Two Fundamental Limits on Dataflow Multiprocessing[*]

David E. Culler
Klaus Erik Schauser
Thorsten von Eicken

Computer Science Division
University of California, Berkeley

Abstract: This paper examines the argument for dataflow architectures in "Two Fundamental Issues in Multiprocessing[5]." We observe two key problems. First, the justification of extensive multithreading is based on an overly simplistic view of the storage hierarchy. Second, the local greedy scheduling policy embodied in dataflow is inadequate in many circumstances. A more realistic model of the storage hierarchy imposes significant constraints on the scheduling of computation and requires a degree of parsimony in the scheduling policy. In particular, it is important to establish a scheduling hierarchy that reflects the underlying storage hierarchy. However, even with this improvement, simple local scheduling policies are unlikely to be adequate.

Keywords: dataflow, multiprocessing, multithreading, latency tolerance, storage hierarchy, scheduling hierarchy.

1 Introduction

The advantages of dataflow architectures were argued persuasively in a seminal 1983 paper by Arvind and Iannucci[4] and in a 1987 revision entitled "Two Fundamental Issues in Multiprocessing"[5]. However, reality has proved less favorable to this approach than their arguments would suggest. This motivates us to examine the line of reasoning that has driven dataflow architectures and fine-grain multithreading to understand where the argument went awry. We observe two key problems. First, the justification of extensive multithreading is based on an overly simplistic view of the storage hierarchy. Second, the local greedy scheduling policy embodied in dataflow, "execute whenever data is available" is inadequate in many circumstances. A more realistic model of the storage hierarchy imposes significant constraints on the scheduling of computation and requires a degree of parsimony in the scheduling policy. In particular, it is important to establish a scheduling hierarchy that reflects the underlying storage hierarchy. However, even with this improvement, simple local scheduling policies are unlikely to be adequate.

In this paper we review the "Two Fundamental Issues" argument and recast it into a simple analytical model of multithreading. This provides a precise statement of the relationship between synchronization cost, number of threads, and latency tolerance. Section 3 shows that the storage hierarchy limits the number of threads among which switching can be cheap and therefore limits the latency tolerance of fast processors. Section 4 shows that latency tolerance and processor efficiency can be improved with a storage directed scheduling policy, but Section 5 shows that this improved scheduling is not fully adequate. This leaves

[*]A version of this report appears in the Proceedings of the IFIP Working Group 10.3 (Concurrent Systems) Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Jan. 1993, Elsevier Science Publishers.

us with an open question of how to incorporate high-level scheduling in the implementation of implicitly parallel languages.

2 Two Fundamental Issues Revisited

In this section, we provide a synopsis of the “Two Fundamental Issues” (TFI) argument and then formalize the argument in a simple algebraic model. This provides a clear statement of how latency, synchronization cost, and processor utilization are related. Formulation of the model also makes clear the assumptions underlying the general argument. In later sections, we will examine the validity of these assumptions.

2.1 Synopsis

The main line of the argument put forward by Arvind and Iannucci is as follows.

Assume a system view of a collection of processors and memory modules connected via a communication network. Each remote reference incurs some latency in traversing the network. This latency will severely reduce the processor utilization if the processor idles throughout remote references. Hence, the processor must continue executing instructions while a remote reference is outstanding.

Overlapping computation with a remote reference requires that the two must synchronize upon completion of the reference. Thus, the cost of synchronization and the ability to tolerate latency are closely linked. If the latency is long compared to the time between remote references, the processor must issue additional remote references before the first completes. “The processor must view the memory/communication system as a logical pipeline[5].” Multiple remote references may be outstanding at once. These requests are likely to complete out of order, since some will follow shorter paths in the network than others or encounter less contention. Therefore, a fairly general synchronization mechanism is required to coordinate each response with the appropriate request.

Since the latency incurred on each request is large, the distance from the point in the program where the request is issued to the point where the response is received must be large. This distance is difficult to achieve within a single task, but can easily be achieved by interleaving requests from completely different tasks within the program. Each overlapped, inter-task remote reference requires a scheduling event to select the next ready task and a synchronization event to enable the issuing task when the response arrives. Thus, the requirements on the synchronization mechanism are substantial. A response may be due to any one of many requests for any one of a many tasks.

Dataflow architectures provide such a scheduling and synchronization mechanism at the instruction level. Each processor maintains a very large set of small ready tasks and every message contains a large tag to identify the task with which it is to synchronize. Specialized hardware is provided to perform the match and enable the task in a few cycles. The processor pipeline is oriented towards scheduling a new task from the ready set. In traditional dataflow architectures, this dynamic scheduling occurs on essentially every instruction. It is claimed that by addressing this synchronization requirement as a fundamental design issue and integrating the mechanism deeply into the processor, the overall synchronization cost will be less than that resulting from a more *ad hoc* treatment.

In addition to this main argument, two other claims are posited. First, it is observed that increasing the processor state reduces the frequency of remote requests, but increases the switch cost since more state must be saved and restored. Traditional dataflow architectures provide only a small amount of state with each task, (the tag and the values on the two input tokens), and no save/restore is required on a switch. Thus, it is claimed that dataflow models provide superior latency tolerance. Arvind and Iannucci suggest that this advantage will carry forward to thread-based dataflow models, with no saving/restoring of state, *e.g.*, Hybrid[14], SCB[19], and ETS[10]. Today, it is widely accepted that if “chunks” of computation are assigned to processors then the latency on certain communication steps is neither long nor unpredictable. The chunk can be linearized into an instruction sequence, or thread, with the small amount of parallelism within a chunk exploited using standard pipelining techniques. The size of these chunks is highly dependent on the strength of the assumptions made about local processor storage, as discussed below.

Finally, it is claimed that general-purpose parallel programming requires that parallel tasks be generated dynamically and that the *synchronization namespace* be large. If remote accesses can be suspended and resumed at the remote site, then the latency observed at the processor can be arbitrarily large. Further arguments can be put forward in terms of ease of compilation and programming generality, but these are hard to codify.

2.2 The formal argument

In order to critique the TFI argument we need to formalize it more carefully. The basic architectural assumption is that each remote operation incurs a latency of some L cycles and that the processor does some R cycles of useful work between remote operations, *i.e.*, there are $\rho = 1/R$ remote references per instruction. Both L and R vary from request to request, but the assumption is that L is often large, *i.e.*, hundreds of cycles, and R is usually small, say ten cycles.[1] If remote requests are not overlapped with computation, the processor repeatedly executes for R cycles and then waits for L cycles, giving an average processor utilization of $U_1 = \frac{R}{R+L} = \frac{1}{1+L\rho}$.

For example, if L is ten times the run length R, the processor utilization is 9%. L reflects the average round-trip time through the network, including the quick servicing of the request on the remote end.[2]

To describe the latency tolerant architecture, we assume a physical processor hosts v *virtual processors* (VP) with at most one remote request each. Each VP alternates between computation and communication, just like the conventional processor. However, at the end of each local run the physical processor switches to another VP with a synchronization cost of S cycles. This cost combines the cost of switching to another enabled VP and of enabling the requesting VP upon arrival of the response.

If v is sufficiently large, then each request will complete before the other $v - 1$ VPs have finished their turns at the processor. Thus, the processor is never idle. The communication latency is entirely masked by the collective computation phases. We say the physical processor is *saturated* with available work. The utilization of the physical processor is given by $U_{sat} = \frac{R}{R+S} = \frac{1}{1+S\rho}$. This shows clearly how synchronization cost determines the effectiveness in tolerating latency. If the synchronization cost is equal to the run length, the processor is half utilized in saturation.

If the processor hosts several VPs, we may consider switching more frequently than the remote reference rate to (hopefully) reduce the switch cost. Switching cheaply on a cache miss is rather difficult, for example, because the miss is discovered very late in the processor pipeline. Switching on every load or on every instruction can be initiated much earlier in the pipeline. This is the primary motivation for interleaved pipelines, as in Hep[15], Tera[3], and Monsoon[16]). However, in these machines there are really two kinds of switches. When the switch does not involve a remote access, the VP immediately re-enters the pipeline. On a remote reference the essential state of the VP is packaged and delivered into the network with the request. An additional mechanism is involved when the request completes and the VP re-enters the pipeline. Thus, the characterization above in terms of R and S applies, however, to saturate the processor an additional $P - 1$ VPs are required, where P is the pipeline depth.

[1]We will ignore the variations in R and L and work with fixed average values. A more detailed analytical model can be found in the references[18]. Also, we will optimistically assume that the network can support the remote request rate[8].

[2]We pose the argument in terms of reads, since the need to wait for the resulting data is obvious. Correct treatment of write operations depends heavily on the consistency model assumed. However, even fairly weak consistency models[1] require an acknowledgement much like a read.

2.3 Extensions of the argument

TFI considers two extremes: a conventional processor that idles on remote references and a multithreaded processor in saturation. Saturation occurs when $v(R+S) \geq R+L$. Our model applies to intermediate design points. If v is small, then the sum of the computation phases does not cover the per request latency. There is some latency induced idle time and this, rather than the switch cost, determines the processor utilization. The utilization is computed as the useful work per compute/communicate cycle, assuming the switch cost is overlapped with message transmission:$U(v) = \frac{vR}{R+L}$. The idle time is $L-(v-1)R-vS$. We call this the *linear* regime, since the utilization is proportional to the number of VPs. For a given v, the utilization in saturation is determined by $S\rho$. As L increases, the processor drops out of saturation into the linear regime, where the utilization falls with L, just as it does for a conventional processor.

2.4 Summary

The TFI argument is at its core a statement about cost/performance. In the presence of frequent remote references subject to long latency, it may be possible to improve performance at a comparatively small increase in cost by supporting several VPs per processor. The argument could as well be stated from the opposite perspective. Given a large number of processors which are all poorly utilized, it may be possible to obtain close to the same overall performance at much lower cost by virtualizing the processors and multiplexing collections of them across a single set of physical resources. The savings depends on how physical resources are shared. Multithreading for latency tolerance is a statement about packaging — it may be cost-effective to package two processors in one.

However, this new dimension of design alternatives must be considered in light of all the other engineering trade-offs, such as pipelining and the design of register files. An alternative that eliminates idle time by drastically reducing the base processor performance or drastically increasing the per processor cost is not advantageous overall. The issues of cost, size, and speed come together in the engineering of the storage hierarchy. We will argue that the realities of the storage hierarchy fundamentally limit the amount of latency that can be effectively tolerated by restricting the number of VPs among which synchronization costs are small.

Let us reconsider the latency tolerance model from an engineering perspective. Suppose that only v_0 VPs can be maintained in the physical processor. Then the amount of latency that can be tolerated is $(v_0-1)(R+S)$. Beyond this point, idle time occurs, just as it does for a conventional processor. Observe that the tolerable latency depends on the remote reference frequency, which is dependent on the program and the storage hierarchy.

The total synchronization cost is the product of the number of remote references and the per reference synchronization cost. This point seems to be missed in TFI. Increasing the processor state eliminates some of the remote references at the expense of increased switch cost. If the reduction in the rate exceeds the increase in cost, there is a performance advantage to providing a large, perhaps less versatile, processor state.

The failure to include the storage hierarchy is where the “high level” perspective of a collection of processors and memory modules interconnected by a communication network breaks down. The performance of each processor, its cost, the number of VPs it can support, and the synchronization cost are all strongly dependent on the storage hierarchy. This relationship is explored in the next section. Furthermore, the effectiveness of the storage hierarchy is strongly affected by the scheduling of computation. This leads us to re-examine, in later sections, the local greedy scheduling implied by the dataflow firing rule, “operators fire whenever data is available.”

3 Limit I: Storage Hierarchy

The design of efficient storage hierarchies is one of the most critical areas in the evolution of computer architectures. For uniprocessors, almost complete consensus has emerged. The top level is typically organized as a multiported register file and various datapath latches. The next one or two levels are static RAMs organized as a cache. The bulk of the primary storage is realized by extremely dense and cost effective dynamic RAM, generally backed by operating system managed disks. With certain notable exceptions, this organization and management policy operates in concert with the way computation is scheduled in sequential programs. Even when operating on large data sets, small regions of data are drawn towards the upper levels and accessed repeatedly. The exception is when computation sweeps through large arrays, as in vector processing, which requires data to be streamed through the processor. The deep question underlying parallel computer architecture is how to organize the storage hierarchy to operate in concert with the way computation is scheduled in parallel programs.

In this section, we relate the structural characteristics of the storage hierarchy to the ability to tolerate latency. We begin with basics and then tie in the multithreaded model developed above. A key point to keep in mind is that the processor only operates directly on the top level of the hierarchy. Data below this level must be brought up before operations can be performed on it. This is the crux of the RISC debate, for example. CISC instructions gave the illusion of operations on memory data, but this data was implicitly moved up to registers. By making this movement explicit, RISC or load/store instruction sets allowed the compiler to use the registers more effectively.

3.1 Physics and dollars

The guiding principles in storage hierarchies are extremely simple: only small memories are fast, big memories are slow, speed is expensive. These are based on several physical and economic factors, including the following. (i) The address must be decoded to isolate the selected storage element. The depth of this decode tree grows with the log of the memory size. (ii) The selected data must be extracted from the storage array. Typically, the storage elements drive a common bit-line or bus to the output drivers. The load on the bit-line increases with the size of the storage array, so the propagation delay increases with size. Alternatively, many bit lines can be brought out of the storage array and multiplexed onto the output. This increases the size of the storage array and also requires a multiplexor tree of logarithmic depth. (iii) Discrete drops performance occur with increasing size as the organization of the basic storage cell changes from a full static flip-flop to a single capacitor requiring refresh. (iv) For extremely small memories, such as register files, multiple bit-lines can be provided to increase the bandwidth through multiple simultaneous read or write ports. (v) The output delay can be reduced by increasing the drive of the storage element, thereby increasing the area and power dissipation. The increase in cost per bit of storage is warranted for small memories, but prohibitive for extremely large ones.

It is difficult to determine precise boundaries between levels from these factors or a precise formula relating size and access time, since the constants vary from generation to generation. For example, Sparc register windows provide a larger first-level register set than other RISC architectures by a factor of roughly three. This does not necessarily limit the cycle time, although it does increase the design challenge of building an adequate register file. It also does not imply that the size could be increased by another factor of three. A rule-of-thumb is that at each level the latency increases by a factor of five, the capacity increases by a factor of a hundred and, the bandwidth decreases by a factor of two.

3.2 Synchronization cost

Our basic observation is that it is only possible to switch quickly to VPs that are resident at the top level of the storage hierarchy. Since this level is small, only a small number of VPs can reside there. Therefore, the amount of latency that can be tolerated is limited. To make this argument more precise, we need to develop a more careful accounting of synchronization cost.

TFI discusses two forms of synchronization. Intra-VP synchronization involves coordinating the response from a remote access with the VP that issued the request. Inter-VP synchronization occurs when coordinating two independent threads of control in different VPs. The second form typically involves repetitions of the first, for example when one thread repeatedly attempts to obtain a lock via a remote location. The effectiveness of a particular high-level strategy for coordination is highly program dependent and largely independent of the virtualization of processors, whereas the first form is primarily architectural. We will focus on intra-VP synchronization.

Each synchronization event involves three components: associating the response with the appropriate VP, enabling the VP, and resuming the VP. The run-time cost of the first two can be reduced by adding special purpose hardware, such as a small matching store and a hardware scheduling queue. These impose some fixed cost in design time and hardware resources. The third component is more interesting. The cost of resuming the VP appears explicitly where exclusive access is required to some shared resource. In addition, there is a cost associated with the reorganization of data within the storage hierarchy, which is often implicit. Specialized hardware support tends to shift the cost from the explicit category to the implicit, where it is obscured, but not absent.

For example, if page tables and translation buffers are maintained on a per-VP basis, it will be necessary to flush and restore the addressing environment on each switch. However, if the virtual address is extended with the VP identifier or shared across VPs, then the cost of the switch will appear only as increased translation misses and larger address tags. Similarly, if there is only one set of registers, these will need to be explicitly saved and restored as part of the switch. If multiple register sets are provided, then an inexpensive switch can be performed to other resident VPs, but switching to a non-resident VP will incur the cost of a register save/restore, and presumably a trap of some kind[13, 2]. In order to increase the number of resident VPs within a fixed number of registers, the number of registers per VP must be reduced[11]. In this case, the cost of data reorganization appears extra instructions to load and unload registers. The other latency tolerant proposals obscure the cost of synchronization more deeply, so more careful examination is required.

Dataflow architectures essentially replace the small register number with a large tag that serves to "name" the value. A realistic view of the storage hierarchy requires that only a small number of such name/value pairs can be resident at a time. Once the number of VPs exceeds the capacity of the top level matching store, the synchronization cost increases dramatically, since some form of overflow store must be used[12, 6]. In dataflow models the problem is made more severe by the lack of distinction between long lived call-frame storage and short lived register storage. Throttling[17] and k-bounding[9] were introduced to constrain the parallelism in dataflow programs so that the VPs would fit within a modest matching store.

The Hybrid proposal[14] makes a clear distinction between call-frame and register storage, and places the requirement that registers must be vacant at the point of a potential switch. Thus, registers are saved and restored *in defense* as part of code generation. The true synchronization cost is the accumulation of the artificial data movement. This is obscured in Hybrid by the inclusion of memory addressing modes.

Monsoon[16] provides a similar two-level store with more of the traditional dataflow structure. It recognizes that the dataflow matching store must be large and implements it as a directly addressed static RAM under the ETS frame-based execution model. Registers are associated with each interleave in the pipeline and must be vacant across potential match points. The cycle time of the machine is determined by the rate of access to the SRAM; the registers serve only to reduce the frequency of matches, since each match costs an ALU cycle. The more dominant synchronization cost is the increase in cycle time resulting from the SRAM access on the critical path. This could be reduced by caching the frame store, but then once again the synchronization cost will appear as the miss penalty if the number of resident VPs is large.

Since the size of the top level of the storage hierarchy determines the amount of latency that can be effectively tolerated, what if the top level is simply eliminated? This is essentially the approach adopted in HEP[15] and Tera[3], which use register addressing modes to access a sizable SRAM. The register access requires multiple cycles, but by interleaving VPs across multiple banks a new access can be initiated every cycle. The top level of the physical storage hierarchy contains only pipeline latches. The argument in favor of Tera is that if a remote request is issued every cycle then it is impossible to utilize a small set of registers effectively anyways. For example, if $L = 70$ as suggested for Tera then more than 70 VPs will be required. Whatever register occupancy was established for a VP when it issued a remote request will be flushed by the time the response completes. On the other hand, wherever the remote reference rate is lower due to a local calculation or recurrence, the execution rate is reduced due to the lack of single cycle register access. Thus, at a small increase in cost, the performance of the processor can be increased where the computation is not fundamentally limited by remote reference bandwidth in the presence of excess parallelism. By eliminating the top level of the storage hierarchy, this opportunity is lost.

In summary, a major flaw in the TFI reasoning is to view the synchronization cost as independent of the number of virtual processors. The synchronization cost can only be small among a small number of VPs. A large set of VPs can be supported in the architecture through various means, but switching to non-resident VPs must incur a larger cost. Thus, a more accurate model of processor utilization under latency tolerance would be the following.

$$U(v) = \frac{1}{C_v} \begin{cases} \frac{R}{R+S_v}, & \text{if } v(R+S_v) \geq R+L, \\ \frac{vR}{R+L}, & \text{otherwise,} \end{cases}$$

where S_v is the switch cost as an increasing function of v and C_v is the increase in cycle time, cycles per useful instruction, and instruction count resulting from switching as an increasing function of v.

This model leads to very different conclusions than the naive model employed in TFI. The naive model implies that for any latency it is possible to achieve peak processor utilization by increasing the number of VPs. In reality, when the synchronization cost increases even slowly with the number of VPs, the utilization in saturation decreases. Thus, the processor utilization reaches a plateau and drops off as either the latency or number of VPs is increased.

4 Storage directed scheduling

We have shown that there is a trade-off between achieving high processor utilization with many VPs and achieving high computational efficiency by making good use of the storage hierarchy. The latency that can be tolerated is proportional to the number of outstanding requests, but each request requires storage resources in the processor so that the response can be handled and tied back into the computation. Thus, the compiler optimization goal should be to achieve maximal latency tolerance using a small, fixed set of processor and storage resources. The key idea is to minimize the resource "footprint" of outstanding requests. This is similar to the optimization goal on sequential machines, where just enough parallelism is exposed to utilize multiple function units. Exposing too much increases the number of registers required to hold variables and intermediate values, adversely affecting register allocation.

In the simple VP model each outstanding request requires the full state of a VP. This can be improved by allowing each VP to have more than one outstanding request, sharing the same VP state. A further improvement is to share resources among VPs. For example, values which are live over an entire function activation may be allocated to registers that are shared among all the VPs of that activation. Finally, switching among VPs which share resources can be made cheaper than switching among unrelated VPs. Therefore, it is desirable to bias the scheduling towards switching among related VPs first.

4.1 The Threaded Abstract Machine

The Threaded Abstract Machine (TAM) provides a well defined compilation framework in which the above optimizations are available[7]. It eliminates the notion of VPs multiplexed onto a physical processor and instead exposes the fixed physical processor resources. Furthermore, the scheduling of computation is exposed, allowing the compiler to favor switching among related computations or to take special action when this is not possible. The effect is that the compiler can manage state across larger pieces of code than a uniform VP view would allow.

As shown in Figure 1, the storage hierarchy exposed in TAM is that of a conventional RISC processor: a fixed register set, the current activation frame which is typically in the cache, other activation frames to which access is typically slower, and heap storage which is distributed across the machine and requires split-phase access.

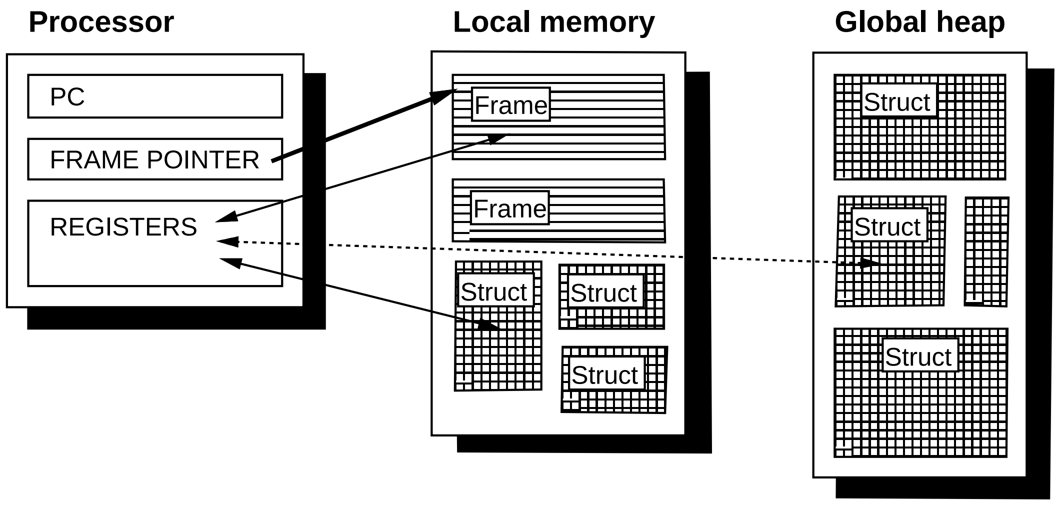

Figure 1: *The TAM storage hierarchy.*

To give the compiler control over scheduling of computation, TAM defines a scheduling hierarchy depicted in Figure 2 that is closely related to the storage hierarchy:

Instructions within threads are scheduled statically by the compiler. A TAM thread is a sequence of instructions that, once scheduled, executes to completion without suspension. To form threads, the compiler analyzes the control and data dependences, groups instructions that can be enabled together into *partitions*[20], and forms TAM threads using traditional instruction scheduling and register allocation algorithms to maximize pipeline utilization.

It is usually desirable to create as large threads as possible. Since a thread can never use the result of a remote reference it issues, the remote reference rate fundamentally limits thread size. However, a thread can issue more than one remote reference and may depend on more than one response.

Threads of a function activation are scheduled dynamically by primitive control operations. A TAM program is composed of a collection of codeblocks, roughly corresponding to functions in the source text. Each codeblock in-turn consists of a number of threads. When a codeblock is invoked, an activation frame is allocated to provide local storage. TAM biases the scheduling toward logically related threads: when a thread completes another enabled thread of the same activation is scheduled if possible. When no thread is enabled for the current activation, another activation frame is scheduled. We call the collection of threads of an activation that execute together a *quantum*.

This storage-directed scheduling can be implemented efficiently on conventional processors. Each processor maintains a *continuation vector* holding the start addresses of all enabled threads of the current

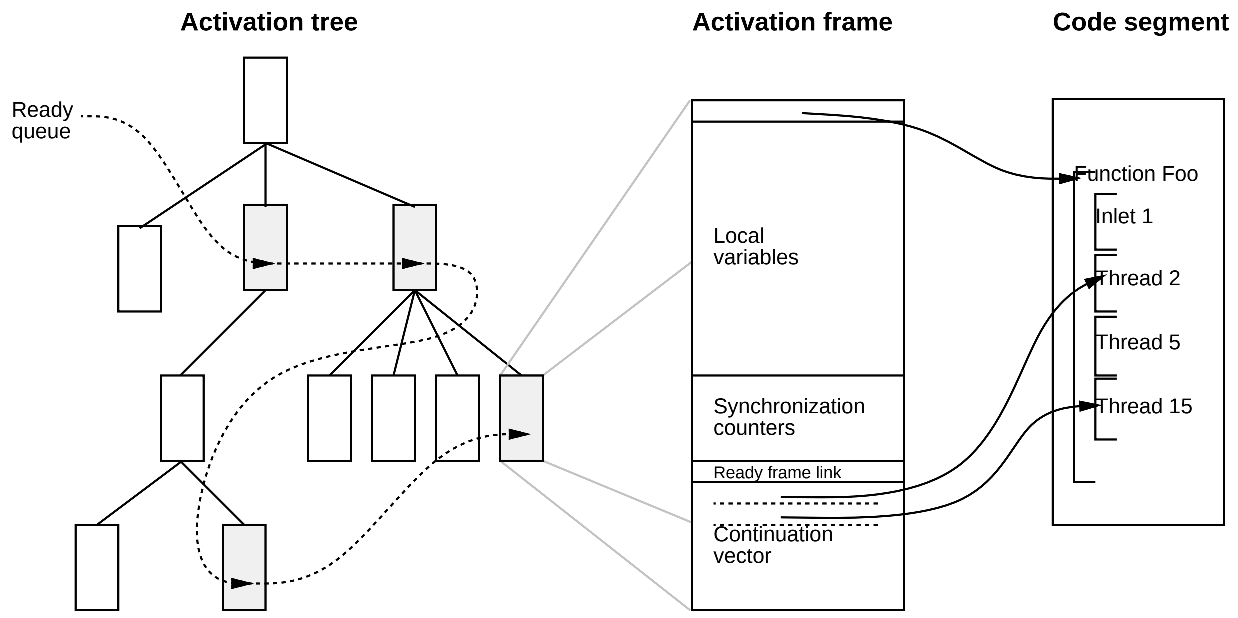

Figure 2: *The TAM scheduling hierarchy.*

activation. The continuation vector is represented as a stack, so a thread is enabled by pushing the thread address and the next thread is scheduled by popping a thread address and jumping to it. Since the compiler is guaranteed that threads of the same activation are executed as long as possible, it can allocate values to registers across thread boundaries where the threads will execute during the same quantum. Values not allocated to registers are stored in the activation frame and can typically be accessed from cache.

Function activations are scheduled dynamically by compiler generated code. When the current activation has no more enabled threads, TAM switches to another activation. The code to perform this switch is emitted by the compiler on a codeblock-by-codeblock basis, so it can save and restore live registers and determine which activation is to be scheduled next. Thus, the compiler can keep values in registers across threads even if it cannot prove that the threads will execute in the same quantum. For example, it can speculate that remote references will return in time for the thread using the responses to be scheduled in the same quantum. The control over the frame scheduling can also be used to keep a small pool of frames in-cache and switch among these with high priority. (Such a frame pool is specially useful on architectures with multiple register sets.) Finally, the compiler can direct the scheduling using high-level knowledge gained through program analysis or explicit programmer directives.

4.2 Evaluation of TAM scheduling

To evaluate effectiveness of the TAM scheduling hierarchy we have collected dynamic statistics for an implementation of TAM on a 64-processor Thinking Machines CM-5. The statistics were gathered using two scientific application benchmarks: Gamteb, a Monte Carlo simulation of neutron transport, and Simple, a hydrodynamics and heat conduction simulation.

Table 1 shows the average cost of a scheduling event at each level in the hierarchy. The average thread length is roughly that of a typical basic block, when measured in TAM operations, which allow direct frame access and single instruction network access. This is not surprising, since the control operation in TAM is to fork a thread. The expansion of TAM operations to Sparc instructions is heavily influenced by the cost of sending messages. Switching from one thread to another costs only 8 cycles (four Sparc instructions) on average. However, more than twelve threads are executed in a typical quantum. Thus, if register allocation is performed across threads, there are several hundred instructions to work with. Also, the cost of switching activation frames is amortized over this amount of work. A typical frame experiences about three periods of activity during its lifetime. This is comparable to a function in a sequential language that makes two calls.

	Gamteb 8096	Simple 128
TAM instructions per thread	4.5	6.4
Sparc instructions per thread	24	33
Instruction switch cost (cycles)	0	0
Threads per quantum	12	13
Thread switch cost (cycles)	8	8
Quanta per function invocation	3	3
Quantum switch cost (cycles)	46	46

Table 1: Dynamic scheduling characteristics under TAM for two programs on a 64 processor CM-5.

5 Limit II: Local Dynamic Scheduling

The storage directed scheduling in TAM overcomes many of the problems associated with traditional dataflow scheduling by focusing the processor on the portion of the computation that is at the top of the storage hierarchy. It reduces the footprint of outstanding requests by pipelining logically related requests through the network. However, like dataflow this is an entirely local scheduling policy. In this section, we show by example that it is prone to serious scheduling anamolies, where many processors are idling even though the program has ample parallelism. (These tiny examples were motivated by circumstances in real programs observed in our Id90 implementation.) At first glance the problem appears to be the coarse grained, unfair scheduling in TAM. However, similar anomolies can occur with fine grain, fair scheduling, as is usually associated with dataflow models. The real problem is relying on a naive, local scheduling discipline with no global strategy.

5.1 Pitfalls of storage-directed scheduling

The potential hazards of TAM scheduling are illustrated in Figure 3. Here two tasks are executed on each processor, one short and one long. The short task spawns off the two tasks executed by the next processor, while the long task just does some computation. There are plenty of independent tasks, but ineffective self-scheduling may fail to expose them. The short task should be scheduled first, so it can spawn work for the next processor. Execution of the long task is overlapped with the computation on succeeding processors, resulting in a nice pipelined execution (top right). However, if the long task is scheduled first only one processor will be busy at a time, resulting in very poor efficiency (bottom).

Fair, fine-grained dataflow scheduling does not exhibit this problem, because neither task is delayed arbitrarily. Execution alternates between small fractions of the two tasks, so even if the long task is scheduled first, the short task is executed and the subsequent tasks spawned before finishing the large task. The execution time is at most a factor of two from the optimal scheduling.

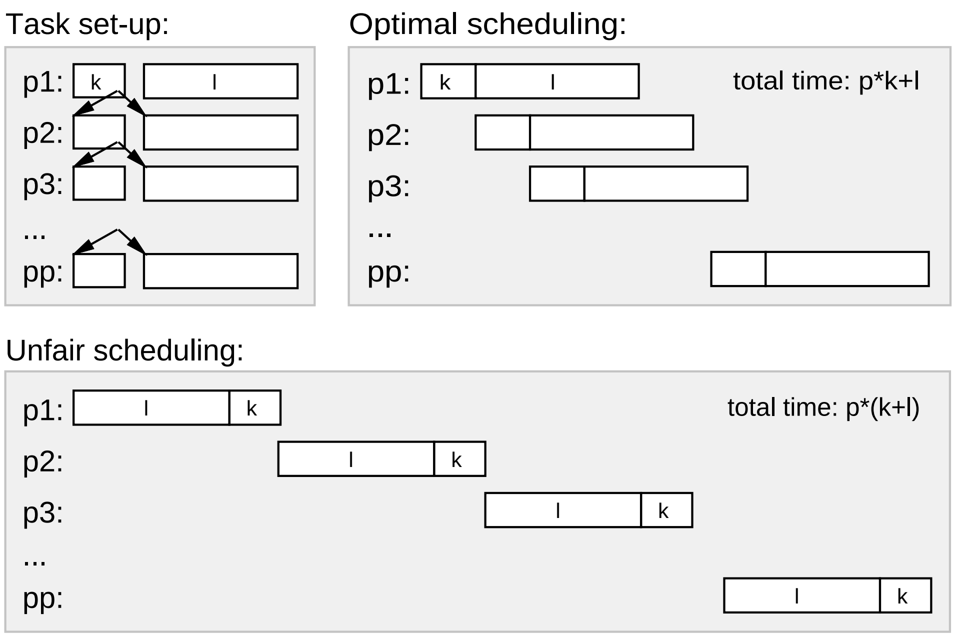

Figure 3: *Potential TAM scheduling pitfall.*

5.2 Pitfalls of dataflow scheduling

Surprisingly, fair scheduling can exhibit the same problem. In the example given in Figure 4, l independent tasks are executed on each processor. One task spawns the l tasks executed by the next processor at the end of its execution. This task should be executed first and to completion, resulting in a pipelined execution on all processors (top right). Fair scheduling would execute a small fraction of all tasks, then another fraction, and so on. Only when the last fraction of the first task is executed are the tasks spawned to the next processor, so only one processor is busy most of the time. Even simple unfair scheduling is likely to be better than fair scheduling: if tasks are chosen at random, unfair scheduling is better than fair scheduling by a factor of two on average.

5.3 Resource requirements

Similar examples can be found that show a large difference in memory resource requirements. Executing a computation represented as a complete binary tree in a fair (breadth-first) manner results in exponentially more resource requirements than an unfair (depth-first) manner. However, if the computation is represented by a chain of nodes, where each node has an additional leaf node attached to it, then depth-first scheduling may follow the chain first and expose all of the leaves, whereas breadth-first exposes them one at a time.

5.4 Non-greedy scheduling

We have seen that the highest levels of the storage hierarchy can support only a fixed number of scheduling contexts. What should be done if the latency is so high, that the number of VPs needed to tolerate it is larger than the storage hierarchy can support? In many cases, it may be better to wait for the remote references to return than to switch to some unrelated context. The state needed when the requests return is already in the highest level of the storage hierarchy, so the processor is ready once the requests return. Switching to some unrelated computation may result in thrashing. In other cases, it is advantages to preempt the current

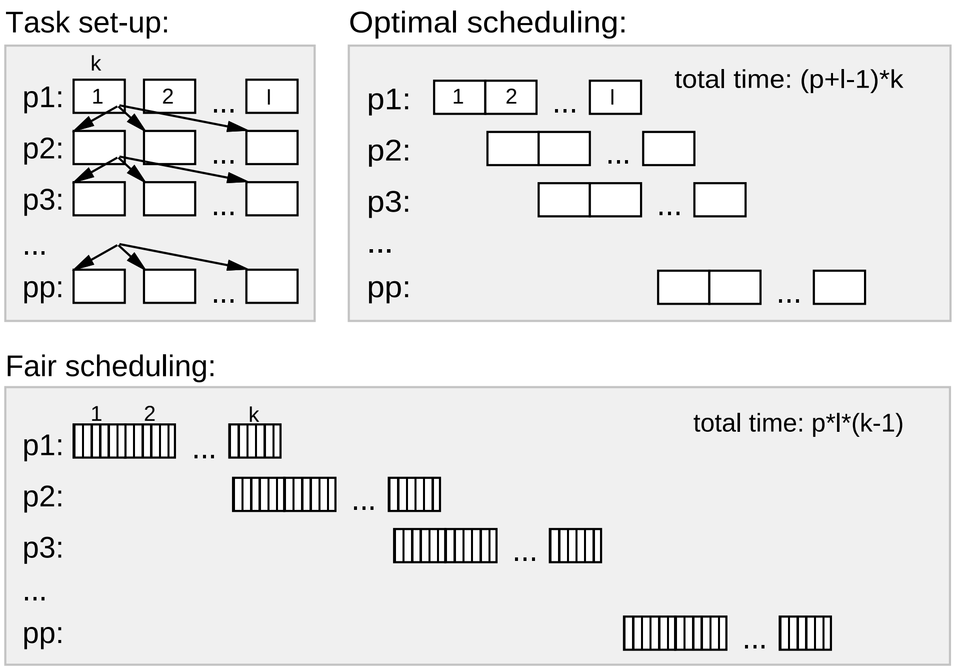

Figure 4: *Potential dataflow scheduling pitfall.*

VP, even if it makes no remote references. The relative advantages of waiting or moving on to other VPs is difficult to determine on a purely local basis.

6 Summary and Open Challenges

The most important argument put forward by proponents of dataflow architectures is the ability to tolerate long and unpredictable latencies by providing cheap, fine-grained dynamic scheduling and synchronization. This allows remote communication requests to overlap with computation. We have shown that, while the basic argument has merit, it ignores the storage hierarchy and the pitfalls of naive scheduling. These pose fundamental limits on the amount of latency that can be tolerated and the efficiency of the overall computation. In a real machine the cycle time is determined by the top level of the storage hierarchy, which can only support a small number of outstanding requests. Latency tolerance and locality can be increased by biasing the scheduling toword collections of logically related VPs and allowing outstanding requests to share resources. Static scheduling, where possible, allows the compiler to efficiently manage registers, and dramatically reduces the overall need for dynamic scheduling.

However, any naive local scheduling policy exhibits unnecessarily low machine efficiency or high resource requirements on some programs. Thus, it would seem unwise to rely solely on low-level hardware mechanisms or run-time system support to determine the scheduling of computation. Most current dataflow and multithreading proposals do not even provide a means of directing scheduling based on some higher level understanding of the program execution, since the scheduling queue is implicit in the execution model. TAM exposes the scheduling queue, so the mechanism exists, however, it remains an open question how this high-level direction should be applied in a consistent manner. Acceleration of the local, naive dynamic scheduling mechansism through hardware enhancements has no impact on this fundamental problem. Work on k-bounded loops[9] and throttling[17] is a step in the right direction, but neither of these approaches has fully considered that scheduling of computation must compete for resources with the computation. If compilers are not able to meet this challenge, the programmer will be forced to express the scheduling in the program, by setting the k-bounds, specifying priorities, specifying where to allocate computation, and so on.

Acknowledgements

We are grateful to Arvind and Bob Iannucci for providing the insights and framework for an on-going examination of the central issues in multiprocessor design. We would like to acknowledge the key contributions of Seth Copen Goldstein to the TAM work and the helpful comments of the referees.

David Culler received support from a National Science Foundation Presidential Faculty Fellowship Award (CCR-9253705). Klaus Erik Schauser is supported by an IBM Graduate Fellowship. Thorsten von Eicken is supported through the Semiconductor Research Corporation. Computational resources were provided, in part, under NSF Infrastructure Grant CDA-8722788.

References

[1] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proc. of the 17th Annual Int'l Symp. on Comp. Arch.*, pages 2–14, Seattle, WA, May 1990.

[2] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. 17th Annual Int'l Symp. on Comp. Arch.*, pages 104–114, Seattle, WA, May 1990.

[3] R. Alverson, D. Callahan, D. Cummings, Koblenz B., A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. of 1990 Int'l Conf. on Supercomputing*, June 1990.

[4] Arvind and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. In *Proc. of the 10th Annual Int'l Symp. on Comp. Arch.*, Sweden, June 1983.

[5] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proc. of DFVLR - Conf. 1987 on Par. Proc. in Science and Eng.*, Bonn-Bad Godesberg, W. Germany, June 1987.

[6] S. A. Brobst. Instruction scheduling and token storage requirements in a dataflow supercomputer. Master's thesis, Dept. of EECS, MIT, Cambridge, MA, May 1986.

[7] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of 4th Int'l Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, Santa-Clara, CA, April 1991.

[8] D. E. Culler. Multithreading: Fundamental Limits, Potential Gains, and Alternatives. In *Proc. of the Supercomputing91 Workshop on Multithreading*, 1992. (to appear).

[9] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proc. of the 15th Annual Int'l Symp. on Comp. Arch.*, pages 141–150, Hawaii, May 1988.

[10] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, (10):289–308, January 1990.

[11] W. Dally. Virtual Channel Flow Control. In *Proc. of the 17th Annual Int'l Symp. on Comp. Arch.*, Seattle, WA, May 1990.

[12] J. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the Association for Computing Machinery*, 28(1):34–52, January 1985.

[13] R. H. Halstead, Jr. and T. Fujita. MASA: a Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proc. of the 15th Int'l Symp. on Comp. Arch.*, pages 443–451, 1988.

[14] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15th Annual Int'l Symp. on Comp. Arch.*, pages 131–140, Hawaii, May 1988.

[15] H. F. Jordan. Performance Measurement on HEP — A Pipelined MIMD Computer. In *Proc. of the 10th Annual Int'l Symp. on Comp. Arch.*, Stockholm, Sweden, June 1983.

[16] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proc. of the 17th Annual Int'l Symp. on Comp. Arch.*, Seattle, WA, May 1990.

[17] C. A. Ruggiero. *Throttle Mechanisms for the Manchester Dataflow Machine*. PhD thesis, University of Manchester, Manchester M13 9PL, England, July 1987.

[18] R. Saavedra-Barrerra, D. E. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the 2nd Annual Symp. on Par. Algorithms and Arch.*, July 1990.

[19] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. of the 16th Annual Int'l Symp. on Comp. Arch.*, Jerusalem, Israel, June 1989.

[20] K. Schauser, D. Culler, and T. von Eicken. Compiler-controlled Multithreading for Lenient Parallel Languages. In *Proc. of the 1991 Conf. on Functional Prog. Lang. and Comp. Arch.*, Cambridge, MA, Aug. 1991.