CHAPTER 5

Dataflow and Multithreading

A Preliminary Architecture for a Basic Data-Flow Processor 315
J. B. Dennis and D. P. Misunas

Executing a Program on the MIT Tagged-Token Dataflow Architecture 323
Arvind and R. S. Nikhil

Architecture and Applications of the HEP Multiprocessor Computer System 342
B. J. Smith

Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor 350
D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm

5.1 Dataflow Computers

Most computers today are based on a *sequential execution model*, where instructions of a program are executed in an implied (sequential) order. This model closely resembles the way processors were implemented in their early days, where program instructions were processed one at a time. The sequential execution model also led to instruction sets and programming languages with sequential execution semantics. Today, almost all programs are written in programming languages and compiled to instruction sets that have an implied execution order.

To maximize parallelism, ordering constraints must be minimized. Ideally, the execution of an instruction should be constrained only by dependence relationships and not by any other ordering constraints. Therefore, to achieve high performance in the sequential execution model, means are needed to discover and extract parallelism from the serial specification, and to execute these operations on parallel hardware. Dataflow is a computing model that has been proposed to overcome the performance limitations of the traditional sequential computing model. Many of the limitations of the traditional (sequential) model have to do with exposing and extracting parallelism hidden as a result of serial program semantics, exploiting it on parallel hardware, and maintaining the implied program ordering.

The idea of dataflow is to have a computing engine that has no ordering constraints on operation execution other than data (and control) dependences that exist in the computation. Computation is represented as a dataflow graph, which captures the data dependence relationships between the operations. Control dependences are treated just like data dependences (by using a SWITCH operator to gate data values using a Boolean condition). By not imposing a "total order" on program execution, opportunities for parallelism are not artificially constrained. In the absence of a total order, other rules are needed to determine the order in which operations should be executed. The dataflow model uses a data-driven (or dependence-driven) rule governing the execution of operations.

A dataflow program consists of blocks of instructions, or *code blocks*, whose execution is governed by the dataflow "firing" rule. The dynamic state of a code block, analogous to a stack frame, resides in a *token store*. When an operation executes, it "wakes up" other operations in the token store that depend on it, and when the operands of an operation are ready, it can be "fired." Different types of dataflow computers use different means to wake up operations. The classic *tagged-token dataflow architecture* (TTDA), which is described in the paper by Arvind and Nikhil, uses an associative token store [3]. When an operation executes, it creates one or two result tokens, each with a tag indicating the instance of an instruction that needs the token. The token store is searched (associatively) to see if another token with the same tag is present. If such a match is found, then the new instruction becomes ready to execute. Otherwise, the token is left

in the token store. Because token stores can be large, associative searching is implemented via hashing [9,14]. A more recent proposal, the *explicit token store* (ETS) dataflow machine eliminates the need to do an associative search in the token store [13] by mapping dynamic instances of code blocks to a linear (addressable) memory; a token can be forwarded directly to the consuming instructions at known memory addresses.

Even though the dataflow execution model eliminates artificial ordering constraints, conventional programming languages have ordering constraints that might be carried through by the translation process, just as a (conventional) program is compiled into a dataflow executable. To overcome this problem, dataflow researchers have proposed dataflow languages such as Id [2], which are devoid of artificial ordering constraints.

A dataflow computing system (language, compiler, instruction set, and execution hardware) allows parallelism to be exposed and exploited to its fullest. However, the abundance of parallelism is not necessarily a good thing: Means must exist to deal with this parallelism [4]. These means include large token stores to handle all the operations that are waiting to execute, means to manage the token store, and means to schedule the (hundreds or thousands) of operations that are ready to execute so that they can execute on the limited amount of available hardware. In many cases, constraining the amount of parallelism actually allows for more efficient solutions!

5. 2 Multithreaded Computers

Multithreaded computers are computers that can execute instructions from more than one execution thread. The term *multithreaded*, however, is a heavily overloaded term in computer science in general, and in computer architecture in particular. A thread is a piece of code together with a state (registers and memory); instructions from a thread modify this state as they execute. In its most common usage, multiple threads imply multiple different programs (or processes). However, computer architects, especially processor architects, are rarely concerned with computer designs that improve the overall throughput when processing a workload consisting of several programs. Rather, they are concerned about improving the time that it takes to execute a single program. In this context, multiple threads refer to different (generally independent) portions of a single program; for example, the different iterations of a loop. In either case, a multithreaded processor can also be viewed as an alternate way of implementing a parallel processor (albeit a small-scale one).

To support multiple threads, multithreaded processors have to provide hardware resources to implement the thread context, that is, the state for each thread. Recall that state for a thread generally resides in two name spaces: registers and memory. Multithreaded processors typically provide a separate register file for each thread. However, the (physical) memory is shared amongst the different threads just as it would be in a nonmultithreaded processor running a multiprogrammed workload.

The most common use of multithreading is to improve the utilization of a resource. For example, when a processor is idling while waiting for the result of a long-latency memory operation, processor utilization (and overall processing throughput) can be improved if the processor switches execution to a different thread, and instructions from that thread are executed during cycles that would otherwise have been idle. By performing work during otherwise dead cycles, the processor is able to tolerate long latencies. Accordingly, many also view multithreading as a means for tolerating long latencies. However, the beneficial effect of tolerating the latency on the execution time of a single program only shows up if the operations that are overlapped with the long-latency operation are from the same program; that is, the multiple threads are threads of the same program.

Multithreading has a cost overhead, in that additional hardware must be provided to handle state from multiple threads. In addition, it has performance overheads related to thread switching that can impact single-thread performance. First, switching to a different thread can incur an overhead, depending on how the register contexts of the different threads are implemented. The impact of this overhead can be reduced by having more hardware that allows switches between threads to take place without any time penalty. Second, once a thread is switched out, it might not be restarted immediately after its long-latency operation is finished, resulting in increased execution time for the thread (versus a scenario where the thread was not switched out, and was able to resume execution immediately on completion of its long-latency operation). The tradeoff between thread switching and single-thread performance has been central to the design of multithreaded processors over the decades, and several different switching policies have been proposed.

One commonly used thread-switching policy is to switch to a different thread on every cycle. This policy is relatively straightforward to implement. Moreover,

if there are as many threads as there are stages in the processing pipeline, there is only one instruction from any thread in the processing pipeline. This obviates the need for hardware dedicated to overcoming the problems of overlapping the processing of instructions from a single thread (for example, dependence checking logic, branch prediction hardware, etc.). The drawback of this approach, however, is that the performance of a single thread is sacrificed.

Another policy is to keep processing a single thread until a long-latency operation (such as a cache miss) is encountered and switch to a new thread at that point [1]. This degrades single-thread performance less than the previous proposal (the issue of restarting a thread on completion of its long-latency operation still exists) but is more involved. There have been several other proposals for thread-switching policies.

Most of the multithreading policies studied until the 1990s were aimed at processors that can execute only a single operation in a clock cycle. For these processors, it made sense only to switch threads after a clock cycle (or after multiple cycles), because at least one operation should be executed from a thread once it has been selected, and this took at least one clock cycle. The advent of multiple-issue long instruction word (LIW) and superscalar processors in the late 1980s and 1990s caused researchers to rethink the notion of thread switching. Multiple-issue processors allow multiple operations to be launched in every clock cycle, and not all of these issue slots might be usable by a given thread in a given clock cycle. In this case, the possibility of thread switching at a granularity of less than a clock cycle (i.e., within a clock cycle) arises: Issue slots within a clock cycle that are unused by the primary thread could be used for operations from another thread. This could potentially be a win-win situation: A single thread continues its execution unperturbed, and other threads get to use execution slots that would otherwise have not been used. Of course, this assumes that the hardware and software complexity is not increased, and that the different threads do not cause interfere in other resources of the machine (e.g., caches, memory systems, and interconnects). Executing operations from multiple threads in a single clock cycle is called *simultaneous multithreading*.

With a simultaneous-multithreading policy allowing multithreading without sacrificing single-thread performance, and with semiconductor technology allowing sufficient hardware resources to build (register) contexts for multiple threads on a single chip, computer architects are considering multithreading support even in general-purpose processors.

The reader may ask: Dataflow and multithreading appear to be very different concepts, so why are we lumping them together in a single chapter? It is true that dataflow and multithreading are different concepts. However, a lot of multithreading research has been carried out within the dataflow context for reasons we discuss next.

The objective of dataflow is to expose and exploit parallelism, and this requires parallel hardware. Parallel hardware introduces latencies that may not exist in non-parallel hardware, such as latencies resulting from traversing interconnection structures, and due to distributing resources such as memory. In the data-driven execution model, where an operation wakes up when its predecessor operation communicates a value to it, latencies introduced by parallel hardware aggravate the interoperation communication latency, and we have to tolerate this latency by overlapping this communication with other (independent) computations. Moreover, a code block of a dataflow program can be viewed as a "thread."[1] Because the dynamic execution of a dataflow program results in several such "threads" being active at the same time, it is natural to design machines that can handle multiple such threads.

5. 3 Discussion of Included Papers

Next, we present a brief discussion of the papers reprinted in this chapter.

5.3.1 Dennis and Misunas's "A Preliminary Architecture for a Basic Data-Flow Processor" [6]

The paper by Dennis and Misunas was the first paper to describe the architecture of an entire processor based on data-driven execution principles. Here, the program is a dataflow program, and the proposal is for a processor that executes a dataflow program. This was not the earliest description of *dataflow execution* principles, however: Tomasulo's algorithm [18], implemented in the IBM 360/91, applied dataflow principles (which were not called dataflow at the time) to come up with hardware that allowed data-driven execution of a small number of instructions. Note, however, that this

[1] If we characterize machines on a multithreading spectrum based on the length of the threads that they support, pure classical dataflow (e.g., MIT TTDA) has threads with a length of one operation. It is possible to think of threads of a dataflow machine consisting of several operations that are executed together (e.g., MIT Monsoon).

group of instructions was not represented as a dataflow program. Rather, they were created dynamically by sequencing through a more traditional control flow program.

5.3.2 Arvind and Nikhil's "Executing a Program on the MIT TTDA" [3]

The early work by Dennis and his colleagues provided the spark for a significant amount of research on dataflow processors at several institutions worldwide in addition to the continuing research at MIT. The paper by Arvind and Nikhil presents a snapshot of the work in dataflow at MIT (as well as at other places) circa 1987. In addition to describing the architecture of the MIT tagged-token dataflow architecture (TTDA) computer, it discusses several other aspects of dataflow computing. The paper starts out by describing the dataflow language Id. As mentioned earlier, dataflow proponents believe that more conventional languages introduce ordering constraints on operations (perhaps unintentionally) that can only serve to obscure parallelism. The paper then describes how Id programs can be compiled into dataflow graphs and an architecture for interpreting (i.e., executing) programs represented as dataflow graphs.

5.3.3 Smith's "Architecture and Applications of the HEP Multiprocessor" [15]

The paper by Smith describes aspects of the heterogeneous element processor (HEP) built by Denelcor. The HEP was an early example of a (publicly documented) machine that used multithreaded execution for processing the "main" instruction stream(s). Earlier machines used multithreading for some aspects of processing (e.g., the input/output units of the MIT TX-2 and the peripheral processors of the CDC 6600). Other multithreaded machines were proposed (and some were even built) prior to HEP, but documentation about such machines is not readily available.

The paper also describes of a novel synchronization mechanism (full-empty bits) and of Fortran extensions that would allow an application to be compiled in order to make use of the multithreaded architecture and the synchronization features of the HEP.

5.3.4 Tullsen et al.'s "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor" [20]

Multithreaded architectures that switched threads every cycle (or every few cycles) were not seriously considered by architects of mainstream general-purpose processors, because they were viewed as sacrificing single-thread latency in favor of multiple-thread throughput. And limited chip resources called into question the utility of expending hardware to support multiple thread contexts, especially if single-thread performance was going to be compromised.

The advent of wide-issue machines resulted in a new form of multithreading, one in which operations from multiple threads were issued *in the same cycle*. This form of multithreading could improve multiple-thread throughput without compromising single-thread latency (assuming that interthread interference did not degrade single-thread latency; e.g., resulting from increased cache misses). There were several proposals for this type of multithreading in the 1990s, both within the context of statically scheduled wide-issue machines [12, 21] as well as within the context of dynamically scheduled wide-issue machines [5, 11, 19, 20, 22].

The paper by Tullsen et al. discusses several issues that need to be considered when implementing simultaneous multithreading on top of a dynamically scheduled superscalar processor. They show how the basic mechanisms of a superscalar processor could be extended to allow multiple simultaneous processing of instructions from multiple threads. Though the threads in this proposal are different programs, it is easy to see how they could be different parts of the same program.

5.4 Looking Ahead

Despite its promise, the dataflow computing model has not been adopted in mainstream processor design. There are several reasons for this. First, it is not practical to give up traditional serial programming languages and instruction sets entirely. Second, the main source of the "power" of the dataflow model (no artificial ordering constraints, and consequently no well-defined, reproducible state) results in several practical problems. Perhaps the most important of these is the difficulty of debugging.[2] A discussion of some of these limitations can be found in a paper by Gajski, et al. [8]. The computer architect is then faced with the question: Can we apply some of the principles of dataflow computing to more practical computing situations? The answer to this question so far has been a resounding *yes*. The dataflow paradigm continues to serve as the inspiration for many innovations in ILP processors, whose goal is to achieve dataflow-like execution, but

[2] Parallel machines without artificial ordering constraints (i.e., synchronization) have very similar problems.

starting out with an imperative language and a practical microarchitecture. Modern dynamically scheduled superscalar processors are one example of this influence, called *micro dataflow* machines by some. In the future, the dataflow model (coupled with suitable speculation techniques) is likely to continue to be the inspiration for many upcoming innovations in exploiting parallelism. Future innovators are likely to benefit from a thorough understanding of dataflow concepts and their evolution. And the dataflow computing model is likely to evolve to include different forms of speculation, including speculation that even removes the one ordering constraint in the dataflow model: data dependences.

Future processors are also likely to be influenced by multithreading techniques. Recall that one way of viewing a multithreaded processor is an implementation of a parallel processor (with several resources that are shared). As technology allows "parallel processing" capability to be put on a single chip, different ways of incorporating "parallel processing" functionality need to be considered. Multithreading techniques that are likely to succeed are techniques that do not hinder single-thread latency. Ideally, multithreading might even be used to *improve* single-program performance rather than simply be a means for improving multiple-thread throughput. This can be done using *thread-level speculation*, as has been proposed in several recent research projects [7, 10, 16, 17].

5.5 References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 104–114, May 1990.

[2] Arvind, K. Gostelow, and W. Plouffe, "An asynchronous programming language and computing machine," Tech. Rep. TR-114a, Department of Information and Computer Science, University of California, Irvine, Dec. 1978.

[3] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.

[4] D. E. Culler and Arvind, "Resource requirements of dataflow programs," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 141–150, May 1998.

[5] G. E. Daddis, Jr. and H. C. Torng, "The concurrent execution of multiple instruction streams on superscalar processors," *International Conference on Parallel Processing*, pp. I:76–83, Aug. 1991.

[6] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic dataflow processor," *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pp. 126–132, Dec. 1974.

[7] P. K. Dubey, K. O'Brien, K. O'Brien, and C. Barton, "Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading," *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pp. 109–121, June 1995.

[8] D. Gajski, D. Padua, D. Kuck, and R. Kuhn, "A second opinion on data flow machines and languages," *IEEE Computer*, pp. 58–69, Feb. 1982.

[9] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, 1985.

[10] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 58–69, Oct. 1998.

[11] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An elementary processor architecture with simultaneous instruction issuing from multiple threads," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 136–145, May 1992.

[12] S. W. Keckler and W. J. Dally, "Processor coupling: Integrating compile time and runtime scheduling for parallelism," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Queensland, Australia, pp. 202–213, 1992.

[13] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 82–91, May 1990.

[14] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi, "Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 226–234, June 1986.

[15] B. Smith, "Architecture and applications of the HEP multiprocessor computer system," *Proceedings of the International Society for Optical Engineering*, pp. 241–248, 1982.

[16] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, June 1995.

[17] J. G. Steffan and T. C. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[18] R. M. Tomasulo "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, pp. 25–33, Jan. 1967.

[19] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 392–403, June 1995.

[20] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 191–202, May 1996.

[21] A. Wolfe and J. P. Shen, "A variable instruction stream extension to the VLIW architecture," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–14, Apr. 1991.

[22] W. Yamamoto and M. Nemirovsky, "Increasing superscalar performances through multistreaming," *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pp. 49–58, June 1995.