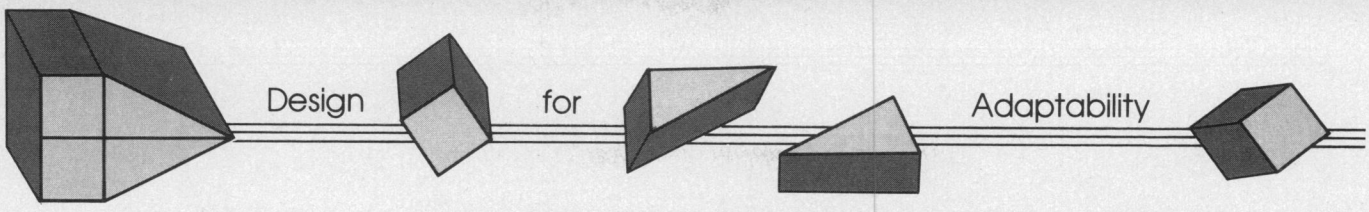

Adaptable Software for Dynamic Architectures

Svetlana P. Kartashev and Steven I. Kartashev
Dynamic Computer Architecture, Inc.

Reconfiguration software takes advantage of performance gains from reconfiguration. Retargetable software allays huge software development costs. Both are adaptable.

Generally, a supercomputing system in which performance is based on successful execution of a mission as specified by quantitative parameters is called a *mission-critical supercomputing system.*[1] Complex real-time missions can undergo rapid or continuous changes during the supercomputing system life cycle, or they can be fixed. The first type of mission requires that the supercomputing system's architecture be able to adapt to the changing algorithms in order to compute them within the times needed; we call these supercomputing systems *adaptable.* The second type of mission is most effectively solved by *dedicated* (real-time) supercomputing systems.[2-5]

Both types of supercomputing system require adaptable software. However, the types of adaptations to be performed by adaptable software depend on the supercomputing system type.

For adaptable supercomputing systems, the main function of their software is to take advantage of the additional performance gains achievable by reconfiguration. Therefore, this software must carry out automatic changes in architectural configurations during program computations. We will call this software *reconfiguration software.*

For dedicated supercomputing systems, their adaptable software must serve not a single but multiple and dedicated (target) supercomputing systems. In other words, the software must be *retargetable* and portable (modular) in order to allay huge software development costs introduced into the software development of every dedicated supercomputing system.[1]

In general, a dynamic architecture possesses the following attributes[3,6-10]:

- it is modular;
- it is dynamically reconfigurable;
- it is capable of configuring into a variable number of computers with selectable word sizes;
- it can assume one of the following architectural types:
  —multicomputer/multiprocessor,
  —array,
  —pipeline,
  —network, and
  —mixed,
  where the latter means coresidence of any combination of the architectures indicated above; and
- it makes transitions, through software, from one architectural type to another with the execution of a single program instruction.

As seen, a dynamic architecture forms a subclass of reconfigurable architectures. They can make fast architectural reconfigurations with the execution of only one program instruction and carry out com-

0018-9162/86/0200-0061$01.00

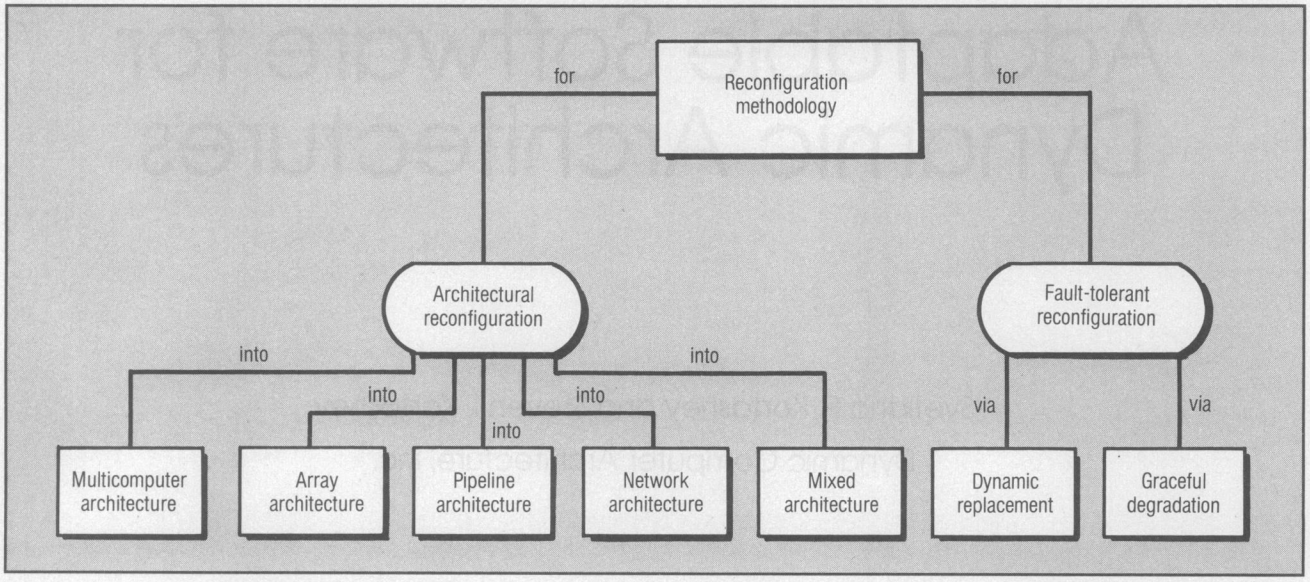

Figure 1. Reconfiguration methodology.

plete resource partitioning into different subsystems with variable parameters during each system reconfiguration (variable pipelines, arrays, multicomputers, networks, and mixed). A number of scientists have described notable representatives of reconfigurable systems.[6-9]

Reconfiguration software for dynamic architectures performs the following functions*:

(1) It finds the flowchart of architectural configurations (states) assumed in execution (called the *reconfiguration flowchart*); and
(2) it organizes fast transition from one state to another in the reconfiguration flowchart.

Therefore, reconfiguration software includes two sets of tools aimed respectively at constructing the *reconfiguration flowchart* and developing a comprehensive set of various system reconfiguration algorithms from one architectural state to another, otherwise referred to as *reconfiguration methodology*.

*These techniques are now under development at DCA, Inc. as part of the research project aimed at developing the principal software tools for dynamic architecture.

Reconfiguration methodology

Reconfiguration methodology can be divided into

(1) *Architectural reconfiguration* from one architectural state to another, and
(2) *fault-tolerant reconfiguration* to purge faulty modules from the system (see Figure 1).

Architectural reconfiguration. In general, any architectural reconfiguration of a set of reconfigurable hardware resources must be supervised by the local monitor interconnected with these resources, since this monitor must find the moment of time when such reconfiguration is possible and should resolve the conflicts among programs with conflicting reconfiguration requests.

If an adaptable supercomputing system, AS, is divided into several (P) subsystems, S_i ($i=1,\ldots, P$), each such subsystem S_i must be provided with the local monitor M_i. Each M_i must be connected with the unique global monitor, GM, of the supercomputing system AS. GM supervises the global reconfiguration of more than one subsystem and controls data exchanges among them. Any monitor (local or global) allows a system reconfiguration only if the respective reconfiguration request wins the reconfiguration conflict (if any) and if the programs assigned to the current architectural state N have either completed their execution in N or have been orderly interrupted, i.e., they can resume their computation in the same state, N, in a warranted computational situation.

Basic steps of architectural reconfiguration. Any system reconfiguration includes

(1) access of a set of new control codes by each computing module of the architectural state to be established,
(2) establishment of new interconnections in the interconnection networks used, and
(3) start-up of programs assigned to a new architectural state.

For dynamic architectures, any architectural reconfiguration is performed by the instruction $N \rightarrow N^*$, where N is a current architectural state and N^* is the next state. $N \rightarrow N^*$ always belongs to the program that requests reconfiguration.

Reconfiguration algorithm for dynamic architecture. When the $N \rightarrow N^*$ instruction is fetched by a dynamic computer of a current architectural state N, it forms the reconfiguration request message $\widetilde{RRM}$, which is sent to the monitor (see Step 1, Figure 2). The monitor first checks

whether or not an arriving $\widetilde{RRM}$ is in conflict with the current *RRM* or next *RRM** available in the monitor, which represent current state *N* or next state *N** (see Figure 3). If it is in conflict, the monitor initiates conflict resolution (Step 2, Figure 2). Otherwise, the monitor initiates task synchronization (Step 7) to start the reconfiguration process specified by $\widetilde{RRM}$.

The actions performed by the monitor during conflict resolution depend on the relationship between the priority code $\widetilde{PC}$ of arriving message $\widetilde{RRM}$ with those already in the monitor: *PC* for current *RRM* and *PC** for next *RRM** (Figure 3). If $\widetilde{PC} < PC^*$ and *PC*, the arriving $\widetilde{RRM}$ requests the state $\widetilde{N}$ with a lower priority than those of current state *N* and next state *N** stored in the monitor. In this case, no reconfiguration actions are performed and the denied $\widetilde{RRM}$ is sent to the denied resource handler, DRH (see Step 4, Figure 2 and Figure 3a).

If the arriving $\widetilde{PC} > PC$ or *PC**, the monitor initiates the architectural interrupt of either the current or next state (Step 5 or 6). Programs assigned to state $\widetilde{N}$ are computed as follows: For the current interrupt, their execution starts immediately; for the next-state interrupt, they wait for completion of current state *N* (see Figure 3b, c). Completion of *N* is established during Step 7 of task synchronization with the use of a special synchronization instruction, **Stop N**, that concludes each task computed in *N*.[10]

During task synchronization (Step 7), the monitor performs a special readiness test, RT, to find the readiness of the requested resources to perform reconfiguration. If the requested resources are ready, the monitor signals the accessing of variable control codes by all requested computing resources (Step 8). Otherwise, a special waiting loop is initialized, in which each iteration ends with the readiness test, RT. The waiting loop ends when the readiness test gives positive results.

The next two steps (Step 8 and Step 9) involve the access of variable control codes (Step 8) and establishment of new interconnections in the network (Step 9).

Each architectural state is characterized by its own set of control codes. For instance, as we have shown,[11,12] to reconfigure into a new multicomputer state requires writing four control codes with changeable values into each computer element of the requested resources. On the other hand, to establish a new network state (rings, trees, and stars) requires only two invariant control codes to be written to all computer elements of requested resources if we use the proper reconfiguration algorithms.[13-15] Therefore, required control codes can be stored either as an array in the memory or, if their number is small (as is true for the techniques developed for network reconfigurations[13-15]), in the reconfiguration instruction $N \rightarrow N^*$. For the first case, storage of the array of control codes should be organized in such a way as to minimize the time they require for sorting and accessing requested computer elements.[11] For the second case, organization of the reconfiguration instruction should allow the control codes stored in the instruction to be sent (possibly by monitor) to the requested computing resources that will be using these codes for formation of *N**.

Finally, formation of the new computing structures of the next architectural state ends with the activation of new interconnections in the interconnection network (Step 9). Such interconnections

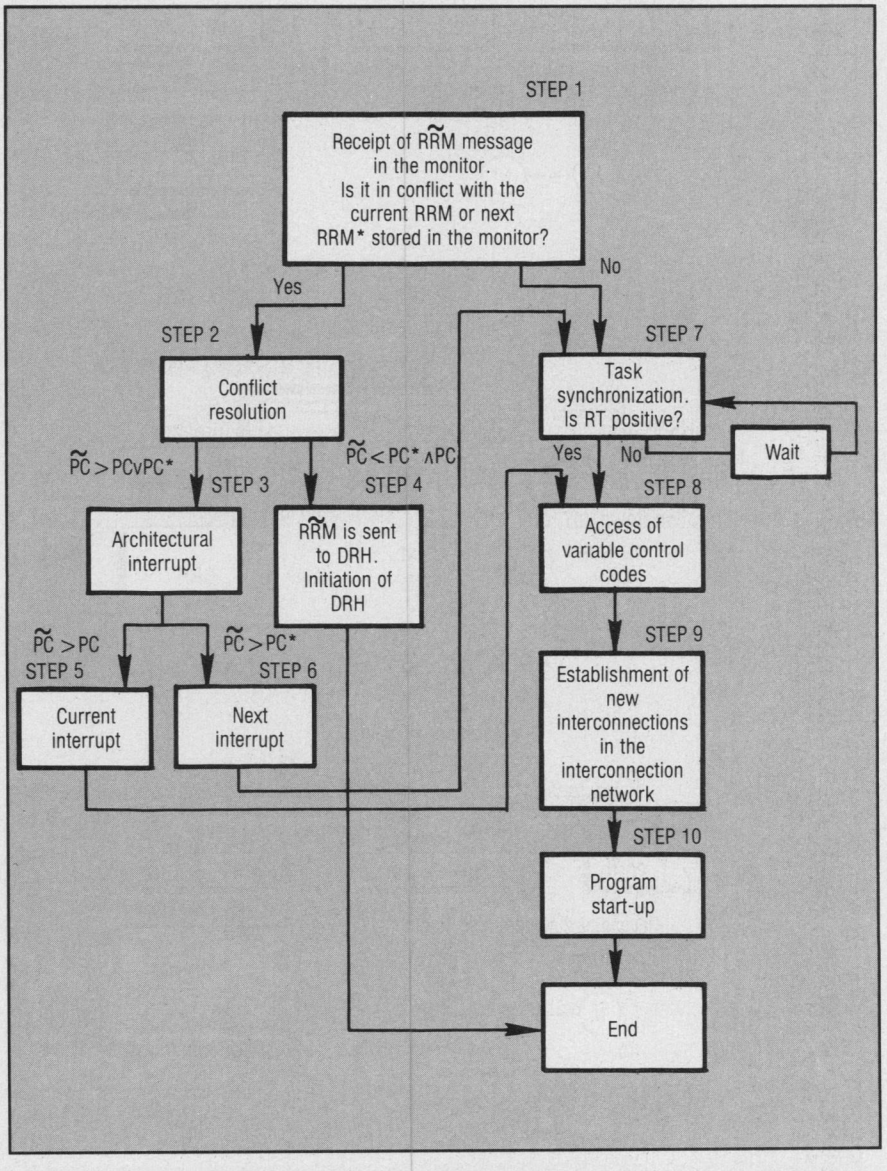

Figure 2. Sequence of reconfiguration steps for multicomputer reconfiguration.

should allow correct formation of *reconfigurable instruction buses,* which broadcast each instruction only to the modules of a newly formed computing structure and prevent broadcast to all other computing structures that coexist concurrently; and *reconfigurable data buses,* which allow fast, concurrent data exchanges within each newly formed computing structure and between different computing structures existing either concurrently or sequentially.

The last step of multicomputer reconfiguration includes program start-up. Each program, P, must be provided with addressing techniques that allow start-up of its execution in state N^*. Consider the following cases:

Case 1. Program P is assigned to both states N and N^* and

(a) P requests transition $N \rightarrow N^*$ (in other words, P is *active*)

(b) P does not request transition $N \rightarrow N^*$ (P is *passive*)

Case 2. Program P is assigned only to N^* and

(a) P is active when performing transition $N \rightarrow N^*$

(b) P is passive when not performing transition $N \rightarrow N^*$

Detailed descriptions of the software and hardware organizations involved in multicomputer and network architectural reconfigurations are found elsewhere.[10-15]

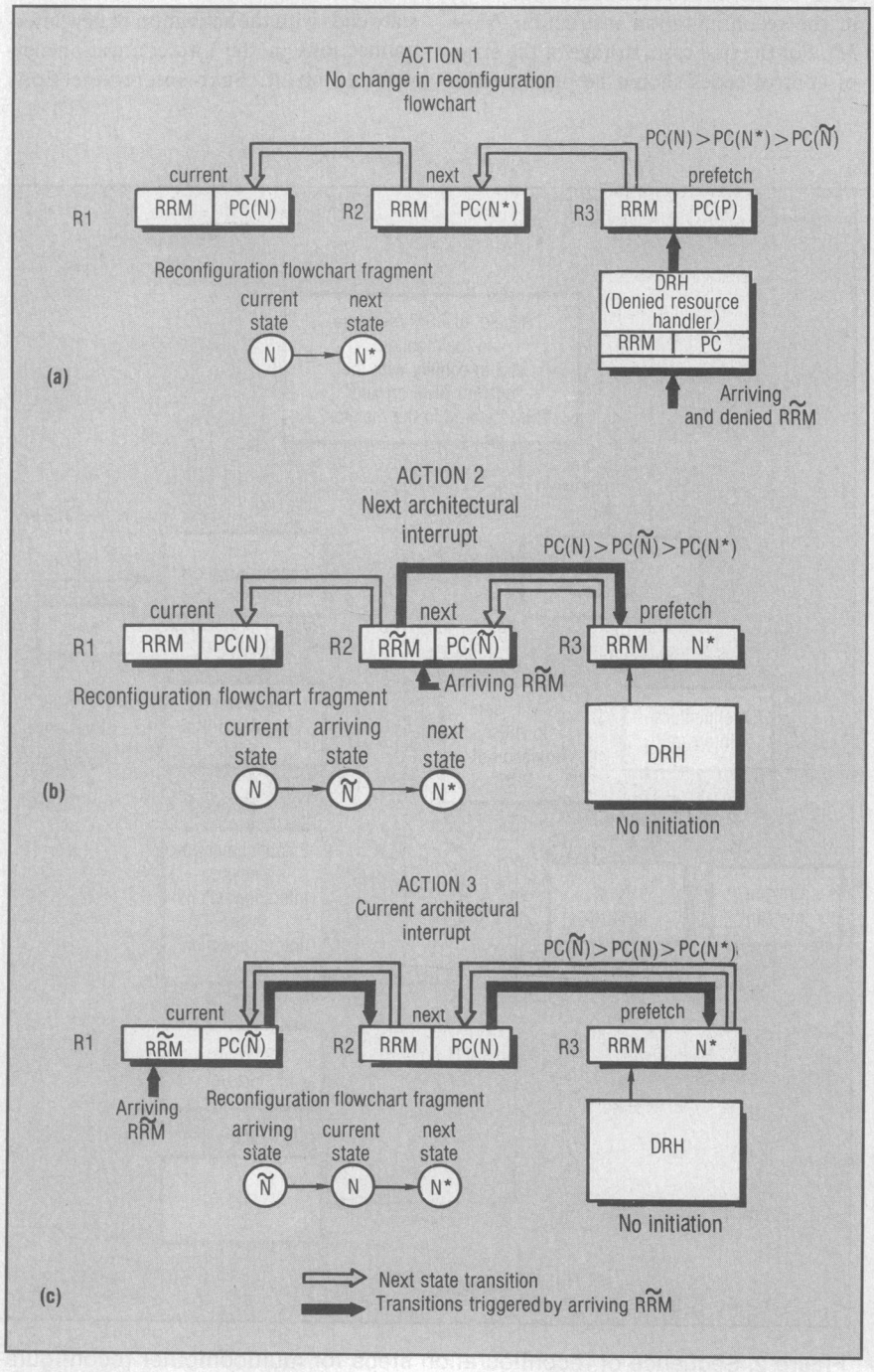

Figure 3. Conflict resolution.

Fault-tolerant reconfiguration. In general, the overall objective of any fault-tolerant reconfiguration is to provide unimpaired performance in a computer system in the presence of faults.

Circuit and module level reconfigurations. Fault-tolerant reconfiguration can be performed at the following levels:

a) *circuit level,* aimed at isolating faulty devices inside a computing module; or

b) *module level,* aimed at isolating every faulty computing module from other fault-free modules of the computer system.

The module-level reconfiguration is performed if and when all circuit-level reconfigurations fail in the sense that they cannot restore correct computations inside a faulty module, or a faulty circuit inside the module is or becomes unique and thus nonreconfigurable (and as such, it prevents fault-free performance of the entire module).

Two types of module-level fault-tolerant reconfigurations can be distinguished: *dynamic replacement* and *graceful degradation.* For dynamic replacement, each faulty node in a system is replaced by a spare assigned the same position code. Therefore, for dynamic replacement, a current computing structure remains unchanged so long as there are spare nodes in the bank. If the bank of spares becomes empty or a computing structure cannot

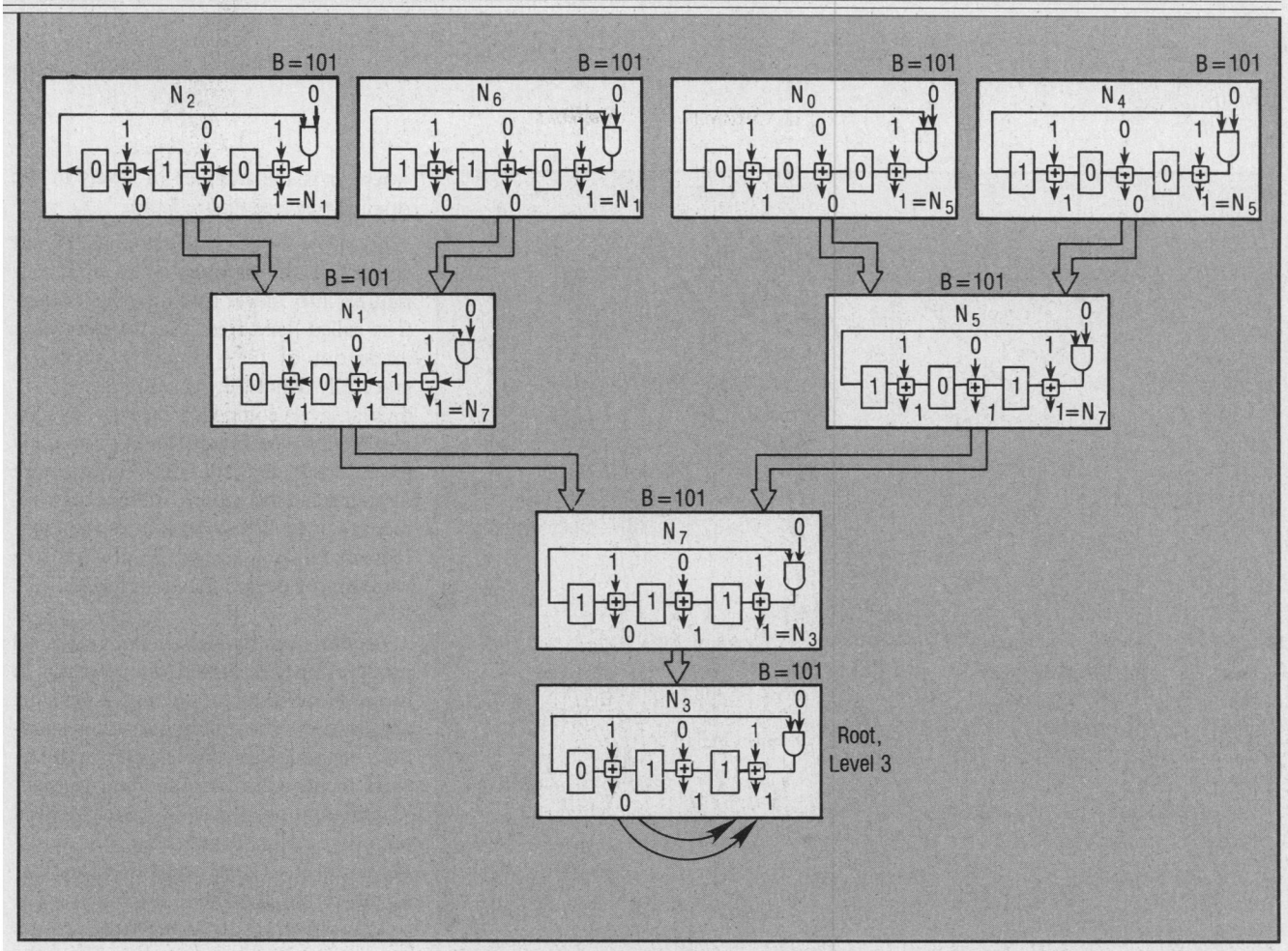

Figure 4. 3-level tree.

tolerate long propagation delays caused by integrating a new spare into the existing structure,[16] graceful degradation follows the dynamic replacement strategy. Graceful degradation turns off faulty modules and forms a connected architectural state from the set of remaining fault-free modules. Therefore, fault-tolerant reconfigurations make the reconfiguration (architectural) flowchart fault-tolerant. While dynamic replacement does not alter the original flowchart, since system throughput remains the same, graceful degradation results in a flowchart with reduced throughput caused by turning off faulty modules.

Gracefully degraded binary trees. Although we discuss here a module-level fault-tolerant reconfiguration for reconfigurable binary trees, these techniques are presented with more detail elsewhere.[15] Every tree configuration is created with a special shift-register, SRVB, residing in every network node and one control code, bias B.

For instance, Figure 4 shows one tree configuration obtained with a 3-bit shift-register in every network node and bias $B = 101$, sent to all nodes of the network. Each node, N, finds its successor, N^*, in the network by performing the following mod 2 addition operation: $N^* = 1[N] + B$, where $1[N]$ is one bit-shift to the left. For instance, N_2 is succeeded in the tree of Figure 4 by N_1 found as $N_1 = 1[N_2] + B = 1[010] + 101 = 100 + 101 = 001$, etc. Since for the n-level tree there are 2^n different biases, it is possible to generate 2^n different trees using these techniques. Each such configuration is obtained during the one clock-period required for node N to form the code of its successor, N^*, and send this code by interconnection network to select the connecting elements joining N and N^* together. Such fast generation of different network structures (of which trees are a particular case) was never considered in the literature.

The availability of 2^n different tree configurations can be construed as a powerful instrument for enhancing the fault-tolerance of a binary tree with multiple faults. Indeed, if multiple faults occupy nonleaf positions in a binary tree, a conventional bypassing procedure no longer works since it distorts the structure of the tree.

Therefore, the task of any meaningful fault-tolerant reconfiguration becomes that of fast formation of a *gracefully degraded tree,* GDT, from fault-free nodes that remain in a system. Faulty nodes must be reconfigured into tree positions that do not disrupt the connectivity of the fault-free nodes in a selected tree configuration. Such reconfiguration allows a simple purge of all faulty nodes from the network. We described these reconfiguration techniques elsewhere.[15]

Two types of gracefully degraded trees can be formed through reconfiguration, using these techniques:

(1) Type-1, or *1-truncated GDT,* in which all faulty nodes are reconfigured into leaves. A gracefully degraded tree will have $(n-1)$ levels of fault-free nodes,

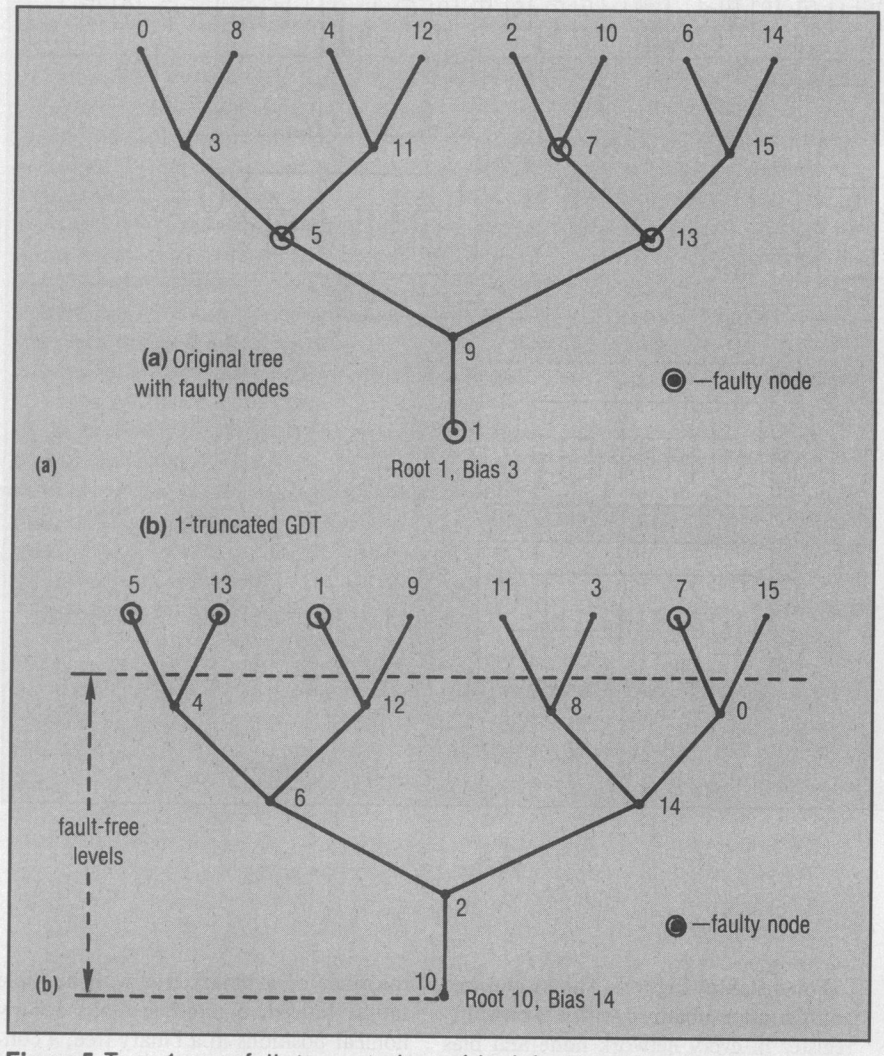

Figure 5. Type-1 gracefully truncated tree: **(a)** original tree type-2 gracefully truncated tree with faulty nodes; and **(b)** 1-truncated GDT.

where n is the number of levels in the original tree (see Figure 5a,b).

(2) Type-2, or *i-truncated GDT,* in which $i>1$ and all faulty nodes are reconfigured into i-level tree branches, otherwise called i-end trees. Each i-end tree is made out of (2^i-1) faulty nodes. A gracefully degraded tree will have $(n-i)$ highest levels composed entirely of fault-free nodes (see Figure 6). As for the i lowest levels, since all faulty end trees are disconnected by reconfiguration, there remain only fault-free i-end trees that contain no faulty nodes. (The original tree with faulty nodes is shown in Figure 7.)

Reconfiguration methodology summed up. We have presented our research in reconfiguration methodology for dynamic architecture. The problem of reconfiguration methodology divides into architectural reconfiguration and fault-tolerant reconfiguration (Figure 1). Architectural reconfiguration organizes an architecdural transition from one state to another in the reconfiguration flowchart. Fault-tolerant reconfiguration enhances the reconfiguration flowchart by adding the element of fault-tolerance.

Architectural reconfiguration summed up. Our research in architectural reconfiguration has focused on solving the problems of architectural reconfigurations in dynamic multicomputer systems and an approach to network reconfigura-

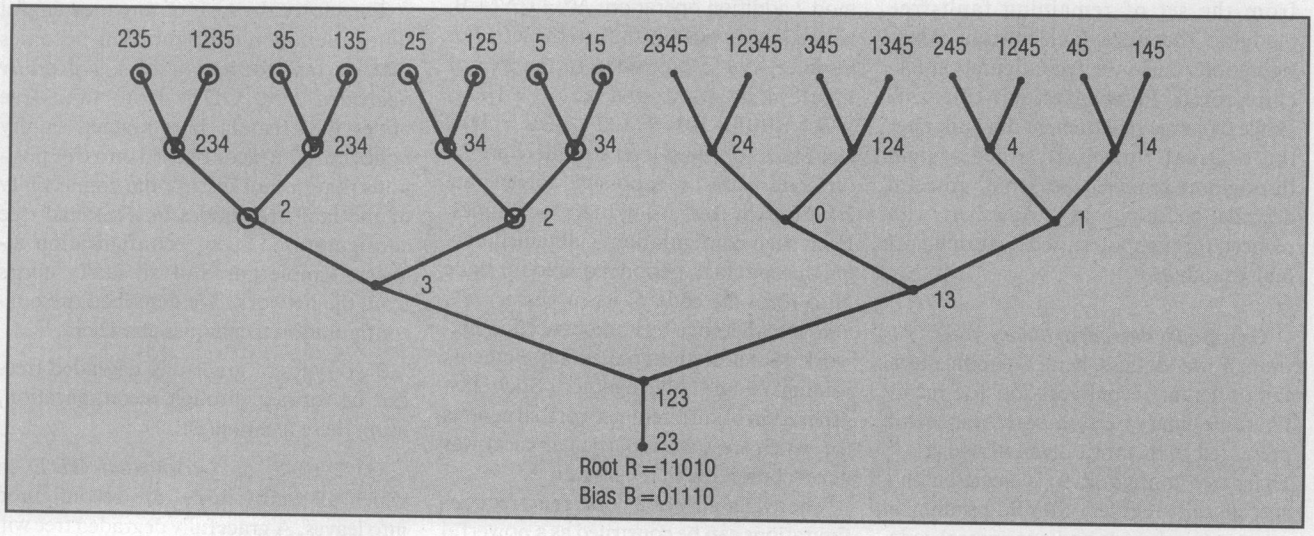

Figure 6. Type-2 gracefully truncated tree: 3-truncated GDT.

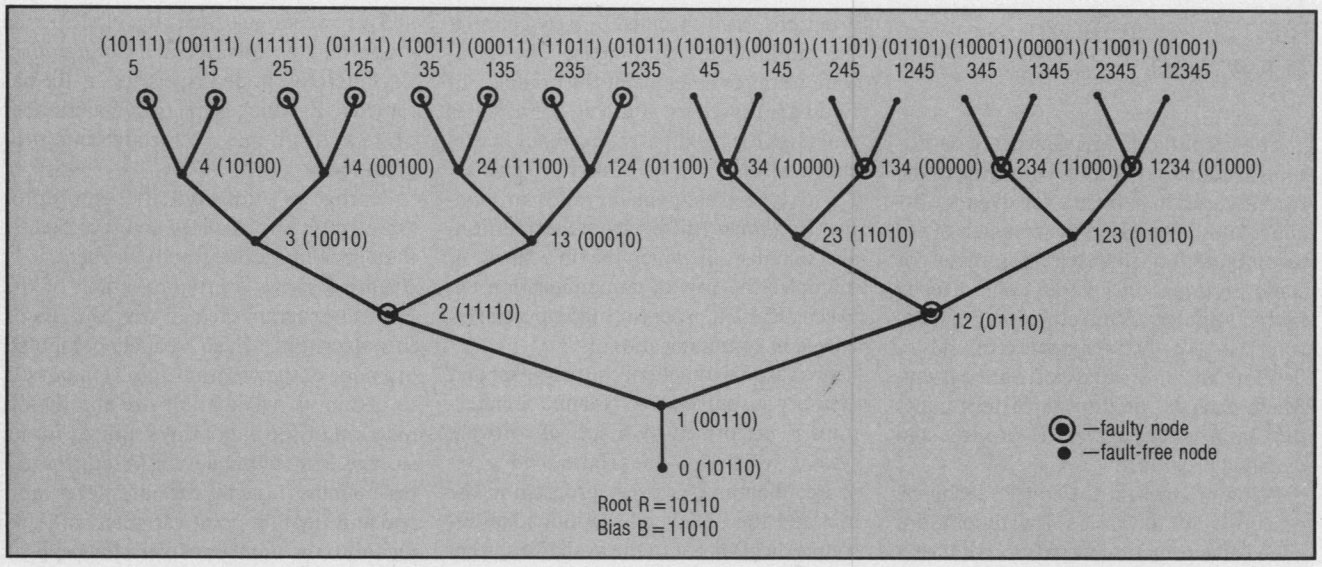

Figure 7. Original tree with faulty nodes.

tions. The techniques we have described and referenced reflect the cost-efficiency of the solutions obtained so far.

Each type of architectural reconfiguration introduced can be organized practically instantaneously with the use of only one program instruction, provided the requested resources have completed execution and the priority of the requesting program warrants such reconfiguration. Therefore, an adopted architectural organization introduces practically no reconfiguration overhead once the program priorities justify such actions. For multicomputer reconfiguration, the reconfiguration instruction takes the same time as 64-bit addition. For network reconfiguration, the duration of the reconfiguration instruction becomes even shorter: The entire reconfiguration process is reduced to executing a simple mod 2 addition ($2t_d$ delay) and can be effected with only two control codes of size $2n-1$ and n, respectively, where $n=\log_2 K$ and K is the number of computer elements, or CE, in the system. These codes can be stored in the reconfiguration instruction, since their total bit size is $3n-1$.

The objectives of our current research in the domain of architectural reconfiguration are to develop comprehensive reconfiguration methodologies for all five architectures (multicomputer/multiprocessor, array, pipeline, network, and mixed) that can be assumed by dynamic architectures.

Also, on the basis of our past involvement in this topic, we believe that the complexity of the entire algorithmic process for each type of reconfiguration will not differ from the time of multicomputer reconfiguration. As a result, multiple system reconfigurations—which will become possible because of well-developed reconfiguration methodologies—can be performed as frequently as necessary to enhance mission-critical computations without introducing significant reconfiguration overhead into system performance.

Fault tolerant reconfigurations summed up. Past research on fault-tolerant reconfigurations focused on developing graceful degradation techniques for reconfigurable binary trees having multiple-fault status. We (and the referenced publications) have demonstrated extreme cost efficiency of the adopted solutions. The entire graceful degradation in a reconfigurable binary tree with multiple faults can be accomplished during the time of one mod 2 addition, using only one control code (bias):

(1) If all faulty modules can be reconfigured into tree leaves, to find this code requires one mod 2 addition.

(2) If all faulty modules are reconfigured into an end subtree having 2^i nodes, to find this code takes i sequential mod 2 additions. (For instance, if $i=2$, then the end subtree is formed from four faulty modules ($4=2^2$) purged from the system during the time of one mod 2 addition. To find the bias B that generates the required gracefully degraded tree takes the time of two mod 2 additions performed sequentially. Thus, the entire reconfiguration process takes the time of $2t_d$ and the time of finding the required bias is $4t_d$.)

In both cases, the fault-tolerant reconfiguration will accomplish the formation of the gracefully degraded tree without faulty modules. For the first case, this tree will function without some old leaves. That is, instead of an n-level original tree, it will act for some nodes as an $(n-1)$-level tree; for others, it will remain an n-level tree. For the second case, this tree will function without one or several of the i-dimensional end subtrees.

Thus, for both cases, the fault-tolerant reconfiguration preserves the structure of the original tree but organizes its performance at a reduced level without violating the tree-type connectivity of the remaining fault-free nodes. This type of performance is impossible to obtain with any bypassing organized in the presence of multiple faults, since any such bypassing will distort the structure of an original binary tree.

Reconfiguration flowchart

The reconfiguration flowchart constitutes a second set of tools belonging to the reconfiguration software for dynamic architectures. We solved the problem of the reconfiguration flowchart for mixes of static programs in Fortran computed by multicomputer architecture.[10,12] Therefore, the task of our current research is to develop an automated software system, which extends the discovered techniques to Ada programs and mixes of static and dynamic programs.

Below we present the concise technical description of the basic software solutions aimed at automatic construction of the reconfiguration flowchart for Fortran programs run on dynamic multicomputer architecture.

Adaptive assignment: statement of the problem. To realize the capability of dynamic architectures in maximizing throughput of the available resources, we must solve the following problem: For each user program we must find a sequence of minimal-size computers to execute it. Next, the available hardware resources must be assigned among user programs, each executed by a sequence of the minimal-size computers. The hardware resource assignment then reduces to finding a flowchart of architectural states that gives the maximal concurrency in execution of the given set of user programs.

This type of assignment, called an *adaptive assignment*, differs from the traditional assignment, in which the objective is to minimize the cost of communication between different processors (computers) involved in communication.[17-19]

Overall assignment of the processor and memory resources for dynamic architectures is performed by a special software system, which first finds the individual resource diagram for every program in the mix and then maps all the individual resource diagrams onto the available system resources.

Program resource diagram. The first stage in constructing the program resource diagram consists of finding for each program written in a high-level language the word sizes for a sequence of computers that could execute this program.

Program graph. Finding word sizes for computed variables is greatly simplified if we construct a *program graph* in which each node includes one or more statements and arrows show sequencing. Nodes in the graph may be *iterative* and *noniterative*. Each iterative node is specified by the parameter, Z, which is the maximal number of iterations it may go through in a program loop.

A program's transformation into a program graph is formalized and specified by the algorithm we presented elsewhere.[10,12] (Figure 8 shows a program graph of the Fortran program created with the use of this algorithm.) Each variable computed in a node of the program graph is analyzed to obtain its maximal bit-size and the dimension of the data array required for its storage. The techniques for finding the upper bounds of the bit sizes of integers and real and floating-point variables with due regard to the number of iterations appear elsewhere.[10]

For each variable X computed in the node d of the program graph, find its maximal bit-size $BS(X)$, and its maximal intermediate bit-size, $BS(X^*)$, using the techniques outlined elsewhere[10] (see Figure 9a). Then the bit size of node d, BSN_d, is obtained as a maximum of all bit sizes assumed by variables computed in this node, i.e., $BSN_d = \max[BS(X)]$. Likewise, $BSN^*_d = \max[BS(X^*)]$.

During the analysis of each graph node, we must specify the dimensions of data ar-

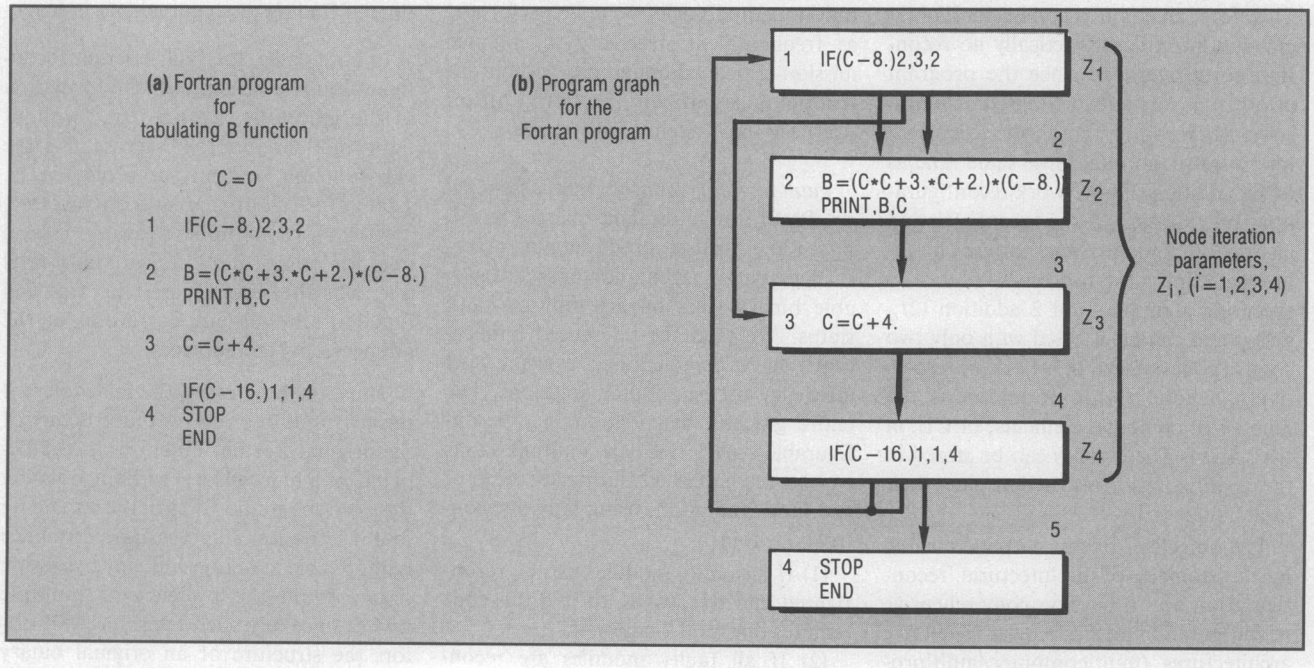

Figure 8. Fortran program and its graph.

rays, DA, for all computed variables. To do this, in a node we find a statement that provides for storage of a computed variable or variables. Let this statement specify that v computed variables, X_1, X_2, . . . , X_v, be stored and the iteration parameter for the node be Z. Then the required dimension of the array to store variables $X_1, X_2, \ldots, X_v$ is $DA = V \times Z$. The dimension of the data array for all variables computed in the node, DAN, is the sum of the DAs obtained for all statements, $DAN = \Sigma DA$.

Diagram of the hardware resource. Analysis of the program nodes is succeeded by construction of the individual diagram of the hardware resource required by a program, or *P-resource diagram*. To construct a P-resource diagram, first find the bit-size diagram of this program graph. The horizontal axis specifies the graph nodes and the vertical axis shows the two bit-size parameters, *BSN* and *BSN** found for each node (see Figure 9a).

The bit-size diagram is aligned next by the assignment system in order to exclude excessive changes in computer sizes (see Figure 9b). Alignment of all graph nodes to dominant bit sizes results in finding the computer-size diagram that specifies the sequence of computer sizes required for computation of the program. This diagram partitions the program into tasks, where each task is composed of nodes requiring the same computer size (see Figure 9b).

To minimize program delays, the adaptive assignment of the resources must be based on the tentative execution time of each task in a given size computer. To find the time of one task, find the time taken to execute one node, using the techniques presented elsewhere.[10] The hardware resource diagram for the entire program (P-resource diagram) is constructed as follows: Its horizontal axis shows the time of functioning of each task. The vertical axis contains two portions. The upper portion shows computer sizes and the lower portion shows the dimensions of the data arrays for all initial and computed variables (see Figure 10).

Assignment of system resources among programs. The assignment of system resources follows construction of individual P-resource diagrams. The assignment system maps all such diagrams onto the following two diagrams, which show the distribution of resources for dynamic architecture:

(1) *the CE resource diagram*, which charts the computer elements assigned to each task and defines the reconfiguration flowchart, and
(2) *the ME resource diagram*, which allocates the primary memory of dynamic architecture among various user programs.

The CE resource diagram is constructed for the assignable resources $R = n \cdot CE \cdot T$, consisting of n computer elements engaged in computations during the total time, T, determined by the assignment. The ME resource diagram is constructed next using the memory portions of assignable resources, consisting of n memory elements, or ME. Figure 11a shows assignable resources R intercon-

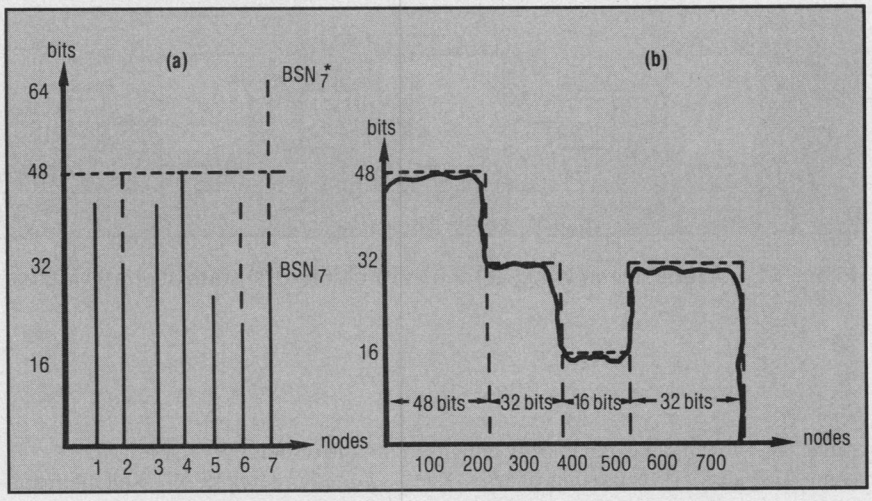

Figure 9. Bit-size diagram **(a)** and computer-sized diagram of program graph **(b)**.

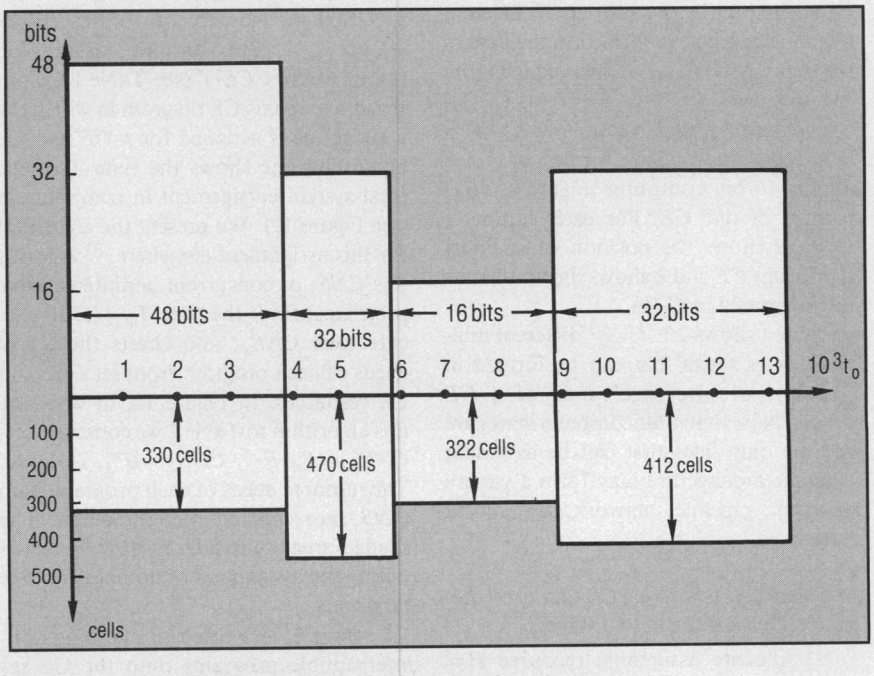

Figure 10. P-resource diagram for the entire program.

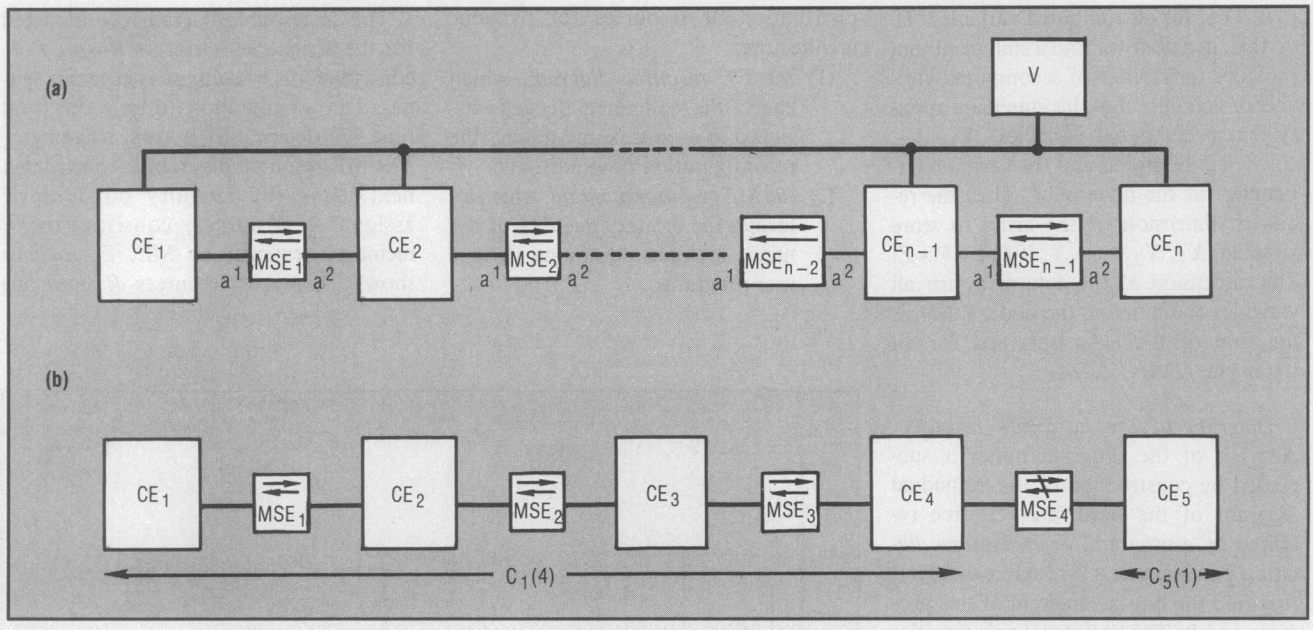

Figure 11. Assignable resources **(a)** and the architectural state of one 64-bit and one 16-bit computer **(b)**.

nected with the minimal complexity interconnection network consisting of $(n-1)$ bidirectional connecting elements, MSE.[11,12]

For multicomputer architecture, the assignable resources R made of n CE can form 2^{n-1} different architectural states N, each specified by the set of different dynamic computers formed in the system through software. For instance, Figure 11b specifies state $N_1 = [C_1(4), C_5(1)]$ consisting of a 64-bit computer $C_1(4) = \{CE_1, CE_2, CE_3, CE_4\}$ formed of 4 CE and a 16-bit computer $C_5(1) = \{CE_5\}$ formed of one CE. For each computer $C_i(k)$, i shows the position of its most significant CE and k shows the number of CE integrated in $C_i(k)$.

Table 1 shows $2^{n-1} = 2^3$ different multicomputer states that can be formed in dynamic architecture made of 4 CE $(n = 4)$. Note that multicomputer states are not the only ones that can be assumed; dynamic architecture may form a variety of array, pipeline, network, and mixed states.

Construction of the CE resource diagram. This is done in two stages:

(1) Allocate assignable resource $R = n \cdot CE \cdot T$ to noninterruptible programs. Find the resource $R' < R$ left.

(2) Assign resource R' to all interruptible programs.

A detailed description of the assignment procedures appears elsewhere.[10]

In stage 1, we are given a set of noninterruptible programs $P_1, P_2, \ldots, P_r$ specified with ordered priorities $PR_1 > PR_2 > \ldots > PR_r$ and the assignable resource $R = n \cdot CE \cdot T$ (see Table 2). Construct a two-axis CE diagram in which the vertical line is assigned for n CE and the horizontal line shows the time T of the total system engagement in computation (see Figure 12). We present the algorithm for this assignment elsewhere.[10] It forms sets CNS_i of concurrent noninterruptible programs; finds the time $T_{max}(i)$ to execute each CNS_i; and charts the actual needs of each program from set CNS_i on CE resources. In Figure 12, by applying this algorithm to Table 2 we construct sets $CNS_1 = \{P_1, P_2\}$, $CNS_2 = \{P_3, P_7\}$, etc. The resource needs of each program from CNS_i are charted in light areas. The shaded areas show idle resources created during the assignment of noninterruptible programs.

In stage 2, as a result of mapping noninterruptible programs onto the CE resource diagram, some CE's are idle. To eliminate their idleness, interruptible programs are first mapped onto these idle CE's. The remaining portions of interruptible programs are then mapped onto those CE resources released after execution of a noninterruptible program.

Figure 13 shows mapping of six interruptible programs, P_9 through P_{14}, onto the idle resources left after mapping eight noninterruptible programs (see Figure 12). A detailed description of algorithms that perform this mapping appears elsewhere.[10]

Minimization of computation delays. The assignment procedures used for noninterruptible and interruptible programs lead to absolute minimization of computational delays in program execution because of the fundamental ability of dynamic architecture to minimize the number of CE resources involved in the computation of each program P. Since P is computed by the sequence of dynamic computers, the following amount of CE (PE and ME) resources are released: Let the CE needs of P be given as

$$P = k_1 \cdot CE \cdot t(TA_1) + k_2 \cdot CE \cdot t(TA_2) + \ldots + k_p \cdot CE \cdot t(TA_p),$$

where $t(TA_i)$ is the time of the ith task.

Find the m parameter of this program, where

$$m = \max(k_1, k_2, \ldots, k_p)$$

In any nondynamic architecture, a computer that may compute this program will contain at least m *CE's*. This means that during the entire time $T = t(TA_1) + \ldots + t(TA_p)$ of P computations, these m *CE*'s will be busy and unable to compute any other program. On the other hand, dynamic architecture releases the following resources, $RR(P)$, involved in computation of P:

$$RR(P) = (m - k_1) \cdot CE \cdot t(TA_1) + (m - k_2) \cdot CE \cdot t(TA_2) + \ldots + (m - k_p) \cdot CE \cdot t(TA_p)$$

All released resources, $RR(P)$, can be used for computing interruptible programs. This results in minimizing the duration of each interrupt and lessening the problem from program delays when a program should wait until the requested resources become available.

Any interruptible program P_i is computed during the time

$$T_{act}(P_i) = T_{min}(P_i) + T_{INT}(P_i)$$

where $T_{min}(P_i)$ is the time of executing P_i in a noninterruptible form and T_{INT} is the total time of all interrupts made during execution of P_i.

In conventional architectures, minimization of $T_{act}(P_i)$ can be accomplished only by adding new resources to the system. In dynamic architectures, it is accomplished by existing resources because of their adaptation capabilities.

The additional concurrency gained because of the adaptive resource assignment for dynamic architectures illustrated in Figure 13 is shown in Table 3.

The total additional concurrency of the resources, or ACR, is

$$ACR = 1 \cdot CE \cdot 45s + 2 \cdot CE \cdot 35s$$

In other words, one CE (16-bit computer) is freed for additional executions for 45 seconds and two CE's (32-bit computer)

Table 1.
Eight different multicomputer states made of 4 CEs.

(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)
N_0	$C_1(4)$	1	1×64	N_4	$C_1(1), C_2(3)$	2	1×16, 1×48
N_1	$C_1(3), C_4(1)$	2	1×48, 1×16	N_5	$C_1(1), C_2(2), C_4(1)$	3	1×16, 1×32, 1×16
N_2	$C_1(2), C_3(2)$	2	2×32	N_6	$C_1(1), C_2(1), C_3(2)$	3	1×16, 1×16, 1×32
N_3	$C_1(2), C_3(1), C_4(1)$	3	1×32, 2×16	N_7	$C_1(1), C_2(1), C_3(2), C_4(1)$	4	4×16

Table 2.
An assignable resource, R.

Col. 1	Col. 2	Col. 3
Priority ordered sequence of noninterrupt programs	Maximal number of CE's required by each P_i m_i	Time of execution of each P_i (seconds)
P_1	2	10
P_2	3	20
P_3	4	30
P_4	4	20
P_5	3	10
P_6	2	30
P_7	1	20
P_8	1	40

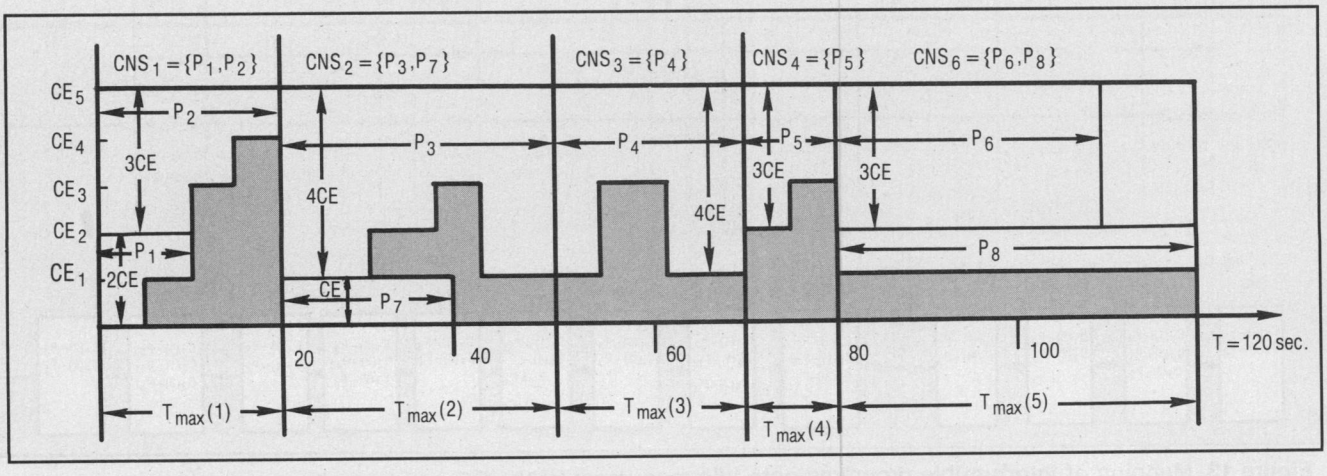

Figure 12. CE resource diagram for the assignable resource of Table 2.

can execute additional programs for 35 seconds.

The ACR is found by adding together all released resources, each one multiplied by the total time of its release. For instance, the total time of releasing CE_1 is made of the following addends (Table 3):

$t(TA_2) = 5$s for P_1;
$t(TA_2) = 10$s for P_2;
$t(TA_2) = 10$s for P_3; and
$t(TA_2) = 20$s for P_{12}.

Thus, during the entire release time (45 seconds), CE_1 is used for additional computations. Similarly, we find the time of 35 seconds for releasing 2 CE's. This concurrency results in minimization of the total engagement time, T, of the assignable resources, R, consisting of 5 CE's and executing 14 programs. As follows from Figure 13,

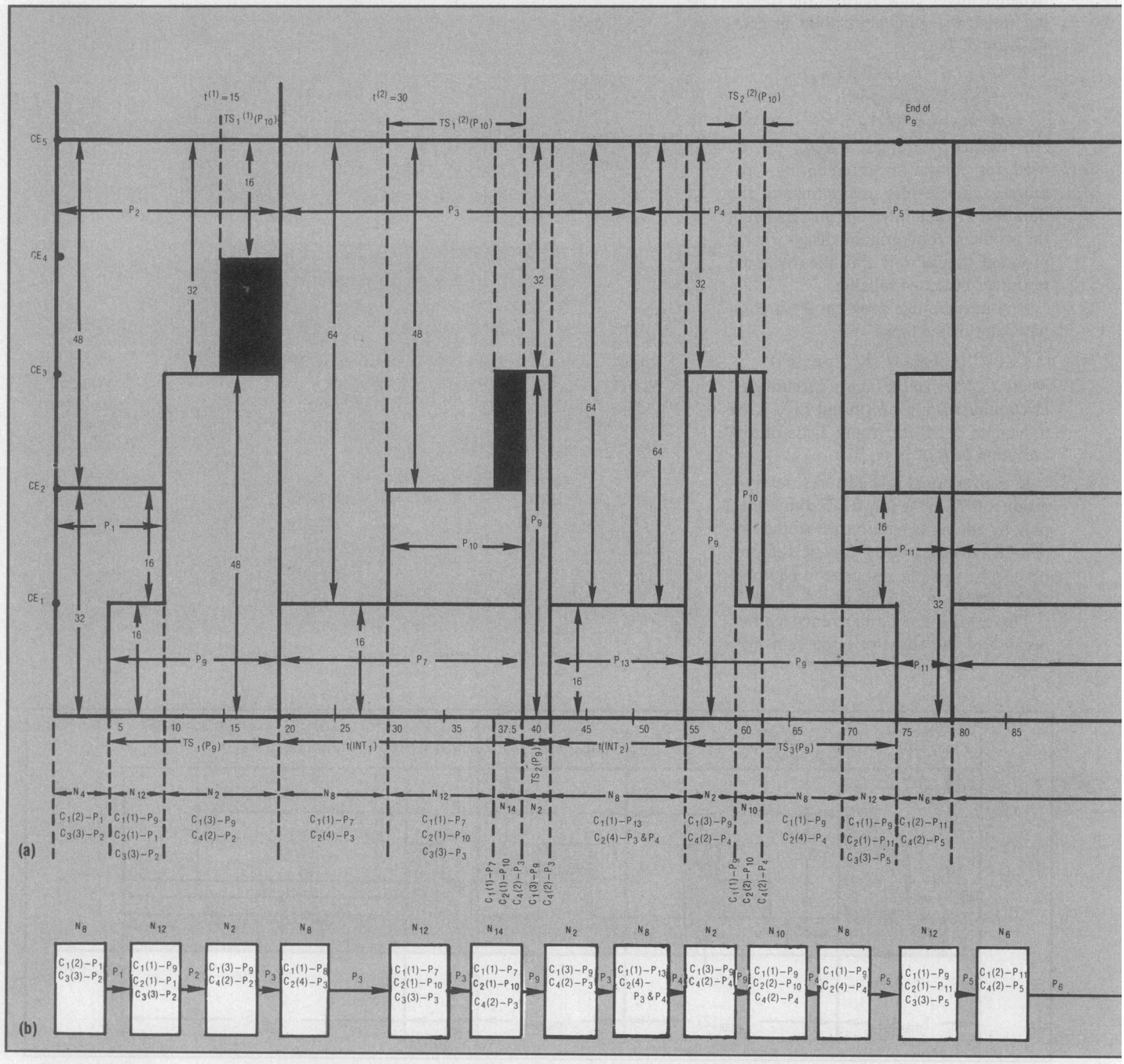

Figure 13. Mapping of interruptible programs onto idle resources: **(a)** the CE resource diagram; and **(b)** the reconfiguration flowchart.

$R = n \cdot CE \cdot T = 5 \cdot CE \cdot 167.5$s

On the other hand, any nondynamic architecture made of 5 *CE*'s can execute the same program mix of 14 programs during 237.5 seconds. Therefore, the hardware resources equipped with dynamic architecture accomplish a 42 percent speedup in execution of the same program mix in comparison with the best nondynamic architecture having the same complexity and fabrication technology.

Finding reconfiguration flowchart. Construction of the CE resource diagram automatically gives the architectural flowchart of the system transitions from one architectural state to another, called the reconfiguration flowchart, where each architectural state N_i is specified by the number and sizes of dynamic computers working concurrently. Figure 13b shows the

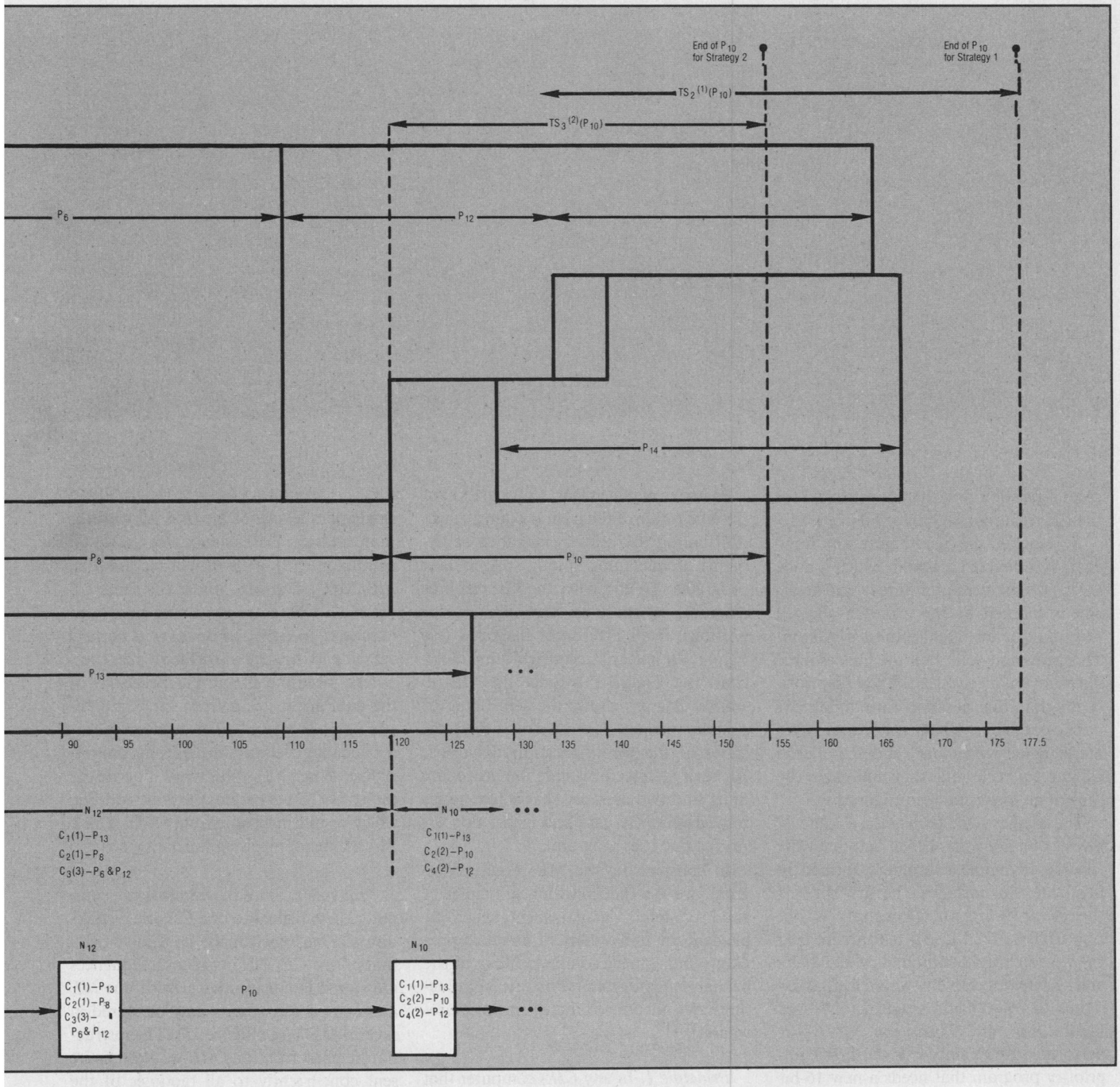

Table 3.
Additional concurrency of resources for the dynamic architectures in Figure 13.

		# (tasks)	m_i/T_i	Task$_1$ m_i-k_1	Task$_1$ $t(TA_1)$	Task$_2$ m_i-k_2	Task$_2$ $t(TA_2)$	Task$_3$ m_i-k_3	Task$_3$ $t(TA_3)$	Task$_4$ m_i-k_4	Task$_4$ $t(TA_4)$
1	RR(P_1)	2	2 / 10	-	5	1	5				
2	used RR(P_1)			-	-	1	5				
3	RR (P_2)	2	3 / 20	-	10	1	10				
4	used RR(P_2)			-	-	1	10				
5	RR(P_3)	4	4 / 30	-	10	1	7.5	2	5		7.5
6	used RR(P_3)			-	-	1	10	2	2.5		
7	RR(P_4)	3	4 / 20		5	2	7.5	-	7.5		
8	used RR(P_4)					2	7.5				
9	RR(P_5)	2	3 / 10	-	5	1	5				
10	used RR(P_5)			-		-	5				
11	RR(P_{12})	3	3 / 56	-	10	1	15	2	31		
12	used RR(P_{12})					1	20	2	25		
13	Total time of freeing one CE			1·CE(5 + 10 + 10 + 20)						45s	
14	Total time of freeing 2 CE's			2·CE(7.5 + 2.5 + 25)						35s	
Total additional concurrency				1·CE·45s + 2·CE·35s							

reconfiguration flowchart constructed for the CE resource diagram of Figure 13a.

In general, the reconfiguration flowchart is a directed graph in which a node means the architectural state N_i and an arrow is marked by the program allowed execution of this architectural transition. This program, found on the CE resource diagram, is conceived as the highest priority program that needs to form a new dynamic computer. The architectural states are assigned with binary codes using the assignment technique that minimizes the time of architectural reconfiguration.

The integer i for architectural state N_i gives its binary code, which automatically specifies dynamic computers formed in this state. For instance, the first state of the reconfiguration flowchart is N_4. $N_4 = 0100 = [C_1(2), C_1(3)]$ is characterized by two concurrent computers with 32 bits and 48 bits respectively, where $C_1(2)$ is assigned to P_1, $C_3(3)$ is assigned to P_2 (see Figure 13b), etc. Transition $N_4 \rightarrow N_{12}$ is performed by P_1 since it is the first high-priority program that needs a new 16-bit dynamic computer $C_2(1)$, etc.

Memory management. Construction of the ME resource diagram is equivalent to distributing the primary memory of dynamic architectures among various user programs. To construct the ME resource diagram, we must use the memory size portions of all P-resource diagrams (see Figure 10) and the reconfiguration flowchart (see Figure 13). Since the ME resource diagram shows the scheduling of primary memory for accessing in different states of the reconfiguration flowchart, for each architectural state the procedure must find two memory spaces that store, respectively, the data and instruction arrays of one user program.

In constructing the ME resource diagram, we use the following principles (1 and 2) of memory management, which implement the universality of executed programs and achieve complete filling of the primary memory for dynamic architectures. We introduced these principles elsewhere.[11,21]

Principle 1. In any $C_i(k)$ computer that includes k CE (CE_i, CE_{i+1}, . . . , CE_{i+k-1}), one 16· k-bit word is stored in a single parallel cell of k *ME*'s, all with the same address. For instance, for the 48-bit computer $C_1(3)$, a 48-bit word is stored in ME_1, ME_2, and ME_3 under the same address A_p. Consequently, a data array containing d 16·k-bit words takes d parallel cells for its storage in different size computers, resulting in the independence of the data array size d from the computer size k (see Figure 14). This leads to the independence of data address from computer size. When a 16·k-bit word is accessed, each of k *CE*'s generates the same address, which results in a concurrent fetch of all k 16-bit bytes comprising the data word.

Principle 2. Since the minimal size computer is equivalent to one *CE*, the instruction size has been made coincident with that of one *CE*. The instruction sequences are stored in consecutive cells of one *ME* and complex programs may be stored in several *ME*'s (see Figure 15). The instruction fetched from one *ME* would then be sent concurrently to all modules of the $C(k)$ computer. Therefore, the same pro-

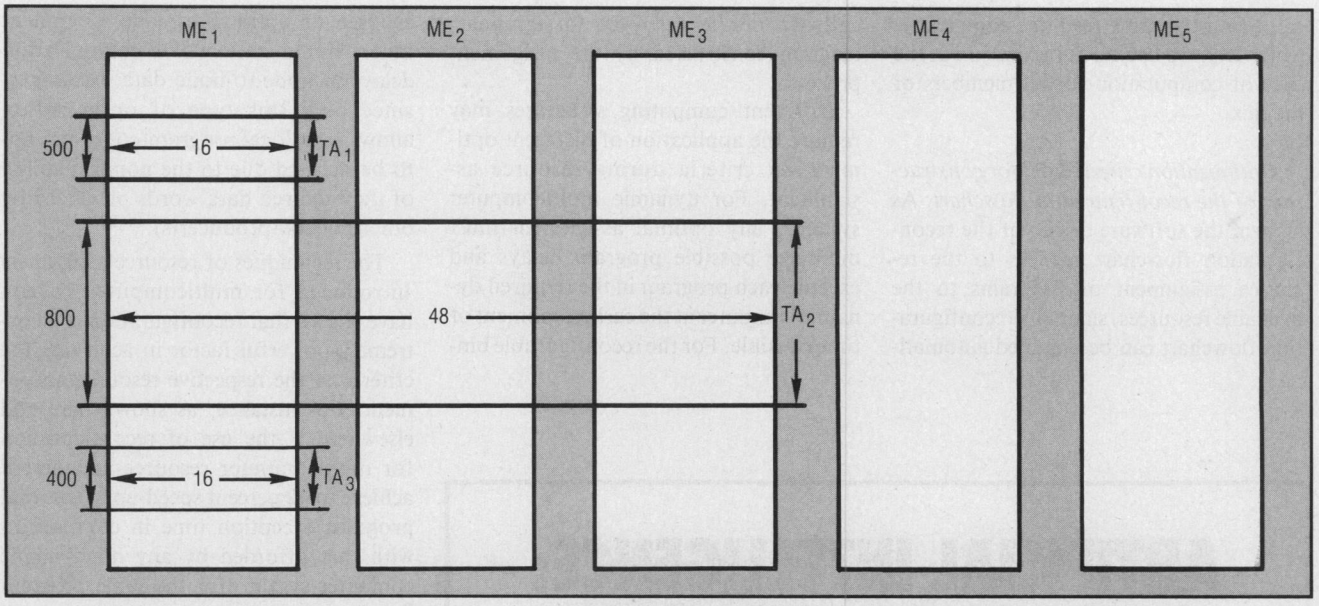

Figure 14. Independence of a data address.

gram occupies the same number of cells in any size computer, achieving independence of the sizes of instruction arrays from computer sizes. This leads to the independence of the instruction address (jump address, etc.) from the computer size.

Therefore, these two principles provide for the independence of both a data address and an instruction address from the computer sizes. This accomplishes the independence of programs from computer sizes, otherwise called *universality of programs*.

Reconfiguration flowchart summed up. We have demonstrated that it is possible to create a complete software design methodology to find a reconfiguration flowchart of dynamic architectures in execution of a given program mix. We have already created such a design methodology for a general-purpose program mix of independent programs presented in Fortran and designated for multicomputer dynamic architectures.

The methodology leads to absolute minimization of the total time of execution required by a given program mix. This results from the ability of dynamic architectures to minimize the amount of processor and memory resources used in executing each user program in this mix. The consequences of this minimization are the additional executional concurrency

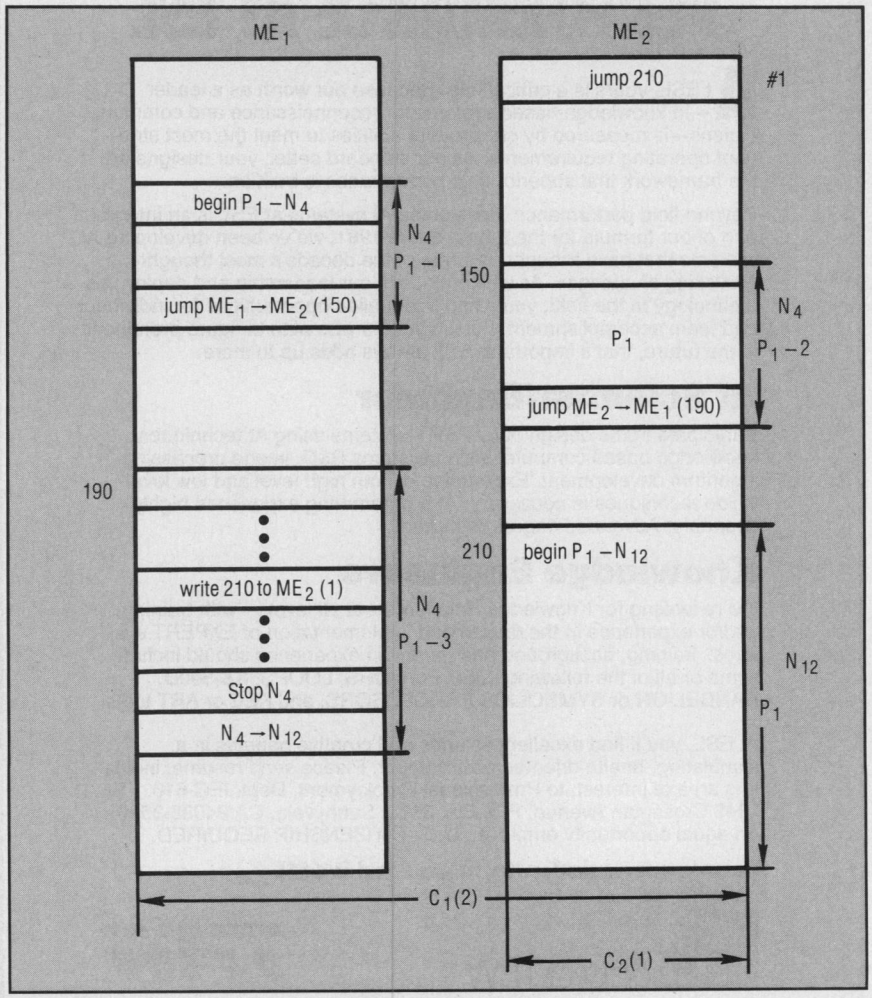

Figure 15. Independence of an instruction address.

gained on the same resource complexity, which leads to overall minimization of the time of computation for all members of the mix.

Optimization criteria used for construction of the reconfiguration flowchart. As shown, the software design of the reconfiguration flowchart reduces to the resource assignment of programs to the available resources, since any reconfiguration flowchart can be obtained automatically from the resource (assignment) diagram constructed by the assignment process.

Different computing structures may require the application of different optimization criteria during resource assignment. For dynamic multicomputer systems, any optimal assignment must minimize possible program delays and execute each program in the required dynamic computer at the earliest moment of time possible. For the reconfigurable binary tree, an optimal resource assignment favors minimization of communication delays in node-to-node data exchanges, since only this type of optimization allows each local program-consumer not to be delayed due to the nonavailability of the required data words produced by other node(s)-producer(s).[20,22]

The techniques of resource assignment introduced for multicomputer systems have shown that reconfiguration is an extremely powerful factor in achieving the criteria of the respective resource assignment. For instance, as shown here and elsewhere,[10] the use of reconfiguration for multicomputer resource assignment achieves a 42 percent speed-up in the total program execution time in comparison with that afforded by any nondynamic computer system (for the same program mixes) having the same complexity and fabrication technology.

The performance improvement for reconfigurable binary trees achieved with the use of assignment algorithms already developed[20,22] is even more dramatic. For the sampled node-to-node exchanges, it achieves 370 percent. A reconfigurable binary tree which we developed may execute the same assortment of sampled node-to-node data exchanges 370 percent faster than any static binary tree. The nature of this improvement involves the ability of reconfiguration to minimize path durations and maximize path concurrency. It does this by performing each critical (main) data exchange in the best tree configuration and assigning this tree configuration with other concurrent exchanges that do not increase the time of the critical exchange.[22]

Static and dynamic assignments summed up. Another division that can be applied to the overall problem of resource assignment for an arbitrary computing structure is to divide a resource assignment into static and dynamic.

Fundamentally, the resource needs of a dynamic program cannot be taken into account during construction of the reconfiguration flowchart. Otherwise, a dynamic program is transformed into a static one. However, in order that the reconfiguration flowchart be responsive to the needs of dynamic programs, the monitor must be involved in the process of dynamic as-

signment aimed at creation of new architectural states. Upon completion of the creation process, each such state must be inserted into the reconfiguration flowchart, causing an interrupt of the current architectural state N or the next architectural state N^* of the available reconfiguration flowchart. In both cases, no idleness occurs in the computer resources because dynamic architecture may interrupt a current or next architectural state only upon the availability of a newly created state.

Therefore, the similarities and differences between static and dynamic assignments can be summed up as follows: Both assignments are aimed at minimization of program delays through the use of reconfiguration. However, without dynamic assignment, the reconfiguration flowchart constructed by static assignment will be unable to meet the challenges of the new and unpredictable computational requirements of dynamic programs. Also, while static resource assignment and the associated reconfiguration flowchart feature no reconfiguration conflicts, for dynamic assignment such nonconflicting planning is fundamentally impossible, since it is impossible to predict the priority of any arriving program without violating the spontaneity and dynamic nature of its arrival.

Static and dynamic assignments thus have different areas of application, which justifies their inclusion as separate research topics dealing with static and dynamic programs represented in Ada. □

References

1. Edith W. Martin, "Strategy for a DoD Software Initiative," *Computer*, Vol. 16, No. 3, Mar. 1983, pp. 52-80.
2. S. P. Kartashev, "Supersystems: Current State of the Art-Guest Editor's Introduction," *IEEE Trans. on Computers*, Vol. C-31, No. 5, May 1982, pp. 345-349.
3. S. P. Kartashev, S. I. Kartashev, C. R. Vick, "Adaptable Architectures for Supersystems," *Computer*, Vol. 13, Nov. 1980, pp. 17-35.
4. R. G. Arnold, R. O. Berg, and J. W. Thomas, "A Modular Approach to Real-Time Supersystems," *IEEE Trans. Computers*, Vol. C-31, May 1982, pp. 385-398.
5. C. G. Davis, R. L. Couch, "Ballistic Missile Defense: A Supercomputer Challenge," *Computer*, Vol. 13, No. 11, Nov. 1980, pp. 37-48.
6. 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.
7. K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. Computers*, Vol. C-29, Sept. 1980, pp. 836-844.
8. E. W. Kozdrowicki and D. J. Theis, "Second Generation of Vector Supercomputers," *Computer*, Nov. 1980, pp. 71-83.
9. H. H. Love, "Reconfigurable Parallel Array Systems," *Designing and Programming Modern Computer Systems*, Vol. 1, Chapter II, *LSI Modular Computer Systems*, eds. S. P. and S. I. Kartashev, Prentice-Hall, Inc., 1982, Englewood Cliffs, NJ, pp. 99-241.
10. S. P. Kartashev and S. I. Kartashev, "Distribution of Programs for a System with Dynamic Architecture," *IEEE Trans. Computers*, June 1982, pp. 488-514.
11. S. I. Kartashev and S. P. Kartashev, "Multicomputer System with Dynamic Architecture," *IEEE Trans. Computers*, Vol. C-28, No. 10, Oct. 1979, pp. 704-721.
12. S. I. Kartashev and S. P. Kartashev, "Designing and Programming Supersystems with Dynamic Architectures," *Designing and Programming Modern Computer Systems*, Vol. 1, Chapter III, *LSI Modular Computer Systems*, eds. S.P. and S.I. Kartashev, Prentice-Hall, Inc., 1982, pp. 245-385.
13. C. G. Davis, S. P. Kartashev, and S. I. Kartashev, "Reconfigurable Multicomputer Networks for Very Fast Real-Time Applications," *Proc. 1982 Nat'l Computer Conf.*, Vol. 51, AFIPS Press, pp. 167-184.
14. S. P. Kartashev and S. I. Kartashev, "Reconfiguration of Dynamic Architectures into Multicomputer Networks," *Proc. 1981 Int'l Conf. on Parallel Processing*, pp. 133-140.
15. S. P. Kartashev and S. I. Kartashev, "Fault-Tolerant Reconfigurable Multicomputer Network," *Proc. 1983 Nat'l Computer Conf.*, AFIPS Press, Vol. 52.
16. R. Negrini, M. G. Sami, and R. Stefanelli, "Fault-Tolerance in Supercomputers: Problems and Solutions for Array Processors," *Computer*, this issue.
17. W. W. Chu, D. Lee, and B. Iffla, "A Distributed Processing System for Naval Data Communication Networks," *Proc. 1978 Nat'l Computer Conf.*, Vol. 47, AFIPS, 1978, pp. 783-793.
18. M. P. Mariani and D. F. Palmer, *Tutorial: Distributed System Design*, IEEE, New York, 1979.
19. W. W. Chu, L. J. Hollway, M. T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, Vol. 13, Nov. 1980, pp. 57-70.
20. S. P. Kartashev and S. I. Kartashev, "Performance Optimization in a Reconfigurable Binary Tree," *Proc. 10th Euromicro Symp. on Microprocessing and Microprogramming*, Copenhagen, Aug. 27-30 1984, pp. 305-315.
21. S. I. Kartashev and S. P. Kartashev, "Dynamic Architectures: Problems and Solutions," *Computer*, Vol. 11, July 1978, pp. 26-40.
22. S. P. Kartashev and S. I. Kartashev, "Data Exchange Optimization in Reconfigurable Binary Trees," accepted for publication in *IEEE Trans. on Computers*.

The Kartashevs' biographies appear following the Guest Editors' Introduction, on page 13.

Questions about this article may be addressed to Svetlana Kartashev, Dynamic Computer Architecture, Inc., 933 Oleander Way S., St. Petersburg, FL 33707.