Pepe

Pepe architecture—Present and future*

by C. R. VICK
U.S. Army Ballistic Missile Defence Advanced Technology Center
and
JOHN A. CORNELL
System Development Corporation
Huntsville, Alabama

INTRODUCTION

PEPE (Parallel Element Processing Ensemble) was designed, developed, and produced for the U.S. Army Ballistic Missile Defense Advanced Technology Center (BMDATC) for research on Ballistic Missile Defense (BMD) systems. With its host computer, the CDC 7600, it is potentially the world's most powerful computer system. PEPE employs parallel processing and associative data-access techniques, and while it can operate in a stand alone mode, it is designed primarily to augment more conventional computers in BMD service. Potential applications besides BMD for which the machine is adapted include weather forecasting, air traffic control, image data processing, signal processing, and other applications exhibiting inherent parallelism and requiring extensive computational power.

The PEPE system can be considered simply as a large master computer, called a host, controlling many smaller slave processors, called elements. In the present design, the host is a CDC 7600, and there are 288 elements. Each element comprises three processors sharing a common data memory. One of these processors, called a correlation unit, is used for inputting data and has an instruction repertoire especially suited for the rapid association of new data with data already on file. The second processor, called the arithmetic unit, has a repertoire similar to that featured in conventional high-power general-purpose machines; i.e., fixed and floating point arithmetic operations, load and store, and logical operations. The third processor, called the associative output unit, is used for ordering and outputting data and is especially designed to perform complex, multidimensional file searches rapidly and efficiently. Each of the three processors is driven by its own control unit, which simultaneously drives all of the corresponding processors in the ensemble of elements. The three control units are also capable of executing their own sequential programs. They are combined into a control console, which drives the ensemble of elements in parallel and interfaces the ensemble with the host. The complete PEPE/Host system, then, is a multiprocessor employing seven processor types in all (host, three sequential processors, and three parallel processors). All seven processor types are capable of simultaneous, overlapped operation.

Software for the PEPE includes the compilers and assemblers for the seven PEPE processors and a linkage editor for binding programs into executable load modules. The entire machine can be programmed in a single language called PFOR, which is a superset of FORTRAN. System software also includes an instruction-level simulator for PEPE, a general-purpose real-time operating system, and a general utilities package.

The PEPE program was a complete system effort, requiring problem analysis; generation of a hierarchy of system, functional, and implementation specifications; hardware design, development, production, and test; system support software design, development, production and test; tactical/problem software design, development, and test; and integration into a total real-time systems environment employing other computers, missiles, radars, and command, communications, and control networks. All of these activities were carried on concurrently. Moreover, much of the hardware architecture and programming techniques had never before been attempted on such a large scale for real-time service. It was obvious, therefore, that conventional procedures for executing a development program of this magnitude and in particular developing the PEPE architecture would not be adequate. Such procedures, employing the traditional sequence of problem analysis, design, hardware development, hardware fabrication and test, program design, development, production and test, and finally system test and validation, often result in major system errors being discovered in the last phase of the sequence when it is too late to do much about them. Because of the complexity of the PEPE development program, major system design errors would almost certainly occur (under the conventional pro-

* This work was supported by the U.S. Army Advanced Ballistic Missile Defense Agency, Huntsville, Alabama, under Contract DAHC60-73-C-0060. Portions of this paper were published in the 1976 Infotech Information Report on Multiprocessors.

cedures) and would propagate through the various stages of the program to be discovered in the final stage, system validation. To avoid the high risk inherent in the conventional procedures, PEPE system designers decided to employ a top-down design approach in which system validation started early and continued throughout the entire program. Thus, the program would be executed in a logical progression of *validated* baseline steps. The validation would be accomplished through a combination of functional and analytic simulations. Early in the project, the architecture and the operating system were validated via functional simulations which modeled the operation of the PEPE in a BMD environment employing a variety of attack scenarios. Further functional simulations of more complex environments employing PEPE in the National BMD Site Defense System were completed during the first half of 1975. A 36-element version of the PEPE hardware was delivered to the BMDATC Advanced Research Center in Huntsville in April 1976. Analytic simulations, employing the instruction-level hardware simulator and the PEPE hardware itself, were conducted during the last three years.

BMD DATA PROCESSING PROBLEM

Figure 1 portrays a simplified subset of the BMD data processing problem; only that part of the total problem associated with detecting, tracking, and classifying targets in the field of view of the radar is considered in this paper. The radar generates a pencil beam capable of being pointed, within a few microseconds, in any direction within the surveillance volume. Beams are either transmit or receive beams; each transmit beam can generate any one of a number of different pulse configurations (carrier frequency, number of pulses, pulse coding, frequency modulation, pulse length, etc., can be varied) and each receive beam can likewise assume any one of a number of configurations (carrier frequency, range extent, matched filter, etc., can be varied). Typically, the radar generates several thousand transmit beams per second and a proportionate number of receive beams. Generally, several hundred transmit and receive beam pairs are reserved for searching the upper part of the surveillance volume; these are generated in a raster scan pattern designed so that no object can penetrate the volume without being detected by at least one search beam. Interleaved at random in both time and space among the search beams, as required by the target environment and battle situation, are a variety of special beams for verifying search detections, precision tracking of previously detected and acquired targets, transmitting guidance commands to interceptors, target discrimination and classification, and several other functions. It is the job of the computer to initiate a file on each detected object, associate each radar return with its proper file, employ the information in each return to update the file, perform complex mathematical functions on the files, and generate requests for additional radar information. Each request becomes part of the file on a particular target. Periodically, the files are searched in some sort of priority fashion to generate orders for radar pulses. Details on each pulse are formulated individually by the computer based on the requests in the target files; those requests which are granted are transmitted to the radar in the form of orders. The objective is to select, for each pulse, that order which will be of most benefit to the defense system at that time. The radar transmits the ordered pulses, new returns are thereby obtained, and the radar computer loop is closed. The foregoing functions must all be performed within rather severe time deadlines. The BMD data processing problem, then, can be described as keeping a large, complex data base updated in real time. Moreover, the means for acquiring the information required for updating is under control of the computer.

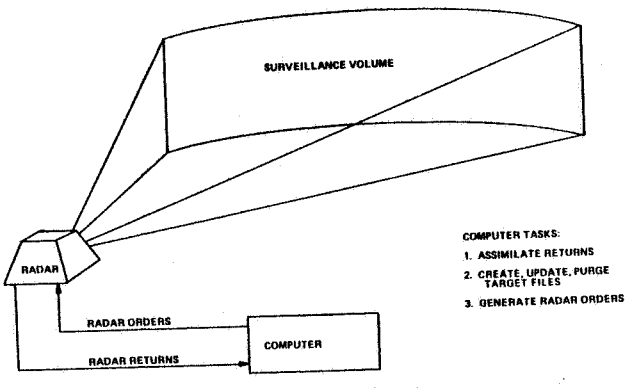

Figure 1—Partial BMD system

Considering the above rather simplified description of a representative part of the BMD data processing problem, one can deduce the types of processing required, and then synthesize a machine architecture to satisfy the requirements; this was done in the case of the PEPE design.

First, means must be provided to associate verified radar search returns with target data already on file in computer memory. Every such return must be compared with all target data on file. Each comparison, or association operation, requires a forward prediction of all filed target positions to the time of the return, calculation of three-dimensional gates (gate dimensions may be more or less than 3, depending on correlation criteria) about the positions, and a match of the radar return against all gate volumes.

A conventional sequential computer must execute the foregoing operations as an $m \times n$ number of operations, where m is the number of new returns in a given time interval and n is the number of targets on file. This means that the data processing resources required for correlation of new returns is an exponential function of the target traffic. Since correlation is one of the major consumers of sequential computer resources in BMD service, an architecture that performs correlation more efficiently is indicated. Associative processors perform the correlation function as an $m \times 1$ number of operations, independent of the number of targets on file. Therefore, a computer architecture for BMD service could advantageously possess associative data input characteristics.

Second, lengthy scientific computations must be made on a large number of separate, independent data sets, as in Kalman filtering of radar returns. Computations are identical

for all targets, so that considerable leverage could be obtained by assigning targets to separate, dedicated execution units operating on data in dedicated memories, all driven simultaneously from a single control unit executing a single program. Therefore, a computer architecture for BMD service could advantageously possess parallel processing characteristics.

Third, multidimensional, prioritized file searches must be made through target files for radar pulse requests, so that the radar pulses can be scheduled efficiently, in observance of a variety of constraints, and in accordance with the immediate needs of the instantaneous total battle environment. Such searches are made inefficiently in conventionally addressed memories, but very efficiently in associative memories. Therefore, a computer architecture for BMD service could advantageously employ associative data retrieval and output characteristics. Moreover, associative data retrieval would be of great assistance in one of the most critical and resource-consuming functions which must be performed in a total BMD system, a function largely ignored in this paper. The function is that of real-time control, or assigning resources and adjusting algorithms, procedures, function execution rates, and overall system behavior dynamically and optimally in accordance with instantaneous system demands, status, and defense strategy. This function, because it is sequential and decision-making in general, is best handled on a conventional sequential computer. However, the complex data interrogation functions which are necessary to efficient execution of the real-time-control task are greatly simplified when the sequential computer has associative access to the dynamic system data base.

In addition to the foregoing three primary BMD data processing requirements of correlation, scientific computation, and multidimensional file search (which are incidentally found to a similar degree in other problems), there are additional requirements common to military computers in general and to BMD computers in particular. First, the computer must be extremely reliable, and this reliability should preferably be an inherent characteristic of the architecture. High reliability is difficult to achieve in a computer suitable for BMD service, because of the very large amount of hardware such computers must contain. The latter is true because, in the final analysis, great computational power (estimates for BMD requirements run from tens to hundreds of millions of instructions per second) is achieved only via the employment of large amounts of CPU hardware. An architecture acceptable for BMD service must allow high reliability despite the large amount of hardware required. Parallel associative architectures provide opportunities for meeting this requirement, so long as the individual elements in the parallel array can be kept independent of one another. Then, individual elements can fail without affecting others, and without affecting the problem solution. Fortunately, many of the data sets dealt with in BMD computations are themselves independent, and lend themselves to assignment to independent processing elements. Since the data sets are independent, there is no need for interelement communication. This permits the arrangement of processing elements to be almost completely unstructured, so that no particular one or combination of elements is needed for successful problem completion.

A final architectural requirement, important for BMD applications, is that the data processing system be capable of easy expansion to meet increased requirements. A parallel, associative architecture, where the number and organizational structure of processing elements can be arbitrary, lends itself to this requirement too. Additional elements could be added to handle additional target traffic, and all of the programs could be written to be independent of the number and/or location of processing elements, so that programs need not be modified to accommodate hardware expansion.

Summarizing, the foregoing requirements led to consideration of a parallel, associative architecture. However, such computers which have been designed and/or built in the past fall generally into two categories, neither of which are really optimized for BMD service. First, past and contemporary associative processors are characterized by very primitive per-element computational capability (usually bit-serial); they achieve high processing speed by virtue of keeping a very large number of elements (say thousands) busy simultaneously. Computers for BMD service would not require the extremely large number of elements possible in such machines, but they do require fairly powerful elements because of the extensive per-target scientific calculations required. The second category is exemplified by the ILLIAC IV, which certainly has sufficient per-element computation capability, but an insufficient number of elements (tens), assuming that one wants to assign one target to an element (this is not really necessary, but not to do so leads to other complications and is beyond the scope of this paper). Moreover, ILLIAC IV contains features such as interelement communication which are not essential for BMD service, but which contribute to lower reliability because of the more or less rigid structure imposed upon the array by the intercommunication facilities. It would seem then, that the parallel associative architecture specified for BMD service should have capabilities somewhere between those offered by associative processors and ILLIAC IV. It should have a moderately large number of elements (hundreds), and each element should have a fairly powerful scientific computation capability. Moreover, each element should comprise three execution units, one each for input data correlation, parallel scientific computation, and associative file search for data output. To maximize throughput, all three execution units in the elements should be capable of simultaneous operation. The elements should have no positional significance; i.e., they should be organized into a completely unstructured array—an ensemble. Interelement communication and local indexing of data, while of considerable value in parallel computers designed to operate on interdependent data sets, can be eliminated because they are not needed and in fact are not even desirable. Finally and perhaps most important, the total data processing system should have high throughput in sophisticated branching and decision-making operations, but these operations should not reduce throughput on the less complex but resource-consuming parallel data processing functions. This means that the total data processing

system should have separate but closely cooperating facilities for both sequential and parallel processing tasks. The architecture which evolved from the foregoing considerations, PEPE, is described in the following section.

PEPE ARCHITECTURAL OVERVIEW

A system block diagram of the PEPE is shown in Figure 2. The Host, a CDC 7600, is connected to the three PEPE Control Units (collectively, the Control Console) via three standard CDC 7600 MUX (Input/Output Multiplexor) channels, one for each Control Unit. The CDC 7600 then sees the PEPE as three independent PPUs (Peripheral Processing Units). Each Control Unit is equipped with a modular Host Interface Unit; by replacing Interface Units, other interface connections with different Host machines are possible. The Host provides overall executive control, through its operating system, of the Host-PEPE configuration. It loads program and initial data into the PEPE memories, schedules and dispatches appropriate tasks to the PEPE, executes those sequential tasks which it does more efficiently than the PEPE, and executes utility functions such as compiling PEPE programs and reading and writing to peripheral devices.

The Control Console contains three independent Control Units (See Figure 3); they are similar but the Correlation Control Unit is optimized for inputting data to the elements, the Arithmetic Control Unit is optimized for scientific calculation in the elements, and the Associative Output Control Unit is optimized for outputting data from the elements. Each Control Unit is a multiprocessor which splits one instruction stream into two parts, one sequential which is executed within the Control Unit, and one parallel which is executed in the ensemble of elements.

Arithmetic control unit and parallel arithmetic units

The Arithmetic Control Unit is shown in block-diagram form in Figure 4. This unit contains a 32,768-word, 32 bit-

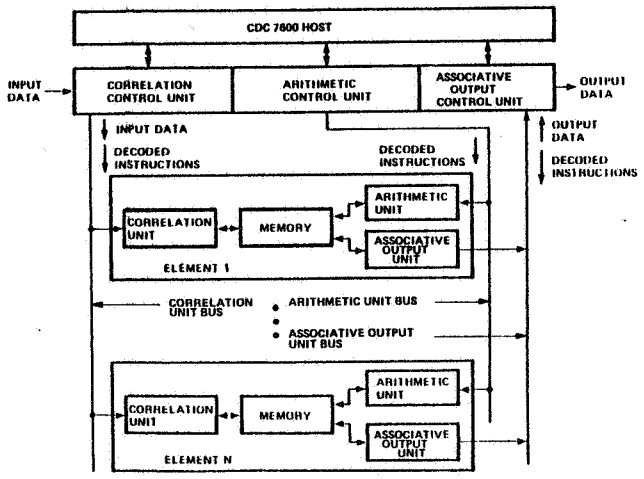

Figure 2—PEPE architecture

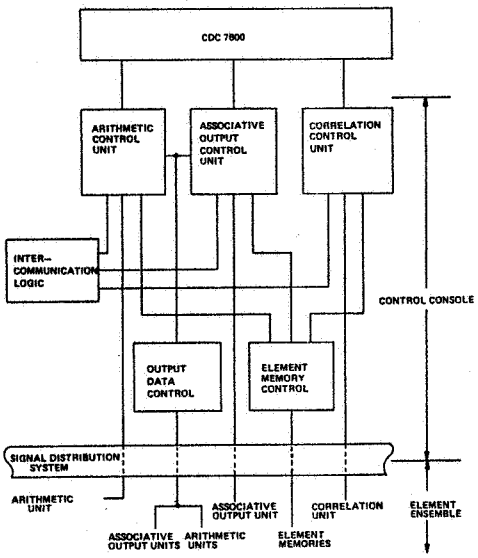

Figure 3—PEPE control console

per-word, random access semiconductor memory with a cycle time of 200 ns. A separate data memory contains 2048 random-access 32-bit words, implemented in ECL with a cycle time of 100 ns. The Sequential Control Logic (SCL) is a fairly conventional processor containing the usual A (Accumulator), B (operand), and Q (quotient/product) registers, plus 15 index registers. It executes integer (24-bit) arithmetic, and logical and branch instructions fetched from program memory on data obtained from data memory. Program memory contains a mixture of sequential and parallel instructions. The SCL executes the sequential instructions itself, but passes the parallel instructions, via the Parallel Instruction Queue (a conventional instruction stack) to the Parallel Instruction Control Unit (PICU). This unit decodes

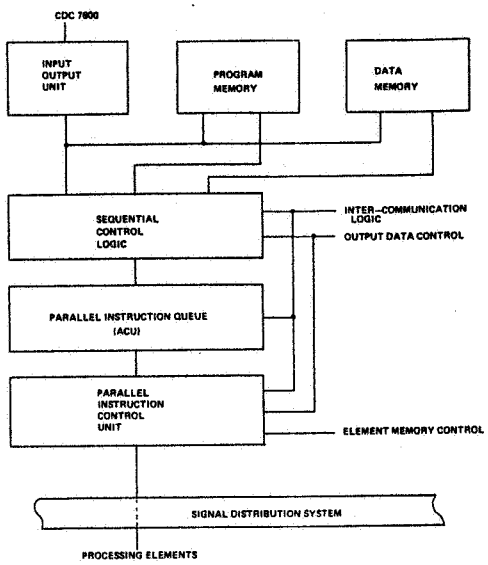

Figure 4—PEPE arithmetic control unit

the parallel instructions and broadcasts the resulting microoperation signals to all of the Arithmetic Units in the ensemble. The Arithmetic Units, provided they are active, all execute the same instructions simultaneously on data obtained from their own local data memories. The parallel instruction set is quite rich, containing the conventional arithmetic operations (plus square root) in both integer and floating point, load, store, and logical operations. In addition, the instruction set contains associative instructions which activate and deactivate elements based on element accumulator content and/or element tag register content. Only active elements execute instructions. Each Arithmetic Unit can execute about one million instructions per second, based on averages over instruction mixes encountered in typical scientific calculations.

Correlation control unit and parallel correlation units

The Correlation Control Unit (CCU) is similar to the Arithmetic Control Unit, but has a smaller, faster program memory and no Parallel Instruction Queue. Moreover, its PICU does not include a floating-point arithmetic capability, nor can it execute sequential floating-point operations. The CCU broadcasts a sequential stream of input data from the Host, or some other source, to all elements, and enters words from the input stream into properly activated elements. The Correlation Control Unit parallel instruction set is rich in associative match and compare operations, so that incoming data can be rapidly and efficiently correlated with data already in element memory, and thus be placed in the proper elements. Instructions are fetched from the CCU program memory and decoded, then the resulting microoperation signals are broadcast to the Correlation Units of all elements. The Correlation Units can be activated and deactivated independently of the element Arithmetic Units; only active Correlation Units execute instructions.

Each Correlation Unit contains a B register and 16 Correlation Registers which support execution of integer and logical operations, plus associative match and comparison operations among entering data and data retrieved from element memory. The Correlation Unit shares element memory with the Arithmetic Unit and the Associative Output Unit. Since the Correlation Unit does not perform floating point operations, its average instruction execution rate is about five million instructions per second, much faster than that of the Arithmetic Unit.

Associative output control unit and parallel associative output units

The Associative Output Control Unit (AOCU) is similar to the Arithmetic Control Unit, but, like the CCU, has a smaller, faster program memory, no Parallel Instruction Queue, and no capability for sequential or parallel floating-point arithmetic operations. Its function is to extract data sequentially from properly activated element memories, based on associative search instruction sequences which it fetches from its own program memory, decodes, and broadcasts (microoperation signals) to all element Associative Output Units (AOU). The AOUs can be activated independently of the element Arithmetic Units; only active AOUs execute instructions.

Each AOU contains conventional A and B Registers which support operation of integer and logical operations plus the aforementioned associative search operations. Parallel searches can be based on multidimensional search criteria, and data can be ordered in several different ways as they are obtained from the ensemble of elements. As in any associative memory, data output is sequential (in PEPE, sequential by 32-bit word). Data are buffered into AOCU Data Memory before transfer to the Host or some other device. Like the CUs, the AOUs execute instructions at an average rate of five million instructions per second, and they share element memory.

Internal interfaces

As shown in Figure 3, three major internal interfaces exist within the Control Console.

First, the Intercommunication Logic provides means for transferring data and control words among the control units. It contains a real-time clock and an interval timer accessible to all control units, and it provides means for the CCU and/or the AOCU to interrupt the ACU.

Second, the Element Memory Control receives requests from the three control units for element memory access. It performs any needed conflict resolution and transmits memory addresses to the elements.

Third, the Output Data Control is provided to resolve conflicts between AU and AOU requests to output data to the ACU and AOCU, respectively, over a common output data bus. The CUs have no capability for outputting data.

Design considerations

Several major design tradeoff decisions were made in assigning various functional capabilities to the different PEPE machine resources. These tradeoffs were made after extensive experiments involving simulated BMD exercises for a variety of attack scenarios and defense configurations. The tradeoffs were optimized for BMD service, and are not necessarily the ones that would have been made for PEPE application to other problems.

First, no provisions were made in the PEPE design for interelement communication. Data can be moved from element to element, but only through the Control Console, and the data transfer is sequential, one word at a time. This lack of a parallel interelement data transfer capability limits PEPE speed on problems involving computation performed on interrelated data sets (as encountered, for instance, in numerical solutions of partial differential equations), but for computations involving independent data sets, it is a real advantage. For the latter class of problem, each independent data set (for instance, a file on an aircraft, a missile, a

person) is assigned to an element, and each element performs identical operations simultaneously on the independent files. Since there is seldom, if ever, any interaction among the files, no interelement communication is required. The hardware simplification achieved by not providing interelement communication is obvious, but there are other more important advantages. The ensemble of elements can be completely unstructured (accordingly, the collection of elements is called an ensemble rather than an array); no element has any positional significance. Thus, the elements can be accessed, ordered, and grouped for data manipulation purposes purely on the basis of their contents, or associatively. An element can fail, with no impact on the calculations occurring in the other elements. Moreover, for many problems where data are periodically updated on the basis of continually arriving new data, only the historical state of an independent file is lost when an element fails. New data destined for that element are unable to correlate, so the data are automatically entered into an empty element to begin a new file. Thus, graceful degradation and automatic recovery are achieved naturally. Further, elements can be added as required with no effect on software or program execution. In fact, PEPE programs are oblivious of the number of elements, and programmers do not know or care either, as long as there are enough.

Another design decision involved floating-point versus fixed-point arithmetic. This decision had to be made for each of the six different processor types in PEPE (one sequential and one parallel processor for each control unit). The sequential processors in the control units are used mainly for comparison, branching, control of program flow, and supervisory operations, all of which can be executed with only integer, branching, load, store, and logical operations. Therefore, no floating-point hardware was provided in the control unit sequential processors. The Arithmetic Units in the elements have scientific calculations as their primary responsibility; therefore, they were provided with a full, powerful, floating-point arithmetic instruction repertoire in addition to all of the other conventional and associative grouping instructions. The CUs and AOUs are mainly required to input/output data and perform associative comparisons, matches, searches, and ordering functions, all of which can be handled with logical and integer arithmetic operations. However, they are occasionally called upon to execute short subroutines containing floating-point operations. Rather than provide expensive floating-point hardware in all of the CUs and AOUs to accommodate the rather infrequent requirements, it was decided to include instead provision for the CCU and AOCU to interrupt the ACU. Then, the ACU could perform the floating-point routines on demand for the CCU and the AOCU. A considerable amount of hardware was thereby saved, and simulations show that the performance degradation caused by lack of floating-point capability in the CUs and AOUs is insignificant.

A third design decision involved the Parallel Instruction Queue, which was provided only in the parallel instruction stream in the ACU. It was provided there because the ACU program memory is much larger and therefore slower than the other PEPE memories (200 ns vs 100 ns cycle time). Also, the average ACU instruction time is one microsecond, so that ACU parallel program execution time could be significantly reduced by inserting a fast stack (the PIQ has 16 ECL registers) between the program memory and the PICU. Since the CCU/CU and AOCU/AOU average instruction speed is 200 ns and the smaller CCU and AOCU program memories have cycle times of 200 ns, no such speedup could be realized in these processors with instruction stacks, and none were provided.

A fourth decision involved the output data bus from the parallel element to the Control Console. Although most element data are output from the AOUs to the AOCUs, occasionally the AUs contain data in registers destined for output to the ACU. To accomplish this transfer, the data could be stored back into element memory, retrieved by the AOU, output to the AOCU, and then transferred to the ACU. Alternatively, a separate AU/ACU data bus could be provided. Since the first alternative was too slow and the second too expensive, it was decided to allow the AUs and AOUs to share the same output data bus, and logic to implement this was provided. Since the AUs output data infrequently relative to the AOUs, output operations are not significantly degraded.

A final decision arose because the element CU, AU, and AOU share a common data memory, and since they can all operate simultaneously, conflicts inevitably occur and must be resolved. Since the element memory has a cycle time of 100 ns, and the average instruction time is one microsecond for the AU and 200 ns for both the CU and the AOU, it would seem that either the CU or the AOU should have first priority. This priority, however, is problem-dependent so simulations were held to establish priorities for memory conflict resolutions. The simulations demonstrated that the priorities should be CU first, AOU second, and AU last. With this priority assignment, the AU program execution time was extended only about five percent over what it would be if the AU had exclusive memory access.

Since memory access conflicts are rare (for typical BMD problems) and since the CU, AU, and AOU can operate simultaneously, average instruction rates for a single element can approach 11 million instructions per second. When all elements are operating, an ultimate rate of over a billion instructions per second can be achieved, but this rate is highly problem-dependent and would occur infrequently.

PEPE APPLICATION IN BMD

A simplified subset of the BMD data processing problem, as implemented on PEPE for its solution, will be described here to convey an idea of how PEPE is used in real-time control applications. Extensions to other similar data processing problems should be obvious, or at least not difficult. The problem comprises the maintenance of data files in real time on all targets (typically several hundred) within view of the phased-array radar. This requires matching each radar return against all target files to determine which file should accept the return data, and placing the return data in that file (typically ten to a hundred instructions per return per

file). If no match is possible, the return is assumed to be from a new target and is used to initialize a new file. Then, the new return is used to update the target file (typically several thousand instructions per update). Finally, based on the updated file, a request for a subsequent radar pulse to gather new data on the target is generated (typically tens to several hundred instructions per update). The request is then scheduled on the radar time line based on a multiplicity of complex radar/environmental/situation constraints (typically several hundred instructions per file per request). The real-time constraint is that, for each return (several hundred to several thousand per second), a request for a subsequent radar pulse must be generated and delivered to the radar within a specified time interval. The time interval, generally called port-to-port time, is usually different for different functions and can be extremely critical for some of them.

The problem is implemented on PEPE by assigning each individual target and its file to one element (other implementations are possible but this is most straightforward). For simplification, initialization of files and starting the problem on the PEPE will be ignored; instead the course of the problem will be picked up in the middle of a BMD engagement and carried on from there. Several hundred target files occupy the ensemble memory, one target file to an element. Periodically, the files are updated in parallel by the ACU/AUs using appropriate mathematical routines. Generally, time enablement of the routines is used; the frequency of enablement is chosen high enough so that port-to-port timing restrictions are not exceeded. For any one enablement, not all files are updated; only those elements which have received new radar return data since the last update are activated and only they participate in the parallel updating.

While the file updating is proceeding in the ACU/AUs, radar return data are streaming sequentially into the CCU data memory, either through the host or directly from the radar. As each return enters (a block of about ten 32-bit words describing target position, time of return, etc.) the CCU interrupts the ACU. The ACU then executes a prediction routine which predicts forward, to the time of the return, the positions of all targets on file. Then, the ACU constructs multidimensional windows around the positions. These operations are performed simultaneously for all targets on file. The ACU is then released, and the CCU orders the transfer of all of the window parameters into the CU correlation registers. The new return data are broadcast to all CUs, and the target position is compared simultaneously to all windows. Where a comparison is successful, the return data are placed in the element where success was achieved, and are used to update the target file during the next AU update interval. When no successful comparisons are made, the target is assumed to be detected by the radar for the first time, and the return data are placed in an empty element to start a new target file. This sequence of events is continuous, and it occurs simultaneously and asynchronously with the updating operations taking place in the AUs. The sophisticated reader will note that the foregoing procedure is a gross simplification of the target correlation problem, no account being given for practical considerations such as crossing targets, ghosts, and redundant targets. However, such problems are common to all computers and depend upon algorithms, not architecture, for their solution. As expected, the associative properties of PEPE make it an especially efficient implementer of algorithms designed to solve practical target correlation problems.

As a final operation in each update occurring in the AUs, requests for subsequent radar transmit/receive pairs, one request per target, are generated. Request formats include time of pulse, type, beam direction cosines, range windows, special pulse characteristics, etc., depending upon target type, priority, and a variety of other considerations. Resolution of conflicting pulse requests, allocation of pulses to targets, and scheduling of requests in the form of pulses and receive windows on the radar time line, are tasks assigned to the AOCU/AOUs. Generally, the objective is to schedule each pulse so as to maximize the BMD system response at the current instant, subject to a large variety of constraints and demands. This requires that target files be searched for highest-priority requests, where priority is a function of many variables. These requests must then be matched against a set of dynamic constraints (generally stored in the Host) in order to build a block of radar orders. Obviously, the procedure calls for complex multidimensional file-searches and matching and comparison operations, and is extremely time and resource-consuming for conventional sequential computers with location-addressed memories. Since the AOCU/AOU conducts the file searches and makes the matches and comparisons associatively, the procedure is considerably simplified and speeded up.

PEPE CONSTRUCTION

PEPE construction is rather straightforward, and employs circuit devices, hardware, structure, and wiring which may be considered state-of-the-art for supercomputers. The Control Console is contained in one cabinet 84″ high, 82″ wide, and 30″ deep. Elements are contained in identical cabinets, 36 elements to a cabinet. Eight element cabinets therefore comprise a full 288-element ensemble.

A complete element is contained in six 16″×18″ printed circuit plug-in boards, two each for the AU and the AUO, and one each for the memory and the CU. All board wiring is contained in eight printed circuit layers, four for signals and four for voltages and ground. Signal layers are designed to maintain a constant 50-ohm impedance. The boards connect to cabinet back-plane wiring through four plug-in connectors per board, 100 pins per connector. All logic is implemented in MECL 10K Dual-in-Line (DIL) integrated circuit packages which plug into sockets on the boards, Each board has space for 300 DILs, but there are only about 1000 DILs per element.

The boards plug into the cabinets vertically in four rows, nine elements to a row. Each row has its own power supply which supplies 5.2 volts at 560 amperes and 2.0 volts at 540 amperes to all of the elements in its row. The right-hand side of the cabinet contains these four power supplies arranged vertically, and the left hand side contains signal distribution

logic and circuitry. The four backplanes (one for each row) into which the elements are plugged are fabricated from three heavy laminated copper plates, which provide dc power distribution, signal ground plane, and power-supply bypass capacitance. Backplane interelement signal wiring is implemented entirely in 50-ohm subminiature coaxial cable. All of the nine elements in a row are connected to one signal distribution bus comprised of about 300 separate balanced-pair high-speed signal lines (PEPE uses a 10-mHz clock). The bus consists of flat belted cables which maintain constant impedance throughout the signal distribution system. The cables are connected to paddle boards which plug into the rear of the back plane to make connection to the elements. The four busses in a cabinet are fanned out from signal distribution circuitry in the left end of the cabinet. The eight cabinets are all connected to the Control Console signal distribution system via balanced-pair, constant-impedance flat belted cables.

Cooling is provided by a water-cooled chill plate located under the bottom row in the cabinet, and forced-air dual blowers located in a plenum beneath the cabinet. Total power dissipation is about 20 kW per element cabinet.

The Control Console cabinet construction is similar to that of the element cabinet, except that the Control Console printed-circuit boards use only six printed circuit layers, plus some conventional wire-wrap signal interconnection.

PEPE SUPPORT SOFTWARE

A detailed discussion of PEPE software is beyond the scope of this paper, but a summary of the capabilities of the support software system developed for PEPE will be given here. The support software system comprises a real-time executive system, a PEPE operating system, an instruction-level hardware simulator, a procedure-oriented language and translation system, and a general utilities package. All of the foregoing operate under control of the CDC 7600 SCOPE 2.0 Operating System.

The real-time executive has two primary responsibilities. First, it schedules tasks for execution on the host and the various PEPE processors in accordance with requests generated by other tasks, data enablements, time enablements, and system status. Second, it dispatches tasks in accordance with priorities and satisfied precedence relationships. The real-time executive resides and operates in both the CDC 7600 Host and the PEPE ACU.

The PEPE Operating System, which includes the real-time executive and supplies support service for it, has components which reside and operate in the CDC 7600 Host and all three control units. It includes mainly interrupt handlers and input/output handlers. Data buffers for Host/Control Console input/output operations, as well as storage space for the process control tables which support the real-time executive are also provided in the CDC 7600 small-core memory and the data memories in the PEPE control units.

The instruction-level hardware simulator is a program which runs on the CDC 7600 and executes interpretively on real PEPE problem code to simulate the operation of the PEPE/Host configuration (except, of course, for run time). It was produced to support the development of PEPE system and problem software prior to hardware availability, but it has additional important uses. Since the initial PEPE hardware installation will comprise only 11 of the 288 elements, it will be used in combination with the hardware to run tests where 11 elements are insufficient. Also, since the entire PEPE Project is at present an experimental one, the simulator will be used to check proposed hardware modifications before they are implemented.

PEPE is programmed in PFOR, a procedure-oriented, higher order superset of FORTRAN. PFOR also includes PAL, the PEPE Assembly Language. In summary, PFOR adds to FORTRAN additional statements which permit the declaration of parallel variables (those variables stored in the element memories), and which implement and exploit the parallel processing and associative access/grouping properties of PEPE. The PFOR compiler is a set of compilers and assemblers, together with a linkage editor, which accepts a single PFOR source program and expands it into object programs and load modules for all of the PEPE processors. Current experience shows that PFOR is an easy language to use and PEPE is easy to program; FORTRAN programmers can learn to program PEPE in a week or so. PEPE programs are generally characterized by a structural simplicity which makes them easy to read, understand, debug, and modify. Of particular interest is the relative absence of complicated program loops and housekeeping operations on data locations.

The PEPE Utilities package is quite conventional; it comprises the usual loaders, debug aids, and data recording programs. However, the actual implementation of the package components is unique because of the PEPE parallel/associative architecture.

PEPE IN NON-BMD APPLICATIONS

BMD data processing can be characterized as parallel computation on independent data sets; this characterization can also be applied to air defense, air traffic control, and several other applications. Such applications, when implemented on parallel architectures, do not benefit from a capability for parallel interelement data transfer among the parallel elements; in fact, there are disadvantages, as discussed previously in this paper, in providing such a transfer capability. However, there is a large class of problems which require operations on interdependent data sets, such as weather forecasting and other fluid dynamics problems requiring the numerical integration of nonlinear partial differential equations. Such problems can be solved much faster when implemented on parallel architectures, but parallel interelement transfer capability seems, at first look, to be required for efficient execution. In this section of the paper, the operation of PEPE, which has no provision for parallel interelement data transfer, on such problems will be discussed.

In fluid dynamics problems, interest focuses on the time-

variant behavior of several dependent variables at a multiplicity of fixed points in 3-dimensional space. This behavior is described (for a continuum) by a set of partial differential equations, generally nonlinear. The only practical procedures for integrating these equations are numerical ones, in which the region of interest is divided into a 3-dimensional grid, and the partial differential equations are replaced by their linear-difference approximations at each point in the grid. The resulting system of linear equations is then solved by any one of a number of methods developed for that purpose. Generally, these methods involve iteration and most of them fall within a broad class of operations called relaxation techniques. Computer architects do not need to know details of the specific algorithms used. It is sufficient to know that the relaxation procedure is started in a computer by assigning to each interior grid point estimated values for the dependent variables. Further, variables at exterior grid points are assigned values fixed by boundary conditions or if not, estimated values. Then, for each grid point, the computer performs several hundred to several thousand mathematical operations (only the instruction mix and count are of interest to a computer architect) to "update" the values of the dependent variables at that point. These operations involve the variables at that point, and all the variables at the surrounding six points. After each grid point in the total volume of interest is thus treated (one relaxation), there are better estimates of the variables at all the grid points. The computer then repeats the whole procedure for as many relaxations as are required to achieve convergence. The foregoing is a simplified and incomplete description of how a computer proceeds to solve problems in fluid dynamics, but it is sufficient for this paper.

A sequential computer can operate on only one grid point at a time; therefore, run times can be extraordinarily long. Obviously, there is a considerable amount of parallelism in relaxation techniques of the type described above, so one should be able to do better with a parallel processor. The most direct approach would be to assign an element to each grid point, store the initial values of the variables at each grid point in its assigned element, and carry out all of the grid-point computations simultaneously. There are, however, at least two problems which make this approach impractical. First, there are requirements for fluid dynamics calculations in regions containing several hundred thousand grid points, and this is not a practical number of elements.

Second, each element needs variables from other elements to perform its calculations, and moving these variables into the proper elements prior to calculations for each relaxation is a formidable engineering problem if a large number of elements are involved.

The first problem can be solved by employing a mass storage device to store a complete grid image, and to have the parallel processor operate on one section of the grid image at a time. Thus, the variables at the grid points in a section of the grid image would be transferred to the parallel elements, calculations would be performed on all of the grid points in that section, and the updated variables would be transferred back to that section in the grid image to replace the old values. Moreover, variables representing more than one grid point could be stored in an element, the number depending on the size of element memory. A complete sweep through the entire grid image would be made per relaxation, with the size of each section depending on the number of parallel elements and the number of grid points assigned per element. Sweeping through the grid image on mass storage and in effect replacing the image with a better estimate of it represents a lot of data transfer between the mass store and the parallel processor, but it cannot be avoided as long as there are not enough elements to contain the entire grid image, which will almost always be the case for the sizes of problems faced by fluid dynamicists. This data-transfer operation, incidentally, is independent of the second problem and is encountered in all parallel processors whether or not they have efficient facilities for transferring data among elements. The mass store data transfer problem must be handled by suitable choice of mass store and its interface to the parallel processor.

The second problem can be handled in several different ways. First, provision can be made in the hardware for efficient interelement transfer, as was done in the ILLIAC IV. This complicates the hardware and forces a rigidly structured implementation of the problem on the processor array, which in turn leads to reliability problems and perhaps topographical mapping problems with grids of unconventional geometry.

Second, an external data manipulator, or permuter, can be attached to the parallel processor array to allow parallel transfer of data from any ordering of elements to any other ordering of elements. This is the approach taken by the STARAN flip network and Feng's "Versatile Data Manipulator." It requires a lot of extra hardware, but can handle any grid geometry.

Third, interelement data transfer can be dispensed with entirely, as in the PEPE, so long as all of the data needed for one relaxation calculation is entered into the proper elements at the time a grid section is transferred from mass memory to the parallel processor array. This requires more storage per element, since each element must store not only the variables it is responsible for updating, but also all those additional variables needed to do the updating. This could mean a factor of up to six times the element storage required by the other two schemes. However, the hardware is simple, complicated grid geometries are no problem, and the parallel array could employ an associative data transfer scheme between the mass storage and the array to counter the reliability problem of failed elements. This scheme would allow use of an essentially unmodified PEPE in fluid dynamic problems.

Assume a global circulation problem, in which we place on the globe a grid point at every five degrees of latitude and longitude, and at six altitude levels. Unwrapped from the globe, this is a rectangular $72 \times 36 \times 6$ grid. At each of these grid points, we are interested in the behavior of eight dependent variables. We will, therefore, be performing calculations at each grid point iteratively to continuously update the values of the variables. To perform one update of the eight variables at one grid point requires that we have available the previous eight values at that point and the

current 48 values from the six neighboring points (north, south, east, west, above, below), and that we must perform 2000 arithmetic operations for each grid point. One complete relaxation requires an update of all $36\times72\times6=15{,}552$ grid points. All of the foregoing numbers are realistic, being derived from the NCAR Global Circulation Model.

To solve the above problem on PEPE, we store the entire grid image on mass storage. We assume the mass storage is connected, via simple interface, to the CCU and the AOCU. Alternatively, it could be connected to the host via a standard interface. Now our general approach is to compute on a section of the grid at a time, to always store the required neighboring grid points in the same element as the point being updated so that interelement data transfer is never needed, and to simultaneously compute, input, and output on three different grid sections. Specifically, for each grid-section relaxation we will assign to each element responsibility for updating the six points in one column. That means we store in each element one column and the four neighboring columns, or $5\times6\times8=240$ words. Now we have all the data needed to update the six points in the center column. Since PEPE will be simultaneously computing on one grid section, inputting from mass store on the next, and outputting to mass store the updated values of the previous section, we need $3\times240=720$ words of storage per element.

To update the center column requires 2000 instructions per grid point, or $6\times2000=12000$ per column. Since the PEPE AU computes at a rate of one MIP, this takes 12 ms. Now, how many columns can be updated simultaneously, or stated another way, how many elements can be kept busy? The answer will give the performance that can be obtained from PEPE, or the number of MIPS that can be applied to the problem.

As soon as one set of columns is updated, computation should start on the next set of columns, to keep the AUs busy continuously. That means the loading of the next five columns in the next grid section, and the unloading of the previous five columns should be complete in 12 ms. Unloading (center column only) of 48 words/element at 2.4 μs/word = 115.2 μs/element (2.4 μs/word is the AOU to AOCU transfer time; we assume that AOCU to mass memory time can keep up with this). Therefore $12000/115.2=107$ elements can be unloaded during the 12 ms compute time.

Loading is a little more complicated. Five columns per element must be loaded in 12 ms, but five elements can be loaded simultaneously, provided we can steer the columns to the right elements. If we can do this perfectly, then loading time will be the same as if we only had to load one column per element. Assume we cannot do it perfectly, so that we have to load the equivalent of two columns per element. Then, loading takes 96 words/element at 1.2 μs/word, so 107 elements can be loaded during the 12 ms compute time (1.2 μs/word is the CCU to CU transfer time).

All of the foregoing means we can keep 107 elements concontinuously busy, giving a computational rate of 107 MIPS on the global circulation problem. This is better than any other existing machine can achieve, especially when one considers that the MIPS required for I/O are overlapped with the compute MIPS and do not subtract from them. In other machines, of equivalent MIP rating, the I/O MIPS (and/or interelement transfer MIPS) must be subtracted, which effectively lowers computational rates.

There are several advantages to the foregoing implementation. The formulation is independent of grid size and geometry; these can be changed easily. The implementation is straightforward and should be easy to program. Data management should be simple, especially when compared to that required for sequential and vector machines. The use of associative input to the elements should permit element failures without program abort, so throughput per shift should be higher than can be obtained from less reliable machines of equivalent MIP rating. Finally, the more sophisticated and complex the updating algorithms are (which is the trend), the better the performance. For instance, 4000 instructions (rather than 2000) would result in keeping 214 elements continuously busy, or 214 MIPS.

ARCHITECTURAL ENHANCEMENT TO THE PEPE

The major architectural improvement that can be made in the present PEPE design is simplification of the signal distribution system. Such a modification would result in a capability for incorporating recent technological advances, primarily the use of available bit-slice microprocessor chips, into the element design. It would also result in an ability of several subsets of the PEPE ensemble to be executing programs at the same time, instead of only one as in the present design. The signal distribution system of PEPE is also the limiting physical factor involved in increasing the number of elements in a parallel processor. Studies show that the PEPE signal distribution system (drivers, receivers, logic transmission lines, connectors) can be simplified about two orders of magnitude. The price paid is some additional complication in the elements. To understand this, consider the original concept of PEPE. A single control unit, containing instruction memory, and instruction fetching, interpretation, and decoding circuitry drives a large number of execution units, each with its own memory. This concept of multiple execution units and only one control unit saves a great deal of hardware in a parallel processor, because the control unit in a computer generally contains a significant amount of complicated hardware. However, the interface between the control unit and the execution unit is the most complicated interface in a computer, and it is this interface which, under the original PEPE concept, must be carried by the signal distribution system to all elements. At the time of the original PEPE concept, engineering tradeoffs favored the single control unit and the complicated distribution system.

Tradeoffs would produce different results now. With thousands of elements, one probably cannot tolerate a complicated signal distribution system. Moreover, control unit circuitry is much cheaper; in fact, microprocessors contain both instruction decoder and execution unit on a single chip. The interface between the instruction decoder and the execution unit is not even available outside the chip.

So, one can put both the instruction decoder and the execution unit in the element much easier and cheaper than

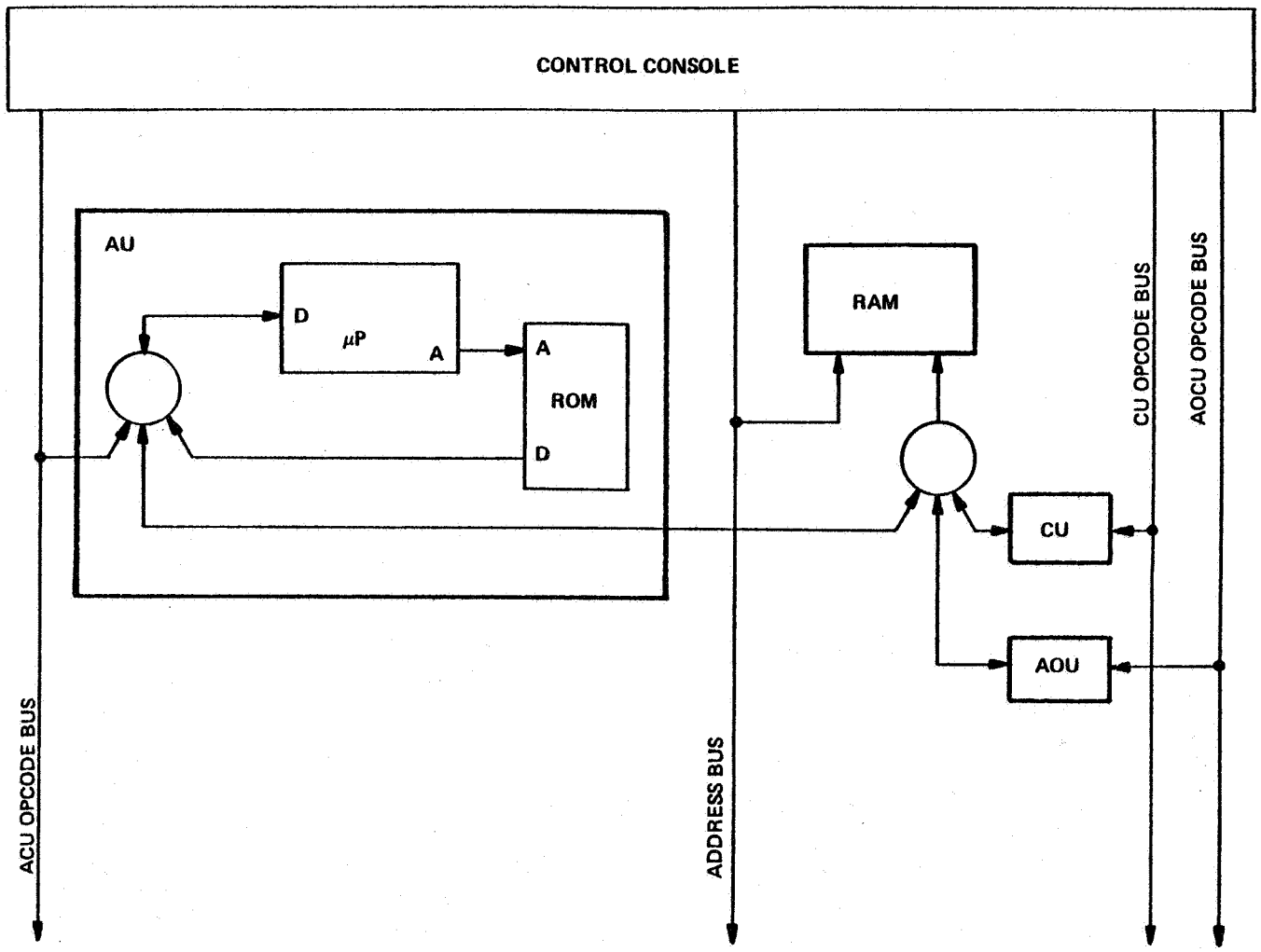

Figure 5—New PEPE scheme

was possible a few years ago, by making extensive use of LSI in the elements (possibly even off-the-shelf microprocessor chips). Then, the PEPE control unit would contain only a program memory and instruction fetching circuitry. This control unit broadcasts only the opcode to the elements, rather than decoded opcode signals. Fewer than ten signal paths are required to broadcast opcodes, rather than a hundred or so, as is the case when the decoded opcode must be broadcast. Moreover, the signal levels on the signal distribution system must change only once every instruction, rather than once every clock pulse. Thus, for the same instruction rate, the switching signals on the signal distribution system can be about ten times slower. These two factors, fewer signal paths and slower switching speeds, will permit great simplification in the signal distribution system.

Figure 5 shows the scheme. Only the details of the AU are shown, although the CU and AOU can be handled identically. The AU contains a microprocessor chip, which contains both instruction decoder and execution unit. In operation, the simplified ACU in the control console fetches an instruction, and places the address on the element address bus and the opcode on the opcode bus. The opcode first enters the data pins on the microprocessor chip, then the data from element memory is entered into the same pins. The instruction is decoded and executed on the chip and the result is returned to element memory. This sequence is conventional for almost all microprocessors. A further refinement can be made by storing special subroutines, such as trigonometric or exponentials, in ROM in the AU. Thus, the control console can include these functions within its opcode set and slow down the signal distribution system switching speed even more, while actually increasing the PEPE instruction rate.

Other useful operations can be done with a PEPE configured as described above. Since the ACU executes on the average about ten microinstructions for every opcode, more than one subset of AUs (same statement goes for CUs and AOUs) can be active at the same time, which is not possible with the present PEPE design. All that is needed is for the opcode bus to carry an activity number with it; only that set

of elements holding that activity number would execute that opcode. Then, while the selected subset of elements is executing the instruction, another instruction could be broadcast on the same bus to a different subset of elements.

This type of operation (it could be viewed as the logical equivalent of two or more ensembles) would keep more elements busy at the same time. Even further, the ensemble elements could store complete subroutines in their program memories, so that the control units need only broadcast subroutine calls rather than instructions. This would decrease signal-distribution system traffic even more, and would allow further exploitation of the ability of the subsets of the ensemble to execute several operations simultaneously.

REFERENCES

Lee, C. Y., and M. C. Paull, "A Content Addressable Distributed Logic Memory with Applications to Information Retrieval," *Proc. IEEE*, Vol. 51, June 1963, pp. 924-932.

Crane, B. A., and J. A. Githens, "Bulk Processing in Distributed Logic Memory," *IEEE Trans. on Electronic Computers*, Vol. 14, April 1965, pp. 186-196.

Huttenhoff, J. H. and R. R. Shively, "Arithmetic Unit of a Computing Element in a Global, Highly-Parallel Computer," *IEEE Trans. on Computers*, Vol. C-18, August 1969, pp. 695-698.

Githens, J. A., "An Associative Highly Parallel Computer for Radar Data Processing," Chapter 3 of *Parallel Processor Systems, Technologies, and Applications*, L. C. Hobbs, D. J. Theis, Joel Trimble, Harold Titus, and Ivar Highberg, Spartan Books, 1970, pp. 71-86.

Crane, B. A., M. J. Gilmartin, P. T. Rux, and R. R. Shively "The PEPE Computer," *IEEE Compcon 72 Digest*.

Wilson, D. E., "The PEPE Support Software System," *IEEE Compcon 72 Digest*, Sept. 1972, pp. 61-64.

Cornell, J. A., "PEPE Applications and Support Software," *IEEE Wescon 72 Digest*, Sept. 1972.

Bergland, G. D. and C. F. Hunnicutt, "Application of a Highly-Parallel Processor to Radar Data Processing," *IEEE Transactions on Aerospace and Electronic Systems*, March, 1972.

Cornell, J. A., "Parallel Processing of Ballistic Missile Defense Radar Data with PEPE," COMPCON 72 Computer Conference, 1972.

Johnson, M. D., "The Architecture and Implementation of a Parallel Element Processing Ensemble," Western Electronics Show and Convention, 1972.

Evensen, A.J., and J. L. Troy, "Introduction to the Architecture of a 288-Element PEPE," 1973 Sagamore Computer Conference on Parallel Processing.

Dingeldine, J. R., H. G. Martin, and W. M. Patterson, "Support & Operating System Software for PEPE," 1973 Sagamore Computer Conference on Parallel Processing.

Troy, J. L., "Computer Simulation of PEPE and its Host at the Instruction Level," 1973 Sagamore Computer Conference on Parallel Processing.

Barrett, A. L., "Process-Construction for a Parallel-Sequential Computer Architecture," 1973 Sagamore Computer Conference on Parallel Processing.

Berg, R. O., H. G. Schmitz, and S. J. Nuspl, "PEPE-An Overview of Architecture, Operation, and Implementation," *Proceedings, National Electronics Conference*, Chicago, Ill., October 1972.

DiVecchio, M., "The Design and Implementation of A High/Low Magnitude Search Instruction on PEPE," 1975 Sagamore Computer Conference on Parallel Processing.

Merwin, R. E. and C. R. Vick, "An Architectural Description of A Parallel Element Processing Ensemble," International Symposium on Computer Architecture, Grenoble, France, 1973.

Marshall, D. D., "A Parallel Processor Approach to Solving Decision Trees," 1977 International Conference on Parallel Processing.

Welch, H. O., "Numerical Weather Prediction in the PEPE Parallel Processor," 1977 International Conference on Parallel Processing.

Blakely, C. E., "PEPE Application to BMD Systems," 1977 International Conference on Parallel Processing.

Evensen, A. J., "PEPE Hardware and System Overview," 1977 International Conference on Parallel Processing.

PEPE—A user's viewpoint; a powerful real time adjunct

by M. P. MARIANI and E. J. HENRY
TRW—Defense and Space Systems Group
Redondo Beach, California

INTRODUCTION

The Parallel Element Processing Ensemble (PEPE) offers a cost-effective means of obtaining significant improvements in data driven real time systems. The inherent parallelism and programmability of the PEPE architecture make it an extremely effective growth option for data driven systems developed around a large-scale centralized Host. With the PEPE architecture, incremental system growth with minimum software impact can become a reality.

The purpose of this paper was to investigate the implementation of PEPE in real time data driven systems. For the purpose of an example, we have chosen the Ballistic Missile Defense (BMD) problem—a real time highly data driven system in its own right. The BMD problem desires a data processing system architecture which is not only highly resilient to the threat but also provides for cost-effective growth to accommodate changing technology. With the PEPE architecture, the system growth can be achieved through the addition of PEPE elements—the system designer can incrementally purchase only those parallel processing elements needed.

THE BMD DATA PROCESSING SYSTEM ARCHITECTURE

The assumed data processing subsystem (DPS) used in this study is responsible for controlling the radar and interceptor subsystems to insure that a sufficient number of defended missiles survives an offensive ICBM attack. To accomplish this objective the DP subsystem performs twelve basic functions (Figure 1). These functions are categorized as either module level or unit level.

The module level functions are oriented toward operational control of multiple Defense Units operating asynchronously. The unit level functions perform the detection, discrimination and intercept of the threatening objects within the Unit's operational sector. Each Defense Unit contains a CDC 7700 data processor, a single, phased array radar and a complement of interceptors. BMD research is currently focused on only the Defense Unit level of the system, and the first eight basic functions.

The eight unit level functions constitute the application programs (AP) operating in each Defense Unit's CDC 7700. The AP contains an array of independent tasks which are either data or time enabled, and scheduled and dispatched by the real time operating system. Of the array of tasks, a subset constitutes the bulk of the real time resource load on the CDC 7700.

Normally, each of the applications tasks consists of multiple subroutine modules which, for testability and growth, are usually limited in size. Within each task is a task control routine which controls the execution sequence of the subroutines within the task, performs all global data base READ operations prior to task execution, and performs all global data base WRITE operations at the termination of the task execution. This access approach reduces the real time data management overhead.

The array of tasks can range in size from less than 2500 executable instructions* to well over 40000 executable instructions for each instance of data processed. In order to prevent any single task from monopolizing the system, a maximum task execution time can be imposed. This design constraint limits the number of data instances which can be processed per task execution. As a result, there are large variations from task-to-task, in the number of instances which could be processed per enablement. These variations when combined with variations in task execution priorities can result in queue buildups. In high threat load situations the queue buildups of some of the more data sensitive tasks can experience increases in execution wait times.

Under severe loads, system saturation will be characterized by queue overflows and corresponding increases in critical response time. The parallelism of PEPE provides the potential for handling such high load conditions by reducing the data processing system's sensitivity to the threat and increasing the system threshold of saturation.

PEPE ARCHITECTURAL OVERVIEW

The PEPE is a 32-bit super-computer of the SIMD and MIMD[1] class. The PEPE[2] (Figure 2) can contain up to 288 parallel processing elements. Each element is itself a multiprocessor containing three independently operating pro-

* Average Machine Language Instructions (MLI) derived from simulations of the applications process over a full engagement scenario.

cessors—an input correlation unit (CU) and an arithmetic output unit (AOU) operating at 5 MIPS*, and an arithmetic unit (AU) operating at 1 MIPS. Each processor in the multiprocessor element is under the control of a separate control unit which broadcasts decoded parallel instructions simultaneously to all 288 of the processing elements. The control units are themselves processors capable of executing sequential instructions while simultaneously issuing parallel instructions to the PEPE elements. The PEPE architecture, therefore, permits the concurrent execution of as many as six instruction streams; three serial instruction streams driving the control unit computations and three parallel instruction streams driving each of the three processors in the parallel elements.

In an operational environment in which PEPE is used as an adjunct processor to a large high speed Host, such as the CDC 7700, the PEPE will execute only highly data parallel computations. The PEPE will operate in a dedicated manner and will contain its own application load modules and operating system. The PEPE-Host communications will be used to only support process control and global data shared between separate processes executing on the Host and PEPE.

*Based on typical BMD computational instruction mixes.

PEPE IMPLEMENTATION—OBJECTIVE

The parallel architecture of PEPE, coupled with its inherent throughput potential offers an available low cost method for responding to the advances in threat technology. An implementation of PEPE which does not accompany a redesign of the software results in an inefficient use of the PEPE

LEVEL	NAME	DEFINITION
Unit Level	1. RRA/RS	Radar Return Assimilation and Radar Scheduling
	2. ODD	Object Detection and Designation
	3. OT	Object Track and Prediction
	4. IPP	Interceptor Planning and Prelaunch
	5. IC	Interceptor Guidance and Control
	6. URM	Unit (Radar and DPS) Resource Management
	7. DMC	Defense Module Inter-Communication
	8. OD	Object Discrimination
Module Level	9. MBM	Module Battle Management
	10. MSA	Module Status Assessment
	11. ESA	Environment Status Assessment
	12. CMP	Command Message Processing

Figure 1—Basic BMD functions

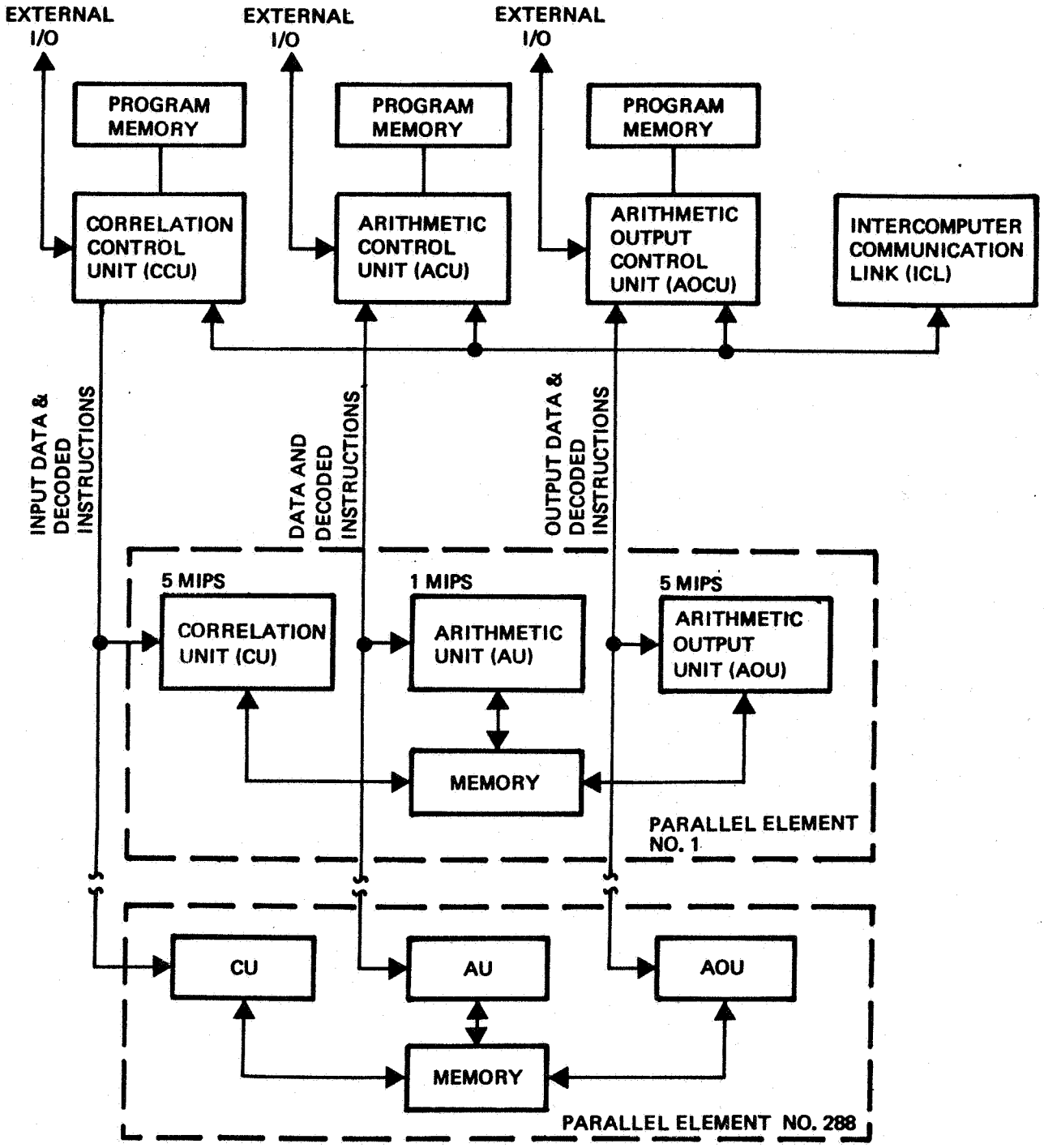

Figure 2—PEPE architecture overview

resources. However, the optimal allocation of a PEPE architecture to the dual CPU CDC 7700 architecture could involve substantial breakage.* The initial objective of the investigation was then to consider alternatives for low-cost (minimal breakage) implementations of PEPE. Therefore, any offload design requiring either a significant software redesign and/or a significant redesign of the PEPE-Host-Radar hardware interface was not considered. Similarly the optimization of the offloaded code was considered less critical than maintaining the original task software.

For each offload design considered both the application

*Breakage as related to the current DP system architecture (both software and hardware) was considered to include:

a. *New* Architecture (SW or HW) components.
b *Modification* of existing Architecture components.
c. *Disposing* of existing Architecture components.

task and operating system software were analyzed for breakage. Results of potential offload designs indicated that the application software breakage was highly sensitive to the number of application tasks included in the offload, while the indirect software breakage of the CDC 7700 operating system is essentially insensitive to the number of application tasks offloaded.

PEPE IMPLEMENTATION

Under the minimum breakage guidelines, there were two fundamental PEPE-Host architectures which were considered (Figure 3). One architecture (Figure 3B) employed PEPE as a radar interface preprocessor to the Host. As a preprocessor the PEPE would perform the radar interface function; correlation of radar returns and scheduling of radar pulses. However, designs of the radar interface software for centralized computers would execute extremely inefficiently on the PEPE, and the necessary software modification to improve the efficiency would be significant. The second architecture (Figure 3C) employed PEPE as an offload processor to the CDC 7700 in its current configuration. As an offload processor the PEPE would be dedicated to performing such highly data parallel operations as verification of detection returns, track initiate and track and discrimination processing.

Analysis of conventional applications programs was used as a basis for developing the offload options and selection of the preferred offload configuration. The analysis provided performance data which expressed resource requirements for: task computation (TASK), operating system support directly related to task execution (TOSC) (e.g., data base access), and operating system support indirectly related to task execution (e.g., external communication). All analysis was conducted within the high load periods of the engagement scenarios investigated.

PEPE IMPLEMENTATION—DESIGN APPROACH

Offload candidate identification

In an attempt to achieve the most cost-effective offload, the applications tasks were evaluated with respect to their current CDC 7700 resource requirements and their PEPE offload efficiency. Three aspects of the task resource requirements were studied,

(1) *Task Execution (TASK):* the resources (msec) required to execute the task instructions only.
(2) *OS Chargeable (TOSC):* the overhead directly associated with task execution (e.g., OS service requests: task load from secondary store, file read and write, task termination, etc.)
(3) *Other OS (TOSO):* the overhead not directly associated with task execution; not occurring during task execution (e.g., external I/O, interrupt handling, scheduling and dispatching).

The resource breakdown was used to provide an indication of each task's overhead sensitivity to the execution of instruction (TASK), handling of global data (TOSC), and external communication and polling loop priority (TOSO). Only those tasks which exhibited high resource requirements were considered as candidates for offload to PEPE. In addition, the general sensitivity of the tasks to the threat traffic was investigated by studying each in further detail during high load periods.

Those tasks which were rated as high resource users were then examined for inherent parallelism (both data and logic), as a basis for assessing the efficiency (breakage) with which a candidate task could be offloaded to the PEPE. Data parallelism was related directly to the task input data rates. With the minimum breakage guideline, algorithmic parallelism was not considered; instead, logic parallelism was evaluated at the subroutine level using the functional flow block diagrams. Tasks with a high degree of conditional branching along equal probability paths (Pb≈40-60 percent) would result in an inefficient use of PEPE resources. The lock-step architecture of PEPE can cause degraded performance in key processing tasks directly proportional to the number of branches in the offloaded processing logic.

A collection of six basic performance parameters was identified as providing the basis for assessing PEPE offload potential; they are,

(1) Total resource requirement
(2) Total number of instances processed
(3) Task resource distribution: TASK, TOSC, TOSO
(4) Total number of Task executions
(5) Peak data rates
(6) Task polling loop wait

Based on these performance parameters a set of five (5) performance criteria was established for paring down the application tasks to preferred offload candidates. These criteria were,

(1) high frequency of task execution
(2) high number of instances processed
(3) high instruction execution resources
(4) low data handling overhead
(5) high data queue buildup

In addition, the task structure was evaluated to identify those tasks with

(1) high probability logic paths ($Pb_n \geq 90$ percent)
(2) low offload instruction count with high frequency execution.

Using the above criteria the applications tasks were evaluated for their resource requirement. The results indicated that a small subset of the applications tasks accounted for over 90 percent of the total system resource requirements.

Analysis during a prolonged engagement interval provided performance profiles for the principal resource users. However, the analysis did little to indicate the sensitivity of the

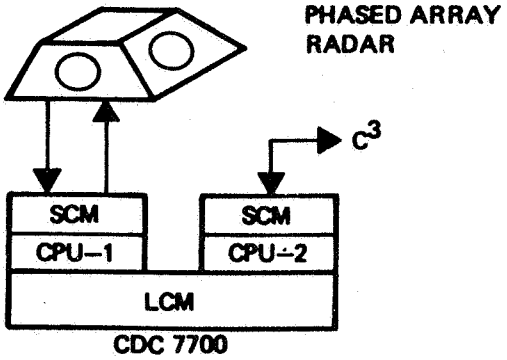

Figure 3A – CURRENT ARCHITECTURE

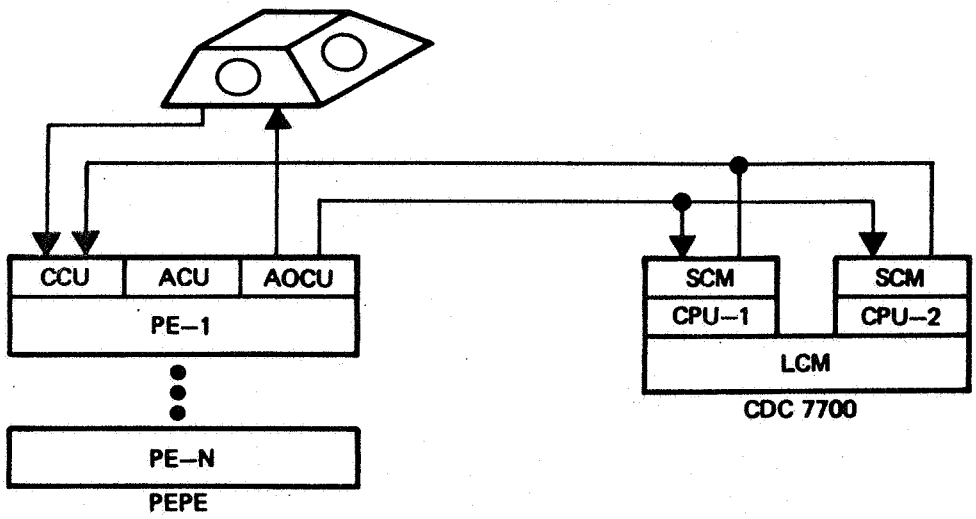

Figure 3B – PEPE PREPROCESSOR

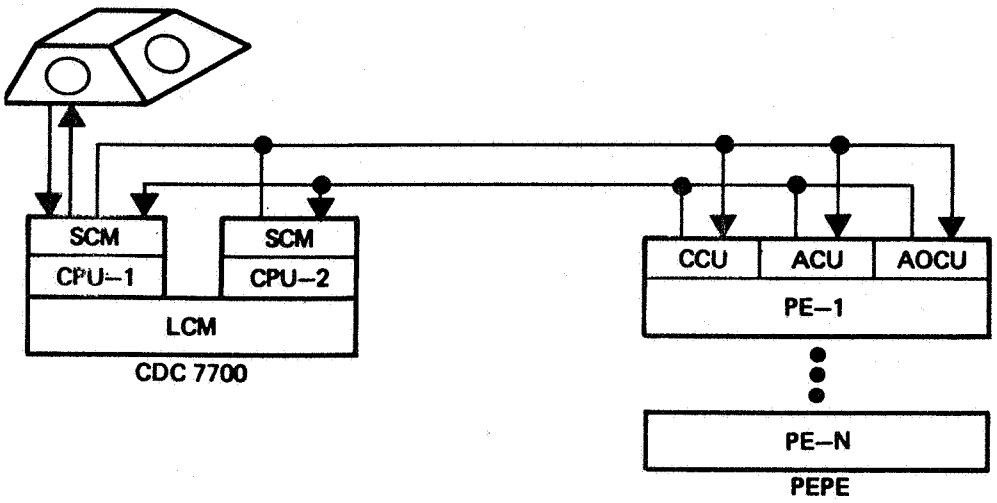

Figure 3C – PEPE OFFLOAD

Figure 3—BMD terminal defense architectures

tasks to threat traffic. Any offload configuration was restricted by the critically short processing response thresholds and limited offload storage available on PEPE. As a result, there was a need to develop criteria with which to assess the relative offload potential of each candidate task. The initial criterion was the resource offload percentage available due to each task. Tasks whose individual Host resource requirements failed to exceed at least five percent of the total applications program, were not considered as potential offload candidates. This test singled out five tasks as having the independent resource potential to justify offload.

The throughput capability of the PEPE relative to the Host, and the inherent parallelism of the PEPE architecture dictated two data related requirements. First, in order to exploit PEPE's parallelism large data loads were desired by the tasks for offload. Second, the cost to process a single instance should preferably be small when taking into account the potential expansion in processing times due to the relative PEPE-CDC clock cycles (100ns−27.5ns).

The first criterion strongly favored three of the tasks, with two of the tasks being marginal. One task did not meet the throughput requirement although the task's resource requirement is among the highest.

The second criterion of single instance processing costs strongly favors four tasks showing a high throughput requirement, with one task (interceptor control) exhibiting extremely high single instance processing costs. In addition, the stringent response requirement of interceptor control also makes it a poor selection for offload to PEPE.

A third performance consideration is the potential PEPE-Host communication overhead accompanying each task offload to PEPE. Large data communication overhead has negative impacts on both the PEPE-Host interface bandwidth requirements, and the responsiveness of the offloaded processing. Since the TOSC resource component is primarily influenced by data management costs associated with the global data, the TOSC component is a strong indicator of the potential PEPE-Host interface overhead. The ratio of TOSC to TASK resource distribution was then used to evaluate the inherent resource offload efficiency available with each offload candidate. The ratio ranged from approximately .25 to 1.14 and was found to remain fairly constant regardless of system loading.

Another consideration for evaluating task offload potential was the frequency of task execution. The execution rate of tasks when compared with their associated data rates provided another indication as to the efficiency with which tasks will execute on the PEPE. Each task execution is accompanied by a basic overhead cost (TOSC) consisting of loading the task module from large core memory (LCM) and the READ/WRITE of global data which support the task execution. This basic overhead is virtually insensitive to the number of data instances processed during a single task execution. If an execution time restriction were not imposed on the process designer, it would be much more efficient to have each task totally deplete the data queue on each execution. The parallel processing architecture of PEPE provides a means of processing a much larger number of data instances per task execution while still maintaining an execution time design restriction.

Offload design—General approach

Minimization of the impact of any offload design required the development of a technique for restructuring the current application software to support offload to PEPE. The technique which we developed was based on partitioning the applications tasks along their current subroutine structure. In addition, the table driven design for the current operating system allowed the offload to be implemented with no impact to the OS software.

The partitioning approach to PEPE offload involved the segmentation of each offload task into three components (Figure 4) operating as three tasks. Two new data files were defined which serve as data enablement queues for the second and third segments. The first and third segments of the task would continue to reside in the Host computer. These segments provide all access to the global data base, and interface with the remainder of the application process. The second segment would reside on the PEPE. One objective of the partitioning was to maintain the total cost of executing the two segments residing on the Host≤the original cost of the task prior to offload.

In general the segmentation will be performed across loosely coupled routine interfaces in the task logic, and produce a segmentation in which the largest segment of the task is placed on the PEPE.

The initial segment will continue to execute in the Host as a data enabled task. The primary purpose of this Host segment will be to format and write into the newly defined PEPE input queue all data necessary to support the second task segment residing in the PEPE. The data will typically include the data instances, such as radar returns, and dynamic data which is needed to support the execution of the PEPE segment.

The second segment would be offloaded to the PEPE and allocated to either the CCU, ACU, or AOCU. The PEPE will contain the complete load module of the second segment loaded into the program memories of the CCU, ACU, or AOCU. Any fixed point arithmetic or logical operations of the offloaded segment, such as unpacking/packing operations, will generally be performed by the CCU or AOCU. The ACU will perform all floating point computations. In

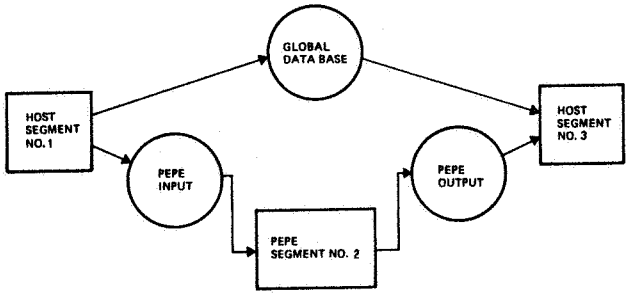

Figure 4—Task offload segmentation technique

addition, new PEPE software must be developed for the CCU and AOCU to support data manipulation within the PEPE, such as loading the PEPE elements prior to parallel computations and retrieving the processed data from the elements for retransmission back to the Host following computations. In addition, for data management operations such as large file searches/correlations, new software must be incorporated into the offloaded segment which exploits the PEPE's associative capabilities. The PEPE Fortran (PFOR) possesses instructions specifically designed to support associative search operations within the PEPE elements. Typically operations such as object redundancy checks and nuclear interference prediction which could be extremely costly when implemented on the Host, will be executed with only a couple of associative search instructions on PEPE.

Alternative offload designs

One of the major applications tasks consistently displayed performance characteristics indicative of efficient PEPE offload. That task performed the object track function. The keys to the offload desirability of the track task were four operational characteristics,

- *high task execution frequency* with highest execution rates occurring during periods of highest threat traffic.
- *high task throughput requirement* with highest data loads (75 percent) occurring during periods of highest threat traffic.
- *high CDC resource requirement* with largest load (80 percent) occurring during periods of highest threat traffic.
- *low task execution overhead* (30 percent) throughout the operational period.

Because of the task's offload desirability it was used to form the foundation of two alternative offload designs.

In addition, the task has a moderate per instance execution cost, a highly favorable characteristic for tasks belonging to short response threads.

Using the track task as a basis, two multi-task offload designs were developed. The first design was directed at significantly reducing the total Host resources requirement. The second design was directed at substantially improving the responsiveness of key processing threads in high threat traffic situations. The first design incorporated the detection and track initiate and discrimination processing tasks with the track task into an offload module for the PEPE. This design provided the largest offload benefit in the early portion of the engagement. In addition, the detection and track initiate tasks are on processing threads with less stringent response requirements than the track thread which reduces the impact of resource contention. The second offload incorporated the interceptor control task with the track task into an offload module for the PEPE. The interceptor control task was a high resource user with an expensive per instance execution cost. In addition, the interceptor control task was on the most stringent response processing thread in the applications program.

A comparison of the two integrated offload designs indicated the first design to be far more preferable. The offload of a sizable portion of the Host resources, also had a significant indirect effect on improving the responsiveness of the threads remaining on the Host. The offload to PEPE of tasks within high response processing threads had less effect on improving system responsiveness than the first offload design. This was attributed to the fact that, (1) less absolute resources were offloaded from the Host to PEPE, and (2) the offload of tasks with equal response priority created resource contention on the PEPE. An important point of both offload designs was the ability of PEPE to substantially improve the performance and predictability of both system throughput and responsiveness.

PEPE IMPLEMENTATION—ANALYSIS

Offload design—S/W portability analysis

Parallel Fortran (PFOR)

Parallel Fortran[3] is a high level, procedural extension to USA Standard Fortran and provides a powerful and efficient set of language extensions for the associative parallel processing capabilities of PEPE. Through the use of the PFOR statements, the user has the capability of simultaneous execution of instructions in all n PEPE elements or a selected subset of the n elements. The user can nest the levels of element activity through PFOR.

In order to identify specific variables which are allocated to PEPE element memory, PFOR has parallel type declaration statements.[4] A set of parallel "system" functions to handle type conversion, square root, and special mathematical functions on parallel arguments and a set of powerful statements are included in PFOR for the subsetting of the current set of active elements. This set includes statements of the form WHERE, CONVERGE, parallel IF, and parallel DO. In addition, a statement is provided for moving data between element memories.

CDC 7700 S/W conversion

In order to determine the amount of code breakage and time required to convert existing code in FORTRAN to PFOR (Parallel FORTRAN), typical code representative of the object track task was converted. The routines were selected as being strongly indicative of the types of code which were contained in the PEPE segments of the offload design.

Some of the code was highly logic oriented and designed to control the hi-fidelity Kalman filter for the track task. The other code contained mostly arithmetic operations and little branching.

There are two options which can be considered when placing the selected subroutines and their associated data base on PEPE. Option one permits only one object maintained per PEPE element and requires a larger set of PEPE

CDC 7700/FTN

```
      SUBROUTINE XYZ

      COMMON/D3T0000/A3T00 (132)
      INTEGER ZDBQN, . . .
      EQUIVALENCE (A3T00(1), ZDBQN)
      IF (Z3DØF .EQ.0) GO TO 10
      Z3BPF = 1
10    CONTINUE

      IF (Z3DØF.EQ.0) GO TO 70
      ZNC (1, Z3ØDV) = ZNC (1, Z3ØDV) +
      FLØAT (ZNCNL (Z3BQV))
      GO TO 80
70    ZNC (1, Z3ØDV) = ZNC (1, Z3ØDV) +
      FLØAT (ZNCNL (Z3BQV)) + 1.
80    CONTINUE
```

PEPE/PFOR (OPTION 1)

```
      SUBROUTINE XYZ, ACU

      PAR COMMON/D3T0000/ZDBQN, . . .
      PAR INTEGER ZDBNQ

      WHERE (Z3DØF.NE.0) 10
      Z3BPF = 1
10    CONTINUE

      WHERE (Z3DØF.NE.0) 70
70    ZNC (1) = ZNC (1) + PFLØAT (ZNCNL(Z3BQV))

      WHERE (Z3DØF.EQ.0) 80
      ZNC (1) = ZNC (1) + PFLØAT (ZNCNL(Z3BQV)) + 1.
80    CONTINUE
```

Figure 5—Relative code comparison

elements. In converting the code for this option, the user must change the doubly subscripted variables currently used throughout the software (e.g., object state estimate) into singly subscripted variables. The second subscript associated with the instance number (e.g., object identification) would be implied by mapping each instance into a unique PEPE element. This option results in significant code modification, however, the code conversion would be simple, straightforward and could be automated.

Option two would permit more than one object maintained per PEPE element and can significantly reduce the size of the PEPE element array required. This option requires doubly subscripted variables to identify the current instance being processed in an active element. This option would result in much less code breakage than option one but would require much larger data storage per element. Once the user develops some familiarity with PFOR a set of existing routines can be transformed into PFOR in relatively short time. As an illustration, Figure 5 provides an example of a logical segment of the subroutine and the FORTRAN code changes required to convert it to PFOR. In practically all cases of code conversion, the code which required modification was directly associated with data management and manipulation. The actual scientific computations written in FORTRAN were directly transportable to the PEPE. For the three subroutines that were selected, approximately 76 percent of the executable statements would require modification for option one and 4 percent for option two.

In the offload design, the current CDC 7700 data base design is not effective when directly installed on PEPE. In each offload design the subroutines executing on the PEPE require only small segments of the data contained in current common blocks. To conserve element memory we elected to restructure the common blocks within the PEPE. The restructuring involves eliminating unused variables and redefining the nonexecutable statements which define these variables. Since the current set of declarative statements (i.e., INTEGER, REAL, EQUIVALENCE) have to be converted to PFOR (i.e., PAR INTEGER, PAR REAL) there is total breakage for these nonexecutable statements.

It is felt that other applications which are suited to the type of parallel processing offered by PEPE could just as easily be converted to PFOR. In some cases the code conversion could even be automated. Applications which would require movement of data between elements may require a longer time to convert to efficient PFOR, but once the user becomes familiar with PFOR, the code conversion is straightforward.

Offload design—performance impact

Analysis indicates that the first offload design exhibits a significant overall potential for reducing the load on the Host computer. Over the complete engagement the offload design provides a net reduction in the average Host resource requirement of the applications program by over 25 percent. This is accomplished by a 54 percent reduction in the resource requirements originally required by the offloaded tasks. The offload design provides the most significant reductions during the major peak load periods of the engagement; specifically early during initial acquisition of the threat and later during track and discrimination prior to intercept

commit of the threat. The early load is caused by high Host resource requirements imposed by the track initiate loads. The later Host load is caused by high Host resource requirements imposed by the object track and discrimination processing. The offload to the PEPE provides a significant reduction in the Host loads during both the early (~28 percent) and later (~23 percent) portions of the engagement.

An analysis of the PEPE under this offload design indicated that both storage and utilization requirements were well within available limits. The program storage requirements for each of the three control units were all less than 35 percent of available storage. Data storage requirements were higher and reached 84 percent of that available in the CCU, and 65 percent of that available in the PEPE element memories. Evaluation of the PEPE resource requirement during the engagement indicated that none of the control units exceeded 80 percent utilization. The average utilization of these units was substantially less, never exceeding 60 percent (ACU), indicating resources are available for additional applications tasks to be offloaded.

The size of the Host resource offload is directly related to the traffic load on the system and as a result varies during the engagement. A major benefit of the PEPE offload design is the reduction in sensitivity of the PEPE-Host system resource requirement to increases and unpredictable perturbations in data (threat) traffic patterns.

The impact of resource offloads on the system responsiveness was also investigated. In the original Host environment there were two principal components of system response, the first component being that of execution wait times, which includes the time from enablement to dispatch. The second component is the actual execution time once dispatched. In high load periods of resource contention, the task wait times become a major component of the overall system responsiveness for the Host alone system. With the offload to the PEPE, a third resource component must be considered, that of PEPE-Host communication. In the offload design, the contention for resources is reduced, and the execution wait times are decreased. In addition, the execution wait times tend to be more stable with the PEPE adjunct than with the Host alone.

When combining the effects of reduced task execution wait time, increased task throughput and the addition of the communication delays, the results indicate that the PEPE-Host system provides improved system responsiveness. This responsiveness is substantially improved during periods of high load conditions.

One of the major offload issues addressed the impact that the PEPE adjunct, and its associated PEPE-Host communication, had on the Host Large Core Memory (LCM) utilization overhead. Analysis indicated that the LCM overhead actually decreased with the offload. Although the Host LCM overhead necessary to support real time input/output (RTIO) increased with the addition of the PEPE, the reduction in the overhead associated with management of some of the larger data files and load modules, now resident in the PEPE, more than compensated for the RTIO overhead increase.

SUMMARY

THE PEPE provides a highly cost-effective method of improving the data handling capacity of a real time system developed around a single large-scale Host. Analysis of conventional terminal defense systems for BMD indicates that the addition of a PEPE adjunct can provide a means to substantially offload the Host computer. As a result of the Host offload, an improvement can be achieved in the capacity and predictability of both the system's throughput and responsiveness.

In addition, the implementation of PEPE can be accomplished with only a minimum breakage to software developed for the Host. In most cases the software breakage is repetitive but not complex and can be automated without significant cost. For the specific offload design investigated, the PEPE storage and throughput requirements were well within limitations and PEPE had the resources to accommodate additional offloads. Based on the current design the apparent threshold in PEPE offload will be the availability of data storage. Analysis indicated that through a redesign of some of the larger common areas the data storage requirements on the PEPE can be significantly reduced.

Further analysis of the offload design indicated that the PEPE adjunct provided several direct performance improvements for the data processing system,

(1) increased data throughput for tasks offloaded to the PEPE,
(2) reduced execution delays for entire application program,
(3) reduced data management overhead,
(4) reduced system sensitivity to data traffic,
(5) improved predictability of the system's real time resource demand.

It is felt that these performance improvements can be achieved as well with non-BMD applications which possess the parallelism which is suited for the PEPE. Once the PEPE user becomes familiar with the operation and the computing power of the PEPE, suitable applications can be mapped onto PEPE in a simple and straightforward manner.

CONCLUSIONS

Results of this PEPE application case study have general applicability to most data driven, real time systems. The inherent parallelism and programmability of the PEPE make it an extremely attractive mechanism for achieving significant improvements in a centralized systems data handling capability.

The implementation of PEPE as a Host adjunct allows the offload of resources to the PEPE. This offload can produce a substantial improvment in the data processing system's throughput and response. The improvement is effected in both the system's capacity and performance predictability to data loading.

REFERENCES

1. Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, September 1972, pp. 948-960.
2. Cornell, J. A., "PEPE Architecture—Present and Future," National Computer Conference, June 1978, Anaheim, California.
3. System Development Corp., "Preliminary Users Manual for the PFOR Language Translation System," SDC, TM-HU-046/400/00, 20 August 1973.
4. Cornell, J. A., "PEPE Application and Support Software," *WESCON* 1972, September 1972, pp. 1/3-1 to 1/3-3.