Final Report: Program Decomposition for Multiple Processor Machines

IBM, Yorktown Grant

July 82 - June 83[*]

Computation Structures Group Memo #244

3 December 1984

Arvind

David E. Culler

[*] This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Work cited here was funded through International Business Machines Corporation agreement dated 5/10/82. This work is part of a larger dataflow project funded primarily by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661.

Abstract

The Functional Languages and Architecture Group at the Massachusetts Institute of Technology have been investigating many aspects of program decomposition, code-generation, and resource management for a dataflow multiprocessor, the Tagged Token Dataflow Architecture. This report summarizes the developments in these areas. Static analysis techniques serve to guide how programs are mapped onto the machine for efficient execution. A certain class of nested loops is shown to exhibit a "hyperplane of parallelism" that can be exploited to assign essentially sequential tasks to processors. In general, dataflow programs require dynamic storage management and dynamic allocation of work to processors. A dynamic resource management system for the Tagged Token Dataflow Architecture is outlined. This resource management system structures the machine resources to reduce the complexity of resource allocation. Blocks of work are assigned dynamically to collections of processors. The mapping of individual activities to processors within a collection is dictated by parameters set by the compiler and is carried out automatically by the hardware.

Key words and phrases: computer architecture, dataflow, multiprocessors, multitask, program decomposition, resource management, von Neumann architectures.

Final Report: Program Decomposition for Multiple Processor Machines

1. Introduction

Members of the Functional Languages and Architectures Group at M.I.T. Laboratory for Computer Science investigated various aspects of program decomposition, code-generation, and resource management in the Tagged Token Dataflow Architecture under funding from the International Business Machines Corporation through T. J. Watson Research Laboratory. Substantial progress was made in a variety of areas. As we came to understand the problems involved, the architecture evolved to meet new requirements. Thus, theoretical investigations were continually mixed with development of simulation tools, compilation techniques, and re-evaluation of the architecture. One prevailing realization of this project is that the ground is very fresh; there is much more work to be done.

The issues in program decomposition for a dataflow multiprocessor are vastly different from those in more conventional contexts. Dataflow programs offer fine-grained parallelism independent of any particular machine feature. Dataflow machines can exploit diffuse, unstructured parallelism. Most parallel computers in use today exploit specific, highly structured forms of parallelism dictated explicitly in the program code. We would like to introduce this report by explaining further the differences between a dataflow machine, such as the Tagged Token Dataflow Architecture, and other parallel architectures in regard to program decomposition.

Consider first the kinds of high-speed parallel computers developed to date (*e.g.*, Illiac IV, Cray 1, and Cyber 205). These machines offer specific vector constructs for parallel operation on arrays of data. A program executes sequentially until a parallel operation is encountered. The parallel operation is performed, and sequential execution resumes. There is no flexibility in how these parallel operations are performed. On a pipelined processor, data is streamed from memory, through the processor, and back to memory. Memory conflicts slow the pipeline beat. On an array of processors, arrays of data are fetched in parallel. An operation is performed and an array is written back. If there is contention for memory or conflicts in transferring data from memory modules to processor modules, the entire machine waits until the last item of data is transferred. In these machines, program performance can be greatly affected by the manner in which arrays are mapped onto storage. However, only initial progress has been made in optimizing storage allocation. The primary program decomposition issue is how to expose the specific structures that the machine can exploit. This can be thrown entirely on the user by offering special parallel constructs in the programming language or can be addressed as a compilation problem; recognize vector constructs in sequential code.

Loops in a sequential language can be transformed into vector constructs if there are no dependencies between iterations. Dependencies arise through the use and subsequent reuse of storage. If a variable is written by statement S_i and later read by statement S_j, with no intervening writes, there is a data dependency between S_i and S_j; information is conveyed from S_i to S_j via this variable. If variables are only written once, only data dependencies are manifested. However, to save space, programmers generally write into variables repeatedly. If a variable is read by S_i and later written by S_j, or written in both S_i and S_j, there is an artificial dependency between S_i and S_j, even though no information is conveyed between the two instructions. If there is any dependency (real or artificial) from S_i to S_j, the two instructions must be executed in the order specified by the sequential program; they can not execute in parallel.

Professor Kuck and others have worked extensively on exposing vector constructs in existing FORTRAN programs. The first step is to recognize dependencies and construct a graph representing the potential dependencies between pairs of instructions. There is a potential dependency from S_i to S_j if (1) some variable appears in both S_i and S_j, (2) this variable is written in at least one of the two instructions, and (3) there is a control sequence in which S_i is executed before S_j such that the shared variable is not written between S_i and S_j. There are a variety of problems with this sort of dependency analysis. (1) It breaks down in the presence of aliasing[1], because syntactically distinct variables may denote the same location when the program is executed. (2) The potential dependency graph is a crude characterization of the actual dependencies (especially in unstructured code, as generally found in large FORTRAN programs) because it must account for all possible control sequences. In many cases, a sequence of jumps, branches, and conditionals that apparently leads from S_i to S_j may in fact be impossible because of mutually exclusive conditions on the branches or it may be rarely traversed. (3) For array references, subscript analysis is required to determine if two references may potentially denote the same location. Banerjee's work [3, 4] offers fairly powerful techniques for detecting dependencies in array references for a restricted class of subscript expressions, appearing in simple loops. A conservative approach must be adopted in constructing the potential dependency graph; a dependency is assumed to exist between a pair of statements, unless proven otherwise. The second step in exposing vector constructs is to transform the program to remove artificial dependencies [17, 5, 11, 4, 8, 12, 14, 18]. Invariably, these transformations introduce additional storage to avoid artificial dependencies. After these transformations have been performed, loops which exhibit recognizable vector structure can be compiled into the specific vector constructs of the target machine.

Another important class of parallel computers exploit unstructured, high-level parallelism in a concurrent sequential process framework. Machines of this ilk have been developed for a variety of real-time applications and as research tools for parallel computing [19][2]. Programs are generally written in languages which support multiple *tasks*, such as Ada. A program is decomposed into a collection of tasks, usually by the programmer. Each task is a fairly large, independent block of computation. Tasks communicate through message passing or shared variables. The programmer has the onus of ensuring proper synchronization between concurrent tasks. The primary program decomposition issue for this class of machines is how to assign tasks to processors. Generally, the collection of tasks is small and stable. An estimate is made (by the programmer) of the communication between pairs of tasks and the processing requirements of each task. A variety of optimization techniques have been developed [15, 16, 6] to determine an assignment of tasks to processors.

---

[1]Aliasing arises when two different identifiers denote the same location. This can happen for example in FORTRAN if variables are equivalenced, if a variable is passed as two of the arguments to a procedure, or if a variable that is declared in COMMON is passed to a procedure.

[2]The current trend in high-speed machines is moving toward exploiting high-level parallelism in addition to structured fine-grained parallelism. The Burroughs BSP, Cray X-mp, and Cray 2 are prime examples.

The situation with dataflow machines is quite different from either of the scenarios depicted above. Dataflow machines exploit unstructured, fine-grained parallelism, but they are not limited to exploiting vector constructs. Dataflow machines also exploit high-level parallelism, but tasks are small, short-lived, and extremely dynamic. Parallelism is exposed without special action on behalf of the programmer and without exotic program transformations. The basic instruction scheduling mechanism takes care of synchronization between concurrent computations automatically. An instruction for a conventional machine specifies (explicitly or implicitly) locations where operands are to be found, a location where the result is to be stored, and the instruction to execute next. An instruction for a dataflow machine is quite different: each instruction specifies the collection of instructions which are to use its result as an operand. Data is transferred explicitly between instructions, rather than implicitly through data storage. An instruction is scheduled to execute when its operands have all been sent to it. There is no instruction counter stepping through the program, as in conventional machines, and no explicit parallel operations. Any instruction can potentially execute in any processor. Parallelism is generated automatically, because whenever two instruction have operands available concurrently, they may execute in parallel. A dataflow program is a data dependency graph. There is no explicit use of storage, and hence no artificial dependencies between instructions. Each procedure or loop in a high-level dataflow program is compiled into a dataflow graph, or *code-block*. As code-blocks are invoked, the program unfolds into a larger graph. The primary program decomposition issue is how to assign work to processors as the program dynamically unfolds. Invoking a code-block is analogous to spawning off a task, but a task of a restricted nature because the only form of interaction is the transfer of arguments and results. Each code-block invocation represents a block of computation which manifests a substantial amount of instruction level parallelism and can potentially be assigned to any collection of processors in the system.

The remainder of this report describes the approach we have adopted for program decomposition on the Tagged Token Dataflow Architecture. This approach involves a combination of static and dynamic assignment of work to processors. The collection of processors in the system are partitioned into smaller collections called *domains*. Code-block invocations are dynamically assigned to domains by the resource management system. The work comprising a code-block invocation is distributed automatically over processors in a domain based on parameters set by the compiler, as a result of static program analysis. Section 2 of the report describes some of the static analysis techniques we have developed. Section 3 describes the hardware structures which allow for automatic distribution of work within a domain and the resource management system which assigns code-block invocations to domain. Section 4 describes some of the simulation tools we have developed for experimenting with program decomposition on the Tagged Token Dataflow Architecture.

2. Static Decomposition of Programs

Experience with conventional multiprocessors has demonstrated that programs perform well only when their structure matches the underlying structure of the architecture [9]. We believe this to be much less of a problem with dataflow multiprocessors for two reasons. (1) The dynamic instruction scheduling mechanism provides tolerance to long, unpredictable communication latencies, because while one instruction is waiting for data, others continue to execute. (2) A significant amount of flexibility is allowed in the way a program is mapped onto the machine. We employ static program analysis to characterize the behavior of portions of a dataflow program, and use this characterization to map programs onto the machine. The basic idea can be illustrated with a simple example. The program shown in Figure 2-1 reduces a matrix to a column vector by summing the elements in each row. A total of *n* instances of the inner loop are invoked, each executes *n* iterations. The data dependencies can be seen clearly by unfolding the loops, as shown in Figure 2-2. Each node represents the computation involved in a single iteration of the inner loop. The arcs represent data dependencies between iterations. We often call this unfolding of loops a dependency lattice, due to its regular structure.

```
procedure Reduce(A,n)
(init R <- <>
 for i from 1 to n do
   R[i] <- (init sum <- 0
            for j from 1 to n do
              new sum <- sum + a[i,j]
            return sum)
 return R)
```

Figure 2-1: A Simple Dataflow Program with Nested Loops

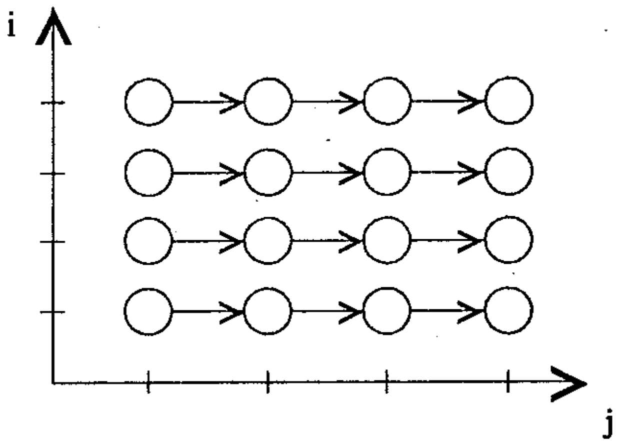
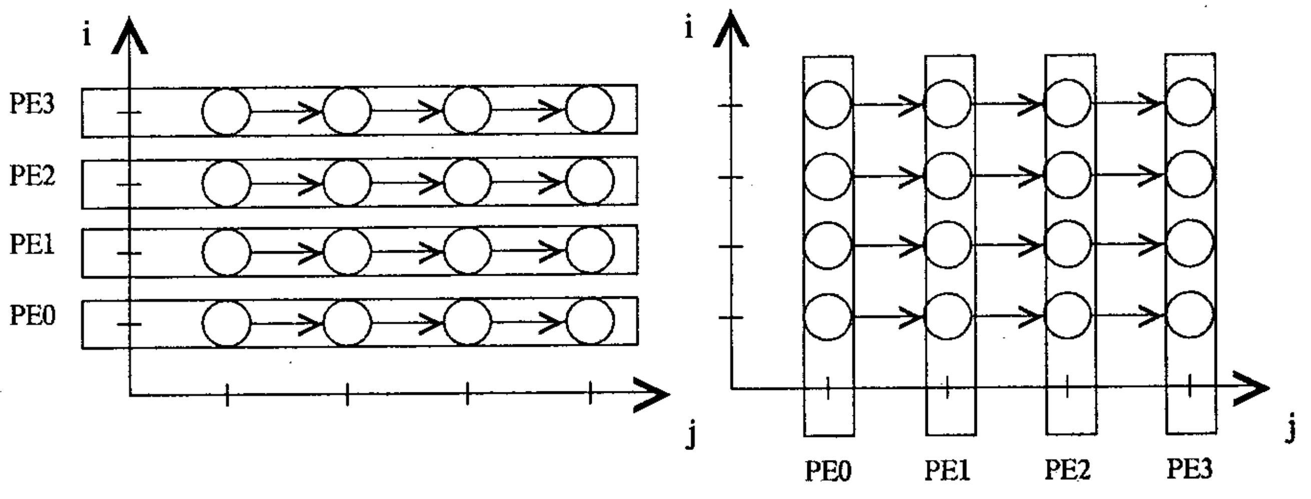


Figure 2-2: Dependency Lattice for Reduce

Given a machine with *n* processing elements (PEs), there are two obvious ways to map this program onto the machine. (1) Assign all iterations of the ith instance of the inner loop to the ith PE. (2) Assign the jth iteration of each instance of the inner loop to the jth PE. These two mappings are described in Figure 2-3. The Assignment 1 is superior because all processors operate concurrently on independent tasks. The parallel structure of the machine is aligned with the parallelism in the program. With the Assignment 2, the computational activity sweeps across the processors, leaving many idle while others work.

If we adopt an abstract view of dataflow program execution, we can see why Assignment 1 is intuitively better than Assignment 2. The basic rule in dataflow computation is that instructions



(1) Project along j axis

(2) Project along i axis

Figure 2-3: Mapping Strategies for Reduce

execute when there operands become available. If we ignore for a moment how a particular machine would perform this computation and focus on the abstract model, this program has a clear, natural behavior. Initially, the first iteration of each instance of the inner loop is enabled for execution. As iterations complete, they enable their successors. A wavefront of activity sweeps across the dependency lattice normal to the j axis, as suggested in Figure 2-4. Assignment 1 effectively projects the dependency lattice along the j axis; thus distributing the parallel activity across the processing elements, and assigning a sequential task to each one. The machine operates in concert with the natural behavior of the program. Assignment 2 assigns a collection of parallel activities to each processor.[3].

These ideas can be formalized to provide a characterization for a broad class of programs. A k-deep nested loop unfolds into an k-dimensional lattice, with one iteration of the inner loop at each node in the lattice. Each vector $(x_1,x_2, ..., x_k)$ defines a collection of k-1 dimensional hyperplanes, normal to the vector. A loop exhibits a *hyperplane of parallelism* if there is a vector $\mathbf{X} = (x_1,x_2, ..., x_k)$ such that for any hyperplane Π normal to $\mathbf{X}$, once all nodes to one side of Π have been computed, all nodes within Π can be computed simultaneously. In the simple example above, the j axis defines such a hyperplane of parallelism. Once the node to the left of the line $j = c$ have been compute, all nodes on the line $j = c$ can be computed in parallel. Culler demonstrated [7] that in general, if node $\mathbf{I} = (i_1, i_2, ... ,i_k)$ depends on nodes $\mathbf{I} + \mathbf{B}_1, \mathbf{I} + \mathbf{B}_2, ...$ for some collection of

[3]Actually, Assignment 2 is not as bad as it might appear; it will require twice the computing time as the Assignment 1. PE_{i+1} will not wait for PE_i to finish all its work; as soon as PE_i finishes one iteration, data becomes available for P_{i+1} to use. The wavefront of activity will tend to sweep across the lattice at a 45° angle to the axes. This kind of adaptability makes dataflow multiprocessor less sensitive to the quality of the program decomposition than conventional multiprocessors.

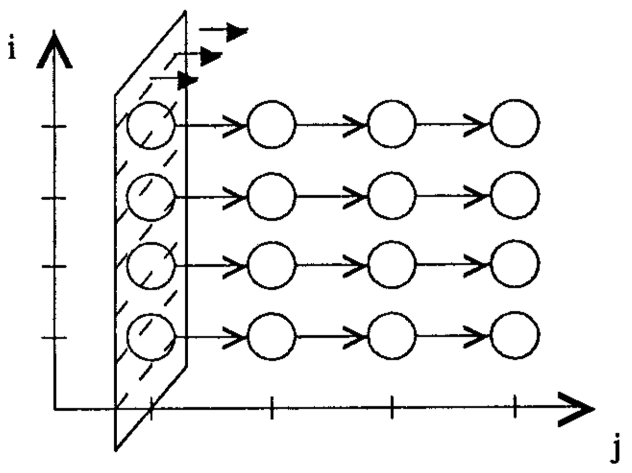


Figure 2-4: Wavefront of Activity for Reduce

constant vectors $\mathbf{B}_1,\mathbf{B}_2,...$ then either the loop exhibits a k-1 dimensional hyperplane of parallelism or the loop is illegal because it specifies cyclic data dependencies. This result follows directly from the Strong Duality Theorem of Linear Programming. Thus, all legal loops which involve array references of the form $A[i_1+b_1, i_2+b_2,...,i_k+b_k]$ and have boundary conditions that are independent of the data computed in the loop exhibit a hyperplane of parallelism. The result suggests an attractive way to map nested loops onto a collection of processors; project the k dimensional lattice onto a k-1 dimensional hyperplane of parallelism to collapse the sequential aspect of the program, then subdivide the resulting hyperplane among the processors. Such a mapping will tend to distribute the parallel activity across processors and keep sequential tasks local to a processor. Standard linear programming techniques (*i.e.*, simplex algorithm) suffice to determine whether such a hyperplane exists, and to find one if it does exist.

As a more realistic example, consider the program for successive relaxation shown below. It appears that a recurrence is present along all three indices. Nevertheless, a sequence of planes can be found such that once all nodes in the region between a plane and the origin have executed, all nodes within the plane may execute simultaneously. In this case, the planes normal to the vector <1,1,2> satisfy this property. Figure 2-5 demonstrates the flow vectors to the point <1,1,1>. The dashed lines describe the plane of parallelism.

```
procedure SOR(X0,f,n,kmax,w,h)
! This computes a solution to the Dirchelet problem in a square
! by Successive over-relaxation.
! X0 gives initial distribution including boundary conditions.
! The grid is square with indices running from 0 to n+1.
! kmax is the number of relaxations to perform.
! w is the relaxation factor.
! h is the time step.
!
(initial u <- <0: X0>
for k from 1 to kmax do
  new u[k] <-
    (initial x <- <0:u[k-1,0], n+1:u[k-1,n+1]>
    for i from 1 to n do
      new x[i] <-
        (initial x_row <- <0: u[k-1,i,0], n+1: u[k-1,i,n+1]>
        for j from 1 to n do
          new x_row[j] <-
            (u[k-1,i,j] + w*(u[k,i,j-1] + u[k,i-1,j] + u[k-1,i,j+1]
                   + u[k-1,i+1,j] - 4*u[k-1,i,j] - h*h*f[i,j])/4)
        return x_row)
    return x)
return u[kmax])
```

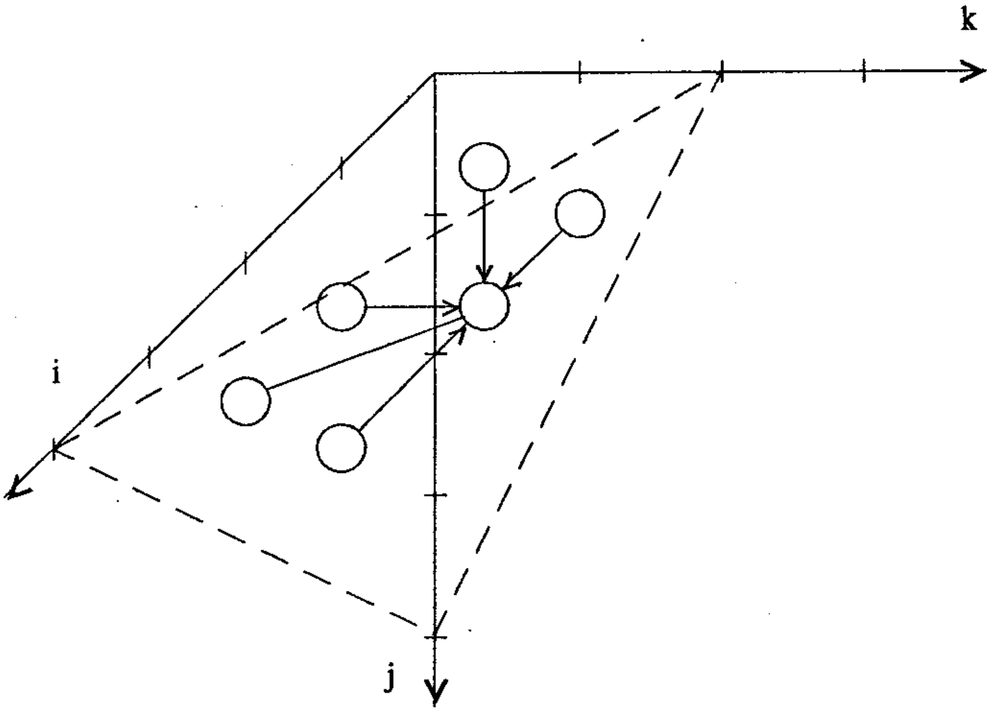


Figure 2-5: Flow vectors in SOR

This mapping technique is appealing, but has its drawbacks. First, complicated hyperplanar projections will require sophisticated hardware to compute where a given node it to be performed. Secondly, the hyperplane of parallelism may not be aligned with the flow of communication. The lattice shown in Figure 2-6 is a prime example. The hyperplane of parallelism is defined by the vector (1,1). However, if the lattice is projected along this vector, all data transfers are external (*i.e.*, from one PE to another). External communication is generally more costly than internal communication. Projecting along either the i or j axis may yield a better execution behavior, even though the parallelism in the machine is not aligned with the hyperplane of parallelism. The wavefront of activity will still tend to sweep across the lattice normal to (1,1).

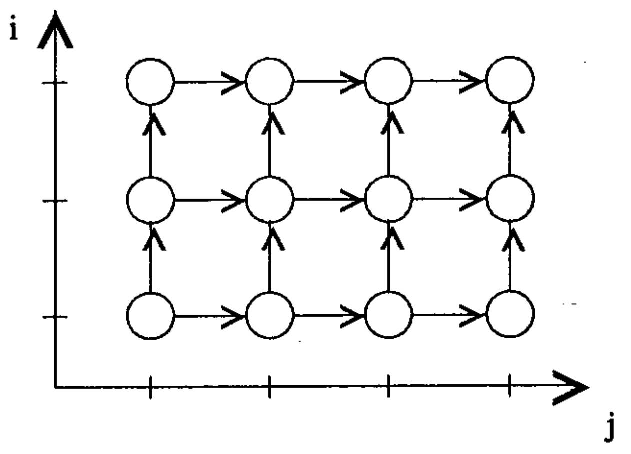


Figure 2-6: Lattice with Unaligned Flow of Data

For the case where the number of iterations is much larger than the number of processors, it is important to determine how to group portions of the lattice to be assigned to one processor. We investigated two kinds of groupings [7]. Assuming that the lattice has been collapsed to form n tasks, the ith task can be assigned to processor i MOD k or to $\lfloor i \cdot k / n \rfloor$. The second approach provides more locality and invariably requires less external communication than the first.

The approach suggested here should be compared with similar work targeted at vector processors. Kuck [11] and Lamport [13]have proposed a similar wavefront method for transforming loops into vector constructs[4]. The collection of nodes within a hyperplane of parallelism can be computed in parallel by a vector operation, before advancing to the next. To exploit this parallelism, the loop must be transformed so that the outer loop index effectively advances along the normal to the hyperplane of parallelism. This introduces a variety of problems. (1) Since the loop is actually transformed, extreme care is required to guarantee that the resulting program is equivalent to the original. (2) Unless the normal is aligned with one of the original loop indices, the index calculations in the transformed loop become very complex. (3) The access pattern for the data involved in each vector operation also becomes complex; generally more complex than vector machines support. Our goal is to characterize the behavior of a program and to determine how to

[4]The theory is much simpler in the case of conventional machines because data dependence arises from reading a value stored by a lexicographically previous iteration. A simple back substitution suffices to determine the hyperplane. There is no possibility of cyclic dependencies.

map it onto the machine, rather than to remove recurrences. Program correctness can not be affected by the mapping. Simple mappings which fail to capture the hyperplane of parallelism execute correctly, exhibit ample parallelism, and often require less external communication than those that capture the hyperplane precisely. The actual execution activity will tend to approximate the hyperplane of parallelism, if the mapping is reasonably close. Also, this technique can be employed for loops which fail to meet the exact conditions of the hyperplane result; since the program is not changed in any way, approximate characterizations provide useful mapping information and still allow the program to execute correctly.

The dynamic scheduling mechanism of dataflow processors influences program decomposition significantly. Decomposition issues are entirely divorced from program correctness issues; a program will execute correctly under any decomposition, but will execute more efficiently under some than others. The efficiency of the machine is affected by the quality of the program decomposition, but to a lesser extent than conventional machines. The machine does not rely on specific parallel constructs, each processor begins computing as soon as data is available. However, the dynamic scheduling mechanism introduces a new class of problems. A program can not accurately be viewed as simply a sequence of loops, as is customary with FORTRAN programs. Rather, it is a collection of concurrent producer-consumer relationships. Thus, it is not necessarily wise to focus on minimizing the time required for a given loop, since all resource allocation decisions effect all other co-existing portions of the program. Moreover, tokens only travel on the arcs in theory; in practice they are stamped with some kind of tag and must be navigated to the place of execution. Mapping policies must be built upon some efficient set of hardware mapping primitives.

3. Resource Management in the Tagged Token Dataflow Architecture

Static analysis techniques, such as those presented in the previous section, aid in mapping portions of a program onto collections of processors. However, they are not powerful enough to deal effectively with dataflow programs in the large. Dataflow programs are extremely dynamic, so it is difficult to predict the overall structure of the collection of co-operating tasks that will exist at any time. Thus, a dynamic resource management system has been developed for the Tagged Token Dataflow Architecture to assign blocks of computation to collections of processors, as a program unfolds dynamically. This assignment is determined primarily by the work load experienced by the various processors. This section outlines the architecture, examines the trade-offs in static and dynamic allocation of resources, and demonstrates how the dynamic resource management system assigns code-block invocations to collections of processors, called *domains*. The distribution of work within a domain is dictated by static analysis, as suggested in the previous section.

3.1. Architectural Background

The Tagged Token Dataflow Architecture (TTDA) is composed of a number of Processing Elements (PEs), connected by a packet switched communications network. Each PE is a complete dataflow computer. The basic organization is shown in Figure 3-1. The PE consists of a number of asynchronous, pipeline stages, connected by FIFO buffers. The various stages form three subsystems. The subsystem shown toward the right in Figure 3-1, performs the basic instruction processing. The stages of this pipeline reflect the essential steps in the processing of a dataflow instruction: detect when data has arrived to enable an instruction, fetch the instruction, compute the result, generate result tags, and finally dispense the result tokens. The subsystem to the left provides storage for data structures. The structure store incorporates a number of innovative ideas to allow for sharing of information without constraining parallelism. The benefits of this approach are presented in Arvind and Iannucci [2]. A detailed design of the controller for the data structure storage is presented in Heller [10]. The center subsystem includes a PE controller, which provides a variety of support operations, including input/output, block transfers, and access to the resource management system.

3.1.1. Tokens and Tags

In the Tagged-Token Dataflow Architecture values are carried on *tokens*, which are passed from one instruction to the next. The arrival of data causes the corresponding instruction to be fetched, unlike a conventional computer in which the execution of an instruction causes data to be fetched. There is no program counter in this machine. Each token carries a *tag*, in addition to a data value, which specifies the instruction to be executed. The tag contains essentially three items of information, the address of the PE which is responsible for executing the instruction, the address of the instruction to execute within this PE, and the *context* in which the instruction is to be executed. The PE address is required because the program, that is, the *code-block*, is spread over many PEs, and tokens must be freely transferred between PEs. The contextual information is required because the TTDA is a parallel computational model; many logically distinct activations of a given code-block may be in execution simultaneously. There must be a way to distinguish the various activations so that token belonging to different activations do not interact. All tokens belonging to a given activation carry the same context identifier or *color*. Thus, two tokens are destined for the same instance of an instruction if and only if their tags match.

3.1.2. Instruction Processing

Upon arriving at a processing element, a token enters the waiting-matching section. The tag it carries is compared against the tags of all the tokens resident in the waiting-matching store. Instructions are limited to two operands, so if a match is found, the corresponding instruction is enabled for execution. The two matching tokens are purged from the waiting-matching store and forwarded to the *instruction fetch* section. The instruction specified in the tag is fetched from program memory, along with any required constants. The data values are aligned, and an operation packet is sent to the *ALU* for processing. In parallel with the ALU, the *compute-tag* section forms tags for result tokens, based on the destination list of the instruction and contextual information on the input tag. The result values and tags are merged to form tokens and passed on to the communication network, whereupon each is delivered to the PE specified in its tag.

3.1.3. Tolerance to Communication Latency

In many respects, a multiprocessor setting presents a fundamental architectural challenge. Communication latency between processors is generally large and unpredictable. Thus, for a multiprocessor architecture to be successful, the individual processing elements must be extremely tolerant to communication latency. The PEs which comprise the Tagged-Token Dataflow Architecture meet this challenge. Note that once an instruction is enabled, it may be processed to completion without further communication with other PEs. The pipeline is never held up by communication latency. Waiting only takes place in the waiting-matching section. Completion detection for external communication is provided naturally by the basic instruction scheduling

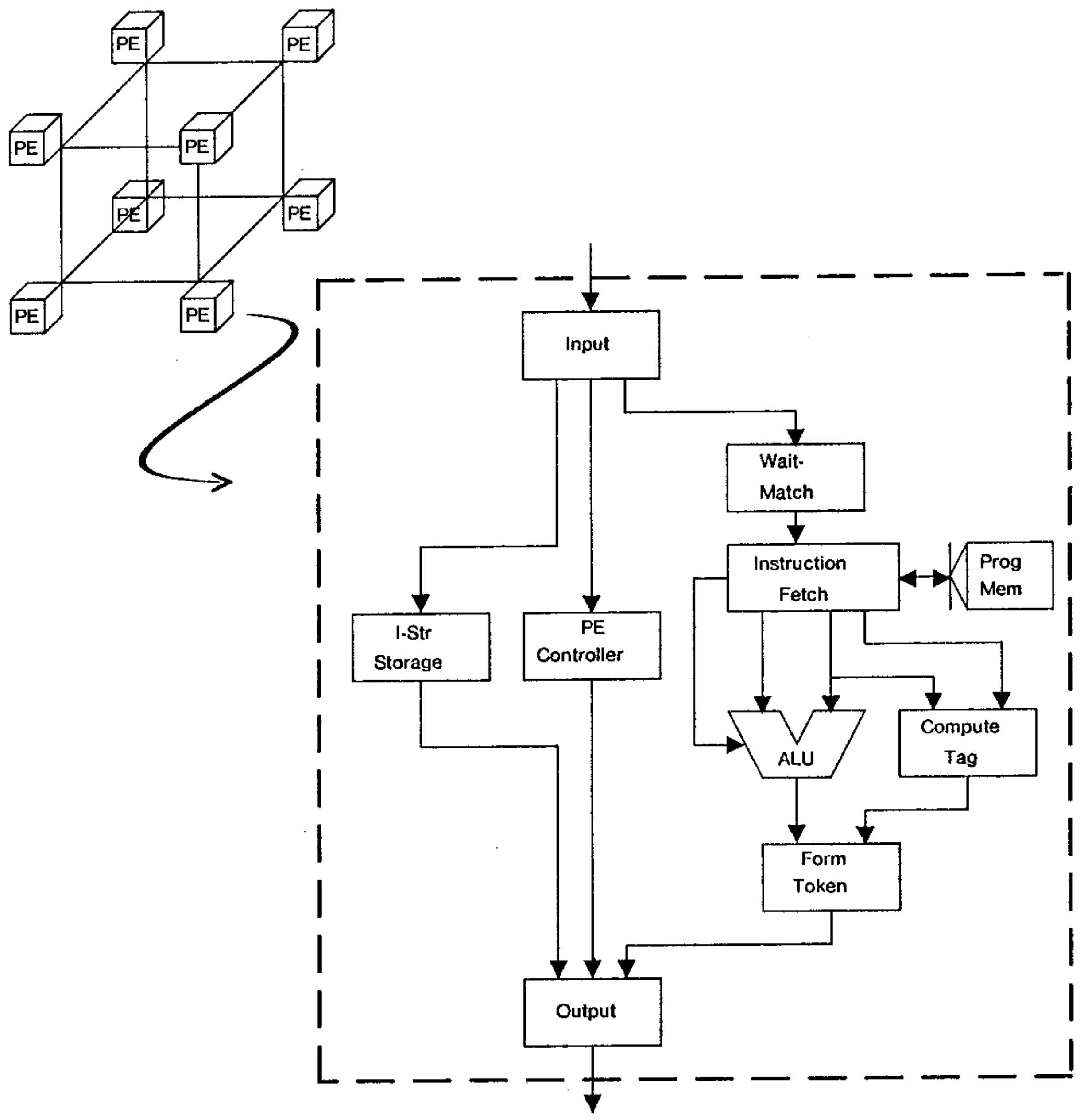


Figure 3-1: The Tagged Token Dataflow Architecture

mechanism; when data arrives an instruction is scheduled. This dynamic scheduling, coupled with the ability to interlace many independent threads of computation, allows for overlapped requests to the communication system, tolerates long latencies, and tolerates unordered responses.

3.2. Managing Resources

The description of the architecture presented above assumes that the program graph is in place, distributed appropriately over the machine. We now examine the process of executing programs on the Tagged-Token Dataflow Architecture at a somewhat higher level. It focuses on how resources are managed and how work is distributed over the machine.

3.2.1. Code-block Activation

Each code-block activation in a dataflow program represents a task, which can be executed anywhere on the machine. These tasks are fairly small; they correspond to individual procedures or loops in a high level dataflow language. A new task is dynamically created every time a procedure or loop is invoked. Thus, the structure and quantity of tasks in execution is extremely variable and dynamic. This raises a fundamental question. In what manner should block activations be distributed over the collection of processors? At one extreme, it is possible to follow a completely static approach; define at compile time a mapping from code-block activations to processors. When a code-block is invoked simply compute the PEs that are responsible for the activation. If sufficient resources are not available to support the activation in the designated PEs, the program aborts. The other extreme is a completely dynamic approach; build a run-time system (a resource manager) to allocate code-block activations to PEs based on the machine status and the availability of resources. When a code-block is invoked, a request is sent to the manager to determine where the activation should be performed.

3.2.2. Static vs. Dynamic Allocation

Each of the two approaches offers advantages and disadvantages A static approach offers very low run-time overhead, but a dynamic approach offers more generality. Most high speed machines today adopt a purely static approach; essentially all scheduling is determined by the code which is generated. Storage is allocated statically. This caters fairly well to scientific applications written in FORTRAN. This kind of approach can be extended to deal with a multiple processor environment, if there are machine language constructs to cause a sequence of instructions to begin executing on a given processor. If the decomposition is just right, the performance can be extremely good. There is little overhead caused by the distribution of work at run-time. However, if the structure of the program is not closely related to the structure of the machine, the performance may suffer dramatically [9]. A static decomposition may fail because work is assigned to a processor with insufficient resource, even though other processor could support the task.

Functional languages require dynamic resource allocation and thus preclude a purely static approach. Various aspects of a program's structure can be gleaned through static analysis, but these only influence, rather than determine, the allocation of resources. Dynamic allocation on the Tagged-Token machine is undertaken at the level of code-block invocations, each of which is assigned to a group of PEs at run-time. This level of granularity reduces the overhead associated with dynamic allocation. The distribution of the individual activities which comprise a code-block invocation is determined statically. If the code-block has a clear structure, *e.g.*, a loop or a recursive procedure, this is exploited by hardware structures which allow instances of the code-block to be distributed automatically. The results of static analysis influence how this automatic distribution is performed.

When a code-block is invoked, the status of the machine can be examined to determine where there are resources available to perform the activation. This provides a way to dynamically balance the workload over the machine, in addition to avoiding the problems with the static approach mentioned above. David Culler has developed a dynamic resource management system as part of a detailed simulation of the Tagged Token Dataflow Architecture along these lines. This resource management system has also been adopted for the emulated dataflow machine. We are currently experimenting with various allocation algorithms. Collections of processors are grouped together to provide an efficient way to exploit the special structure of loops and recursive procedures. As static analysis techniques develop further, we will consider more aggressive optimizations to exploit special structures. The remainder of this section outlines the design of the resource management system.

3.2.3. Hierarchy of Resources

When a code-block is invoked, the task of executing it is assigned to a collection of PEs. A variety of resources are required to support the activation: program memory, code-block registers, colors, etc. Co-ordinating resource allocations between a variety of PEs can be extremely cumbersome, if PEs are allowed to co-operate in arbitrary ways. In order to keep the complexity of resource management tractable, a hierarchy is imposed on the set of system resources. In effect, the manager deals with blocks of resources and allows the hardware to distribute them at a finer grain automatically. This also allows the number of requests to the manager to be reduced.

The notions of *physical domain* and *physical subdomain* are introduced to facilitate selecting a set of PEs for an invocation. The collection of PEs in the system are divided into disjoint *physical domains* (PD), each being a set of consecutively addressed PEs. This partition is not allowed to change while programs are running. Each Physical Domain may be further partitioned into a set of disjoint *physical subdomains* (PSD), again each is a set of contiguously address PEs. The size of the PSD is dynamic and will vary for different code-blocks in a given PD. A code-block activation is assigned to a domain, and a complete copy of the code is placed in each subdomain.

To simplify memory management, we require that memory allocation be identical in every PE in a given domain. The code-block is split into as many sub-blocks as there are PEs in a subdomain. Each PE receives one sub-block, all starting at the same base address. This allocation is replicated for each subdomain in the domain, as shown in figure 3-2. For computational efficiency we require that all PD and PSD be a power of two in size.

Each copy of the code-block may be used by many concurrent activations and each of these may require mapping information to be stored local to the PEs, so that result tags can be generated. Again, by grouping activations together this mapping information can be kept within reason. Each PE contains a set of 256 *code-block registers* (CBR). Each code-block register contains a code-base address and information pertaining to the mapping of the code-block onto the domain. Hues, which serve to distinguish concurrent activations, are partitioned into a set of *Colors.* Each Code-block register can support 16 Colors; additional information can be associated with each color. Code-block registers, like program memory, are allocated uniformly throughout each domain. With this partitioning, the manager views the system resources in terms of domains, code-block registers, and colors.

To illustrate the role of the manager, consider the process of code-block activation. A request is sent to the resource manager. It first selects a domain with sufficient resources to support the activation. If the code-block is not present in this domain, it will have to be loaded. In this case, the subdomain size may be set as suggested by the compiler, or as determined by the resource manager.

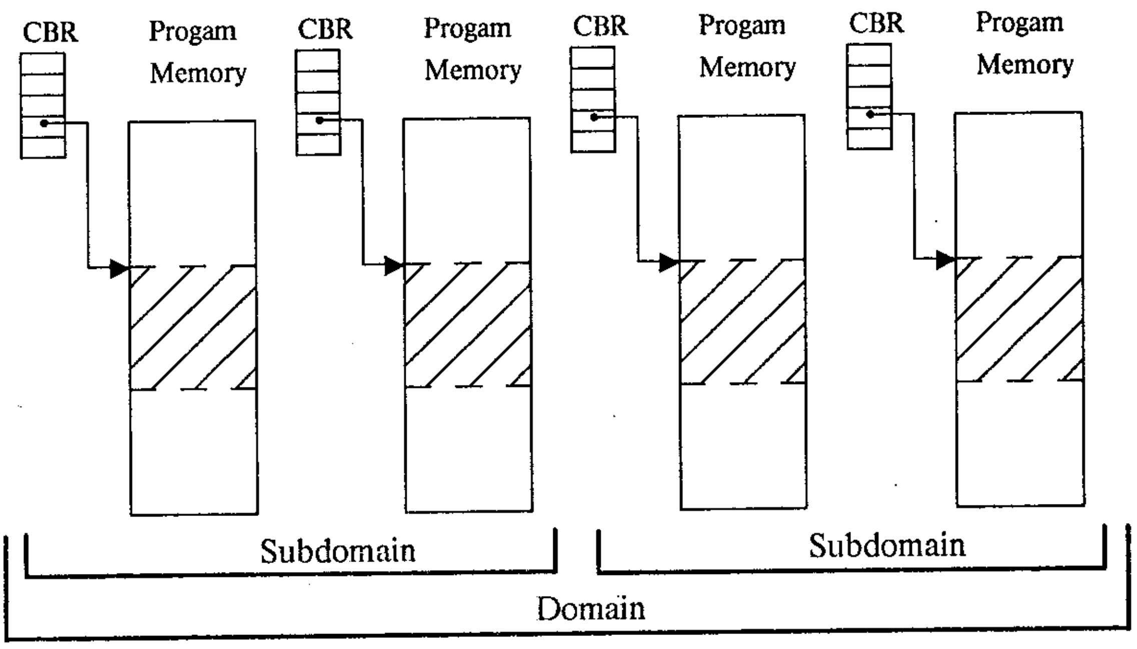


Figure 3-2: Allocation of Program Memory Across a Domain

A code-block register must be allocated, and a set of colors, relative to this CBR, is assigned to this activation. Finally, program memory is allocated throughout the domain and the code is loaded. Mapping information is established in each PE to describe the disposition of the code-block, as part of the CBR. Further information pertaining to each color can also be provided. The code-block is now ready to execute. The manager allocates argument and result structures and sends descriptor to both the caller and the newly activated code-block. The code-block may now execute on its own except for invocation and resource allocation requests. Subsequent invocations of the same code-block may share the CBR, if colors are available. If not, a new CBR must be allocated, but it may share the copy of the code. In either case the mapping parameters will have to be the same as the original, since the code has the same disposition. New color information may be provided.

These resource management concerns essentially dictate the structure of the tag carried on tokens. It consists of five fixed size fields <PE#, CBR, Color, Initiation #, Relative instruction Address>. The code-block register gives the base address of the code and the base address of the color information. This allows instructions and constants to be fetched from program memory. The CBR also describes the disposition of the code-block across the domain, so the hardware can generate tags for result tokens.

3.2.4. Distribution of Work Within a Domain

The resource manager provides a mechanism for initiating a code-block activation. The distribution of work within a domain is dictated by parameters set by the compiler. These parameters are the mapping primitives upon which static decomposition strategies are based. Two kinds of local distribution of work are supported: distribution of activities within a subdomain, and distribution of repeated initiations across subdomains. These mechanisms are described below.

The simplest class of code-blocks are acyclic graphs, which arise form non-recursive procedures with no internal loops. Each instruction in such a code-block is executed once per invocation. The work involved in an invocation can be distributed over a collecion of processors by simply partitioning the graph and assigning the various partitions to different processors. This partitioning is determined by the compiler and expressed by the order in which nodes appear in the representation of the graph.

Highly repetitive program structures, such as loops and recursive procedures, execute each instruction in the graph many times. Numerous iterations of a loop, or numerous recursive invocations of a procedure may execute in a domain without interaction with the resource manager. Each color has associated with it 256 initiation numbers. Loops and recursive procedures make use of the initiation numbers to distinguish tokens belonging to different iterations, or different recursive invocations. Loops utilize the D operator, which increments the initiation number. Recursive procedures may make use of the R operator, which computes a new initiation number of the form $A * I + B$.

The hardware provides a mechanism to distribute iterations and recursive invocations over processors in a domain efficiently. Recall, each domain can be partitioned into subdomains, and when a code-block is invoked in a domain a copy of the graph is placed in each subdomain. Repeated initiations can be assigned to subdomains by hashing the initiation number in the tag. Instructions for a given initiation are distributed over processors in a subdomain by partitioning the graph as describe above. The subdomain size is determined by the compiler. Initiations can be assigned to subdomains in blocks, *i.e.*, k initiations on the first subdomain, the next k on the second, and so on. So initiation i is assigned to subdomain $\lfloor i/k \rfloor$ MOD s, where s is the number of subdomains in the domain. This provides enough flexibility to express the two mappings for the Reduce example presented in Section 2. The rationale follows along the lines discussed in Arvind, *et. al.* [1].

The results of the static analysis are expected to guide the resource manager in making allocations. In particular, it enters into the process of choosing among domains, choosing subdomain size, choosing k (the number of iterations to keep together), and for deciding where to allocate data structures. Conveying this information is somewhat subtle, however. Completely static information, such as nesting relationships, can be included in the compiler output along with the code. Execution dependent information is supplied by augmenting the graph slightly and sending information along with the request. Currently, only simple information is conveyed in this way, such as the expected number of iterations. But we are investigating more powerful aids of this form

Many loops may exceed 256 iterations, hence a facilities are provided for allocating blocks of colors and for using the next color in the case of initiation number overflow. The compiler generates code to test whether all colors and initiation numbers have been exhausted and to issue a manager request for more colors in case they are.

3.2.5. Compiler Responsibilities[5]

Introducing a resource management system into a dataflow model raised a number of interesting compilation issues. The U-interpreter is entirely independent of resources; it assumes unbounded computational resources. In order to execute dataflow programs on a practical machine, resources must be explicitly requested. They must also be released in some manner. The basic dataflow graphs must be augmented so that the resource manager request is generated when resources are required. Also, the compiler must generate code to make proper use of blocks of resources. For example, to use initiation numbers properly, it is necessary to handle overflows, and so on. The compiler must generate code to determine when resources can be released. This involves detecting completion of code-block activations and proper handling of reference counts for structures.

Detecting completion is somewhat tricky in a dataflow system; some instructions may be waiting for data, even though the value which is to be the final result has been produced. This situation arises particularily in the presence of conditionals. Thus, termination of activity within a code-block activation is not synonymous with returning a result, as it is with conventional machines. Program graphs must be embellished to detect the termination of activity for each code-block activation. For acyclic graphs there is little trouble. A signal token is generated for any operators which have no destinations. These and all the output arcs of the graph form the inputs to a binary reduction tree. The token that falls out the bottom of the tree is the last token of the activation. This technique can be extended to loops, but care must be taken to avoid serializing loops that would otherwise provide parallelism. The special signal token of one iteration is part of the completion tree for the next. In effect, completion detection is serial, even though the loop may be parallel. The detection simply lags behind the loop execution.

One important implication of this incremental completion detection is that it makes it possible to run loops of arbitrary number of iterations using a few colors. The loop is given multiple colors. As it exhausts one color it goes on to the next or requests more. The completion follows behind detecting and releasing colors that are no longer needed and, hence may be recycled.

The compiler must also assist in the detection of parallelism in loops in data structure constructions. When certain restrictions apply, arrays can be modeled as I-structures which allows for a more parallel implementation than for ordinary structures Detecting those arrays which can be implemented as I-structures is difficult in general; it involves analysis similar to the that done by vectorizing compilers during code vectorization. There are, however, several important special cases where I-structures can be detected easily. We have also found several situations in which the techniques of vectorizing compilers can be borrowed and applied. Once I-structures are created by the compiler, they must eventually be reclaimed. Since I-structures are acyclic, reference counting can be implemented. Overhead of maintaining reference counts can be substantially reduced if they are incremented at procedure call and exit only.

[5]Development of the compiler has been carried out by Vinod Kathail under funding from ARPA. Guidelines for compiler development were established through work described in this report.

4. Architecture Development and Simulation

Over the past year, a detailed simulation of the Tagged Token Dataflow Architecture has been developed in conjunction with IBM T. J. Watson Research Center[6]. This simulates the behavior of the hardware down to the individual stages in the pipeline and the finite buffers connecting them. It implements the functional characteristics of each stage as Pascal procedures and carries explicit timing information to model the temporal behavior. Currently this is over 10,000 lines of Pascal code. The basic simulated machine has been operational since June 1983. The period since then has been devoted to developing the resource management system and responding to evolutions in the architecture.

The simulation serves as a 'soft' prototype of the machine. It has allowed the design to be carried through in substantial detail, without the overhead of building hardware. Progress has been incremental in both the simulation and the architecture. In many cases the process of writing simulation code caused us to solidify the design. In some cases, the design was simply insufficient for the task and had to be modified. In particular the relationship of the instruction fetch, tag generation, and token forming stages evolved substantially after we had a very detailed understanding of their tasks.

Modeling the finite buffers proved to be the most subtle aspect of the simulation. The basic requirement was that each stage be represented by a Pascal procedure which could be implemented with no knowledge of buffers or simulator scheduling; it simply receives a token at some time and produces a list of tokens at some later time. The fabric of the simulation interfaces these procedural modules to the time-dependent buffers. The problem is that because a given stage is allowed to run to completion, it actually gets ahead of the downstream stage, and it may appear that a buffer is full when indeed it should not be. We solved this problem by having the downstream station issue a receipt for each token it removes. Whenever the two stations meet in time, it is possible to reconstruct the precise history of insertions and deletions to the buffer. With this mechanism we are able to guarantee that no insertion and no removable is performed too early or too late.

Perhaps the most valuable aspect of the simulation development was the experience with building a resource manager. Since the compiler does not yet support streams or managers, we developed the resource manager as a Pascal program. The basic issues were the same, however, and in the simulation we try to account for the manager overhead appropriately. The current version supports the means-ends analysis described above and block color allocation for loops. We plan to experiment with more sophisticated techniques in the coming months.

5. Conclusion

Our research over the past year has dug deeply into the exigencies of running programs on our machine: generating good code, distributing the work, and managing resources. In many ways this is a very new area of research, primarily by nature of the highly parallel environment. However, the issues addressed in this report are not restricted to dataflow; they are applicable to any

[6]This work is being carried out on an IBM 4341 on long term loan from the IBM Glendale Laboratory in Endicott, New York. The simulation development is primarily through ARPA funding, however, partial support for Keshav Pingali was provided under the IBM grant reported here.

multiprocessor system comprising many autonomous processors. Dataflow simplifies some aspects and complicates others. We have yet to study how restricting the programming language could simplify resource management and architecture. For example, Id treats procedures as first class data objects; restricting this would simplify the program analysis at the expense of programming generality. Also, we have not dealt specifically with issues like splitting a code-block into two pieces efficiently. The techniques for these kinds of problems are well represented in the literature and are not as essential to the success or failure of the approach as the issues presented here. The topics discussed in this report are still under active investigation. In particular much work needs to be done to be able to abstract that information from a program which would be most useful in guiding the resource management system during execution. This involves analyzing programs to determine which information is essential for resource management, developing compilation techniques to generate it, and resource management techniques to make use of it.

References

1. Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali. The Tagged Token Dataflow Architecture. To be published as an MIT Technical Report.

2. Arvind, and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. Proc. of the 10th International Symposium on Computer Architecture, June, 1983.

3. Utpal Banerjee. Data Dependence in Ordinary Programs. Master Th., University of Illinois at Urbana-Champaign,November 1976. Gives necessary conditions for dependence in the case of polynomial subscripts..

4. Utpal Banerjee, Shyh-Ching Chen, David J. Kuck, Ross A. Towle. "Time and Parallel Processor Bounds for FORTRAN-Like Loops". *IEEE Transactions on Computers c-28*, 9 (September 1979), 83-93.

5. J. Cocke and F. E. Allen. "A Program Data Flow Analysis Procedure". *Comm. ACM 19*, 3 (March 1976), 137-146.

6. Coffman, E. and Denning, P.. *Operating System Principles.* Prentice Hall, Inc., 1973.

7. David E. Culler. Decomposition of Nested Loops in a Dataflow System. Unpublished Paper.

8. Ronald Cytron. Improved Compilation Methods for Multiprocessors. Master Th., University of Illinois at Urbana-Champaign,May 1982.

9. Deminet, J. "Experience in Multiprocessor Algorithms". *IEEE Transactions on Computers C-31* (April 1982).

10. Heller, S. and Arvind. Design of a Memory Controller for the Massachusetts Institute Technology Tagged Token Dataflow Machine. 230, Computer Structures Group Memo, October, 1983. Presented at IEEE/ICCD '83.

11. Kuck, D.J. "A Survey of Parallel Machine Organization and Programming". *ACM Computing Surveys 9* (march 1977), 29-59.

12. Kuck, D. J., Kuhn, R. H. Padua, D. A. Leasure, B. and Wolfe M. Dependence Graphs and Compiler Optimizations. Proceeding of ACM Symposium on Principles of Programming Languages, January, 1981.

13. Lamport, L. "The Parallel Execution of DO Loops". *Communications of the ACM 17* (February 1974).

14. Padua, D. A., D. J. Kuck, and D. H. Lawrie. "High Speed Multiprocessors and Compilation Techniques". *IEEE Transactions on Computers C-29*, 9 (September 1980), 763-776.

15. Perng-yi, R., Edward, Y. S. Lee, and Masahiro Tsuchiya. "A Task Allocation Model for Distributed Computing Systems". *IEEE Transactions on Computers c-31*, 1 (January 1982).

16. Stone, H.S. "Multiprocessor Scheduling with the aid of network flow algorithms". *IEEE Transactions on Software Engineering se-3* (January 1977).

17. Towle, R. A. *Control and Data Dependence for Program Transformation.* Ph.D. Th., University of Illinois at Ubana-Champaign, March 1976.

18. Michael J. Wolfe. Techniques for Improving the Parallelism in Programs. Master Th., University of Illinois at Ubana-Champaign,July 1978.

19. Wulf, William, Levin and Harbison. *Hydra/C.mmp - An Experimental Computer System.* Mcgraw-Hill, 1981.