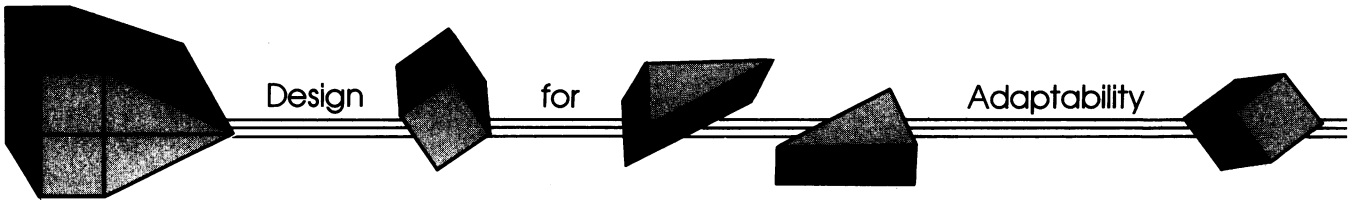


Adaptable Software for Supercomputers

Thomas Schwederski and Howard Jay Siegel
Purdue University

Software used to control and program reconfigurable supersystems must efficiently exploit the hardware flexibility available. If not, the system does not fulfill its potential.

Steady increases in computer hardware performance show no sign of slowing. Today's supercomputers such as the Cray-1,[1] Cyber 205,[2] and MPP (Massively Parallel Processor)[3,4] are capable of a few million hundred-floating-point operations per second. However, even these computers do not always satisfy the speed requirements of the scientific and defense communities. Consequently, researchers are now exploring new supersystem architectures, such as data flow computers and reconfigurable computers that restructure the organization of their hardware to adapt to computational needs. If a supersystem can be reconfigured, it can more likely efficiently execute tasks that previously required a set of dedicated systems. Reconfiguration is especially important if the programs executed by the supersystem have widely differing computational requirements. Computations needed to control some sophisticated weapon systems are of this variety.[5]

The utilization and performance of any supersystem depends strongly on the software available for it. Necessary software includes system software, such as compilers and operating systems, and application software. Designing the system software to make efficient use of complex supercomputers is difficult. It is further complicated if the supercomputer is reconfigurable. The application programs for such systems are typically very large and complex, such as those for mission-critical military tasks. Thus, two problems facing system designers are

(1) the development of system routines to efficiently control system reconfiguration, and
(2) the writing of complex application programs.

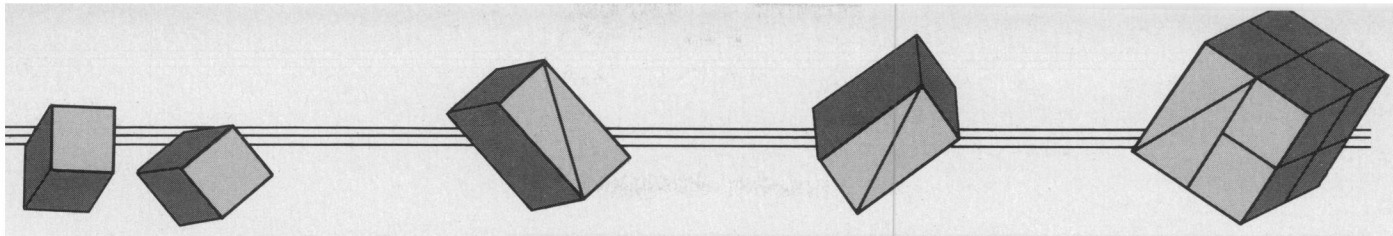
Software used to control and program reconfigurable supersystems must efficiently exploit the hardware flexibility available. If not, then the system does not fulfill its potential. Frequently in the past, customized application software packages were developed for every new computer, often resulting in programs that could be executed on only one type of machine. This machine dependence leads to duplication of effort since the same algorithms have to be coded repeatedly, thus increasing software cost. It is therefore important to look for solutions to these two problems in order to make the efficient use of reconfigurable supersystems practical. We consider possible solutions to these problems.

Adaptable software

Adaptable software offers such a solution. It encompasses all software that manages reconfigurable computer systems and all software itself adaptable to various computers. As noted in the Guest Editors' Introduction, compiler and operating system software used to manage a reconfigurable computer will be termed *reconfiguration software*. Machine-independent software is termed *retargetable software*.

Reconfiguration software is essential for supersystems with reconfigurable ar-

0018-9162/86/0200-0040$01.00 © 1986 IEEE



chitecture. The actual reconfiguration of the hardware must be handled by the operating system of a reconfigurable system to shield the user from details of the hardware implementation and to avoid erroneous hardware settings. The operating system can be directed to perform reconfiguration either explicitly by the user or implicitly by software tools that analyze a given problem automatically, determine the optimum hardware configuration for the problem, and supervise program execution. The explicit approach gives the programmer flexibility in choosing the system configuration needed for a particular algorithm, while the implicit approach reduces programming effort and speeds up program development. Kartashev and Kartashev describe a system that implements the implicit approach for Fortran programs.[6] However, currently no automatic approach exists for all levels of algorithm design and execution. In fact, an exclusively automatic approach may not be desirable for all computer architectures and problems. In some cases explicit reconfiguration enhances efficiency. For example, many highly efficient algorithms for specific computational tasks and specific computer architectures have been developed in the past. The efficiency and speed of such hand-honed algorithms cannot be rivaled by automatically generated programs. For such cases, it is desirable for an adaptable system to assume the architecture for which such an algorithm was designed. To do this, it must provide the user with the tools to easily execute this algorithm on the reconfigurable system.

The life cycle of software can be prolonged if it is made retargetable. Only programs that do not use machine-specific instructions will be retargetable, since any machine-specific instruction needs to be changed when the program is ported to a different computer. Thus, only programs written in high-level languages will be retargetable.

Similar problems arise for the transportation of programs that use explicit operating system calls. If a program uses explicit operating system calls (which are inherently machine dependent), it is not retargetable. Furthermore, if the operating system is changed or exchanged, the program will have to be rewritten or at least modified to still execute on the same machine. Machine-independent software can survive hardware or operating system changes more easily. The elimination of changes due to hardware or operating system modifications automatically decreases the overall cost of a software system. If a program is portable, it can be used on a wide variety of computers, which eliminates duplicate programming efforts and further reduces cost. Since a machine-independent program is expected to remain useful longer, it is feasible to put more development effort into a program. This results in better, more efficient programs.

We have introduced two different types of adaptable software: retargetable and reconfiguration software. Most reconfiguration software will not be retargetable since it has to be tailored towards machine-specific capabilities. In some cases, components of a reconfiguration software system could be retargetable, such as a reconfiguration control strategy coded in a high level language. Thus, reconfiguration and retargetable software may overlap.

Reconfiguration software

The reconfigurability of a supercomputer can take many different forms. Such a computer could consist of a number of processors with a certain numeric precision. It might be possible to combine several processors to achieve higher precision, such as the reconfigurable varistructure array processor,[7] DCA,[8,9] and TRAC.[10] A supersystem could be switched between the single instruction stream—multiple data stream, or SIMD, and multiple instruction stream—multiple data stream, or MIMD, modes of operation.[11] It could also be partitionable and able to form subgroups of processors. The subgroups could operate in SIMD or MIMD mode, such as PASM.[12] A supercomputer might have even more states, such as array processor, pipeline computer, and multiprocessor, such as DCA[13] and Reddi's and Feustel's machine.[14] The reconfiguration capability might be static, so that the system has to be stopped in order to perform reconfiguration. Alternatively, it could be dynamic, so that architectural states could be switched during program execution. Clearly, static reconfiguration has limited use in mission-critical supersystems since these must perform their tasks at high speed and without interruptions.

The processors of a supersystem must communicate with each other to share data and results. This communication can be handled through shared memory or through processor-to-processor interconnection hardware.[15] In shared-memory systems the processors may be connected to the shared memories through an interconnection network. Examples include C.mmp (which used a crossbar network),[16] CHoPP (which proposed a hypercube structure),[17] and the NYU Ultracomputer (which will use a multistage network).[18] For processor-processor communication a variety of types of networks may also be employed. Hardware interconnections can be fixed as the non-edge connections in the MPP,[3,4] where processors communicate only with their four nearest neighbors. If a processor needs to talk to a processor not its nearest neighbor, multiple data transfers are necessary. Reconfigurable processor-to-processor interconnection networks can allow communication among a large subset or all of the processors, as in PASM,[12] where processors use a multistage cube network. Software running these systems must make efficient use of the interconnection capabilities.

Thus, to take optimal advantage of the system, reconfiguration software needs to know about the hardware's capabilities. It must therefore be tailored towards the hardware on which it will run. If the reconfiguration software is retargetable, the machine-specific hardware description would have to be entered at compile or run time, since it could not be part of the retargetable software.

Reconfiguration of a supersystem can be handled by the programmer through explicit use of reconfiguring instructions, or implicitly by system software which automatically generates reconfiguring instructions without the need of programmer intervention. Hybrids of these methods are also possible.

Explicit reconfiguration. If instructions for reconfiguring the system are available to the user, the programmer can take explicit advantage of various architectural states. With careful program analysis and good coding, this results in programs gaining the most advantage from the reconfiguration capability. Programs for each

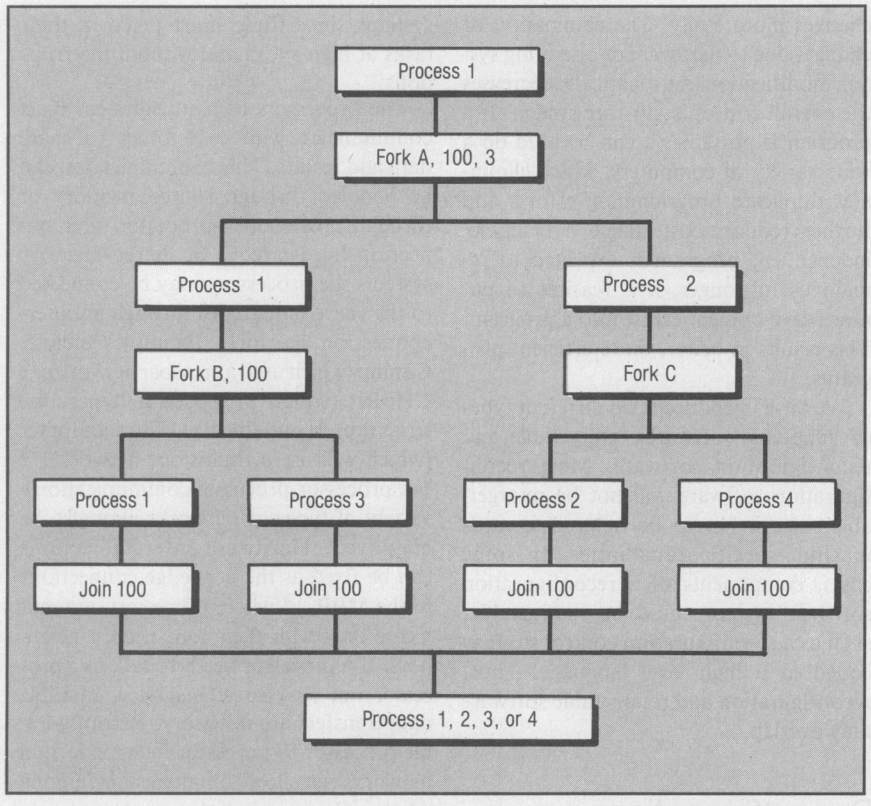


Figure 1. Flowchart for a parallel program using **Fork** and **Join** statements.

architectural state like array, multiprocessor, pipeline, etc., must be written in a language suited for this state, and the language must include constructs for reconfiguration. High level languages for MIMD and SIMD machines give good examples of the specific requirements. Consider an MIMD computer. If the number of processors for the given task is fixed, a standard assembly or high level language such as Pascal, C, or Fortran can be used to program the task. The programmer has to decide which processor handles what particular task, and write the appropriate programs. The programs are loaded into the processors needed for the task. The operating system of the machine starts the task and coordinates activities. Many applications, especially in image processing, require a set of processors that all execute the same program. An example is an image processing algorithm in which each processor finds features in a part of the image. In this case, each processor has its own set of data, but all processors execute the same program. Only one program has to be coded, independent of the number of processors. Depending on the task, this may involve just MIMD operations, just SIMD operations, or both.[19]

If the number of processors varies during task execution, the programming language has to be expanded and must include special statements to allocate and deallocate processors. Conway[20] proposes the use of **Fork** and **Join** instructions. Whenever a **Fork** is encountered, a new process is created that proceeds independently from the original process. The new process can run on another processor. If no processor is available, the new process can be queued until a processor becomes available. If multiprogramming capability is available, such as CHoPP,[17] two or more processes can run on the same processor in a timesliced fashion. Conway proposes three kinds of **Fork: Fork A, J, N; Fork A, J;** and **Fork A.** In this implementation, a counter keeps track of the number of concurrent processes. **Fork A** generates a new process; the old process continues to execute the instruction following the **Fork**, the new process starts at instruction **A**. The **Fork A, J, N** instruction sets a counter in memory location J to the value N, then executes **Fork A**. The programmer must match the number of **Forks** in his program with the count specified in the **Fork A, J, N**, since the **Fork A** alone does not affect the counter. **Fork A, J** increments counter J by one and then executes a **Fork A**. This instruction is needed if the decision to fork is made at run time. The **Join** statement merges concurrent instruction streams. When a **Join J** is encountered, the counter at location J is decremented. The processor that decrements the counter to zero executes the statements following the **Join**. All other processors are released and become available for other computations.

Figure 1 illustrates the use of the three different **Fork** and the **Join** statements. Process 1 executes a **Fork A, 100, 3,** which sets the counter in memory location 100 to 3 and starts a concurrent process (Process 2). Then Process 1 executes a **Fork B, 100,** which increments the counter at location 100 by one and starts another process, Process 3. Meanwhile, Process 2 executes a **Fork C**, generating a fourth process, Process 4. Thus, four concurrent processes are now active. When a process encounters a **Join** 100, it decrements the counter. The process continues if the counter reaches zero; otherwise, it ceases. **Fork** and **Join** are examples of two explicit reconfiguration commands.

Dijkstra[21] proposed a more structured way to express reconfiguration for multiprocessor programs: **parbegin** *S1;S2;* .... *Sn*; defines *S1* through *Sn* as statements which can be executed in parallel. The parallel execution ends when a **parend** is found. Only after all processors executing the parallel statements have found their **parend** are statements following the parend executed. A program that needs to run two processes in parallel might have the following structure:

```
parbegin
process1 : begin
    statements for process 1
    end
process2 : begin
    statements for process 2
    end
parend
```

Both **parbegin-parend** and **Fork-Join** blocks could be conditional. In other words, whether or not the block would be entered could depend on the result of an **if**

statement. This results in a dynamic reconfiguration capability.

Parallelism of an SIMD computer cannot be expressed explicitly in any conventional programming language. One type of SIMD computer consists of an array of processor/memory pairs called processing elements, or PEs, which do the actual computations, and a control unit. The control unit executes all program flow instructions and broadcasts data processing instructions to the PEs. All PEs execute the same instruction at the same time, but on different data, e.g., on different sections of an image. The control unit also determines which of the PEs execute the instruction and thereby varies the extent of parallelism of the computer. An interconnection network provides communication between the PEs. The network can be fixed as in the MPP or the Illiac IV,[22] or it can be reconfigurable as in PASM.[12] If the network is reconfigurable, instructions that reconfigure the network in the desired fashion must be provided in the programming language. Constructs to enable and disable PEs need to be available as well. Extensions of existing high level languages have been proposed.[23-30] Such extensions can be machine dependent or machine independent.

An example of a machine-dependent language is Glypnir,[24] specifically developed for the Illiac IV computer and based on Algol 60. Illiac IV was an SIMD machine with 64 PEs, designed in the 1960's. Its fixed interconnection network connects PE i to PEs $i+1, i-1, i+8, i-8$, all modulo 64, providing a mesh-type interconnection. The system cannot be partitioned into smaller machines; the only reconfiguration capability involves enabling and disabling PEs. In Glypnir, variables for PEs and for the control unit can be defined. A PE variable has an element in every PE and is called a super word, or *sword*. Therefore, a sword always has 64 elements. For example, the statement

```
PE REAL A,B
```

declares the variables *A* and *B* to be swords.

```
PE REAL C(20)
```

declares *C* to be an array of 20 elements in each of the PEs. By using Boolean expressions, PEs can be selectively enabled or disabled. Consider, for example, the statement

```
if <Boolean expression> then
   <statement1>
else
   <statement2>
```

The Boolean expression will be evaluated in all processors. The processors finding a true value execute statement1. The others do not participate in the instruction stream broadcast for statement1, but execute only statement2.

Glypnir's architectural dependence on the Illiac IV severely limits its use. An attempt for machine independence was made by the developers of Actus,[26] a language based on Pascal. Actus was designed for SIMD machines of arbitrary size. A variable can be declared parallel. The extent of parallelism must be specified for each parallel variable. In the range specification of a parallel variable declaration, a colon denotes that a dimension is to be accessed in parallel. For example,

```
var array1: array [1:m,1..n] of integer
```

declares an array of *m* by *n* elements; *m* elements can be accessed at a time. The underlying operating system takes care of reconfiguring the computer into a machine that can handle the specified parallel variable. **if, case, while,** and **for** statements are expanded for use with parallel variables. Alignment operators are also provided. For example, a **shift** operator moves data within the declared range of parallelism, a **rotate** shifts data circularly with respect to the extent of parallelism. **shift** and **rotate** will be implemented by using the processor interconnection network.

Parallel-C is a proposed programming language based on C that handles both the MIMD and SIMD modes of operation.[30] The SIMD features include declarations of parallel variables and functions; an indexing scheme to access and manipulate parallel variables; expressions involving parallel variables; extended control structures using parallel variables as control variables; and functions for PE allocation, data alignment, and I/O. The language supports simple parallel data types like scalars and arrays as well as structured data. For example,

```
parallel [N] int a;
parallel [N] char line[MAXLINE];
struct node {
   char *word; struct node *next;
} parallel [N] nodespace[100], *head;
```

declares an integer "a," an array of MAXLINE characters, an array of 100 nodes, and a node pointer for each of the N processors. Indexing along the parallel dimension can be done by using selectors; they enable or disable subsets of the N processors. Functions to send data between processors are provided. They are not part of the language, but library routines, thus avoiding machine dependence. If the compiler is retargeted, these communication routines must be adapted. No change of the compiler itself is required. To facilitate MIMD mode, a few new features and keywords are added. A preprocessor recognizes these constructs and translates the parallel algorithm into a standard serial C program, which can then be compiled by a standard C compiler and loaded into the program memories of the processors of the MIMD machine.

Implicit reconfiguration. On a supercomputer with a fixed architecture, it is desirable to have the ability to automatically analyze a serial program and restructure it so that it can be executed efficiently. As reconfigurable supercomputers become more and more complex, having users use explicit reconfiguration commands to construct efficient algorithms for the system may no longer be feasible. On a machine with reconfigurable architecture, it is desirable to automate the process of finding an optimal architectural state and then structuring the task for the architecture. This may involve more than one architec-

A task may be most efficiently performed using different architectural states for different subtasks.

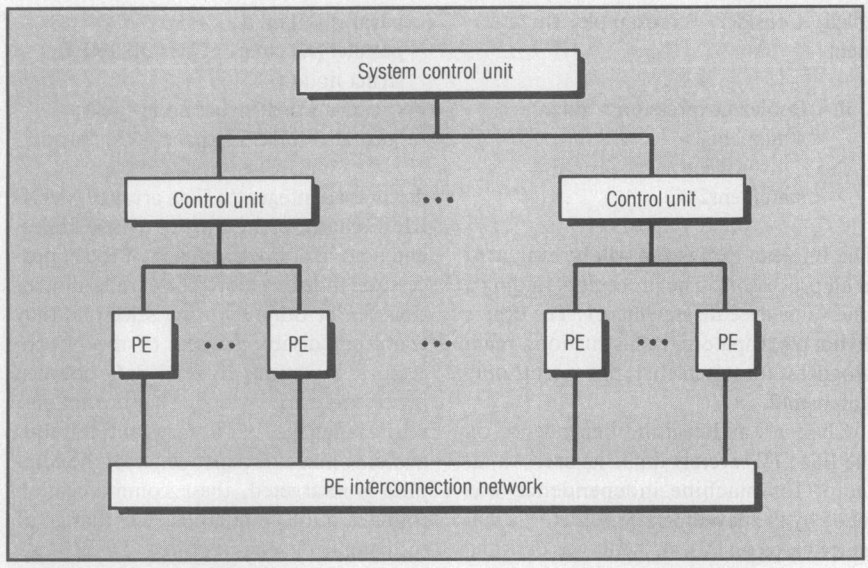


Figure 2 Simplified block diagram of PASM.

tural state, because a task may be most efficiently performed using different architectural states for different subtasks.

Much effort has gone into developing algorithms capable of detecting parallelism in serial programs.[31,32] Consider a program to add to vectors A and B:

FOR i=1 **TO** N **DO** C[i] := A[i] + B[i];

Obviously, the addition statements do not depend on each other; therefore, all additions can be executed in parallel on N processors. This example typifies the operations performed by a parallelizing compiler. The compiler analyzes the program and schedules statements that do not depend on each other to exeecute in parallel. It must also determine the number of processors needed to execute the program and have the operating system allocate them. Since the order in which two statements, I and J, execute in a sequential program is not preserved if they are executed in parallel, three conditions must be met if I and J are to be parallelized. Statement I must not write to locations that statement J reads; statement J must not write to locations that statement I reads; and statement I must not write to the same location as statement J if that location is used by a later statement.

One approach to handle a system with reconfigurable architecture, developed by Kartashev,[6] is called the reconfiguration flowchart. The algorithm assumes a dynamic architecture that can partition its processors into groups of computers of different sizes and can combine processors to form higher-precision computers. It analyzes a Fortran program and its program flow structure and constructs a program graph. Each node in the graph includes one or more program statements. The arrows between nodes represent the program flow. For each node, the necessary computational precision is determined and by that the minimum number of processors required for the computation of the node is found. Since a minimum number of processors is found for each node, a maximum number of nodes can be processed simultaneously.

Several programs can be run concurrently on the system. Whenever a program finishes, a new program should be started, necessitating a switch to a new architectural state. If the task causing the switch runs for a longer time than other tasks, the processors executing the short tasks will wait for the long task to finish. Therefore, the execution time for each node is determined to minimize processor idle time. Then a resource diagram is constructed, where nodes are assigned specific processors, and execution can begin.

Ongoing research at Purdue involves the design of expert-system-based intelligent operating systems for controlling reconfigurable supersystems.[33] To optimize system performance, the intelligent operating system uses information about which subtasks need to be executed to perform an overall task, the performance/system-requirement characteristics of the algorithms for the subtasks, and the current state of the system.

Reconfiguration methodology. An important consideration in reconfigurable systems is the reconfiguration methodology. The switch from one architectural state to another takes time. To maximize performance, the sum of execution time and reconfiguration time must be minimized. Clearly, a very short reconfiguration time is desirable, since otherwise advantages gained by the reconfiguration are lost due to the reconfiguration overhead. Of course, the time to reconfigure has to be less than (hopefully, much less than) the reduction in execution time gained through the reconfiguration.

As an example of a reconfiguration methodology, consider the PASM system. PASM is a partitionable SIMD/MIMD system. A prototype with 16 Motorola MC68000-based PEs in the computational engine and four control units is being constructed at Purdue University.[34] Figure 2 shows a simplified block diagram of the system. In SIMD mode, the PEs receive their instructions from a control unit and process data from their private memory. In MIMD mode, the PEs have data and program in their private memory. One type of reconfiguration is the switching from SIMD to MIMD mode and vice versa. Whenever a PE addresses a certain address range, AR, this PE is in SIMD mode. As soon as an access to AR is detected, an instruction request is issued to the control unit, which then broadcasts an instruction to the requesting processors. The control unit accomplishes a switch from SIMD to MIMD mode by broadcasting an unconditional jump to the beginning of the MIMD program to the PEs, where the address of this MIMD program is outside the range AR. The PEs can return to SIMD mode by jumping into the address space AR. The reconfiguration overhead for PASM is therefore just one instruction. For all practical applications, this overhead is negligible and program sections can be executed in SIMD or MIMD mode, whichever is best suited.

Apart from reconfiguration for optimum performance, reconfiguration due to faults or other undesired conditions

such as unbalanced load must be considered in a supersystem. If a single processing element of a supersystem fails, the system as a whole should continue to function, eventually with decreased performance. The necessary reconfiguration has to be automatic and must be handled by the operating system, since a programmer cannot predict a fault. Ma[35] proposes reconfiguration control algorithms for reconfigurable computers with and without centralized control. As an example for the methodology, consider the straightforward reconfiguration procedure with centralized control. The control unit monitors the state of the processing nodes and detects undesirable conditions. It then broadcasts reconfiguration commands to the appropriate nodes, which perform the necessary reconfiguration and signal completion to the control unit. After all nodes have completed their respective reconfiguration tasks, the control unit signals the nodes and computation can resume.

An additional consideration for reconfiguration software is the ability to concentrate computing power.[33] For example, assume that numerous subtasks of some overall task are being executed concurrently. If, as a result of some derived intermediate result, one subtask becomes more time-critical than the others, the system should dynamically reconfigure to provide additional resources to the subtask. Reconfiguration software capable of such operation would have to have some form of intelligence and knowledge of a system model. This is an area for future research.

Retargetable software

As mentioned previously, software that can be used on a wide variety of computers is very desirable. An early attempt towards that goal is the standardization of the programming language Fortran by ANSI. Nevertheless, many Fortran dialects exist, and compilers are not verified. Therefore, it cannot be guaranteed that the same program will exhibit identical performance on different machines. A standard Fortran program using only standard I/O routines like **read** and **write** might very well be portable. As soon as explicit operating system calls such as **seek** for a disk are included, however, the program is restricted to machines with compatible operating system calls.

Other important tasks that depend on the operating system are task creation and interprocess communication. They are necessary for expressing concurrent operations. Concurrent operations are often used in real-time systems and are essential for parallel processing. Low-level operations like interrupt control are not possible in Fortran. All this seriously limits the portability of Fortran programs. The same problems arise with other languages, such as Pascal and Algol. Current efforts on the design of Ada[36,37] are attempting to circumvent these problems.

There are basically three ways to make a program portable.[38]

Transportable language approach. This approach, designed for serial computers, is based on widely used languages like Fortran and Pascal. The dialects of such a language are analyzed and a hypothetical parent language with a grammar common to all dialects is defined. The syntax and semantics of the parent language are rigorously specified. For each computer which is to run portable code, parameters which completely specify the computer architecture are determined. These parameters are byte size, word size, byte or word orientation of the CPU, main memory size, and maximum program module size. Then a compiler with two modes of operation is constructed: the transportability testing mode and the compilation mode. In the transportability testing mode, a program coded in a dialect of the language is converted to the hypothetical parent language, using the architectural parameters. In the compilation mode, hypothetical parent-language code is translated into a dialect of the language. Thus, existing programs can be converted to the parent language, making them portable. Also, programs can be written directly in the parent language and thus be guaranteed easily portable.

Certain instructions must be modified for the parent language. For example, all variable declarations need to include a precision declaration so that the parent language compiler can determine the appropriate data type in the language dialect. Desirable language features not available in a given dialect (such as data structures or **while** loops) can be made available in the parent language, thus making programming in the parent language more flexible.

A major advantage of the transportable language approach is that existing programs are not rendered obsolete.

A major advantage of the transportable language approach is that existing programs are not rendered obsolete but can be converted and made portable. The necessity to write bifunctional compilers for every machine, and the continued existence of multiple dialects for the language are disadvantages.

Emulation approach. Another approach to transportable software is to transform the machine rather than the program. Making machines capable of emulating some class of computers allows those machines to execute code written for any of the computers in that class.[39,40] Programs can be coded in any high-level or assembly language supported by the computers in that class. The programs are compiled and the resulting machine code is transferred to one of the emulation machines. This machine emulates the computer for which the code was written and thus is capable of executing the machine code. One problem with the approach is decreased performance. Another is that all machines, of a set of machines among which code is to be transportable, have to be capable of emulating all the others.

New language approach. In this approach, all programs to be ported are written in a standard new language known by all machines, thus making programs portable. The definition of this new language incorporates all desired features and the compilers for this language are rigorously verified to ensure portability. The Department of Defense chose this method when

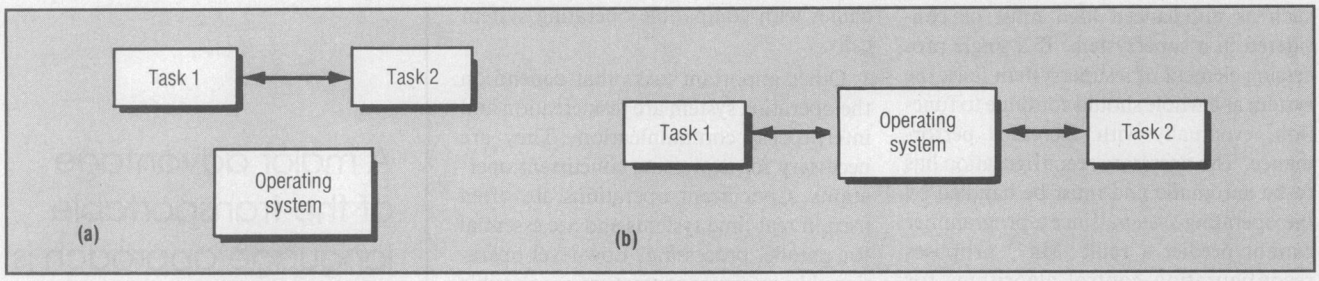


Figure 3. (a) User's view of task communication in Ada. (b) User's view of task communication in a standard programming language.

developing the programming language Ada.[37,41,42]

Ada is a structured programming language. In addition to being designed for portability, it supports many features necessary for executing tasks on reconfigurable systems. Functions necessary for process creation, deletion, and interprocess communication are part of the language. The compiler actually generates operating system calls from those functions, but this is hidden from the user. If the program is transferred to a different machine and recompiled, the appropriate new operating system calls are generated.

Depending on the machine architecture, concurrent processes could either be run in a time-sliced mode, or they could be allocated on different processors. The compiler for a particular machine, in cooperation with the operating system, has to handle the allocation procedure.

Concurrent processes are called tasks in Ada. A task is declared by defining a task header and a task body. If one or more tasks are declared inside a procedure, they execute concurrently with the procedure. The task header specifies the task name and entries that handle interprocess communication. The task body contains all statements executed in the task. To send a message to task T, other tasks call an entry of T like a procedure. The destination task T can accept the message.

As an example, consider a process that accepts single-character messages from other tasks. The header has the form

```
task ACCEPT_CHAR is
  entry MESSAGE ( C: in
  CHARACTER)
end;
```

The task name is defined as ACCEPT_CHAR. Other processes can send messages to ACCEPT_CHAR calling the entry:

```
MESSAGE (CHAR);
```

The task ACCEPT_CHAR reads the message when it encounters an **accept** statement, which must be part of the task body:

```
accept MESSAGE (C: in
  CHARACTER) do
  MESS : = C
  end;
```

This statement accepts an incoming message and assigns its value to the variable MESS. Only between the **do-end** part of the **accept** is the passed message accessible. Data is actually exchanged only when the sending task calls an entry and the receiving task executes an **accept** on the same entry. If a process encounters an **accept** without another process executing the appropriate entry call, the process will wait until an entry call is executed. An entry call also waits for the appropriate **accept**. Thus, sending and accepting of messages is synchronized between tasks.

The execution of an entry call and its associated **accept** statement is called *task rendezvous*. The task rendezvous facilitates interprocess communication by the ability to exchange data. It facilitates process synchronization by executing the task only if both the sending and receiving tasks have reached their respective rendezvous statements. It facilitates mutual exclusion since only one of all tasks requesting communication with another process will be served at any given time. The difference between an Ada task rendezvous and task communication in other programming languages is illustrated in Figure 3. Since no explicit operating system calls are needed, the task handling is not machine specific and can easily be ported.

Another important portability problem with programming languages arises from machine-dependent data types. A Cray-1 supercomputer,[2] for example, has an integer precision of 24 bits, whereas the Fujitsu VP-200[43] uses 32-bit integers. Programs utilizing 32-bit integer precision could not run on a machine of 16-bit integer precision. In Ada, a programmer has access to such machine-dependent datatype attributes as

```
INTEGER_SIZE
```

Using such attributes, a program can check at runtime whether a specific machine will be able to execute a program as desired.

The task concept of Ada can be used to explicitly handle parallel processing in MIMD multiprocessors. This is insufficient to express parallelism of an SIMD machine. Extensions to Ada facilitating SIMD programming have been proposed[29]; these extensions provide functions similar to those for SIMD programming as described above. Due to the rigid Ada policy allowing no subsets or supersets of the language in a verified compiler, this approach may not be widely used. To code SIMD programs explicitly nevertheless, subprograms coded in a language suitable for SIMD programming, such as Actus or extended Ada, could replace a standard Ada subprogram if the program is executed on an SIMD machine. This procedure is analogous to replacing a high-level language subroutine by a faster executing assembly program. The main body of the program would then still be portable. Only the special subroutine has to be changed when the program is retargeted.

As more and more adaptable supersystems are designed and built, the need for adaptable software to run these systems efficiently will continue to grow. The adaptable software required includes retargetable software, which makes application programs transportable, and reconfiguration software, which controls the system organization. Thus, adaptable software is an essential element in the development of supersystems or mission-critical objectives. □

Acknowledgments

The authors would like to thank Steven and Svetlana Kartashev and Andre van Tilborg for their careful reading of the manuscript and many helpful suggestions.

This work was supported by the Rome Air Development Center under contract number F30602-83-K-0119.

References

1. R. M. Russell, "The Cray-1 Computer System," *Comm. ACM,* Jan. 1978, pp. 63-72.
2. E. W. Kozdrowicki and D. J. Theis, "Second Generation of Vector Supercomputers," *Computer,* Nov. 1980, pp. 71-83.
3. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. Computers,* Vol. C-29, Sept. 1980, pp. 836-844.
4. K. E. Batcher, "Bit Serial Parallel Processing Systems," *IEEE Trans. Computers,* Vol. C-31, May 1982, pp. 377-384.
5. E. W. Martin, "Strategy for a DoD Software Initiative," *Computer,* Vol. 16, Mar. 1983, pp. 52-59.
6. S. P. Kartashev and S. I. Kartashev, "Distribution of Programs for a System with Dynamic Architecture," *IEEE Trans. Computers,* Vol. C-31, June 1982, pp. 488-514.
7. G. J. Lipovski and A. R. Tripathi, "A Reconfigurable Varistructure Array Processor," *1977 Intl. Conf. Parallel Processing,* Aug. 1977, pp. 165-174.
8. S. I. Kartashev and S. P. Kartashev, "Dynamic Architectures: Problems and Solutions," *Computer,* July 1978, pp. 26-42.
9. S. I. Kartashev and S. P. Kartashev, "A Multicomputer System with Dynamic Architecture," *IEEE Trans. Computers,* Vol. C-28, Oct. 1979, pp. 704-720.
10. G. J. Lipovski, "The Banyan Switch in TRAC the Texas Reconfigurable Array Computer," *Distributed Processing Technical Committee Newsletter (IEEE Computer Society),* Jan. 1984, pp. 13- 26.
11. J. Keng and K-S. Fu, "A Special Computer Architecture for Image Processing," *1978 IEEE Computer Society Conf. Pattern Recognition and Image Processing,* June 1978, pp. 287-290.
12. H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Trans. Computers,* Vol. C-30, Dec. 1981, pp. 934-947.
13. S. I. Kartashev and S. P. Kartashev, "Problems of Designing Supersystems with Dynamic Architecture," *IEEE Trans. Computers,* Vol. C-29, Dec. 1980, pp. 1114-1132.
14. S. S. Reddi and E. A. Feustel, "A Restructurable Computer System," *IEEE Trans. Computers,* Jan. 1978, pp. 1-20.
15. H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, D. C. Heath and Co., Lexington, MA, 1985.
16. W. Wulf and C. Bell, "C.mmp—A Multi-Miniprocessor," *AFIPS 1972 Fall Joint Computer Conf.,* Dec. 1972, pp. 765-777.
17. H. Sullivan, T. R. Bashkow, and K. Klappholz, "A Large-Scale Homogeneous, Fully Distributed Parallel Machine," *Fourth Ann. Symp. Computer Architecture,* Mar. 1977, pp. 105-124.
18. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing an MIMD Shared-Memory Parallel Computer," *IEEE Trans. Computers,* Vol. C-32, Feb. 1983, pp. 175-189.
19. D. L. Tuomenoksa, G. B. Adams III, H. J. Siegel, and O. R. Mitchell, "A Parallel Algorithm for Contour Extraction: Advantages and Architectural Implications," *1983 IEEE Computer Society Symp. Computer Vision and Pattern Recognition,* June 1983, pp. 336-344.
20. M. E. Conway, "A Multiprocessor System Design," *AFIPS 1963 Fall Joint Computer Conf.,* 1963, pp. 139-146.
21. E. W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages,* ed. F. Genuys, Academic Press, New York, NY, 1968, pp. 43-112.
22. W. J. Bouknight, S. A. Denenberg, D. E. McIntryre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV System," *Proc. IEEE,* Vol. 60, Apr. 1972, pp. 369-388.
23. K. G. Stevens, "CFD—A Fortran-like language for the Illiac IV," *ACM Conf. Programming Languages and Compilers for Parallel and Vector Machines,* Mar. 1975, pp. 72-76.
24. D. H. Lawrie, T. Layman, D. Baer, and J. M. Randall, "Glypnir—A Programming Language for Illiac IV," *Comm. ACM,* Vol. 18, Mar. 1975, pp. 157-164.
25. S. F. Reddaway, "DAP—A Distributed Array Processor," *1st Ann. Symp. Computer Architecture,* Dec. 1973, pp. 61-65.
26. R. H. Perrott, "A Language for Array and Vector Processors," *ACM Trans. Programming Languages and Systems,* Vol. 1, Oct. 1979, pp. 177-195.
27. L. Uhr, "A Language for Parallel Processing of Arrays, Embedded in Pascal," *Languages and Architectures for Image Processing,* eds. M. J. B. Duff and S. Levialdi, Academic Press, London, England, 1981, pp. 53-88.
28. A. P. Reeves and J. D. Bruner, "The Programming Language Parallel Pascal," *1980 Int'l Conf. Parallel Processing,* Aug. 1980, pp. 5-7.
29. C. Cline and H. J. Siegel, "Augmenting Ada for SIMD Parallel Processing," *IEEE Trans. Software Engineering,* Vol. SE-11, No. 9, Sept. 1985, pp. 970-977.
30. J. T. Kuehn and H. J. Siegel, "Extensions to the C Programming Language for SIMD/MIMD Parallelism," *1985 Int'l Conf. Parallel Processing,* Aug. 1985, pp. 232-235.

31. D. J. Kuck and D. A. Padua, "High-Speed Multiprocessors and Their Compilers," *1979 Int'l Conf. Parallel Processing,* Aug. 1979, pp. 5-16.
32. Arvind, "Decomposing a Program for Multiple Processor Systems," *1979 Int'l Conf. Parallel Processing,* Aug. 1979, pp. 7-14.
33. E. J. Delp, H. J. Siegel, A. Whinston, and L. H. Jamieson, "An Intelligent Operating System for Executing Image Understanding Tasks on a Reconfigurable Parallel Architecture," *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Mangement,* Nov. 1985, pp. 217-224.
34. H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "PASM: A Reconfigurable Parallel System for Image Processing," *Workshop on Algorithm-guided Parallel Architectures for Automatic Target Recognition,* July 1984, pp. 263-291. (Also appears in the ACM SIGARCH newsletter, *Computer Architecture News,* Vol. 12, No. 4, Sept. 1984, pp. 7-19.)
35. Y. W. Ma, "Reconfiguration Control Algorithms for Reconfigurable Computer Systems," *Int'l Computer Software and Applications Conf.,* Nov. 1982, pp. 70-77.
36. J. G. P. Barnes, *Programming in Ada,* Addison-Wesley, London, England, 1982.
37. U.S. Department of Defense, *Reference Manual for the Ada Programming Language,* Washington, D.C., 1982.
38. P. A. D. de Maine and S. Leong, "Transportation of Programs," *15th Southeast Symp. System Theory,* Mar. 1983, pp. 158-164.
39. T. A. Marsland and J. C. Demco, "A Case Study of Computer Emulation," *Canadian J. Operational Research and Information Processing,* Vol. 16, June 1978, pp. 112-131.
40. J. Hayes, *Computer Architecture and Organization,* McGraw-Hill, New York, NY, 1978.
41. H. Ledgard, *Ada, An Introduction,* Springer-Verlag, New York, NY, 1983.
42. R. J. A. Buhr, *System Design with Ada,* Prentice Hall, Englewood City, NJ, 1984.
43. K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, NY, 1984.

Thomas Schwederski is currently working toward a PhD degree at Purdue University. He received the Diplom-Ingenieur degree in electrical engineering from Ruhr-Universitaet Bochum in 1983, and the MS degree in electrical engineering from Purdue University in 1985. His research interests include computer architecture, parallel processing, multimicroprocessing systems, and VLSI design.

Howard Jay Siegel is a professor of electrical engineering and Director of the PASM Parallel Processing Laboratory at Purdue University, where he has been involved in research and teaching since 1976. His research interests include parallel/distributed processing, computer architecture, and image and speech understanding.

Siegel received BS degrees in electrical engineering and management from the Massachusetts Institute of Technology in 1972. He received the MA and MSE degrees in 1974, and the PhD degree in 1977, all three in electrical engineering and computer science, from Princeton University.

Siegel has served as an IEEE Computer Society distinguished visitor and is currently an associate editor of the *Journal of Parallel and Distributed Computing*.

Questions regarding this article can be directed to either author at the PASM Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.