Program Partitioning for Multithreaded Dataflow Computers

Ben Lee

Department of Electrical and Computer Engineering
Oregon State University

Krishna Kavi

Computer Science Engineering Department
The University of Texas at Arlington

Abstract

In recent years, there have been numerous proposals that represent the current generation of dataflow computers. Although the spectrum of such architectures is very broad, a special class of these machines called multithreaded dataflow multiprocessors have recently received considerable attention. In multithreaded dataflow computers, one of the major challenges is the issue of program partitioning. Therefore, this paper presents a compile-time partitioning strategy for multithreaded dataflow computers. The proposed scheme is based on the minimum granularity required to maximally utilize a given processor pipeline. To determine such a partition, architectural and program characteristics, such as the frequency and the delays of split-phase loads and their effect on the parallelism required, are considered. Once these parameters are known, a partition can be chosen that best matches the capability of a given multithreaded dataflow multiprocessor.

1. Introduction

Advocates of the dataflow execution model argued that highly parallel multiprocessors based on the conventional control-flow concept are limited by two fundamental issues: *memory latency* and *synchronization* [2]. Memory latency is the time that elapses between issuing a memory request and receiving the corresponding response. In general, as the system becomes large so does its latency. The processor idling due to memory latency can be avoided by providing a facility to perform context switching. However, conventional multiprocessors incur a very large overhead during a context switch due to the existence of substantial local state associated with each process. Synchronization on the other hand is needed to enforce the ordering of instructions according to their data dependencies. In a conventional multiprocessor system, a program is partitioned into processes based on programming constructs such as loops, procedures, etc. Synchronization primitives are placed in parallel programs either explicitly by the programmer or by the compiler to enforce the execution ordering of the processes. The problem is the high overhead cost involved in the implementation of these synchronization primitives [11]. Therefore, the natural tendency is to choose a large granularity to limit the number of synchronization and context switching. The net effect is that parallelism is exploited usually at the large-grain level and thus sacrificing the potential fine-grain parallelism that exists in many application programs.

In contrast, the dataflow model of computation offers many attractive properties for parallel processing. First, the dataflow model of execution is asynchronous, i.e., the execution of an instruction is based on the availability of its operands. Therefore, the synchronization of parallel activities is implicit in the dataflow model. Second, instructions in the dataflow model do not impose any constraints on sequencing except the data dependencies in the program. Hence, the dataflow graph representation of a program exposes all forms of parallelism eliminating the need to explicitly manage parallel execution of a program. For high-speed computations, the advantage of the dataflow approach over the control-flow method stems from the inherent parallelism embedded at the instruction level. This allows efficient exploitation of fine-grain parallelism in application programs.

Due to its simplicity and elegance in describing parallelism and data dependencies, the dataflow execution model has been the subject of many research efforts. Since the early 1970s, a number of hardware prototypes have been built and evaluated [1, 8], and simulation studies of different architectural designs and compiling technologies have been performed [8]. The experience gained from these efforts has led to progressive development in dataflow computing.

One of the most important development is the emergence of a novel and simplified process of matching tokens called *direct matching* [5]. Most recent generation of dataflow machines, such as Monsoon [5], Epsilon-2 [9], P-RISC [14], and EM-4 [17, 21], all use direct matching for detecting enabled nodes thereby eliminating the expensive and complex process of associative search used in previous dynamic architectures [1]. Another important

0-8186-1060-3425/93 $03.00

development is a shift in viewpoint on the concept of dataflow and its implementation. Rather than viewing the dataflow execution model as the consumption of tokens and the firing of enabled nodes, it can be thought of as the execution of multiple interacting sequential threads, where fork and join are extremely efficient [15]. The importance of this observation is two-fold: First, there is a convergence between control-flow and dataflow concepts. Second, the latency due to split-phase loads can be masked by interleaving the execution of independent threads.

In light of the aforementioned discussions, one of the major challenge is the proper utilization of a processor pipeline through multithreading. It is apparent that to mask long and unpredictable latency due to split-phase transactions an appropriate number of independent threads must be allocated to a processing element. Therefore, this paper presents a compile-time partitioning strategy for multithreaded dataflow machines. The goal of the proposed scheme is to determine the optimum granularity that results in maximum utilization of a processor pipeline. To achieve this, the proposed strategy considers programming and architectural parameters, such as the frequency and the delays of split-phase loads, and evaluates a measure of required parallelism that can be used to construct an appropriate partition.

The organization of this paper is as follows: Section 2 presents the general characteristics of multithreaded dataflow computers. Section 3 discusses the problem of program partitioning for dataflow multiprocessors. In Section 4, we develop a simple analytical model for multithreading. The discussion is based on the speedup attainable and the relative parallelism needed to fully utilize the available resources in a pipeline processing element. Section 5 presents a partitioning strategy based on a measure of required parallelism developed in Section 4. Finally, Section 6 provides a brief conclusion.

2. Multithreaded Dataflow Architectures

In this section, a brief overview is provided to identify and analyze the characteristics of multithreading. In multithreaded dataflow computers, a thread of computation is completely described by an instruction pointer (IP) and a frame pointer (FP) [5]. The pair of pointers, <IP, FP>, is regarded as a *continuation* and it corresponds to the tag part of a token (terminology used in TTDA [1]). Thus, on each clock cycle, a different continuation or thread descriptor can be inserted into the processor pipeline allowing an arbitrary number of independent thread to be interleaved. The advantage of multithreading is that rather than blocking the execution of other threads a number of different threads can be inserted into the pipe. This effectively masks long memory latencies due to split-phase loads.

Dataflow multithreaded architectures can be viewed as either an evolution of dataflow architectures in the direction of more explicit control over instruction execution order, or as an evolution of von Neumann machines in the direction of better support for synchronization and tolerance of long latency operations. We present both possibilities by describing the architectural organizations of Monsoon and P-RISC.

The basic organization of the Monsoon processing element (PE) consists of an Execution Pipeline connected by a Token Queue (Figure 1). The Execution pipeline implements the dataflow instruction cycle based on the Explicit Token Store (ETS) model [5]. Monsoon provides a fine-grain dataflow capability by having continuations of the form <IP+r, FP>, where r is encoded as a literal in the instructions that represents the next instruction in a logical sequence (i.e., self-scheduling). Monsoon can also implement a more conventional form of sequential thread execution by letting r equal to 1. This is accomplished using a simple recirculate scheduling paradigm where continuations are immediately reinserted into the Execution Pipeline (via Direct Recirculation Path). Since a continuation completely describes a process, the processor pipeline can possibly contain eight unrelated threads. Therefore, the processor pipe can tolerate unpredictable latency due to split-phase loads as long as there is sufficient number of tokens (i.e., parallelism) in the Token Queue.

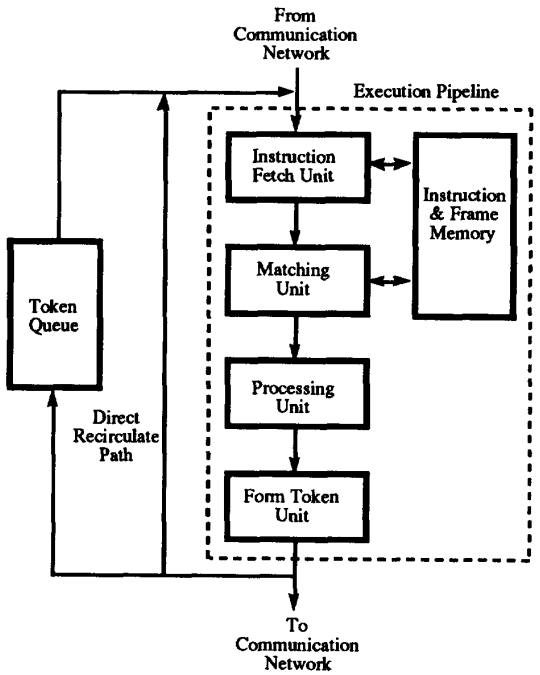

Figure 1. Organization of Monsoon Processing Element.

P-RISC was proposed by Nikhil and Arvind at MIT [14]. The organization of a P-RISC processor pipeline is shown in Figure 2. As the name suggests, P-RISC is based on an existing RISC-like instruction set where the instructions are 3-address frame-to-frame operations. Note that since P-RISC executes 3-address instructions on data stored in the frames, "tokens" only carry continuations. For normal arithmetic/logic instructions, the continuation is simply the next instruction in the thread—as in von Neumann scheduling (i.e., <IP+1, FP>). However, unlike the Monsoon, IP basically represents a program counter in the conventional sense and is incremented in the Instruction Fetch stage. In case of a non-local memory instruction, such as a procedure call or an I-structure read, it is directed to an appropriate PE or I-structure memory via Load/Store and Start units. To exploit fine-grain parallelism, the instruction set is extended with FORK and JOIN operations for thread initiation and synchronization, respectively. These operations are simple instructions, not operating system calls, that are executed directly within the normal processor pipeline.

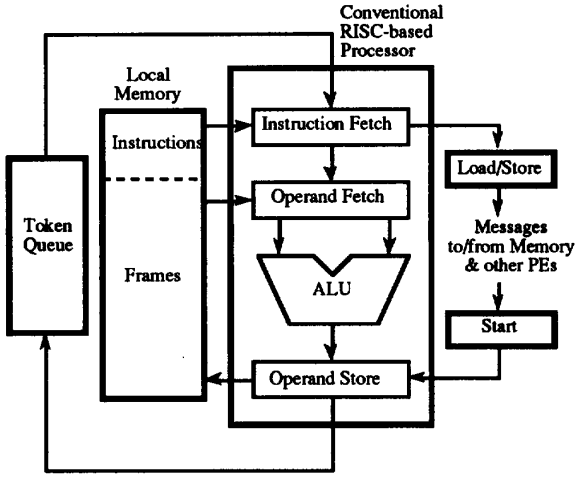

Figure 2. Organization of a P-RISC Processing Element.

3. Program Partitioning

There have been a number of proposed methods for partitioning programs on multithreaded dataflow multiprocessors. These approaches can be classified as either medium-grain or coarse-grain. In the medium-grain approach, the dataflow graph representation of a program is transformed into set of threads. A *thread* is a sequence of statically ordered instructions where once the first instruction in the thread is executed the remaining instructions execute without interruption [20]. Therefore, a thread defines the basic unit of work from the dataflow model point-of-view, where synchronization is required only at the beginning of a thread. There is a number of thread partitioning algorithms that convert dataflow graph representation of programs to threads [10, 19, 20]. The basic objectives of these algorithms are to maximize thread lengths and minimize inter-thread synchronization, and generate "safe partitions" that can be mapped onto processors [19].

The second approach is to construct a more coarse-grain partition. The two major factors that have a direct affect on the granularity of a partition are (i) the inter-processor communication overhead and (ii) parallelism. In general, the total inter-processor communication cost is minimized by clustering as many instructions as possible in a partition. An extreme case of this is a uniprocessor system, where inter-processor communication costs are non-existent. On the other hand, as granularity becomes larger the opportunity to exploit parallelism is reduced. Obviously, these are two conflicting objectives and therefore a proper compromise must be provided.

The question is then "what is a proper compromise between parallelism and communication costs?" Since the problem of finding the optimum partition is NP-complete [18], heuristic solutions are generally used to solve the problem. Sarkar and Hennessy proposed a method that attempts to partition a program graph on the basis of the optimum granularity dictated by a cost function that considers an appropriate balance between communication overhead and the execution time of the partition [18]. This is achieved by starting with the finest granularity partition that places each node (i.e., an instruction) in a separate subgraph. Then, iteratively subgraphs are merged to generate a new partition that provides the lowest cost function. This process is repeated until an entire program has been merged into one subgraph. A cost history is then used to reconstruct the partition with the lowest cost function.

Although the aforementioned method succeeds in providing a compromise between exploiting parallelism and limiting the communication overhead, it has neglected to consider an important factor in multithreaded dataflow multiprocessors—split-phase transactions. In other words, to find an optimum granularity, a partitioning strategy must consider the proper utilization of the hardware resources available. For example, a grain of computation assigned to a processor contains a set of independent threads. Since an arbitrary number of threads is interleaved in the processor pipeline, an important consideration is the amount of parallelism required in a particular grain to provide fast context switching capability that masks the memory latency during split-phase operations. Therefore, the following sections develop a quantitative measure for determining the required parallelism in a grain and an algorithm that

utilizes such a measure to construct an appropriate partition.

4. A Simple Analytical Model

In this section, a simple analytical model is derived for a pipelined processing element. The general assumption of the architectural characteristics is based on the dataflow proposals discussed in Section 2. Each processing element consists of a number of pipelined stages connected in a circular ring fashion via a token queue. We consider the execution of a grain (i.e., a partition) that consists of a set of threads. The execution of these threads can be based on two different approaches. In the first approach, a token is removed from the token queue at each pipeline clock cycle and inserted in the pipe. In the second approach, a thread is executed sequentially until it dies (due to split-phase operations or synchronizing points). The analysis presented in this section is based on the latter assumption; however, regardless of the approach, the net effect is that different threads are interleaved to mask the latency involved in split-phase operations.

4.1. The Speedup

In order to model the effect of split-phase load operations, we can consider the fraction f of a code that involves such delays. In addition, we define a *thread* as any sequence of statically ordered instructions. This is in contrast to the more restricted definition provided in [20]. The following notations are used in the modelling:

- N_i - the length of a thread i
- P - the number of independent threads
- k - the number of stages in a pipe
- τ - the average delay in each pipeline stage
- t - the total delay in the execution of an instruction (i.e., $t = \tau \cdot k$)
- f - the fraction of the instructions performing split-phase load operations
- $\bar{t}_{sp}$ - the average delay involved in a split-phase load operation

With these parameters, we can define the serial execution time Ts required to execute a grain of computation on a nonpipelined processing element as

$$Ts = t \cdot \sum_{i=1}^{P} Ni \cdot \left[(1-f) + f \cdot \frac{\bar{t}_{sp}}{t} \right], \tag{5.1}$$

where $t = \tau \cdot k$ and $\sum_{i=1}^{P} Ni$ represent the total number of instructions in a grain. If the same grain of computation is executed on a pipelined processing element, the ideal execution time is given as[1]

$$Tp = \tau \cdot \sum_{i=1}^{P} Ni. \tag{5.2}$$

This is possible since multithreaded dataflow computers provide instruction-level context switching capability and hence a processor pipe can accept a token at each clock cycle.[2] The *maximum speedup* S_{max} that can be obtained is

$$S_{max} = \frac{Ts}{Tp} = \frac{k \cdot \tau \cdot \sum_{i=1}^{P} Ni \cdot \left[(1-f) + f \cdot \frac{\bar{t}_{sp}}{k \cdot \tau} \right]}{\tau \cdot \sum_{i=1}^{P} Ni} \tag{5.3}$$

$$= k \cdot (1-f) + f \cdot \frac{\bar{t}_{sp}}{\tau}.$$

Equation 5.3 gives the maximum speedup achievable in a pipelined processing element, given the fraction of a code that performs split-phase load operations. The average delay involved in split-phase loads depends on the characteristics of the communication network and the structure memory. In general, for scalable architectures $\bar{t}_{sp}$ will increase with the number of PEs and can be expected to be greater than the delay of a pipeline stage $\tau \cdot k$ that yields the following bounds:

$$k \le S_{max} \le \frac{\bar{t}_{sp}}{\tau}. \tag{5.4}$$

Equation 5.4 indicates that depending on the delay $\bar{t}_{sp}$ the upper bound on speedup is limited by the number of pipeline stages or $\bar{t}_{sp}/\tau$. An important characteristic noted from Equation 5.3 is that the speedup can be extended beyond the number of pipeline stages k by a factor of $\bar{t}_{sp}/\tau$. The additional parallelism obviously comes from masking the latency of split-phase operations by interleaving a number of threads through the pipeline.[3]

Since the masking of delays due to split-phase loads can occur only as a result of sufficient parallelism within a grain, it is of interest to determine the parallelism needed to keep the pipe full. In order to acquire such an expression, the parallelism needed to mask the latency of split-phase loads must be isolated from the parallelism obtained from pipelining itself. The general *speedup* S that can be obtained from masking the latency of split-phase loads is given as

---

1 We assume Ni and P are sufficiently large such that start-up delays are negligible.

2 Although a split-phase operation causes a thread to die, we assumed that these operations are conceptually executed by all the stages in the pipeline, thus a "result" token can be generated every clock cycle.

3 This is consistant with empirical results obtained from studies of HEP, which is a multithreaded MIMD architecture [11].

$$S = \frac{Ts}{Tp} = \frac{\tau \cdot \sum_{i=1}^{P} Ni \cdot \left[(1-f) + f \cdot \frac{\bar{t}_{sp}}{\tau} \right]}{\tau \cdot \sum_{i=1}^{P} Ni} \quad (5.5)$$

$$= (1-f) + f \cdot \frac{\bar{t}_{sp}}{\tau},$$

where Ts represents the sequential pipeline execution time of P threads.

4.2. A Measure of the Required Parallelism

Equation 5.5 provides an expression for the speedup attainable by interleaving a number of threads through a processor pipeline. In order to determine the parallelism needed to maintain the processor pipeline busy, consider the relationship between speedup S, *parallelism* P, and *efficiency* E. Since the maximum utilization is achieved when $E = 1$, the parallelism required is given as [7]

$$P = \frac{S}{E} = f \cdot (\frac{\bar{t}_{sp}}{\tau} - 1) + 1. \quad (5.6)$$

The significance of Equation 5.6 is that it provides a general insight to the required parallelism within a grain. It states that the parallelism needed to fully utilize a processor pipe directly depends on the frequency and the latency of split-phase loads. For a grain that contains no split-phase loads, only a single thread is sufficient. On the other hand, as f increases more threads are required to mask the latencies. For an extreme and probably unrealistic case where f is unity, the number of parallel threads required is $\bar{t}_{sp}/\tau$.

The question is then how accurate of a measure is Equation 5.6 for evaluating the necessary parallelism? The following theorem is provided to answer this question:

Theorem 4.1: For a set of equal length parallel threads, where each threads length is equal to $m \cdot (\bar{t}_{sp}/\tau\text{-}1)$ for $m = 1, 2, 3...$, containing a fraction f of split-phase load instructions each incurring a delay of $\bar{t}_{sp}/\tau$, the *lower bound* on the number of parallel threads and thus parallelism P required to keep a k-stage pipeline full is $P \geq f \cdot (\bar{t}_{sp}/\tau\text{-}1)+1$.

Proof: For a given set of threads (i.e., a grain), f is defined as the ratio of the number of split-phase loads to the total number of instructions in the grain. Thus, the proof for Theorem 4.1 can be provided by considering the maximum number of split-phase loads a grain can have and yet provide full utilization of a processor pipeline. For a split-phase load operation j, at least $\bar{t}_{sp}/\tau$-1 other instructions are needed to fill the pipe before the succeeding instruction j+1 can be executed. To determine the maximum number of allowable split-phase loads within a grain, consider how often split-phase load instructions can be encountered without disrupting the instruction flow through the pipe.

For P parallel threads, at most P-1 successive split-phase load operations (i.e., interleaving of different threads) can occur before $\bar{t}_{sp}/\tau$-(P-1) instructions are forced to execute sequentially, of which the last instruction is a split-phase load. Once this requirement is satisfied, the blocked node j can continue its execution. Therefore, this process can be observed as a repetitive execution sequence of P-1 consecutive split-phase loads followed by $\bar{t}_{sp}/\tau$-P normal (i.e., non split-phase) instructions. Since the thread lengths must be the same, the total number of instruction executed is $P \cdot m \cdot (\bar{t}_{sp}/\tau\text{-}1)$, where $m = 1, 2, 3...$ The minimum fraction of the split-phase loads to the total number of instructions in a grain is then given as $f \leq (P\text{-}1)/(\bar{t}_{sp}/\tau\text{-}1)$, which yields $P \geq f \cdot (\bar{t}_{sp}/\tau\text{-}1)+1$.

Q.E.D

Consider an example of such a case shown in Figure 3 consisting of three parallel threads (i.e., $P = 3$) and $\bar{t}_{sp}/\tau = 7$. The shaded nodes indicate split-phase load instructions and the dotted lines imply the precedence relationships between nodes in a thread. The dark lines illustrate the execution order of the nodes in the threads. As can be seen by this example, the maximum number of consecutive split-phase loads is two before five instructions from a different thread are forced to execute sequentially. When this condition is satisfied, the blocked node at cycle 1 can resume its execution at cycle 8. This process repeats every 6 clock cycles and yields a result from the processor pipe every clock cycle.

It is clear from the Theorem 4.1 that when a grain contains a set of threads, the required parallelism obtained from Equation 5.6 is the absolute minimum. However, the above analysis assumes a restricted case where a grain consists of P parallel threads and the distribution of split-phase loads within the grain is such that the aforementioned condition is satisfied. However, such assumptions are not realistic in practice and to quantify the amount of parallelism required a more general equation is needed. This is obtained by the following equation that considers the *required parallelism* P_{REQ}, where

$$P_{REQ} = P + \varepsilon \quad (5.7)$$

and ε is the *error factor*. The error factor ε reflects the amount of additional parallelism needed when the distribution of split-phase loads within a grain is such that $\varepsilon \neq 0$.

Before we determine the error factor needed for the required parallelism, it is important to characterize the meaning of parallelism in a given program graph. In general, a grain of computation contains threads that are not of equal lengths and therefore it is often more natural to determine the amount of parallelism that exists in a grain in terms of its *average parallelism* P_{AVG} [7]. The

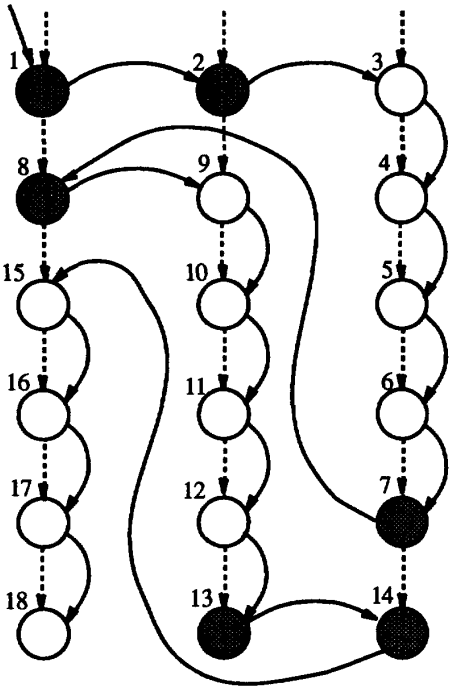

Figure 3. Ideal case execution sequence of a grain with $P = 3$, $f = 1/3$, and $\bar{t}_{sp}/\tau = 7$.

average parallelism of a program graph can be expressed as a ratio of the total serial execution time to the length of the critical path in a program [7], i.e.,

$$P_{AVG} = \frac{T_{S'}}{T_{Cr}} = \frac{S \cdot T_P}{T_{Cr}} = \frac{\sum_{i=1}^{P} N_i}{N_{Cr}}. \tag{5.8}$$

For the simplest case, we have $N_{Cr} = N_i$ for $1 \le i \le P$ and thus $P_{AVG} = P$.

The amount of parallelism needed can now be estimated for an arbitrary distribution of split-phase loads within a grain by considering the error factor ε that occurs when a portion of split-phase loads does not have enough parallelism (i.e., nodes) to mask the delays. However, this depends on the characteristic of a program and the method used to partition a program into valid threads [19]. Therefore, we take a more heuristic approach to determining the additional parallelism needed in a grain of computation by considering how often split-phase loads can be interleaved before additional parallelism is required to mask the latency. For P parallel threads, this situation occurs when P consecutive split-phase loads are processed and the P^{th} split-phase load instruction causes a stall of $\bar{t}_{sp}/\tau - P$ cycles. This situation is depicted in Figure 4. The threads are identical to that of Figure 3; however, the distribution of the split-phase loads is such that additional nodes (and thus addition parallelism) are needed to maintain full utilization of the processor pipeline. Thus, the error factor is given by

$$\varepsilon = \frac{\left\lfloor \frac{f}{P} \cdot \sum_{i=1}^{P} N_i \right\rfloor}{N_i} \cdot \left(\frac{\bar{t}_{sp}}{\tau} - P\right) \ge f \cdot \left(\frac{\bar{t}_{sp}}{\tau} - P\right). \tag{5.9}$$

The parallelism required to keep the processor pipe busy can be rewritten as

$$P_{REQ} \ge P \cdot (1 - f) + f \cdot \frac{\bar{t}_{sp}}{\tau}, \tag{5.10}$$

where P is defined by Equation 5.6.

Equation 5.10 is a measure that can be used to determine the minimum granularity of a partition. However, a set of threads within a grain may provide the appropriate average parallelism P_{AVG} and yet does not really satisfy the necessary requirement $P_{AVG} \ge P_{REQ}$. This can be illustrated by the worst case scenario that occurs when there exists a partially ordered graph that is of the "reverse -T" form shown in Figure 5. The average parallelism in such a graph is given by

$$P_{AVG} = 1 + \frac{N_{P\,max} - 1}{N_{Cr}}, \tag{5.8}$$

where $N_{P\,max}$ represents the *maximum parallelism* in the graph. As can be seen, the characteristic of a "reverse-T" graph is such that P_{AVG} can satisfy a required parallelism P_{REQ} as long as $N_{P\,max}$ and N_{Cr} are selected properly. For example, as long as $2 \cdot N_{Cr} = N_{P\,max} - 1$, $P_{REQ} = 3$ can be obtained. Yet, it is obvious that an additional parallelism of $(N_{Cr}-1)\cdot(P_{REQ}-1)/N_{Cr}$ is needed and we can define the *maximum parallelism required* $P_{REQ'}$ as

$$\begin{aligned} P_{REQ'} &= 1 + \frac{N_{P\,max} - 1}{N_{Cr}} + \frac{(N_{Cr} - 1)\cdot(P_{REQ} - 1)}{N_{Cr}} \\ &= P_{REQ} + \frac{N_{P\,max} - P_{REQ}}{N_{Cr}}. \end{aligned} \tag{5.9}$$

Thus, the bounds on parallelism required to keep a processor pipeline full are

$$P \le P_{REQ} \le P_{REQ} + \delta, \tag{5.10}$$

$$\text{where} \quad \delta = \frac{N_{P\,max} - P_{REQ}}{N_{Cr}}.$$

Note that $N_{P\,max} = P_{REQ}$ yields $P_{REQ'} = P_{REQ}$ In effect, Equation 5.9 detects the skewness of the parallelism available in a grain of computation and provides aconservative estimate on the additional parallelism needed to keep a processor pipeline full.

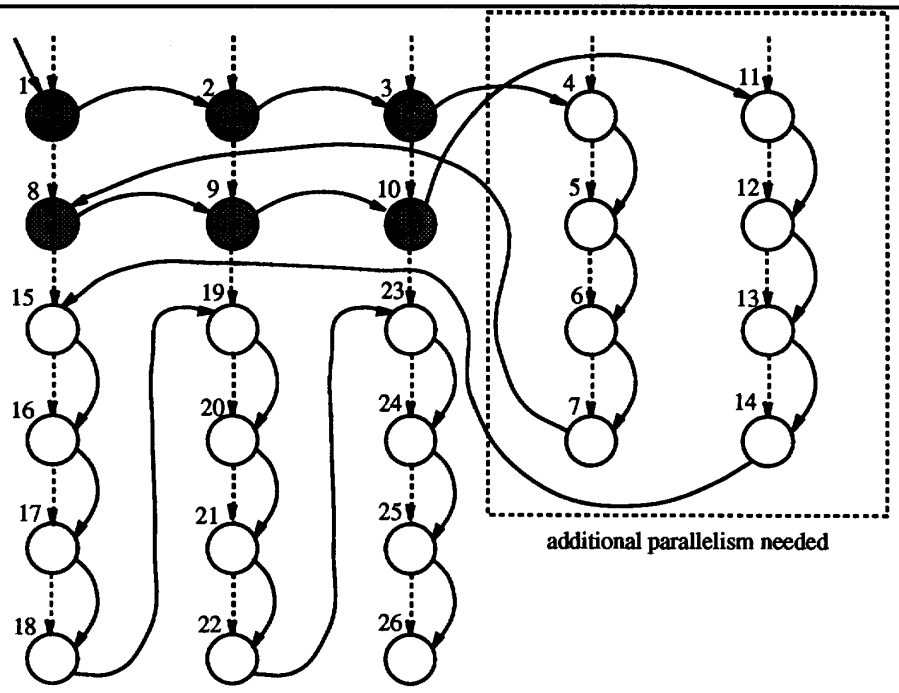

Figure 4. An execution sequence of a grain with a distribution of split-phase loads that requires additional parallelism ($P = 3, f = 1/3$, and $\bar{l}_{sp}/\tau = 7$).

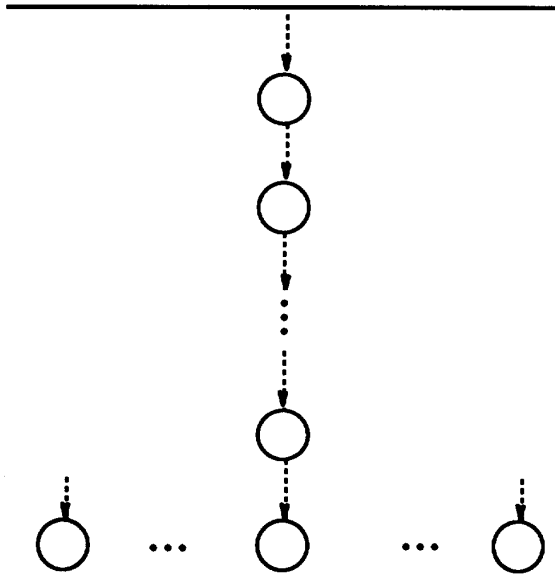

Figure 5. An example of a "reverse-T" graph.

5. Partitioning Strategy

In the previous section, a minimum granularity required to keep a processor pipeline full was discussed. In this section, an algorithm that constructs such a partition is presented. In order to illustrate the partitioning algorithm, consider an arbitrary acyclic dataflow graph $G \equiv G(N, A)$,[4] where N represents the vertices each containing a thread and A represents the partial ordering $\rightarrow$ between the threads. Thus, G represents a partially ordered graph of fully ordered instruction. In order to locate the various parallel threads in a program graph, the algorithm uses a variant of topological sorting.

The objective is to identify as many parallel threads as possible for merging. This is done by processing the program graph layer by layer where threads in each layer can be executed in parallel and the layers are linearly ordered with respect to their precedence constraints. The

[4] Before a directed graph G is processed, it is transformed into an acyclic graph. Such a transformation is done by a depth-first traversal marking all backward-pointing arcs which close loops [12]. This is necessary only for the partitioning process—once the partitioning is done, the actual program graph is processed.

algorithm starts the partitioning process by labeling the root vertex as layer l (e.g., l=1).[5] This node is also considered as a *parent* vertex. In succeeding steps, all edges (arcs) originating from the parent vertices (e.g., root vertex) in layer l are removed. We then find all the vertices with zero in-degrees and label these vertices as layer l+1. These vertices are *siblings* of the parent vertices. At this point, note the following observations: First, the parent vertex may belong to a subgraph that has already been formed or by itself represents a subgraph. For example, the root vertex itself represents a subgraph since there are no other parallel threads that can be combined. Second, the siblings in layer l+1 represent parallel threads and thus parallelism. On the basis of these observations, a program graph is processed recursively in a parent-sibling order between layers l and l+1.

In order to construct a subgraph of appropriate granularity, the siblings in layer l+1 are merged one-by-one. The merging continues until either the average parallelism is greater than the required parallelism of the subgraph or no more siblings exist in layer l+1. If a subgraph is formed and there remains additional siblings, the process is repeated to form a new subgraph. Once all the siblings in layer l+1 are merged into subgraphs, each subgraph represents a *compound* parent vertex for the threads in layer l+2. The algorithm continues until all the threads in G have been merged into subgraphs. The general outline of the partitioning algorithm is given as follows:

Partitioning Algorithm:

- Step 1: Label the root vertex as l.
- Step 2: Repeat Steps 3-7 until all the vertices in G are merged into subgraphs.
- Step 3: Remove all arcs originating from the subgraph(s) in layer l.
- Step 4: Find all vertices with zero in-degrees. In other words, find all the sibling of the parent subgraph(s) and label them l+1.
- Step 5: In layer l+1 merge a sibling to form a subgraph and recalculate f to reflect the new fraction of split-phase loads in the subgraph.
- Step 6: Determine P_{AVG} of the subgraph. If P_{AVG} is less than P_{REQ} of the subgraph and siblings not yet merged still exist, repeat Step 5; otherwise, goto step 7.
- Step 7: If siblings that have not been merged still exist, goto Step 5 and start the constructing of a new subgraph, otherwise, increment l and goto step 3.

Note that the partitioning algorithm may not always construct a subgraph with sufficient parallelism. The merging process may terminate before the parallelism requirement has been met due to lack of threads in a given layer. This is an indication of insufficient parallelism in certain parts of the program graph rather than the inability of the algorithm to find such parallelism (e.g., a root vertex represents a subgraph that is inherently sequential). It is also important to mention that the proposed algorithm makes no attempt to merge two sequential threads. The reasons are two-fold: First, merging two partially ordered threads provide no additional parallelism to the subgraph. Second, for scheduling purposes it is important to distinguish between a parent subgraph and its sibling subgraphs. Once a parent is merged with one of its siblings, it may introduce unnecessary sequential ordering between siblings.

[5] If no root vertex exists, a *dummy* root vertex is introduced.

6. Conclusion

In this paper, we presented a partitioning strategy for multithreaded dataflow computers. The proposed scheme is based on the minimum granularity required to mask latencies due to split-phase loads. In order to quantify such a measure, an expression was developed that reflects the amount of parallelism required for a grain, given the programming behavior and characteristics of the processor pipeline. This measure was then utilized in the proposed algorithm to construct a partition that best matches the program characteristics with the capability of a given multithreaded dataflow computer.

Once a program graph has been partitioned, the next task is the scheduling of the subgraphs to processing elements. A number of methods have been proposed in the literature can be used to provide an efficient schedule [13, 18]. In particular, the scheme proposed in [13] that allocates dataflow graphs to dataflow multiprocessors can be used to schedule subgraphs. The importance of our partitioning strategy is that it provides the suitable granularity that best matches the programming behavior and the characteristics of a processor pipeline.

References

[1] Arvind and Culler, D. E., "Dataflow Architectures," *Annual Review in Computer Science*, 1986, Vol. 1, pp. 225-253.

[2] Arvind and Iannucci, R. A., "A Critique of Multiprocessing von Neumann Style," *Proc. 10th Annual Symposium Computer Architecture*, June 1983, pp. 426-436.

[3] Arvind, Culler, D. E., and Ekanadham, K., "The Price of Fine-Grain Asynchronous Parallelism: An Analysis of Dataflow Methods," *Proc. CONPAR 88*, Sept. 1988, pp. 541-555.

[4] Arvind, Nikhil, R. S., and Pingali, K. K., "I-structures: Data Structures for Parallel Computing," *Proceedings of the Workshop on Graph Reduction*, Los Alamos, NM, 1986.

[5] Culler, D. E. and Papadopoulos, G. M., "The Explicit Token Store," *Journal of Parallel and Distributed Computing*, Vol. 10, No. 1, December 1990, pp. 289-308.

[6] Culler, D. E. *et al.*, "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstracted Machine," *4th Int'l. Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[7] Eager, D. L. *et al.*, "Speedup Versus Efficiency in Parallel Systems," *IEEE Transaction on Computers*, Vol. 38, No. 3, March 1989, pp. 408-423.

[8] Gaudiot, J.-L. and Bic, L., "Advanced Topics in Data-Flow Computing," Prentice Hall, 1991.

[9] Grafe, V. G. and Hoch, J. E., "The Epsilon-2 Multiprocessor System," *Journal of Parallel and Distributed Computing*, 10, 1990, pp. 309-318.

[10] Hoch, J. E. *et al.*, "Compile-time Partitioning of a Non-strict Languages into Sequential Threads," *Proceedings of the 3rd Symposium on Parallel and Distributed Processing*, Dec. 1991.

[11] Iannucci, "Towards a Dataflow/von Neumann Hybrid Architecture," *Proc. 15th Annual Int'l. Symposium on Computer Architecture*, 1988, pp. 131-140.

[12] Jordan, H. F., "Performance Measurement on HEP - A Pipelined MIMD Computer," *Proc. 10th Annual Int'l. Symposium on Computer Architecture*, June 1983.

[13] Lee, B., Hurson, A. R., and Feng, T. Y. "A Vertically Layered Allocation Scheme for Dataflow Computers," *Journal of Parallel and Distributed Computing*, Vol. 11, 1991. pp. 175-187.

[14] Nikhil, R. S. and Arvind, "Can Dataflow Subsume von Neumann Computing?," *Proc. 16th Annual Int'l. Symposium on Computer Architecture*, 1989, pp. 262-272.

[15] Papadopoulos, G. M. and Traub, K. R., "Multithreading: A Revisionist View of Dataflow Architectures," *Proc. 18th Annual Int'l. Symposium on Computer Architecture*, 1991, pp. 342-351.

[16] Saavedra-Barrera, R. H. *et al.*, "Analysis of Multithreaded Architectures for Parallel Computing," *2nd Annual ACM Symposium on Parllel Algorithms and Architectures*, July 1990.

[17] Sakai, S. *et al.*, "An Architecture of a Dataflow Single Chip Processor," *Proc. 16th Annual Int'l. Symposium on Computer Architecture*, 1989, pp. 46-53.

[18] Sarkar, V. and Hennessy, J., "Partitioning Parallel Programs for Macro-Dataflow," *ACM Conf. on Lisp and Functional Programming*, 1986, pp. 202-211.

[19] Schauser, K. E. *et al.*, "Compiler-Controlled Multithreading for Lenient Parallel Languages," *5th ACM Conference on Functional Programming Languages and Computer Architecture*, August 1991, pp. 50-72.

[20] Traub, K. R., "Global Analysis for Partitioning Non-Strict Programs into Sequential Threads," *Proceedings of the ACM on LISP and Functional Programming*, June 1992, pp. 324-334.

[21] Yamaguchi, Y. *et al.*, "An Architectural Design of a Highly Parallel Dataflow Machine," *Proc. IFIP Congress 1989*, pp. 1155-1160.