TOWARD A DATAFLOW / VON NEUMANN HYBRID ARCHITECTURE

Robert A. Iannucci

IBM Corporation
- and -
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

ABSTRACT

Dataflow architectures offer the ability to trade program level parallelism in order to overcome machine level latency. Dataflow further offers a uniform synchronization paradigm, representing one end of a spectrum wherein the unit of scheduling is a single instruction. At the opposite extreme are the von Neumann architectures which schedule on a task, or process, basis.

This paper examines the spectrum by proposing a new architecture which is a *hybrid* of dataflow and von Neumann organizations. The analysis attempts to discover those features of the dataflow architecture, lacking in a von Neumann machine, which are essential for tolerating latency and synchronization costs. These features are captured in the concept of a *parallel machine language* which can be grafted on top of an otherwise traditional von Neumann base. In such an architecture, the units of scheduling, called *scheduling quanta*, are bound at compile time rather than at instruction set design time. The parallel machine language supports this notion via a large synchronization name space.

A prototypical architecture is described, and results of simulation studies are presented. A comparison is made between the MIT Tagged-Token Dataflow machine and the subject machine which presents a model for understanding the cost of synchronization in a parallel environment.

Key Words and Phrases: architecture, context switching, dataflow, hybrid, I-structure storage, latency, multiprocessor, name space, process state, split transaction, synchronization, von Neumann

1. Introduction

It is becoming increasingly apparent that the lessons learned in 40 years of optimizing von Neumann uniprocessor architectures do not necessarily carry over to multiprocessors. Compiler technology coupled with simple pipeline design is now used effectively [20, 25, 26, 28] to cover bounded memory latency in uniprocessors. Unfortunately, the situation is qualitatively different for multiprocessors, where large and often unpredictable latencies in memory and communications systems cannot be tolerated by using similar techniques. This is attributable at the architectural level to poor support for inexpensive dynamic synchronization [4]. Specifically, latency cost is incurred on a per-instruction basis, but synchronization on a per-instruction basis is impractical. A scalable, general purpose multiprocessor architecture must address these issues. Traditional compile time sequencing is too weak a paradigm for general purpose machines (*c.f.*, ELI-512 [15], the ESL Polycyclic processor [27]), and traditional run time sequencing mechanisms are not sufficiently flexible (*c.f.*, The IBM 360 Model 91 [1, 31], the Cray-1 [28]).

The overall goal of this study is to discover the critical hardware structures which must be present in multiprocessor architectures to effectively tolerate latency and synchronization costs. This investigation is based on demonstrating that a tradeoff exists between von Neumann instruction sequencing simplicity and dataflow sequencing generality. To explore this tradeoff, a new architecture is developed as a synthesis of the best features of von Neumann and dataflow ideas. Evaluation of this architecture is based on characterizing the differences in various architectural figures of merit (*e.g.*, number of instructions executed, instruction complexity) between the new machine and the well-studied MIT Tagged Token Dataflow Architecture (TTDA) [2, 5, 11, 14].

The remainder of this paper provides technical justification for, and details of, the new architecture. Section 2 describes the work leading up to this proposal, including a brief discussion of von Neumann and dataflow architectures and the ways they address the issues of latency and synchronization. Section 3 discusses how these two apparently dissimilar architectures may be combined into a new architecture. The architecture and instruction set of the new machine are described. Using a newly-developed code generator and simulator for the architecture, Section 4 presents results of the first set of experiments along with relevant comparisons to the TTDA.

2. Background

2.1. Two Fundamental Issues

In [4] we argue that architects of any scalable, general purpose multiprocessor must face two very basic issues in order to exploit parallelism in programs. The first issue is *latency*: the time which elapses between making a request (*e.g.*, a memory reference) and receiving the associated response. Latency often incurs a cost in the form of induced processor idle time and is directly attributable to *physical partitioning of the machine*. The second issue is *synchronization*: the time correlation of related activities. Synchronization also incurs a cost in the form of the fixed time required to perform a synchronization operation plus the time lost to waiting or context switching, and is directly attributable to *logical partitioning of the program*. That is, in order to exploit parallelism in a program it must be *decomposed* into fragments which communicate. Managing this communication while preserving precedence constraints is one primary job of a multiprocessor's synchronization mechanism.

These issues are not only fundamental, they also appear to be strongly related. For example, one possible solution to idling the processor during a long-latency remote memory reference is to switch the processor to another task in the same manner that an operating system will switch contexts when an input/output (I/O) operation is begun. Unfortunately, this requires a highly efficient synchronization mechanism to manage the matching of memory responses with idled, or deferred, tasks.

It was our conclusion that satisfactory solutions to the problems raised for von Neumann architectures can only be had by altering the architecture of the processor itself. Questions raised in this study regarding the near-miss behavior of certain von Neumann multiprocessors (*e.g.*, the Denelcor HEP [22, 30]) led to the belief that dataflow machines and von Neumann machines actually represent two points on a continuum of architectures.

Arvind has suggested that an architecture formed on the principles of *split transaction* I-Structure memory references in a von Neumann framework coupled with data driven rescheduling of suspended instructions in the local memory of each processor would be interesting. Such a machine has the potential of tolerating memory latency and of supporting fine-grained synchronization, and yet (in the strict sense) is neither a von Neumann machine nor a dataflow machine. This suggestion has led me to develop the hybrid architecture presented here. In order to better understand the motivations, the next sections re-examine the strengths and weaknesses of von Neumann and dataflow architectures.

2.2. von Neumann Architectures

Advocates of non-von Neumann architectures (including the author) have argued that the notion of sequential instruction execution is the antithesis of parallel processing. This criticism is actually slightly off the mark. Rather, a von Neumann machine in a multiprocessor configuration does poorly because it fails to provide efficient synchronization support at a low level. Why is this so?

The participants in any one synchronization event require a common ground, a *meeting place*, for the synchronization to happen. This may take the form of a semaphore [9], a register [28, 22], a buffer tag [31], an interrupt level, or any of a number of similar devices. In all cases, one can simply think of the common ground as being the *name* of the resource used (*e.g.*, register number, tag value, etc.). The participants also require a mechanism to trigger synchronization action.

When viewed in this way, it should be clear that the number of simultaneously pending synchronization events is bounded by the size of this name space as well as by the cost of each synchronization operation. More often than not, this name space is tied to a physical resource (*e.g.*, registers) and is therefore quite small, thereby limiting support for low level dynamic synchronization. For most existing von Neumann machines, synchronization mechanisms are inherently larger grain (*e.g.*, interrupts) or involve busy waiting (*e.g.*, the HEP[1] [22, 30]). Therefore, the cost of each event is quite high. Such mechanisms are unsuitable for controlling latency cost. Moreover, since task suspension and resumption typically involve expensive context switching, exploitation of parallelism by decomposing a program into many small, communicating tasks may not actually realize a speed-up.

It is important to observe that these arguments favor the alteration of the basic von Neumann mechanism, and not its total abandonment. For situations where instruction sequencing and data dependence constraints can be worked out at compile time, there is still reason to believe that a von Neumann style sequential (deterministic time order) interpreter provides better control over the machine's behavior than does a dynamic scheduling mechanism and, arguably, better cost-performance. It is *only* in those situations where sequencing cannot be so optimized at compile time, *e.g.*, for long latency operations, that dynamic scheduling and low-level synchronization are called for. One must also keep in mind that, despite any desire to revolutionize computer architecture, von Neumann machines will continue to be the best understood base upon which to build for many years.

2.3. Dataflow Architectures

The MIT Tagged Token Dataflow Architecture, and other dataflow architectures like it, provide well-integrated synchronization at a very basic level. By using an encoded dataflow graph for program representation, machine instructions become self-sequencing. One strength of the TTDA is that each datum carries its own context identifying information. By this mechanism, program parallelism can be easily traded for latency because there is no additional cost above and beyond this basic mechanism for switching contexts on a per-instruction basis.

However, it is clear that not all of the distinguishing characteristics of the TTDA contribute towards efficient toleration of latency and synchronization costs. One very sound criticism is that intra-procedure communication is unnecessarily general. Intuitively, it should not be necessary to create and match tokens for scheduling *every* instruction within the body of a procedure - some scheduling can certainly be done by the compiler. In a dataflow machine, however, data driven scheduling is *de rigueur*. This implies, for instance, that the time to execute the instructions in a graph's critical path is the product of the critical path length and the pipeline depth. One is left to wonder if it might not be possible, even desirable, to optimize this by performing the *necessary* synchronization explicitly, and relying on more traditional (read: *well-understood*) mechanisms for instruction sequencing in the remainder of the cases. The uncertainties in this argument are the fraction of time wherein synchronization is necessary, and the complexity of the mechanisms required.

[1]The HEP also exhibited several synchronization namespace problems: the register space was too small (2K), there was a limit of one outstanding memory request per process, and there was a very serious limit of 128 process status words per processor.

3. Synthesis

A simple view is that von Neumann and dataflow machines are not, in fact, orthogonal but rather sit at opposite ends of a spectrum of architectures. One might speculate that there are families of machines along this spectrum which trade instruction scheduling simplicity for better low level synchronization. One might further speculate that for some figure of architectural merit, taking into account hardware complexity, instruction scheduling flexibility, and synchronization support, that there exists some optimum point between the two extremes, *i.e.*, a hybrid architecture which synergistically combines features of von Neumann and Dataflow.

Starting with the observation that the costs associated with dataflow instruction sequencing in many instances are excessive, others have suggested that dataflow ideas should be used only at the inter-procedural level [23] thereby avoiding dataflow inefficiencies while seemingly retaining certain advantages. This view is almost correct, but ignores the importance of the fundamental issues discussed above. Restricting architectures to this "macro dataflow" concept would amount to giving up what is possibly a dataflow machine's biggest feature - the ability to context switch efficiently at a low level to cover memory latency.

Given this, one is led to ask the following question: *what mechanisms at the hardware level are* essential *for tolerating latency and synchronization costs?* Based on various studies of parallel machines [2, 7, 12, 22] the following conclusions are drawn:

- In general, on a machine capable of supporting multiple simultaneous threads of computation, executing programs expressed as a total ordering of instructions will incur more latency cost than will executing a logically equivalent partial ordering of the same instructions. In fact, for a class of programming languages which are *non-sequential* [33], expressing programs as a partial ordering is a necessary condition for avoiding deadlock. It is assumed, therefore, that the machine language must be able to express partial ordering.
- In any multiprocessor architecture, certain instructions will take an unbounded amount of time to complete (*e.g.*, those involving communication). Such operations can be either atomic, single phase operations or split, multiphase operations[2]. Multiphase processing will always minimize latency cost over single phase processing because the potential exists for covering processor idle time. Based on the frequency of the occurrence of such long latency operations [2] in all but the most trivial parallel computations, efficient multiphase operation requires specific hardware mechanisms [3, 12]. Multiphase instructions are commonly referred to as *split transactions*.

The remainder of this section describes a new, hybrid architecture along with its instruction set and programming model. The architecture can be viewed as either an evolution of dataflow architectures in the direction of more explicit (*i.e.*, compiler directed) control over instruction execution order, or as an evolution of von Neumann machines in the direction of better hardware support for synchronization and better tolerance of long latency operations. The study of this architecture will focus on the frequency of unavoidable run-time synchronization and, therefore, the applicability of compiler-directed control over instruction scheduling in a general-purpose multiprocessor.

3.1. Scheduling Quanta

The central idea of this new architecture involves some reconsideration of the basic unit of work in both dataflow and von Neumann architectures. The unit of parallel computation in a von Neumann machine is the *task*. Inter-task synchronization is typically expensive when it relies on software-implemented mechanisms. Such cost favors large tasks which synchronize infrequently. Within a task, synchronization of *producer* and *consumer* instructions is entirely implicit in the ordering of instructions. Between tasks, barrier synchronization is done explicitly with guards, semaphores, or some other similar mechanism. Context switching is usually done when necessary at the synchronization points, so an important performance metric is the *run length*, or number of instructions between synchronization operations. During such a run, instructions from the same context can enter the pipe at each pipe beat. This kind of locality can often be exploited at the hardware level; however, increasing the locality may imply a loss of parallelism.

[2]A *multiphase* operation is one which can be divided into parts which separately *initiate* the operation and later *synchronize* prior to using the result.

This is in sharp contrast to the dataflow model where the basic unit of parallel computation is the instruction. Inter-task (*i.e.*, inter-instruction) synchronization is performed implicitly by the hardware; the single instruction "task" is not awakened until its operands are available. Context switching can and does occur at each pipe beat; any instruction *n'* which is enabled as a result of the completion of instruction *n* may not enter the pipeline for a number of cycles equal to the pipeline depth. The intervening cycles must be filled by instructions from another thread of execution, possibly but not necessarily from a related context. Not surprisingly, this model is highly parallel, but the parallelism comes at the expense of some lost locality.

3.1.1. Repartitioning Dataflow Graphs

Consider a graph for a simple code block (Figure 1). Note that there is some potential parallelism (lack of interdependence between instructions) in this graph. For example, instructions **I1** and **I5** do not depend on one another. They depend only on the availability of the values **a**, **b**, and **c**.

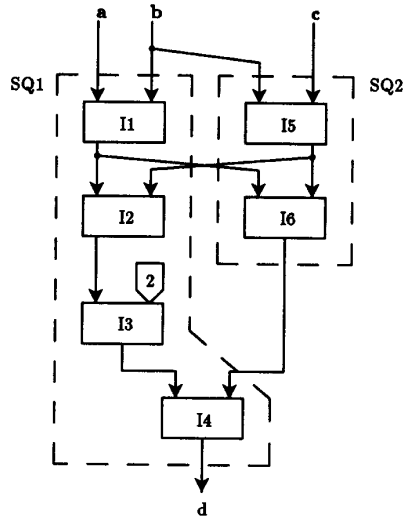

Figure 1: A Sample Dataflow Graph

More pertinent to this discussion are the instructions of the graph which *directly* depend on one another. Instructions **I2** and **I3**, for example, have an interesting dependence. Having executed **I2**, it is known from the graph that instruction **I3** can be executed because it only depends on **I2** and a compile time constant. In some sense, then, the pair (**I2** , **I3**) form a new instruction which has the same input and output characteristics as any other instruction, and which has similar synchronization requirements.

There has been some suggestion within the dataflow community that such aggregation be exploited, if only to improve performance. There is a danger in doing this by altering the machine instruction set, because any statistically beneficial aggregation will have been highly dependent on compilation and code generation techniques used while collecting said statistics. That is, the choice of aggregated instructions may vary as improvements are made to the compiler(s). This suggests that the issues of synchronization should be separated from the issues of opcode semantics.

A slightly more sophisticated view is to permit the compiler to aggregate an *arbitrary* collection of instructions according to any criterion of optimality into a unit of schedulability. Each such unit is called a *scheduling quantum*, or *SQ*. Their size, inter-SQ dependences, and content are determined at compile time. In the Figure, two SQ's are shown, but many other aggregations are possible.

3.1.2. Partitioning Strategies

Although the present discussion is oriented toward machine architectures, it is illuminating to look briefly at methods of partitioning programs expressed as graphs into SQ's, partly to lend credibility to the approach, and partly to better understand the relationship between the static and dynamic scheduling requirements of programs. Starting with a dataflow graph, partitioning may be done in a number of ways. Issues of concern include

- **Maximization of exploitable parallelism:** Poor partitioning can obscure inter-procedural and inter-iteration parallelism. The desire to aggregate instructions does not imply any interest in restricting or limiting parallelism - in fact, those cases where instructions may be grouped into SQ's are quite often places where there is little or no easily exploitable parallelism.
- **Maximization of run length:** Longer SQ's will ultimately lead to longer intervals between context switches (run length). Coupled with proper runtime support for suspension and resumption, this can lead to increased locality. Run lengths which are long compared to the pipeline depth have a positive effect on shortening critical path time. Short run lengths (frequent instruction aborts due to suspension of a frame reference) tend to bubble the pipeline.
- **Minimization of explicit synchronization:** Each arc which crosses SQ boundaries will require dynamic synchronization. Since synchronization operations are pure overhead[3], it is desirable to minimize them.
- **Deadlock avoidance:** Non-sequentiality implies that instruction execution order cannot be made independent of program inputs or, said another way, instruction execution order cannot be determined *a priori*. It is necessary to understand where this dynamic ordering behavior will manifest itself in the generated code. Such dynamic ordering must be viewed as a constraint on partitioning since two instructions whose execution order is dynamically determined cannot be statically scheduled in a single SQ.
- **Maximization of machine utilization:** Given a set of costs for instruction execution, context switching, synchronization, and operand access, partitions can be compared on the basis of how well they "keep the pipeline full." This metric is fairly machine specific and is in that sense less general than those previously described but no less important.

Extant partitioning algorithms [6, 13, 21] can be classified as *depth-first* or *breadth-first*. Depth-first algorithms [6] partition by choosing a path from an input to an output of a graph and making it into an SQ, removing the corresponding instructions from the graph in the process. The algorithm is repeated until no instructions remain unpartitioned. Such partitionings tend to be best at minimizing critical path time and rely heavily on pipeline bypassing since, by definition, instruction *n* depends directly on instruction *n-1*. Breadth-first algorithms [13, 21] tend to aggregate instructions which have similar input dependences but only weak mutual dependences. The *method of dependence sets* as presented in [21] is discussed in the next section.

3.1.3. The Method of Dependence Sets

In order to guarantee liveness of the partitioned graph, it is essential no cycle be introduced which cannot be resolved. Such cycles can be either static or dynamic.

> **Definition 1:** An *unresolvable static cycle* is a directed cycle of SQ's in a partitioned dataflow graph for which no schedule of SQ executions can terminate.

An example of how partitioning can give rise to a static cycle is shown in Figure 2. It can be shown [21] that a graph interpretation rule which includes *suspension* and *resumption*, *i.e.*, partial execution of SQ's, is a sufficient condition for preventing such static cycles from becoming unresolvable. Sarkar and Hennessy [29] avoid the unresolvability issue entirely by imposing a *convexity constraint* on the partitioning - static cycles can therefore never arise.

A much harder problem is that of preventing unresolvable dynamic cycles. Dynamic cycles arise due to the *implicit* arcs between **STORE** and **FETCH** instructions which refer to identical elements. Such arcs are implicit because they are generally input-dependent.

[3]Coming from a von Neumann uniprocessor mind set where explicit synchronization is virtually unheard of except in situations which require multitasking, it is natural to view synchronization in this way. Coming from the dataflow world where synchronization is unavoidable in every instruction execution, and where there is no opportunity to "optimize it out," it is also reasonable to view explicit synchronization instructions as overhead. In a later section, these perspectives are reconciled with the view that explicit synchronization instructions are both necessary and, in some sense, beneficial.

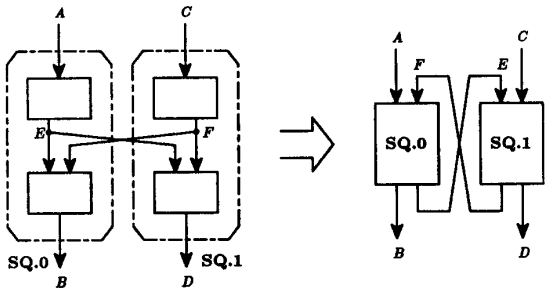

Figure 2: Partitioning which Leads to a Static Cycle

Definition 2: An *unresolvable dynamic cycle* is a directed cycle of SQ's in a partitioned dataflow graph, augmented with all possible input-specific dynamic arcs, for which no schedule of of SQ executions can terminate.

It is necessary to constrain partitioning such that unresolvable dynamic cycles provably cannot arise for any possible set of program inputs. An example will make this clearer. Consider the following Id program fragment:

```
{   a = vector (0,2);
 a[0] = 0;
 a[1] = a[i] + 1;
 a[2] = a[j] - 2;

 in a[1] - a[2]}
```

and its associated graph[4,5] in Figure 3. Such a graph *would* terminate under a dataflow instruction execution rule. However, without exercising some care, partitioning this graph into SQ's can lead to deadlock. Putting all of these instructions into a single partition won't work, nor will a partitioning such as that shown in Figure 4. Such partitionings result in code which can never terminate, despite the adherence to static dependences in deriving the individual SQ schedules.

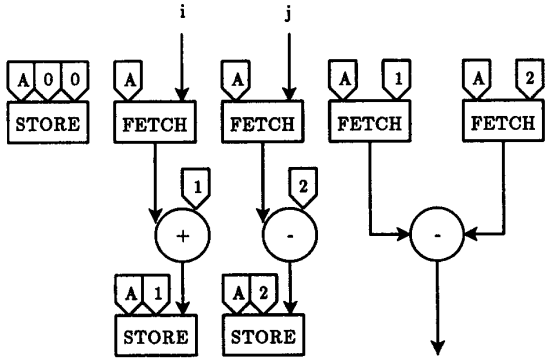

Figure 3: Program Graph Fragment

The problem, of course, is that the actual instruction execution order in the dataflow case depends on the indices used in the structure operations, where no such dependence is allowed in the partitioned case. Figure 5 shows two instruction execution orderings which must be possible in any correctly compiled version of this program. These orderings demonstrate the dynamic dependences between **STORE**s and **FETCH**es. If these dependences were fixed, and if it were possible to determine them at compile time, SQ partitioning to avoid deadlock would be straightforward. Since this is not the case, the problem is one of developing a safe partitioning strategy which is insensitive to the arrangement of dynamic arcs. One approach is to make each partition exactly one instruction long, *i.e.*, the dataflow method. This, of course, is at odds with the desire to exploit static scheduling.

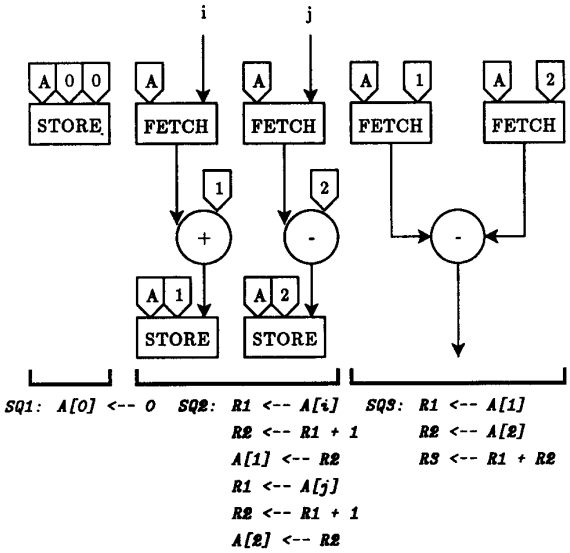

Figure 4: Partitioning which Leads to Deadlock

Another method is to give names to *sets of dependences*. The following definitions are in order:

Definition 3: A **FETCH**-*like output* of an instruction is one which is associated with a dynamic dependence. An instruction itself is **FETCH**-like if at least one of its outputs is **FETCH**-like, implying that the instruction initiates a split transaction (long latency) operation.

Definition 4: The *input dependence set* for an instruction in a well-connected graph [32] is the union of the output dependence

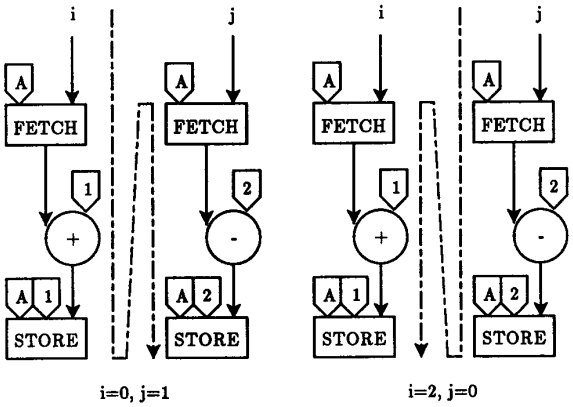

Figure 5: Input Dependent Execution Order

[4]The descriptor for vector *A* is depicted as a constant to simplify the drawings. This is done without loss of generality.

[5]In the sequel, it is assumed that global storage is read by multiphase operations, and that the memory controller implements I-Structure-like synchronization [18]. In that sense, **FETCH** and **STORE** behave as **I-FETCH** and **I-STORE**.

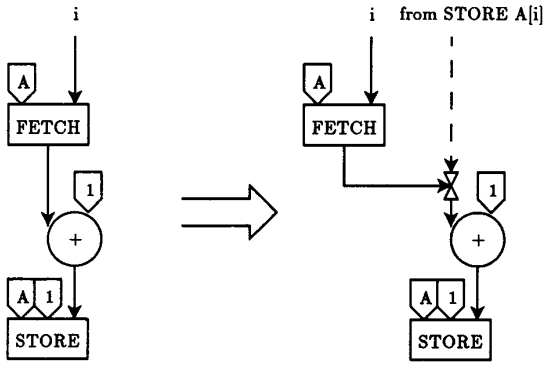

Figure 6: Gating Effect of **FETCH**-like Instructions

sets of all instructions from which it receives input. The input dependence set of the root instruction is defined as { α }.

Definition 5: The *output dependence set* for a given output of an instruction is either the instruction's input dependence set if the output is not **FETCH**-like, or the union of the instruction's input dependence set with a singleton set which uniquely names the given output if it is.

Note that it is a **FETCH**-like instruction's *output*, and not the instruction itself, with which is associated a change of dependence set. The intuition is that **FETCH**-like instructions themselves can never suspend while waiting for their output. Rather, the instructions which receive the **FETCH**-like instruction's output are the ones which will suspend. Figure 6 makes this clearer. A **FETCH**-like instruction can be viewed as gating the value of a **STORE**-like operation; the dynamic arc terminates on the virtual gate. It has the effect of suspending the "+" instruction until both the **STORE** and the **FETCH** have completed.

Applying the definitions to the graph in Figure 3 and using β, γ, and δ (in that order) for unique names results in the following assignments of input dependence sets to instructions (assume that vector *A* and the indices *i,j* are derived from the root with dependence set {α}):

Instruction	Input Dependence set
STORE(0)	{ α }
FETCH(i)	{ α }
FETCH(j)	{ α }
FETCH(1)	{ α }
FETCH(2)	{ α }
+1	{ α β }
STORE(1)	{ α β }
-2	{ α γ }
STORE(2)	{ α γ }
-	{ α δ }

The assignment of instructions to SQ's is now straightforward: an SQ is associated with each unique dependence set. Instructions are assigned to the SQ corresponding to their input dependence set in an order corresponding to their topological ordering in the unpartitioned graph. Since each distinct combination of dynamic arcs denotes a single SQ, dynamic scheduling can change to match the dynamic dependences. The correctly partitioned graph is shown in Figure 7. The determination of synchronization points is also straightforward: each dependence (arc) which crosses SQ boundaries must be explicitly synchronized by the consumer, or *sink*, SQ. Consumers in the same SQ as the instruction producing a value need not perform synchronization - it is implicit in the static scheduling of instructions within the SQ.

In [21], the deadlock-avoidance property of this algorithm is proved. Moreover, if procedure calls are compiled as **FETCH**-like operations, the method of dependence sets naturally allows inter-procedural parallelism. A simple extension to *k-bounded loops* [10] also allows inter-iteration parallelism. Run length, explicit synchronization, and machine utilization properties of this algorithm are studied in a later section.

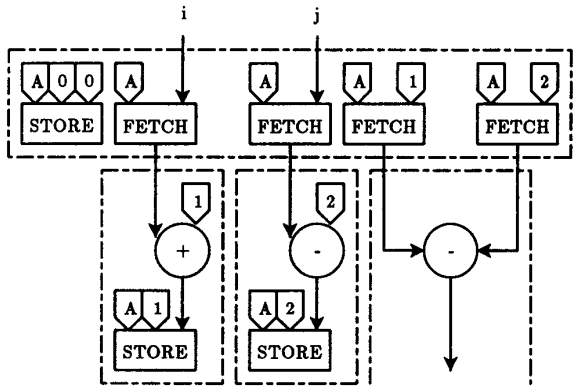

Figure 7: Properly Partitioned Graph

3.2. Parallel Machine Language

Let's review the essential conclusions so far. Latency and synchronization have been shown to be fundamental issues in the development of scalable, general purpose multiprocessors, and the issues seem related in fairly incestuous ways. Basic changes to traditional architecture are necessary for dealing with them. One such change is that the execution time for any given instruction must be independent of latency (giving rise to split transactions). A second change is that synchronization mandates hardware support: each synchronization event requires a unique name. The name space is necessarily large, and name management must be efficient. To this end, a compiler should generate code which calls for synchronization when and only when it is necessary. A natural approach is to extend instruction sets to express the concepts of *both* implicit and explicit synchronization. Such an instruction set, which captures the notions of bounded instruction execution time, a large synchronization name space, and means of trading off between explicit and implicit synchronization is called a *parallel machine language* (PML).

It has been shown that adding partitioning to a dataflow graph is tractable. Doing so moves dataflow graphs into the realm of parallel machine languages. The question remains of how to organize a machine to efficiently implement a PML. One method would be to start with the explicitly-synchronized dataflow paradigm and to augment it with facilities for compiler-directed instruction scheduling (*e.g., Monsoon* [24]). Another approach, described below, would be to start with the implicitly-synchronized von Neumann paradigm and to augment it with facilities for dynamic instruction scheduling.

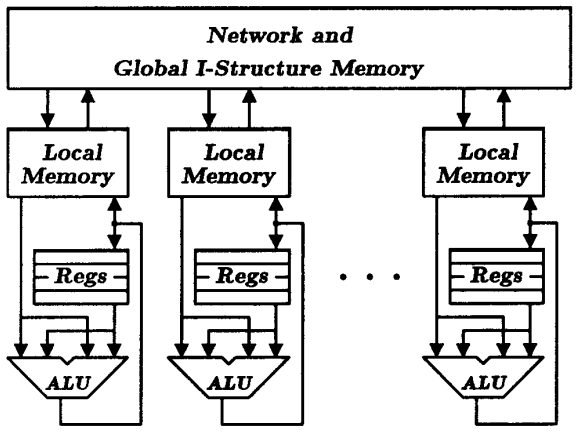

Figure 8: The Hybrid Machine

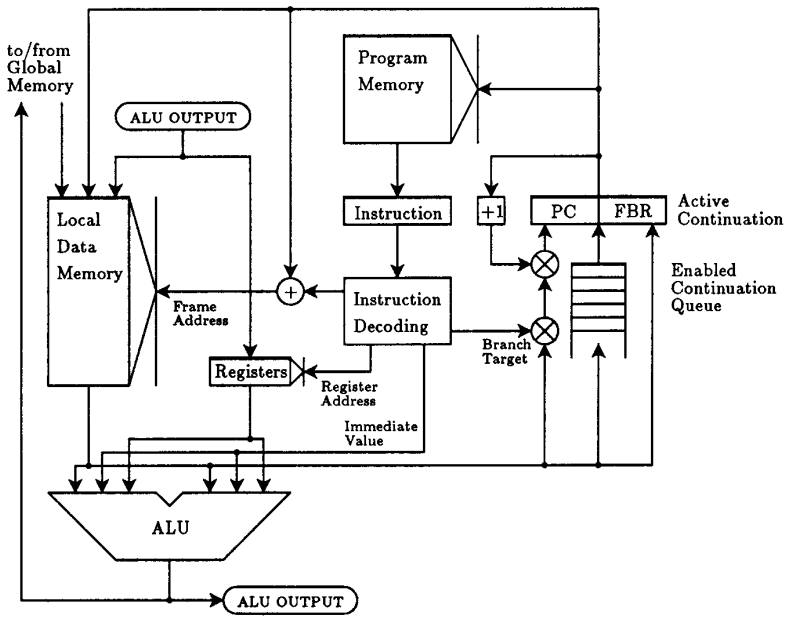

Figure 9: The Hybrid Processor

3.3. The Hybrid Multiprocessor

The architecture is modeled as an array of n identical processors, connected through a suitable switching network to a globally addressed I-Structure memory[6]. Each processor is made up of a pipelined datapath, a collection of high speed registers, and a local memory. Instructions are provided which allow movement of data between local and global memories, and between registers and local memories. All inter-processor communications can be thought of as going through *global memory*[7]. The *local memory* is both physically and logically local to a processor. For each invocation of each code block, a *frame* is allocated in the local memory of *exactly one processor* to hold local variables[8]. References to frame slots can be synchronizing or non-synchronizing.

3.3.1. Processor Hardware

The hardware which makes up a hybrid processor is strongly similar to that of a von Neumann machine, but with a few important differences (Figure 9). The datapath (ALU, etc.) and registers are conventional. The hardware datatypes are integers, floating point numbers, memory addresses, and the like. The most significant new datatype is the *continuation* which is a tuple of a program counter (PC) and a frame base register (FBR). Logical continuation states are depicted in Figure 10 and are encoded by the *location* of the continuation. *Enabled* continuations reside in the Enabled Continuation Queue. The *running* continuation resides in the Active Continuation Register. *Suspended* continuations reside in frame slots. *Uninitiated* and *terminated* continuations are not explicitly represented.

The PC of the running continuation denotes the instruction to be dispatched next. Instructions may make operand references to the registers or to slots in the local data memory. The local memory's behavior is similar to I-Structure storage in that each slot has several presence bits associated with it. A *non-synchronizing* reference to a slot behaves as a normal memory read operation. *Synchronizing* references invoke suspension of the running continuation if the slot being read is marked as EMPTY. Synchronizing reads of a WRITTEN slot behave just the same as nonsynchronizing reads.

The key hardware extension lies in the efficient state-transition management for continuations. Because continuations are word-sized objects, they can be easily fabricated when an SQ is invoked. When the running continuation encounters a blockage (via a synchronizing reference to an empty frame slot), the hardware simply stores the running continuation into the slot, marking it as now containing a continuation. An enabled continuation can then be selected from the queue. Upon satisfaction of the blockage (some other continuation writes into the slot), the suspended continuation is extracted and re-queued. A continuation may be suspended a number of times between initiation and termination. This behavior is reminiscent of the traditional *task* in a demand paged system which, upon encountering a missing memory page, becomes suspended until the page is made available.

Registers may be used freely *between* instructions which make suspensive references to the frame, but since no register saving takes place when a suspension occurs, register contents cannot be considered valid across potentially suspensive instructions. The cost of a register reference (maximum two per pipeline beat) is less than a local memory reference (maximum one per pipeline beat) which is, in turn, less than a global I-Structure reference (split transaction, possibly with an explicit address arithmetic instruction). Local memory is referenced relative to the FBR in the current continuation. I-Structure address space is global and is shared.

One can imagine a number of implementations of suspension which incur costs ranging from nothing to many tens or hundreds of instructions. In order to keep the implementation from distorting the kinds of code a com-

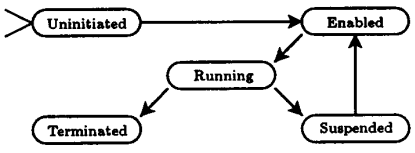

Figure 10: Continuation States

[6]The behavior of an I-Structure Storage unit is discussed extensively in [18, 19] and will not be repeated here. It is sufficient to note that all I-Structure references are split transactions and, therefore, never block the processor pipeline. Moreover, one can view the functions of an I-Structure storage as a superset of the functions of a traditional store, *i.e.*, at the hardware level, imperative reads and writes can be performed as easily as I-Structure reads and writes.

[7]This restriction can be relaxed somewhat. It is possible to pass around local memory pointers if they are used for *remote store-in*, *i.e.*, direct forwarding of values from one processor to another. This can be used to great advantage in procedure linkage. The ability to do this is a function of the lifetimes of local memory addresses.

[8]Frame sizes are determined at compile time.

piler might generate, it is imperative that the cost of performing a context switch must be exceedingly low - on the order of a single pipe beat. To make this practical and general, context state must be easily saved in a single cycle. Moreover, it must be nearly trivial to create or destroy such context states as SQ's are initiated and terminated. Lastly, the number of such extant contexts cannot be bounded by some small hardware resource. For these reasons, context state *at suspension points* is represented solely by the continuation and its associated frame. By making the continuations no larger than a frame slot, saving and restoring becomes nearly trivial (a single memory reference) compared to schemes wherein registers are also saved[9].

The simple minded *dispatch instructions sequentially until blocked* paradigm works well, and has a very positive effect on locality. Other approaches are possible, however. The high-level goal is to dispatch instructions so as to keep the pipeline full of useful work. *Non-useful* work includes execution of **NO-OPs** (*i.e.*, pipeline bubbles) and instructions which suspend. Each processor maintains a queue of enabled continuations. One may view each continuation as logically contributing one instruction (the one pointed to by its PC) to the set of *enabled instructions*. At each time step, the processor's instruction dispatcher can freely choose one instruction from among this set. Optimal dispatching of instructions is impossible without foreknowledge of which instructions in the set *will* suspend. However, simple decoding of instructions allows the dispatcher to know if the instruction *cannot* suspend (*e.g.*, those which only reference registers or which make nonsuspensive references to frame memory) or if it *might possibly* suspend. A good strategy, then, is to divide the set of enabled instructions along these lines and to dispatch first from the subset of those which cannot suspend, delaying as long as possible execution of instructions which might suspend (thereby reducing the probability of suspension in many cases)[10].

3.3.2. Instruction Set

The instruction set is simple and regular in structure, with addressing modes and instruction functions being largely orthogonal. Instructions are readily implemented in a single cycle. The basic addressing modes are IMMEDIATE, REGISTER, FRAME NONSUSPENSIVE, and FRAME SUSPENSIVE. All unary and binary ops for arithmetic and logicals can take immediate, register, or frame slot operands and produce register or frame slot results. The **MOVE** opcode encodes all intra-process data movement. It is capable of moving an immediate, register, or frame slot to a register or frame slot. The **MOVE-REMOTE** opcode initiates movement of a value to a remote (non-local) frame slot. This instruction is used for procedure linkage, and is the only way one procedure can store into another's frame. The **LOAD-FRAME-INDEX** opcode and its variants initiate an indexed read from I-structure memory to the frame. **STORE** and its variants initiate a store to I-Structure memory. The **TEST** and **RESET** opcodes are provided for explicit synchronization and frame slot re-use, respectively. **RESET**s occur within the body of a multiple SQ loop to re-enable synchronization prior to iteration. The **BRANCH** and **BRANCH-FALSE** opcodes do the obvious things, causing the PC in the continuation to be replaced (conditionally in **BRANCH-FALSE**). The **CONTINUE** opcode causes a fork by creating and queueing a new continuation. The corresponding join operation is implemented implicitly through frame slots. A number of other instructions unique to TTDA-style program graphs have been implemented. The simplest are the **CLOSURE** ops which construct and manipulate closures as word-sized objects (rather than memory-bound structures). These are arguably easy to implement in single machine cycles. The remainder are instructions which form manager message packets to allocate and deallocate various resources.

In translating program graphs to machine language, arcs are mapped to frame slots. Slots may be re-used within a code block but it is the responsibility of the compiler to guarantee that all reading of a slot is complete before it is re-written. Synchronizing operand reads are used to implement inter-SQ communication including the synchronization associated with **FETCH**-like instructions. Note that it is the *reader* of the slot which chooses to synchronize or not; it is not a property of the slot itself. Each slot may have multiple readers, some synchronizing and some non-synchronizing.

[9]Intuition leads one to believe that such a scheme results in degraded performance in the form of additional memory references. As discussed in [21], this is an oversimplification because frame storage can be cached easily *without* a coherence problem. See Section 5. Experience generating code from dataflow graphs shows this strategy to work. More sophisticated code generation techniques can make even better use of such non-saved registers.

[10]This scheme is not ideal, however. Consider the case of having postponed execution of a long-latency but potentially suspensive **FETCH**. If executing the instruction does not, in fact, cause suspension, delaying it will give up cycles which could otherwise have been used to mask the latency.

4. Characterization of the Hybrid Architecture

Using the method of dependence sets, a new code generator for the Id compiler [32] has been constructed to generate PML code. A heavily instrumented simulation model of the hybrid machine has been built which allows study of the following, using compiled Id programs:

- the effects of architectural assumptions (*e.g.*, code partitioning) on program behavior. This *idealized* model imposes only very weak physical constraints, *i.e.*, the number of processors is assumed unbounded, all instructions are assumed to execute in unit time, and communication latency is assumed zero. Continuations execute when and only when the synchronization constraints are satisfied.
- the effects of physical constraints on otherwise idealized program behavior. To the idealized model are added *realistic* limits such as nonzero communication latency, finite processors, and realizable scheduling.

This section presents experimental results from the first set of simulation studies of the hybrid architecture along with a comparison to similar results from studies of the TTDA. One very interesting metric is the dynamic instruction count. Because synchronization is not entirely implicit in the hybrid model, it is reasonable to expect that compiling for and executing a program on the hybrid machine should result in more instructions executed than in the case of the same program compiled for and executed on the TTDA. By experimental results, it is shown that this is not necessarily so.

4.1. Power of a Hybrid Instruction

In the TTDA model, each invoked instruction constitutes its own continuation. Each instruction can *synchronize* or *join* two threads of computation and can *fork* two new threads in addition to performing some computation. In the hybrid model, each instruction is typically one of many in a single SQ. As in the TTDA, each instruction can join two threads of computation by making two suspensive references in the frame. However, each instruction can *continue* only a single thread of computation - forks are done explicitly by the **CONTINUE** instruction and therefore represent additional overhead. Assuming all other things equal, *e.g.*, the opcode set, hybrid instructions are strictly less powerful than TTDA instructions.

An interesting question, as alluded to earlier, is whether the full generality of TTDA instructions is used frequently or infrequently. By using the identical program graphs in generating code for both the TTDA and the hybrid machine, it has been possible to study this question in some detail.

Figure 11 shows some dynamic instruction counts for various Id benchmark programs. These numbers were derived from simulation programs written specifically for the TTDA and the hybrid architecture, and count only program, not system or manager, instructions[11,12]. The counts do not favor either architecture but rather show that, for a variety of program types, instruction counts are comparable to first order. If hybrid instructions are less powerful, how can this be?

One part of the answer lies in the reduced number of overhead operators in the hybrid code resulting from fewer independent threads. In the TTDA, termination detection is done via trees of **IDENTITY** instructions. The leaves of these trees are the instructions which otherwise produce no tokens, *e.g.*, **STORE** operations. In the hybrid model, it is only necessary to test for termination of the SQ in which such instructions reside. Hence, *n* **STORE**s in one SQ imply only *one* explicit synchronization operation instead of a binary tree of *n-1* **IDENTITY** instructions.

Another part of the answer is elimination of the need to perform explicit fan-out of **FETCH**ed values; the associated frame slots can simply be re-read. In the TTDA, however, **FETCH** operations can have only a single destination instruction. Multiple destinations imply the need for an **IDENTITY** instruction as the destination for the **FETCH**.

Because, in general, it would take two hybrid instructions to mimic the function of a single TTDA instruction (join two threads, compute, continue two threads), one might expect the hybrid count to be roughly double the TTDA count, yet the counts are nearly the same. The effects described above, when combined, represent roughly 30% of this discrepancy. It is

[11]In neither case is the type of simulation model (idealized or not) relevant for instruction counts. Instruction counts do not vary across these models.

[12]Programs in the Procedure Calling Overhead section are essentially trivial routines enclosed in the standard procedure call framework. List programs (implemented with CONS, CAR, and CDR) are memory intensive. Vector programs (implemented with ARRAYs) include both a significant amount of memory traffic and ALU operations.

	Problem Size	Dynamic Instruction Counts TTDA	Hybrid
Procedure calling overhead:			
CONS		14	14
CAR		9	4
CDR		9	4
Fibonacci	10th	3,708	3,265
Lists:			
Reverse	9	497	585
Compute Length	9	506	439
Multiplicative Reduction	9	1,014	909
Vectors and Matrices:			
Trivial Sum	500	5,023	3,513
Sum of Squares	500	19,072	23,058
Linear Recurrence	500	18,047	15,275
Pointwise Product	20	290	445
Inner Product	20	285	359
Matrix Multiplication	10x10	20,716	26,255

Figure 11: TTDA and Hybrid Instruction Count Comparison

likely that the remainder is attributable to the fact that it does *not* in general take two hybrid instructions to displace a single TTDA instruction. There are many instances of TTDA instructions in typical programs where the full generality and power of the instruction is not being used in the sense that the hybrid partitioning strategy chooses to eliminate it rather than mimic it. This conclusion is borne out by comparing the dynamic instruction mixes (instruction types) executed by the two architectures when running the same program. In the hybrid model, parallelism is retained in the machine code only when dictated by dynamic arc constraints. According to this view, the remainder of the parallelism in TTDA code is superfluous. In the next section, we examine the effect of this reduced parallelism in terms of the hybrid machine's ability to tolerate latency.

4.2. Parallelism and Latency

No amount of "optimization" by packing instructions into larger chunks is worth much if it negates the architecture's ability to synchronize efficiently or to tolerate latency. It is reasonably clear that the hybrid architecture provides the necessary synchronization support at a basic level for the purposes of program decomposition. But what about the hybrid machine's tolerance of long latency operations?

Consider as an example the recursive Fibonacci program of the previous section. Running this program under the idealized model yields a *parallelism profile* (Figure 12) showing the number of concurrently executable continuations as a function of time. This is more a characteristic of the program than of the machine - it shows the available parallelism subject to the chosen partitioning. In such an experiment, the cost of communicating across processor boundaries is the same as the cost of communicating within a processor: results of executing an instruction are available in the following cycle. The parallelism profile is a metric of the method of *logically decomposing* a program.

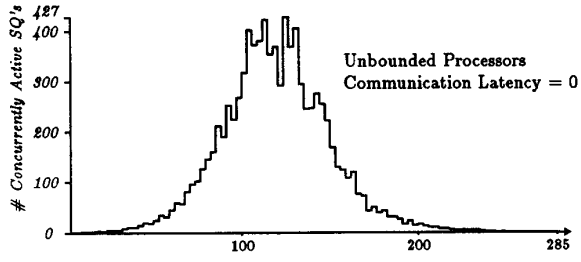

Figure 12: Idealized Parallelism Profile for FIBONACCI(14)

The effect of *physical partitioning*, or distributing a program can be estimated by assigning a cost to each inter-processor communication in terms of a delay between production and use of results (*i.e.*, a latency), by setting a limit on the number of processors (each processor can execute at most one instruction at any time t), and by allocating procedure invocations to individual processors by some rule. Note that this is more restrictive than a simple "finite processor" limit which simply forces k instructions on p processors to take $\lceil k/p \rceil$ time. The top profile of Figure 13 shows the effect of these assumptions with the latency still zero, using a random policy for assigning invocations to processors. By increasing the latency and measuring the increase in execution time, it is possible to quantify the architecture's ability to use excess parallelism to cover the latency. In the lower half of Figure 13, the inter-processor latency has been increased from 0 to 10 pipe steps, yet the increase in critical path time is only 13.2%.

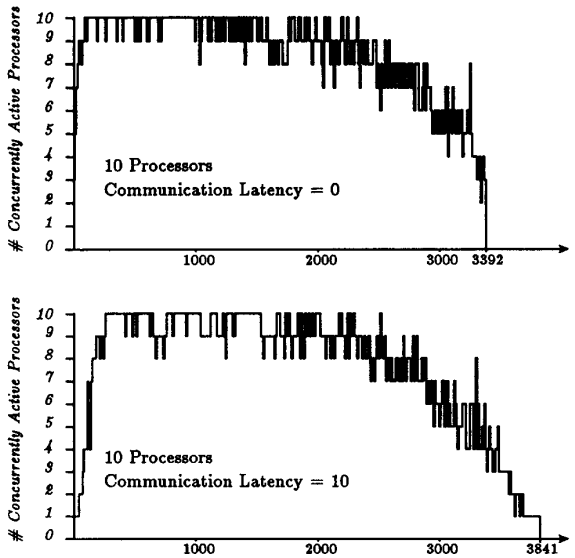

Figure 13: Comparison of Execution Time for Various Latencies

4.3. Dynamic Run Length

An important issue in partitioning as addressed earlier is the *dynamic run length*, or number of instructions successfully executed between suspensions. Figure 14 shows the instruction counts (from Figure 11 - this is the number of instructions which ran to completion), the number of aborted instructions (aborted instructions are not included in instruction count), the ratio of aborts to instructions expressed as a percentage, and the mean dynamic run length. With the exception of CAR, CDR, and CONS, aborted instructions are 8 to 16% of instructions successfully executed, and run lengths of 2.4 to 6.5 are typical.

5. Conclusion

A new architecture has been described which supports a parallel machine language, capturing the notions of split transaction operations, a large synchronization namespace, and means for trading between implicit and explicit synchronization. It has been demonstrated that the architecture is capable of effectively exploiting parallelism in partitioned dataflow graphs, of trading program parallelism for latency cost, and of enabling a compiler to control locality to first order.

From the preliminary results presented above, it appears that little of the full power of the TTDA's synchronization mechanism is actually used in typical programs. This leads to the observation that *explicit* synchronization instructions, used when necessary, may in some sense be cheaper than paying the full cost of synchronization at each instruction. This is, perhaps, the equivalent of the RISC argument applied to multiprocessing.

	Problem Size	Instr Count	Aborts	%	Run Length
Procedure calling overhead:					
CONS		14	4	28.57	1.3
CAR		4	1	25.00	1.0
CDR		4	1	25.00	1.0
Fibonacci	10th	3,265	265	8.12	3.4
Lists:					
Reverse	9	585	67	11.45	3.3
Compute Length	9	439	68	15.49	2.4
Multiplicative Reduction	9	909	121	13.31	2.7
Vectors and Matrices:					
Trivial Sum	500	3,513	2	0.06	585
Sum of Squares	500	23,058	3,004	13.03	3.8
Linear Recurrence	500	15,275	2	0.01	2,182
Pointwise Product	20	445	42	9.44	6.5
Inner Product	20	359	41	11.42	5.4
Matrix Multiplication	10x10	26,255	2,445	9.31	5.0

Figure 14: Hybrid Instruction Aborts and Run Length

As yet unanswered is the question of the effectiveness of the hybrid architecture, or architectures like it, for other parallel programming models (*e.g.*, Halstead's MultiLisp [16]). It is conjectured that simple extensions to the frame slot synchronization mechanism can effectively support demand-driven, or EVAL-when-touched scheduling. Of considerable practical interest is the possibility of targeting FORTRAN compilers to the hybrid paradigm.

Work from the present project reported elsewhere [21] includes integration of a local memory cache into the processor's design. Since local memory can only be read by the local processor, there is no issue of global coherence. Such a cache can lift the restriction of a single frame access per pipe beat. Many improvements to code generation have also been made including the *k-bounded* loop schema [10], while others have only been briefly considered, *e.g.*, peephole optimization and other criteria for SQ partitioning.

It is encouraging to see how the research efforts of various groups are converging on hybrid models such as the one presented here. Space constraints prohibit review of these projects; the interested reader is directed to the the work of Buehrer and Ekanadham [8], Halstead and Fujita [17], Papadopoulos [24] Sarkar and Hennessy [29], and Bic [6].

6. Acknowledgments

The author wishes to thank Andrew Chien, Ken Traub, Steve Heller, and the anonymous referees for their criticisms and suggestions for improvement of this manuscript. Special thanks go to Arvind for encouraging me to explore ideas of a hybrid architecture. The author also wishes to thank Lee Howe, Ev Shimp, Bobby Dunbar, Bill Hoke, Lucie Fjeldstad, Frank Moore, Dick Case, Hum Cordero, Bob Corrigan, Carl Conti, John Karkash and his other sponsors at IBM who made this work possible.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-83-K-0125 and N00014-84-K-0099. The author is employed by the International Business Machines Corporation.

References

1. Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo. "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling". *IBM Journal 11* (January 1967), 8-24.

2. Arvind, S. A. Brobst, and G. K. Maa. Evaluation of the MIT Tagged-Token Dataflow Architecture. Computation Structures Group Memo 281, MIT, Laboratory for Computer Science, Cambridge, MA 02139, December, 1987.

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

4. Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, June, 1987.

5. Arvind, G. K. Maa, and D. E. Culler. Parallelism in Dataflow Programs. Computation Structures Group Memo 279, MIT, Laboratory for Computer Science, Cambridge, MA 02139, December, 1987.

6. Bic, L. A Process-Oriented Model for Efficient Execution of Dataflow Programs. Proc. of the 7th International Conference on Distributed Computing, Berlin, West Germany, September, 1987.

7. Bouknight, W. J., S. A. Denenberg, D. E. Mcintyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. "The ILLIAC IV System". *Proceedings of the IEEE 60*, 4 (April 1972).

8. Buehrer, R. and K. Ekanadham. "Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution". *IEEE Transactions on Computers C-36*, 12 (December 1987), 1515-1522.

9. Cox, G. W., W. M. Corwin, K. K. Lai, and F. J. Pollack. "Interprocess Communication and Processor Dispatching on the Intel 432". *ACM Transactions on Computer Systems 1*, 1 (February 1983), 45-66.

10. Culler, D. E. Resource Management for the Tagged-Token Dataflow Architecture - S.M. Thesis. Technical Report 332, MIT, Laboratory for Computer Science, Cambridge, MA 02139, January, 1985.

11. Culler, D. E. and Arvind. Resource Requirements of Dataflow Programs. Proceedings of the 15th Annual International Symposium on Computer Architecture, IEEE Computer Society, Honolulu, Hawaii, June, 1988.

12. Deminet, J. "Experience with Multiprocessor Algorithms". *IEEE Transactions on Computers C-31*, 4 (April 1982), 278-288.

13. Ekanadham, K. Multi-Tasking on a Dataflow-like Architecture. Tech. Rept. RC 12307, IBM T. J. Watson Research Laboratory, Yorktown Heights, NY, November, 1986.

14. Ekanadham, K., Arvind, and D. E. Culler. The Price of Parallelism. Computation Structures Group Memo 278, MIT, Laboratory for Computer Science, Cambridge, MA 02139, November, 1987.

15. Fisher, J. A. Very Long Instruction Word Architectures and the ELI-512. Proc. of the 10th, International Symposium on Computer Architecture, IEEE Computer Society, June, 1983.

16. Halstead, R. H., Jr. "MultiLisp: A Language for Concurrent Symbolic Computation". *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), 501-538.

17. Halstead, R. H., Jr., and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. Proceedings of the 15th Annual International Symposium on Computer Architecture, IEEE Computer Society, Honolulu, Hawaii, June, 1988.

18. Heller, S. K. An I-Structure Memory Controller (ISMC). MIT Department of Electrical Engineering and Computer Science, Cambridge, MA 02139, JUNE, 1983.

19. Heller, S. K. and Arvind. Design of a Memory Controller for the MIT Tagged-Token Dataflow Machine. Computation Structures Group Memo 230, MIT, Laboratory for Computer Science, Cambridge, MA 02139, October, 1983. Proceedings of IEEE ICCD '83 Port Chester, NY.

20. Hennessey, J. L. "VLSI Processor Architecture". *IEEE Transactions on Computers C-33*, 12 (December 1984), 1221-1246.

21. Iannucci, R. A. *A Dataflow / von Neuamnn Hybrid Architecture*. Ph.D. Th., MIT Department of Electrical Engineering and Computer Science, May 1988.

22. Jordan, H. F. Performance Measurement on HEP - A Pipelined MIMD Computer. Proceedings of the 10th Annual International Symposium On Computer Architecture, Stockholm, Sweden, June, 1983, pp. 207-212.

23. Kuck, D., E. Davidson, D. Lawrie, and A. Sameh. "Parallel Supercomputing Today and the Cedar Approach". *Science Magazine 231* (February 1986), 967-974.

24. Papadopoulos, G. M. *Implementation of a General Purpose Dataflow Multiprocessor*. Ph.D. Th., MIT Department of Electrical Engineering and Computer Science, May 1988.

25. Patterson, D. A. "Reduced Instruction Set Computers". *Communications of the ACM 28*, 1 (January 1985), 8-21.

26. Radin, G. The 801 Minicomputer. Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, March, 1982. Same as Computer Architecture News 10,2 and SIGPLAN Notices 17,4.

27. Rau, B., D. Glaeser, and E. Greenwalt. Architectural Support for the Efficient Generation of Code for Horizontal Architectures. Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, March, 1982. Same as Computer Architecture News 10,2 and SIGPLAN Notices 17,4.

28. Russell, R. M. "The CRAY-1 Computer System". *Communications of the ACM 21*, 1 (January 1978), 63-72.

29. Sarkar, V., and J. Hennessy. Partitioning Parallel Programs for Macro Dataflow. Proceedings of the ACM Conference on Lisp and Functional Programming, ACM, August, 1986, pp. 202-211.

30. Smith, B. J. A Pipelined, Shared Resource MIMD Computer. Proceedings of the 1978 International Conference on Parallel Processing, 1978, pp. 6-8.

31. Tomasulo, R. M. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". *IBM Journal 11* (January 1967), 25-33.

32. Traub, K. R. A Compiler for the MIT Tagged-Token Dataflow Architecture - S.M. Thesis. Technical Report 370, MIT, Laboratory for Computer Science, Cambridge, MA 02139, August, 1986.

33. Traub, K. R. *Sequential Implementation of Lenient Programming Languages*. Ph.D. Th., MIT Department of Electrical Engineering and Computer Science, May 1988.