to build locking structures around relatively small pieces of code. (Semaphores are a standard device to avoid interference between processes; they are flags that indicate whether an object, e.g., a piece of code or a piece of data, is in use by another process.) The problem is that the operations on semaphores that HYDRA provides are much too slow relative to the frequency of use and size of code they are locking. Because of the time constant, we had to build another level of semaphore that would only make use of the HYDRA semaphore on rare occasions. This is an example of a functional capability that could not be used because of the time constant problem.

Another place where the time constants become critical is in real-time applications. The basic functions like context swap, relocation register change, inter-process communication, and interrupt handling take much more real time on a mini than on a large computer. The difference can be attributed to the differences in the raw processing power and in the complexity of instruction sets. When these differences are taken into account, the relative overhead on a mini and a large computer are about the same. However, the real-time overhead becomes critical when doing real-time computation, or when minimizing terminal response time.

The time-constant problem thus comes down to understanding the time constants and their relationships with respect to a given application system. This understanding is necessary if the application system is to make effective use of the multi-mini. The problem exists in all computer systems, of course. But it is much

more critical in systems with highly flexible and general operating systems (such as HYDRA). Such operating systems provide functional capabilities of great power and elegance, but the time constants often deny their use. The situation is especially critical in multiprocessors where exploitation of the system requires working with many processors in some coordinated scheme. This can only be done by working through the operating system. It is almost impossible in a multiprocessor to avoid the time-constant problem by withdrawing to your own world to avoid interaction with the operating system.

ISSUE 6: RELIABILITY

The sixth major issue is total system reliability. Multi-mini-processors are complex, and much can go wrong in both hardware and software. Also, the hardware that provides for multiprocessing provides redundency, which if exploited can permit more flexibility in recovering from hardware failures. Because of these factors, reliability plays an important role in all multi-minis. There are several known approaches to the reliability problem.

CRASH AND DUMP

The most common approach in existing large computer operating systems is to bring the system to a grinding halt when a failure is detected. The system is usually dumped at this point so that system maintainers can try to determine what caused the failure. Then a fresh copy of the system is brought up. The obvious flaw with this strategy is that all users lose their current run, even if the failure would not have otherwise affected them. This approach is slowly disappearing as more experience is gained with smooth recovery from failures.

AMPUTATION AND EXTERNAL BACKUP

The PLURIBUS IMP stresses reliability as its most important attribute. Their system is highly modular and redundant. Every structure in hardware and software is isolated and duplicated. The system makes periodic validity checks and amputates any structure that appears suspicious. If the amputation causes some data to be lost, an external backup provides the data to be dealt with again. The interaction between an IMP and the ARPA Net involves much handshaking. When data is accepted by an IMP, it acknowledges the reception. If no acknowledgement is received within a certain time frame, the data is sent again. In this way, data is distributed across all IMPs in the network. Thus, the specialized nature of the application, in this case the ARPA Net, provides an external backup for lost data, no matter what the cause for the loss. This permits good solutions to the local reliability problem.

The reliability of the PLURIBUS IMP is so high that the first time the system was ever brought up they discovered that the only way to stop it was to pull the plug on the whole system. Since then, their system has grown to be one of the most reliable known to us.

RECOVERY BY RECONSTRUCTION

The nature of reliability on C.mmp and HYDRA is quite different but still very important. The stress in HYDRA is on recovery after a failure has been detected. C.mmp does not have the kind of backup that PLURIBUS IMP has with the ARPA Net. The method in

HYDRA is to maintain a global symbol table (GST) which contains information about every structure in the system. The GST is maintained so that any destroyed structured can be recreated, including parts of the GST. To detect failures, the hardware has been modified to do parity checking and the software maintains checksums of all critical structures. In addition, whenever an error is detected while running a user system, the error information is passed back to that system. Thus, the end user can build reliable application systems. HYDRA's reliability is still under research and its success has not been fully determined.

PARTITIONED SYSTEMS

Another aspect of reliability in C.mmp is the ability to partition the system into several smaller systems. This allows concurrent system development, general user facility use, and hardware maintance and development. The PLURIBUS IMP can also be partitioned, but only by recabling. The C.mmp partitioning is controlled by switches and is changed on very short notice (a couple of minutes). This ability is being used on a day-to-day basis to aid is selecting a stable configuration of C.mmp. We also use it frequently to allow several groups to work independently.

ISSUE 7: PERFORMANCE EVALUATION

The last major issue is how to analyze and evaluate the performance of running systems on a multi-mini-processor. This issue is perhaps the least understood of all of the issues. Programmers are notoriously wrong in guessing what their programs are actually doing and where the time is really going. There is reason to believe that on a multi-mini, the problem is going to be much, much worse. The decomposition of algorithms to take advantage of parallel processing is currently a rich research field. Imagine how difficult it will be to determine the dynamic characteristics of several cooperating parallel processes.

Traditionally, the analysis of performance of a computer system or a program is undertaken as a study. Often this study is primarily of academic interest, though sometimes with a view to balancing the computer system or making the algorithms run more efficiently. However, we believe that for multiprocessors there will be a major shift in emphasis of performance evaluation from analysis tools to operational tools. They will become as important to a multiprocessor user as the traditional debugging tools.

The solution to the problem on C.mmp has been to start a research project on a hardware device, called a hardware monitor [Fuller, 1973], which will allow us to measure specified kinds of activity on one processor's bus. This device, used in close conjunction with software in HYDRA, should give the user a chance of obtaining the dynamic job statistics he needs to analyze the performance of his programs. We also hope to use the device to help understand the real performance characteristics of HYDRA in order to improve system performance.

An example within HYDRA illustrates the use of the hardware monitor. We have a real-time device that connects C.mmp to the PDP10. We recently discovered that characters were occasionally being lost, presumably because HYDRA was running blind to interrupts for too long. We were able to verify this with an oscilloscope. However, we have not been able to find the code in HYDRA responsible for the excess blind time. We expect the hardware monitor to be able to isolate the offending code. The important point is that a multi-mini is so complex that new techniques must be developed to aid in performance evaluation.

CONCLUSION

Though other programming issues could be discussed, seven is all anyone should be called upon to remember. Let us sum up.

We believe that multi-mini-processors such as C.mmp and PLURIBUS IMP will come to provide a substantial amount of computational power. Although the technical capability for creating multiprocessors has existed for quite awhile, only with the development of the minicomputer (and now the microprocessor) has the cost-benefit structure pointed to multiprocessors as an important technical solution. As a result we know almost nothing at this point about the actual programming and use of genuine multiprocessors, i.e., those where the multiprocessor structure is sufficiently general and available to affect the structure of application systems.

Several of the issues we have discussed in our list, e.g., how to get all the software, protection, reliability, and performance analysis, are well recognized problems and are subject to intensive independent research. The work with multiprocessors gives them a new twist, however, raising to consciousness aspects that are of little interest in other kinds of systems. Though still speculation on our part, performance analysis as a real-time dynamic debugging tool represents a new world.

Two items on our list, the small-address problem and the time-constant problem, do not represent areas that are well explored in computer science. We have seen no solutions to the underlying programming issues in the literature. Both items seem critical and worthy of considerable attention.

The small-address problem seems inherent in mulitprocessors built with mini- or micro-computers. Possibly the problem will be solved by avoiding it. Some new minis are appearing on the market now with large physical address spaces but maintaining the other attributes of a mini. However, a large address requires many bits, both in memory to retain it and in bandwidth to communicate it.

We might point out to psychologists that the problem is in essence faced by a population of intercommunicating humans. No one has internal symbols (i.e., addresses) designating all the things that all individuals designate internally. That is, they do without large addresses in the hardware. Instead they use language, which is a set of software-maintained large addresses, for their intercommunication. They continue to think their private thoughts in separate representational worlds. Thus the problem of communicating with small addresses is a fundamental one not restricted to the world of multi-mini-processors.

The time-constant problem seems critical if we are to make effective use of multiprocessor architectures. We must understand what various patterns of relative and absolute time constants imply for the processing systems built on top of them. Only then can we design multiprocessors with a balance between their

functional capabilities and the dynamic capabilities (i.e., the speed with which they carry out various functions). What can be done within a system which has already been designed is still quite unclear. Some overheads, such as swapping time, are built into the hardware. Others, such as protection, may be subject to ingenious trade-offs between flexibility and computing cost (for checking protection). For example, one can think of compiling out restricted protection schemes so that a minimum of checking has to be done dynamically.

We have attempted to expose a set of programming issues that we have encountered in beginning to use a multi-mini-processor. We confess our fundamental ignorance of the correct formulations -- to say nothing of the correct solutions -- for most of these problems in this new environment. Perhaps these will no longer look like the important problems after we obtain more experience. That experience is now enveloping us day after day.

[Fuller, 1973]
S.H. Fuller, R.J. Swan, and W.A. Wulf
The Instrumentation of C.mmp: A Multi-Mini-Processor
Proceedings of COMPCON 73, New York, N.Y., March 1973,
pp. 173-176

[Heart, 1973]
F.E. Heart, S.M. Ornstein, W.R. Crowther, and W.B. Barker
A New Minicomputer/Multiprocessor for the ARPA Network
Proceedings of the National Computer Conference, 1973,
pp. 529-537

[Lesser, 1974]
V.R. Lesser, R.D. Fennell, L.D. Erman, and D.R. Reddy
Organization of the HEARSAY II Speech Understanding System
Proceedings of the IEEE Symposium on Speech Recognition,
April 1974, pp. 11-21

[Newell, 1971]
A. Newell, P. Freeman, D. McCracken, and G. Robertson
The Kernel Approach to Building Software Systems
1970-71 Computer Science Research Review
Cargegie-Mellon Univ.

[Newell, 1973]
A. Newell, J. Barnett, J. Forgie, C. Green, D. Klatt,
J.C.R. Licklider, J. Munson, R. Reddy, and W. Woods
Speech Understanding Systems: Final Report of a Study Group
Pub. by North-Holland, 1973

[Robertson, 1974]
G. Robertson, A. Newell, and D. McCracken
On Doing Software Experiments
1973-74 Computer Science Research Review
Carnegie-Mellon Univ.

[Wulf, 1971]
W.A. Wulf, A.N. Habermann, and D. Russell
BLISS: A Language for Systems Programming
Communications of the ACM, December 1971
See also: "BLISS-11 Programmer's Manual", DEC, December 1972

[Wulf, 1972]
W.A. Wulf, and C.G. Bell
C.mmp -- A Multi-Mini-Processor
Proceedings AFIPS 1972, FJCC. Vol. 41, AFIPS Press,
pp. 765-777

[Wulf, 1974]
W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin,
C. Pierson, and F. Pollack
HYDRA: The Kernel of a Multiprocessor Operating System
Communications of the ACM, June 1974, pp. 337-345