The RMIT Data Flow Computer: A Hybrid Architecture

https://academic.oup.com/comjnl/article/33/3/230/365140

This paper examines the two conventional models of data-flow machines, static and dynamic. The advantages and disadvantages of each scheme are described. A hybrid architecture is presented which gains the advantages of both. This architecture has been incorporated into a new data-flow machine built at the Royal Melbourne Institute of Technology. Some simulations are presented which illustrate the advantages of the hybrid approach.

1. INTRODUCTION

Data-flow machines are multiprocessors which execute parallel program graphs rather than sequential programs. The order of execution of the nodes in the graph (or instructions) is determined by the availability of their operands rather than the strict sequencing of instructions in a von Neumann machine. Consequently the program statements are executed in a non-deterministic manner, and concurrency is obtained if more than one node executes at the same time.

There are currently two main classifications for data-flow architectures, static and dynamic. The static scheme was first proposed by Dennis, and has been used by various research architectures such as TI’s DDP and the LAU architecture. The dynamic scheme is used in Arvind’s research group at MIT, at Manchester University, the DDM architecture and the EDFG system. In the static data-flow model only one token (or instruction operand) is allowed on a program arc at any time. In the dynamic model many tokens are allowed on arcs, and their order is determined by special tag fields. A data flow machine being built at Royal Melbourne Institute of Technology around an architecture originally devised by Egan at Manchester University is a hybrid between the classic static and dynamic architectures. The hybrid attracts the advantages of both schemes and avoids most of their disadvantages.

2. DYNAMIC AND STATIC ARCHITECTURES

In the static data-flow architecture program instructions, or nodes, execute when all of the operands, or tokens, appear at their inputs (e.g. two input nodes require two tokens before the node executes). The ordering of the tokens on the arcs can be guaranteed by constructing the data-flow graph so that only one token can exist on any input at a time. Because a node can only have one input waiting there is no advantage in placing the node on more than one processor, as it cannot execute more than one computation at the same time. Thus, program graphs are statically allocated to the real processors, and the destinations are statically coded in the graph. Static architectures work well for processing “sets” of data which are streamed through the graph, because the data is output in the same order that it is entered. For any single “data set” the amount of parallelism is basically limited to the width of the program graph. If data values are fed back into the graph then the number of nodes active concurrently may exceed the graph width. More concurrency can be extracted when multiple data sets are introduced because a processor can start work on the next piece of data when it has completed a computation.

The hardware required by the static model is quite simple. Because an instruction can only have one outstanding operand, the matching unit only needs to retain which operand is waiting and the valve. When the other operand arrives the match can occur. The main disadvantages of the static architectures are that it can be difficult to construct the graph to guarantee one- token per arc restriction. The one-token per arc requirement can be achieved by balancing the arc lengths in the program graph and inserting sequencing code in conditional expressions. The amount of parallelism in a static architecture is severely restricted because the only way to achieve more concurrency than that dictated by the width of the program graph is to introduce multiple data sets. Another problem with those machines which statically allocate the graph to processing elements is that because nodes are permanently assigned to processors, a particular processor may attract an unequal share of the work load. There is no possibility of dynamically distributing the work load across the machine. This problem is not apparent on all static machines. The parallelism on these machines is further restricted because a sending node cannot transmit a result until the receiving node is ready to accept a new token. This has the effect of halving the number of active processors, because only alternate levels of the graph can be active at any time. A modified form of the static architecture has been used in the NEC dataflow chip. In this machine a small number of tokens may be queued on the inputs of nodes. The queues solve the problem of a sending node being unable to compute a result whilst a destination node is busy. However, because the queues are quite small, they can overflow and block computation as in the static scheme.

In the dynamic model every token holds a tag field, which determines the sequence number of the token. An instruction may have multiple tokens waiting for matching on an arc, and they are only removed and the instruction executed, when the token with the correct tag value arrives at the other arc. The addition of tags alone does not increase the concurrency in a data-flow graph as it is still limited to the width of the graph and the number of concurrent data sets. However, because a node can have more than one token waiting at a time, it may be sensible to allow more than one invocation of the node to execute at a time, each with a different set of data. Thus if two sets of tokens arrive for a node in a dynamic architecture and can match simultaneously then two processing elements can be occupied. In a static data-flow machine they would be forced to execute sequentially. This feature is called loop-unfolding, and can increase the amount of parallelism in a data-flow machine significantly above the width of the graph. In fact, loop unfolding may produce so much parallelism that it exceeds the number of real processors and much of the advantage is lost. Loop unfolding also helps to distribute the work load because a given node may execute on more than one processor. When a token is produced the destination processor is calculated dynamically, usually by applying a hashing function to the tag field. Nodes can only execute if there is a copy of the graph, or relevant part, held, at the destination processor. Thus it is necessary to duplicate the program graph across the processors or a subset of processors.

The addition of tags to tokens makes matching much more complex and time consuming. Thus, the cost of the large amounts of parallelism is a much more complex matching unit, which can increase the cost of each processing element over the static scheme. The amount of network traffic may also increase as the token need to carry tag information. A major disadvantage of the dynamic model is that data may be output in any order and must be resorted if ordering is important. This resorting may add a considerable overhead to the computation. Thus in cases when pipelining is the most important form of parallelism and there is little loop unfolding or feedback, as in many real world control problems, the dynamic architectures have the added complexities of tagging and untagging, the increased network traffic and the resorting of data. For example, each iteration of a loop in a dynamic machine must include tag generation code, even if the loop has a data dependency in it that forbids loop unfolding. This overhead is not present in the static model.

The architecture described in the next section allows efficient execution of pipelined data sets, without the disadvantaged of one-token per arc, and the high degree of parallelism obtained in dynamic schemes, without the necessity of always tagging data.

3. THE HYBRID ARCHITECTURE

Combining Static and Dynamic Architectures

The hybrid architecture allows more than one token per arc by having two different modes of operation. In the first, tokens may be queued on an arc for a particular node, and are executed in the order that they were placed on the queue. If there is a queue of tokens on one arc of a two input node, and a token arrives at the other input then the head of the queue is removed and the node executed. This arrangement is similar to the classic static architecture, but allows more than one token per arc by providing queues. The queues must be large enough for the probability of queue overflow to be small. Thus, nodes should never be blocked from transmitting results because a destination queue is full. In this mode tokens do not carry any tagging information. In the second mode, tokens may be tagged and “heaped” at a particular input until a token of a matching tag arrives at the other input. This is identical to the dynamic scheme. In this mode tokens must contain tag values which distinguish the different data sets.

Furthermore, tokens with the same tag value may be queued on an arc. Thus, it is possible for a computation to generate a new tag value and place it on many tokens. This provides a combination of a queued static model and the purely dynamic scheme.

The allocation of the graph to the processing elements is done in two ways. It can either be allocated to processors with any given node appearing on only one processor, or the node can be copied and the copies allocated to multiple processors. If the machine is executing a section of code using the staticqueued model then only one copy is required. In this case the nodes are allocated to processors using a uniform random probability distribution. Whilst this may not achieve optimal graph placement, it is much simpler than calculating optimal or even suboptimal placements. This random placement has been shown to perform well, especially if the graph is large. If the dynamic tagged scheme is used it may be necessary to duplicate the graph on many processors to achieve the required concurrency. When a token is created as a result of a node firing, the destination address is calculated by hashing the tag field. This has the effect of randomly distributing the work to the processors and balancing the workload.

Providing both data-flow models achieves the advantages of both. If the programmer wishes to flow “sets” of data through the graph then no tags will be added to the tokens. There are a number of advantages with this approach. First, the cost of maintaining and manipulating tags is removed. Thus, loops which have data dependencies in them, which prevent loop unfolding, may be coded without tag generation logic. The queues preserve the loop ordering even if the data has some initial tag value when it enters the loop. Second, the network traffic is reduced because the tag need not be transmitted. Third, the matching function becomes relatively easy, and even a simple implementation of matching store yields predictable, scalable and acceptable performance. Fourth, unlike the dynamic model, the data does not need to be resorted when it emerges from the graph, and may be used directly. Because the architecture allows more than one token per arc it is not necessary to balance arc lengths in the graph as in the pure static case. However, some additional code may still be necessary to ensure correct sequencing of conditional expressions. Also no interlocks are necessary between the sending node and the destination node. The addition of queues allows a sending processor to compute results and send them even if the destination processor is busy and has a queue of pending tokens. A further advantage of the queued static model is that it makes the implementation of ordered token streams easier, because the elements of the stream remain in their intended order. The implementation of streams on dynamic machines is much harder because the stream order can only be maintained by tagging each stream element with a unique value, and including code which examines the tag before the stream is processed. Unlike the static model, the hybrid architecture allows token tagging and loop unfolding if it is required.

data-flow machine – потоковая (вычислительная) машина