University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Advances in the Dataflow Computational Model

Walid A. Najjar†
Edward A. Lee‡
Guang R. Gao

CAPSL Technical Memo 29
April 1, 1999

† Department of Computer Science, Colorado State University
‡ Department of Electrical Engineering and Computer Science, Univesrity of California, Berkeley

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

The dataflow program graph execution model, or dataflow for short, is an alternative to the stored- program (von Neumann) execution model. Because it relies on a graph representation of programs, the strengths of the dataflow model are very much the complements of those of the stored-program one. In the last thirty or so years since it was proposed, the dataflow model of computation has been us ed and developed in very many areas of computing research: from programming languages to processor design,and from signal processing to reconfigurable computing. This paper is a review of the current *state-of-the-art* in the applications of the dataflow model of computation. It focuses on three areas: multithreaded computing, signal processing and reconfigurable computing.

Contents

1 Introduction

Dataflow is a sound, simple, and powerful model of parallel computation. In dataflow programming and architectures there is no notion of a single point or locus of control — nothing corresponding to the program counter of a conventional sequential computer. The dataflow model describes computation in terms of locally controlled events; each event corresponds to the "firing" of an actor. An actor can be a single instruction, or a sequence of instructions (per se, the dataflow model does not imply a limit on the size or complexity of actors). An actor fires when all the inputs it requires are available. In a dataflow execution, many actors may be ready to fire simultaneously (locally controlled by their operand availability), and thus these actors represent many asynchronous concurrent computation events.

Work on dataflow computer architecture emerged in the early 1970s with the use of dataflow program graphs to represent and exploit the parallelism in programs [25, 27]. In a Dennis dataflow graph [27], operations are specified by actors that are enabled just when all actors that produce required data have completed their execution. The dependence relationships between pairs of actors are defined by the arcs of a graph, which maybe thought of as conveying results of an actor to successor actors, and by the firing rules, which specify exactly what data is required for an actor to fire. Decision and control actors may be included to represent conditional expressions and iterations. They alter the routing of tokens to effect data-driven control. Data structures may be constructed and accessed by appropriate dataflow actors.

Also in the early 1970s, a rather different style of dataflow models emerged [60]. Called Kahn process networks, this model replaces actors with sequential processes. These processes communicate by sending messages along channels that conceptually consist of unbounded FIFO queues. Kahn dataflow can be viewed as a generalization of Dennis dataflow [73]. While Dennis dataflow was originally applied to computer architecture design, Kahn dataflow was used by concurrency theorists for modeling concurrent software. Multithreaded architectures, with dataflow roots, use a style of dataflow that can be viewed as having elements of both.

In computer architecture, dataflow program graphs were originally used as a machine-level program representation. Two forms of dataflow architecture have become known: In a static architecture, the arc connecting one instruction to another can contain only a single result value (a token) from the source instruction. In this scheme there can be only one instance of a dataflow actor in execution at any time. In a dynamic dataflow architecture, tags are conceptually or actually associated with tokens so that tokens associated with different activations of an actor may be distinguished. This enables arcs to simultaneously carry multiple tokens, thereby exposing more data parallelism.

Because of the inherently parallel nature of dataflow execution, dataflow computers provide an efficient and elegant solution to the two fundamental problems of von Neumann computers: the memory latency and synchronization overhead, as described by Arvind and Iannucci [6]. The ability of the dataflow model to tolerate latency, by switching dynamically between ready computation threads, and to support low overhead distributed synchronization in hardware, has made it the candidate of choice for what has later been called "latency tolerant" architectures.

A history of the evolution of the dataflow computation model(s) and the variety of architecture and computational models that have been inspired by it is beyond the scope of this paper. The reader is referred to, among many others, the following texts [63, 55, 41].

As a computation model, the dataflow approach has had influence on many areas of computer science and engineering research. Examples include programming languages, processor design, multithreaded architectures, parallel compilation, high-level logic design, signal processing, distributed computing, and programming of systolic and reconfigurable processors. This paper reviews the recent developments in three of these areas: multithreaded execution, signal processing, and reconfigurable computing. The first two areas are relatively mature and have generated a substantial body of research results and commercial application. The last area is relatively young and is still in an emerging state. The first two sections review the state of the art of the dataflow computational model in multithreaded architectures and signal processing, respectively. The third section proposes the dataflow model as a programming model for reconfigurable computing.

2 Dataflow and Multithreading Execution

Issues and challenges for multithreaded computer architecture suitable for use in parallel computers for general purpose computation are the subject of intensive debate. These issues and challenges depend heavily on the choice of an appropriate program execution model that will affect the programming model, the organization of the underlying system as well as the development support of complex applications on to the proposed architectures – compilers, runtime systems, and tools. The choice of program execution models may also have a profound impact on programming methodology [33, 34].

In this section, we introduce basic concepts, characteristics and evolution of multithreaded computer architectures with dataflow origin. Our focus will be the review of issues and advances in program execution and architecture models upon which the design of these architectures are based.

In our terminology, a program execution model defines a basic low-level layer of programming abstraction of the underlying system architecture upon which the architecture model, programming model, compilation strategy, runtime system, and other software components are developed. It serves as an interface between the architecture and the software. Program execution model is in a broader (or higher) level than *instruction set architecture* (ISA) specification. An ISA usually provides a description of an instruction set for a particular machine that involves specific details of instructions such as instruction encoding and the organization of the machine registers set. In the context of this paper, a program execution model for multithreaded machines includes the following aspects: the thread model, the memory model and the synchronization.

The rest of this section is organized as follow. In Section 2.1, we briefly review the evolution of multithreaded architectures with dataflow origin. The subsequent sections will review the issues, challenges and advances in three areas: the thread models in (Section 2.2), the memory models in (Section 2.3). and the synchronization models in (Section 2.4).

2.1 The Evolution of the Hybrid Dataflow/Von-Neumann Model and Multithreading

The dataflow model and von Neumann serial control-flow model are generally viewed as two extremes of execution models on which a spectrum of architecture models can be based. The two models are in fact not orthogonal. Starting with the operational model of a pure dataflow graph, one can easily extend the model to support von Neumann style program execution. A region of actors within a dataflow graph can be grouped together as a thread to be executed sequentially under its own private program counter control, while the activation and synchronization of threads are data-driven. The new *hybrid* model is flexible in combining dataflow and control-flow evaluation, as well as in exposing parallelism at a desired level.

Such hybrid multithreaded architecture models have been proposed by a number of research groups with their origins in either static dataflow or dynamic dataflow. A number of articles have been published on multi-threaded execution and architecture models with dataflow origin and can found in a survey article by Dennis and Gao [29]. Principal projects and representative work before 1995 have been discussed in two monographs [55, 41] as well as other survey articles [61]. Below we briefly mention a few research projects on each model.

Inspired by the static dataflow model, the *McGill Dataflow Architecture Model* [39] has been proposed based on the *argument-fetching* principle [28]. The architecture departs from a direct implementation of dataflow graphs by having instructions fetch data from memory or registers instead of having instructions deposit operands (*tokens*) in "operand receivers" of successor instructions. The completion of an instruction will post an *event* (called a signal) to inform instructions that depend on the results of the instruction. This implements a modified model of dataflow computation called *dataflow signal graphs.* The architecture includes features to support efficient loop execution through *dataflow software pipelining* [37], and the support of threaded function activations. A good summary of the McGill Dataflow Architecture model and its extensions has been presented in H. Hum's Ph.D. thesis [52].

Based on his experience with the MIT dynamic (tagged-token) dataflow architecture [5], Iannucci combined dataflow ideas with sequential thread execution to define a hybrid computation model described in his Ph.D. thesis [54]. The ideas later evolved into a multithreaded architecture project at IBM Yorktown Research Center as described elsewhere [55]. The architecture includes features such as a cache memory with synchronization controls, prioritized processor ready queues and features for efficient process migration to facilitate load balancing.

The P-RISC [88] is an interesting hybrid model exploring the possibilities of constructing a multithreaded architecture around a RISC processor. The Star-T project – a successor of the Monsoon project – has defined a multiprocessor architecture using an extension of an off-the-shelf processor architecture to support fine-grain communication and scheduling of user microthreads [10]. The architecture is intended to retain the latency-hiding feature of the Monsoon split-phase global memory operations. Later development of the Start-T project can be found in [22, 4].

Several other interesting projects have been carried out elsewhere in the US [56, 97, 57, 85, 84] and in the world such as in Japan [65, 89]. For example, the RWC-1 project [95] of the Real World Computing Partnership in Japan was initiated based on the experience of working with dataflow architectures and their multithreaded extensions.

It is possible to define a thread model combining the advantages from both the static and dynamic dataflow models. An example is the EARTH model [53] which provides the full generality of threaded function invocation as in the dynamic dataflow model, while maintaining the simplicity of the static dataflow at the finer level of threads through software pipelining.

2.2 The Thread Models

Multithreaded execution models with dataflow origin provide support for fine-grain threads at two levels. For example, under the EARTH model, the first level of thread is called *threaded function invocation:* parallel function invocation forks a thread to execute the function in parallel. Note that the caller continues its own execution without waiting for the return of the forked threaded function. At a lower (finer) level, the body of a threaded function can be further partitioned (by a user or a compiler) into fibers [100]: a collection of operations that can be forked as a separate thread.

The support of such finer level of threads is a distinguishing characteristic of multithreaded architecture models with dataflow origin. Such a thread may be generated from a region of a dataflow graph derived from functional programming such as the so-called "strongly connected regions" in the EM-4 architecture and its successors [64, 65], or the superactors in the Super-Actor machine [52] based on the McGill Dataflow Architecture Model (MDFA [39]). With the recent advances in compiler technology, new algorithms and methods have been developed to perform thread partitioning for programs written in conventional programming paradigms [49, 98, 99]. A more in-depth discussion on such multithreaded compilation technology can be found in X. Tang's Ph.D. thesis [94]. Such compilers may also require the use of a cost model for the underlying multithreaded machines. This is a non-trivial task due to the features of dynamic thread scheduling and latency tolerance — which has motivated the study to develop such models [87]. The evolution of a combination of compiler advances and architecture concepts has led to the proposal of the *superstrand* execution and architecture model [80, 81].

Theobald in his Ph.D. thesis [100] has illustrated that such finer-level threads — also called *fibers* in his thesis — have played an important role for efficiently exploiting producer-consumer style fine-grain parallelism in scientific applications. He has demonstrated how to exploit a style of *dataflow software pipelining* between producers and consumers through fine-grain synchronization between fibers (discussed below in Section 2.4), and how such "pipes" can be established smoothly through threaded function invocations.

2.3 The Memory Models

Before we discuss the synchronization model, we need first to discuss the issues and advances in memory models. This is, perhaps, one of the most important areas where the evolution of such multithreaded execution and architecture models have departed from the dataflow models. The pure dataflow models do not use the conventional notion of an update-able memory. The programming models (e.g. functional or dataflow programming languages) are based on a semantic model where there are no side-effects due to execution of any operations. As can be seen from the following discussion, researchers in multithreaded execution models have advanced far beyond the pure dataflow models in developing memory models and cache memory schemes that take advantage of situations where single assignment rule can apply, while allowing normal load/store operations (with side-effects) when required without imposing unnecessary constraints for parallelism and efficiency.

Following the convention in [93], a memory model in our terminology contains two essential aspects: (1) the support of a shared space addressing and (2) the so-called *memory consistency model*. Advances have been made in addressing both issues.

Global Addressing Capability

It is reasonable to assume that part of the storage space of the system is globally addressable. Experience has shown that a shared addressing capability makes naming logically shared data much easier for the programmer because any thread can directly reference any data in the shared space and the naming of model is similar to that on a uniprocessor. This should greatly enhance the programmability for certain applications (although it is still a subject of debate). For a multi-threaded execution model (no matter with dataflow origin or not) globally addressable memory provides a seamless extension of the memory model viewed by threads assigned to the same processing node. For applications with irregular and unpredictable data needs, the lack of global naming capability can hurt the programmability and efficiency [96, 104]. The ability to name memory objects globally will also facilitate the support of dynamic load balancing of threads. As a result, shared-address space is adopted by most multithreaded execution models with dataflow origin.

Consistency and Replication

A memory consistency model represents a binding "contract" between software and hardware in a shared-memory multiprocessor system. It is important to provide a memory consistency model that is easy to understand and that also allows for an efficient hardware implementation. Issues arise as to how the replication of non-local data is managed. In a classical shared address space, since non-local data may be accessed through ordinary processor reads and writes, opportunities exist for the underlying system to replicate data in a way that is transparent from the users.

The design of most shared-memory multiprocessors has been based on a multiprocessor memory consistency model, usually one derived from Lamport's concept of sequential consistency [68] which requires the execution of a parallel program to appear as some interleaving of the memory operations on a sequential machine. Substantial research has been performed on how to develop relaxed memory consistency models based on the sequential consistency [44, 45, 75]. However, there have been observations that "the performance gained by using relaxed models does not justify their complexity [50], with the implication that perhaps multiprocessors should only support simple consistency models.

Some multithreaded program execution models with dataflow origin are taking an alternative point of view that the function of a memory system is to support correct execution of programs by the computer system of which it is a part [42]. A computer system, with its memory subsystem, must be a correct implementation of the program execution model. While the formulation of a memory consistency model may be part of the process of building a multiprocessor computer system, the assumed sequential consistency "requirement" is not essential to the construction of a useful and efficient parallel processor, and may even be a distraction from the primary task of correctly implementing a desirable model of parallel computing. The challenge is to investigate and develop a sound memory model fully integrated into the underlying multithreaded execution framework and design an efficient cache management scheme for its implementation.

Most scientific applications are determinate, and such programs written under a suitable execution model will not incur any scenario where memory races may occur. Therefore, memory coherence is not an issue here. Under a multithreaded execution model with dataflow origin, the so-called *single assignment* style is extended and enriched in such a way that writes in the same threads can be considered as ordinary "updates" to memory. However, the dataflow style synchronization between threads ensures that producers and consumers of values are guaranteed to be properly ordered.

In particular, under the single assignment property of the I-Structure memory [7] once a data element is defined, it will never be updated again. The copies of the data elements in the local cache will never be updated. Therefore, cache coherence should not be an issue in I-Structure memory systems. It makes the design of I-Structure cache much simpler without having to take care of the cache coherence problem. Such a memory model and its cache management has been proposed and described in [31, 32]. Implementation of I-structure caches in multithreaded architectures with dataflow origin have also been pursued in a number of research groups [1, 76, 7, 3, 65, 89].

For applications where non-determinacy is required – such as transaction processing applications – several schemes based on an extension of functional programming and I-structures have been reported. One scheme is based on the M-structures proposal [9]. Another scheme is to propose a program execution model where sound non-determinacy operators are introduced so the programs can be written in such a way that non-determinacy can be isolated, and memory operations are properly ordered, such that its effect of the ordering of memory system can be ensured to have expected behavior as anticipated by the users without resorting to traditional memory coherence requirement. A proposal along this line is based on the non-deterministic merge operator as described in [30].

Another proposal is to develop a memory consistency model, as an alternative to the SC-derived model that does not make the coherence assumption. One such model is the *Location Consistency* [40, 42, 43]. Instead of assuming that all writes to the same location are serialized according to some total order, the state of a memory location is modeled as a partially ordered multi-set (pomset) of write operations and synchronization operations. The partial order in the pomset naturally follows from the ordering constraints defined by a concurrent program. A cache management mechanism under the LC model has been proposed in [43] which eliminate the unproductive traffic for maintaining cache coherence.

The implementation of these proposals and performance evaluation based on shared memory programs are yet to be carried out so their advantages and tradeoffs can be judged both qualitatively and quantitatively.

2.4 Synchronization Models

This section will focus mainly on the synchronization among the finer-level threads (as described in Section 2.2) — a distinctive feature of multithreaded models with dataflow origin. The semantics of threaded function invocations themselves is well understood and has been discussed elsewhere [29].

Recall that when a program is divided into these finer-level threads, the data and control dependencies among the threads must be preserved. Under a program execution model with dataflow origin, such control and data dependencies are made explicit in the code. A synchronization event is posted from one thread to another to inform the recipient that a specific control or data dependence has been satisfied. For instance, the sending thread may have produced data required by the receiving thread according to some data dependence, and the producer thread must tell the consumer thread that the data is ready. If a thread depends on more than one data or control event, it has to make sure that all dependences have been satisfied before it becomes enabled. The underlying architecture (and runtime system) should provide atomic operations for sending data (possibly to a global memory location) and posting an event to guarantee that the data has been properly transferred before the receiving thread that depends on the data is enabled by the synchronization event. Some recent studies on the effectiveness of these operations have been reported in [100]. As a remark, these operations are very powerful: they can name any location in the shared address space for the transaction, they also name a thread which should be informed when the transaction is completed. Note that a designated thread may or may not become enabled after the transaction — because it may wait for more than one synchronization event. So these operations are related to both traditional shared-memory synchronization operations as well as message passing operations (including *active messages* [101]) but are different from both.

3 Dataflow in Signal Processing

Signal processing has its intellectual roots in circuit theory. Algorithms tend to be modeled as compositions of components that conceptually operate concurrently and communicate via signals that are functions of time. The components are typically filters, which transform the input signals to construct output signals. Traditionally, the signals are continuous functions of the time continuum.

Contemporary signal processing is more likely to involve discrete-time signals than continuous-time signals. Here, instead of time continuum, a discrete clock globally regulates the computation. Signals have values at the discrete clock ticks, but not in between. Multirate systems involve multiple clocks, but with clear relationships among them, so that ticks in the multiple clocks can be unambiguously associated.

A fairly direct model of discrete-time systems is obtained using the synchronous/reactive principle [11]. In this model, a concurrent program executes in a sequence of discrete steps, which correspond to ticks of a global clock. At each clock tick, the value of each signal is obtained by solving for a fixed point of a system of equations. One of the possible outcomes of this fixed point solution is that a signal has no value at a particular clock tick. By this mechanism, multirate systems are easily modeled. Some synchronous/reactive languages, notably Lustre [58] and Signal [70], have a dataflow flavor, in that signals are viewed as streams of values, where each value is aligned with a clock tick. However, the concurrent semantics of these languages is very different from Dennis [27] or Kahn [60] styles of dataflow, which are distinctly asynchronous. Whereas synchronous languages are analogous to synchronous circuits in their treatment of concurrency, Dennis and Kahn-style dataflow are analogous to self-timed circuits.

The synchronous model globally orders the tokens according to a global clock. In Dennis and Kahn dataflow, by contrast, signals are streams of tokens, where the relative ordering of tokens within a stream matters, but there is not necessarily any ordering of tokens across streams. Partial ordering constraints on tokens in distinct streams are imposed by the data precedences in the actors or processes.

In Dennis dataflow, actors are enabled by the presence of tokens on the input streams, and once enabled, can fire to produce tokens on the output streams. This enabling/firing sequence implies a partial order in which output tokens are required to occur after the input tokens that trigger the firing.

In Kahn's model, sequential processes communicate via FIFO queues that are conceptually unbounded. A process can write to a queue at any time, but if it reads from a queue, then it will block if there is no data. This blocking is a kind of dual of the Dennis notion of firing, where an actor can be fired once input data becomes available. In Kahn's model, a process replaces a Dennis actor, and it is active unless there is no input data. A sequence of firings of a Dennis actor can be viewed as a Kahn process [73], so we will henceforth refer to these two models simply as "dataflow." For their detailed semantic connection, see [74].

The synchronous/reactive model provides an abstraction of discrete-time systems where the metric properties of time are eliminated. That is, there is no measured or even specified time interval between clock ticks. Rather, there is a sequence of clock ticks. But one key property of time is retained: its global ordering of events. Dataflow abstracts things still further by eliminating this global ordering.

Streams of tokens provide a suitable abstraction for signals. Dataflow actors or Kahn processes operating on these streams provide a suitable abstraction for the components. The notion of time is lost, but the data precedences are preserved. Thus, the abstraction is more highly concurrent even than the circuit-theory roots of signals processing, since the tight coordination implied by a global ordering based on time is no longer required. Instead, only the data precedences need to be respected. It is arguable that this reduces stream-based processing to its computational essence.

3.1 Industrial Practice

The signal processing community has embraced dataflow modeling. A major reason for this is the widespread use of block diagrams for documenting and explaining signal processing algorithms. Dataflow semantics work well with a block diagram syntax. Moreover, the use of block diagrams evokes a circuit metaphor, thus making a connection with the historical roots of signal processing. But it also proves to be a convenient and lucid way to specify reasonably complex algorithms. By contrast, attempts to use dataflow principles directly in the computer architecture of embedded signal processors have largely failed.

Commercial software environments that combine dataflow semantics with a block diagram syntax include SPW from Cadence, Cossap from Synopsys, and several programs from smaller players. These software environments are used for algorithm-level modeling of systems, for specification, and as a starting point for synthesis of customized hardware designs and embedded software.

Some closely related block diagram environments have semantics that more closely resemble synchronous/reactive languages, such as Simulink from The MathWorks. These are more explicit about time. Simulink in particular has its roots in continuous-time modeling of control systems, and therefore has a very physical model of time as an integral part of its semantics.

Block diagram languages are by no means the only tool used in signal processing. A very popular alternative is matrix-oriented imperative languages such as Matlab from The MathWorks. These languages speak to the mathematical roots of signal processing more than to its roots in circuit theory. Indeed, these languages do not directly model signals. Instead, signals must be decomposed into finite and classical data structures, particularly vectors and arrays.

Since many of the more sophisticated algorithms in signal processing are based on matrix formulations, Matlab and related languages prove very convenient for describing them. These algorithms, however, most often get embedded in systems where stream-based processing is much more natural. A fairly common compromise, therefore, is to use coarse-grain dataflow, where actors can represent large scale computations that are specifically defined using a matrix-oriented language. At least one environment specifically permits the functionality of dataflow actors to be given using Matlab [17].

3.2 Decidable Dataflow

Many signal processing systems involve repeated (infinite) execution of a well-defined finite computation on an infinite stream of data. Implementations have real-time constraints, and often take the form of embedded software (such as assembly code for programmable DSPs [69]). This raises a number of interesting issues. In particular, it is important that the schedule of actor firings be predictable in order to ensure that real time constraints are met. It is also critical that a program never deadlock. Because of the embedded system context, it is also important that the total memory devoted to storing unprocessed tokens be bounded.

In dataflow, deadlock occurs when all actors are starved. Deadlock is equivalent to halting. For general dataflow models, it is undecidable whether a program will deadlock. It is also undecidable whether a program has an infinite execution that consumes bounded memory for storing pending tokens [16, 15]. One approach, therefore, is to use a subset of dataflow where these questions are decidable.

So-called synchronous dataflow [71] and its extensions [14, 38], which is not synchronous in the same sense as synchronous/reactive languages, is one such decidable model. In SDF, an actor is characterized by the number of tokens that it consumes and produces on its inputs and outputs on each firing. The inputs and outputs of each actor are labeled with an integer constant. For an input, this is the number of tokens that are required on that input stream in order to fire the actor. For the outputs, it is the number of tokens that are produced by a firing.

In a block-diagram syntax, an arc connecting the two actors represents the flow of tokens between them. If we assume that the topology of such flows is fixed, and that the number of tokens produced and consumed by each actor is fixed, then both deadlock and bounded memory are decidable. To get an idea of how this is possible, we review the notion of balance equations [71].

Suppose an actor A produces N tokens on an output that is connected to actor B. Suppose further that actor B requires M tokens on its input to fire. Suppose that actor A fires f_A times, and actor B fires f_B times. Then the balance principle requires that

$$f_A N = f_B M.$$

One such equation can be written for each arc in an interconnection of actors.

The balance principle is trivially satisfied if either $f_A = f_B = 0$ or $f_A = f_B = \infty$. A more interesting situation arises, however, if there is a bounded positive solution for f_i for each actors i such that all balance equations are satisfied. In this case, there may be a finite but nonzero set of firings that achieve balance.

It has been shown that for a connected dataflow graph, if the balance equations have a non-trivial solution, then they have a unique smallest positive integer solution [71]. If they have no solution, then there is no bounded memory infinite execution of the dataflow graph.

If the balance equations have a non-trivial solution, then there is a simple finite algorithm that will determine whether the graph deadlocks. Thus, both the bounded memory question and the deadlock question are decidable for SDF. Hence, SDF graphs can reliably be implemented in embedded real-time systems.

Although the decidability of the SDF model is convenient for certain applications, it comes at a very high cost in expressiveness. In particular, the fact that the token production/consumption patterns are fixed means that an application cannot use the flow of tokens to effect control. That is, conditional variations in the flow are not allowed. Even signal processing applications involve a certain amount of conditional computation, so this restriction becomes very limiting.

Consider for example an actor which, under the control of a boolean-valued input stream, routes tokens from another input stream to one of two outputs. Such an actor is often called a "switch." The partner of the switch is the "select," which uses a boolean input stream to guide its decisions about which of two other input streams to read for the next token. After reading an input token from either of these two inputs, it outputs that token on its single output. These two actors thus serve the same function as railway switches, splitting or merging streams of data under the control of boolean signals. However, the introduction of these two actors alone to the SDF formalism makes both deadlock and bounded memory undecidable.

One solution is to broaden the dataflow model that is used by accepting the undecidability of the key propositions but nonetheless attempt to decide. Undecidability merely states that no algorithm can decide in finite time for all programs. It does not prevent us from constructing a formalism where a compiler can decide for most programs.

This is the approach taken by Buck [16], who based his approach on the token flow model of Lee [72]. This model generalizes the balance equations to permit the number of tokens produced and consumed to vary dynamically. To do this, it attaches a symbolic value rather than a numerical value to each port of each actor. The balance equations therefore include symbolic coefficients, and have to be solved symbolically. Frequently, it is possible to solve them. The resulting solution can be used to construct a schedule of actor firings that is provably free from deadlocks and that requires only bounded memory for token storage.

Buck's model came to be called Boolean dataflow, although the basic concept has been extended to non-boolean control signals [18]. BDF, however, has two problems. First, because the key underlying implementation questions remain undecidable, a programming environment that depends on being able to decide is somewhat fragile. Small and seemingly innocuous changes to a program may cause the scheduling algorithms to fail. Also, experience constructing programs using this model indicates that it is not often the most intuitive way to specify a computation. Programs often resemble puzzles, where considerable cleverness is required to construct them and to understand what they do. Using the routing of tokens for control flow may not be such a good idea.

3.3 Reactive Systems

Signal processing systems almost never do only signal processing. Consider for example a cellular telephone. It includes intensive numerical signal processing in both the speech coder and radio modem. It may also include advanced features such as speech recognition for hands-free dialing. These signal processing components are each quite sophisticated, and invariably involve algorithms that stress the limitations of SDF. For example, equalization of a fading radio channel may involve the use of distinct algorithms during the establishment of a connection vs. steady state. Also, power conservation dictates the use of simpler algorithms when the channel is benign, suggesting mode changes that would be driven by channel estimators. SDF by itself does not match these requirements well. BDF is a possibility, but it is not a complete solution if one considers the entire design of the phone.

Signal processing applications fall into a broader class of so-called reactive systems [79], which are those that interact with their environment at the speed of the environment. Much of what characterizes many reactive systems is randomly structured control logic that bears little resemblance to the regular algorithmic structure of core signal processing algorithms. A cellular phone, for example, includes not only sophisticated signal processing, but also a substantial amount of embedded control logic for call processing and multiple-access protocols. BDF no longer matches well, since even relatively simple control logic becomes very difficult to understand.

One possible solution is to embrace a heterogeneous approach, where modeling techniques can be combined systematically to construct more complex systems. Decidable dataflow models would be used for the hard-real-time signal processing in the cellular phone, but other formalisms would be used for the control logic. Some researchers advocate melding state machines with dataflow [46], much the way StateCharts melds a synchronous/reactive model with state machines [59]. State machines are good at representing intricate control logic. Moreover, they often yield to formal analysis, and with careful language construction, remain decidable.

4 Dataflow as a Programming Model for Reconfigurable Computing

The dataflow graph program representation model has been proposed as a platform for the programming of reconfigurable computing machines (RCMs). Reconfigurable computing consists of the dynamic mapping, and re-mapping, of computations directly onto hardware. Even though the reconfigurable computing model is not new [36], it has received renewed attention in recent years due to the development and rapid evolution of FPGA devices in particular [12, 19] and of VLSI technology in general.

The design of computing systems is a trade-off between generality and efficiency. One of the great attractions of the reconfigurable computing model is its ability to increase efficiency with a smaller sacrifice in generality than vector processors for example. As a computation model, reconfigurable computing avoids one of the major limitations of the von Neumann stored-program model, namely the fetching of instructions from a storage, it can therefore achieve a much higher operation density than general-purpose processors [26].

Reconfigurable computing is a parallel execution model that combines aspects of the SIMD, systolic arrays and MIMD models. The granularity of the computation in this model can vary, within a same program, from bit-level manipulations to loop-level parallelism. Because of the wide variation in granularity and the flexibility of the execution model, in particular on machines that support partial dynamic reconfiguration, the programmability of RCMs is a major obstacle to their wider acceptance. The development of an accessible yet efficient programming paradigm for reconfigurable computing is an important research challenge.

4.1 Reconfigurable Computing Machines

In its most general form, a reconfigurable computing machine (RCM) consists of a processor and an array of programmable blocks (RC array). One of the main distinguishing features of RCMs is the granularity of the programmable blocks. At one extreme are the FPGA-based machines (FRCMs) where the programmable block consists of a logical collection of gates. Their programming model is that of a gate-level net-list [77, 78]. On the other extreme are the coarse-grain reconfigurable machines (CRCMs) where the logic block consists of an ALU and possibly a register file. In either of these categories, the architecture of the RCM can be a specialized processor (such as the Ganglion, built for neural network computations [23]), a coprocessor (such as the PRISM [8] and the Thinking Machine CM-2X [24]) or an attached processor (such as the DEC PAM and PeRLe-0 and PeRLe-1 [12, 13, 102] and the SRC Splash 1 and 2 [47, 19]).

Fine-Grained RCMs. FRCMs rely on FPGAs as the reconfigurable array. The computation to be mapped onto hardware is written in a hardware description language (HDL) such as VHDL or Verilog. The HDL program is then compiled using a synthesis tool and the resulting net list is placed and routed for the specific FPGA device used. The resulting configuration codes are then down loaded to a board that comprises one or more FPGA devices. The programming model for these machines is, therefore, essentially that of hardware design. It is a lengthy process and requires expert knowledge of HDLs as well as the particular FPGA device family being targeted. Its advantages are its low cost and its very high flexibility: the configurable circuit is designed to exactly fit the computation. It was shown that the efficiency of computations on FRCMs, measured in bit operations per λ^2 second, is one to two orders of magnitude that of conventional computers [26]. This model, therefore, tends towards the efficiency levels of ASICs while preserving a much higher flexibility. Examples of this model are the DEC PAM and PeRLe machines, the SRC Splash 1 and 2 and the Annapolis Microsystems Wildforce board [2]. FRCMs are generally designed as attached processors communicating with a host processor on an I/O bus.

Coarse-Grained RCMs. At the other extreme are the RCMs where the programmable blocks consist of complex ALUs with local registers and possible support for floating-point operations. In these machines the programming model is similar to a conventional processor or multiprocessor. Examples of this model include the UC Irvine Morphosys chip [67], the UC Berkeley Garp [48], the MIT RAW machine [103], the CMU Pipe-Rench [20] and the University of Washington RaPiD [35]. The advantage of this model is its simpler programming paradigm. It also benefits better from traditional compiler optimizations, both sequential and parallel. Its hardware structure, however, is more rigid and hence less flexible than that of its FRCMs. CRMs are generally designed as coprocessors or integrated in the processor design itself [67, 20, 21]. These two RC architecture models are by no means exclusive. As a matter of fact, the model proposed by Gerald Estrin in 1963 [36] has been called the "Fixed plus Variable" model: It combines, in one processor, a traditional CPU and a fine-grained reconfigurable engine. In a way, most reconfigurable architectures follow this same model of fixed plus variable. However, the relative sizes and the degree of integration of the fixed and variable parts vary. The CMU PipeRench, the UC Berkeley Garp and the UC Irvine M1 processors consist of a CPU and the reconfigurable component on the same processor chip.

4.2 The Programmability of RCMs

The poor programmability of RCMs is probably the most serious obstacle to their wider adoption. It derives from two distinct and orthogonal problems: (1) the complexity of the model and (2) the use of HDL (VHDL or Verilog) as a programming language in FRCMs.

The Complexity of the Model. Reconfigurable computing is powerful model that can exploit program parallelism at most levels and in an *integrated* way. The parallelism in this model ranges from bit-level manipulations, to instruction, to loop and task levels (such as producer-consumer relationships). While in a von Neumann processor, these level of parallelism are exploited in a somehow hierarchical manner (reflected for example in the storage hierarchy: registers, cache, memory, virtual memory), in a reconfigurable hardware these levels of parallelism are integrated in a flat "address" space (the address space becomes physical space). Dynamic reconfiguration adds another dimension to the programming complexity of this model. Dynamic reconfiguration is the ability to "re-program" the RCM, all or just part of it, dynamically and based on the outcome of a computation. In summary, this computational model combines the power, and complexity, of the SIMD, systolic arrays and the MIMD models. The challenge is therefore to migrate, at least in part, some of that complexity from the user to the compiler.

The Use of HDLs. At the root of the resurgence of the reconfigurable computing model is the emergence of high-density and high speed FPGAs. It is only natural that such machines have been programmed using HDLs. However, both Verilog and VHDL are languages designed, initially, for system level simulation. They are both based on a discrete-event simulation paradigm and are not easily accessible to conventionally applications programmers. The challenge is to interface between traditional high-level languages and HDLs.

4.3 Dataflow Graphs and Reconfigurable Computing

A dataflow graph represents the essence of a computation.The two major characteristics of a dataflow graph program representation are:

- Data is represented by values (arcs) and not by memory location.
- Operations are functional: the output depends only on the inputs.

As a result, the advantages of this model for reconfigurable computing are:

- The ordering of operations is determined only by true data dependencies. It is not constrained by artificial dependencies introduced by the stored program model, such as memory aliases.
- It displays parallelism in a computation at all levels. Fine-grained parallelism, at the operation level, can be exploited within a processor. Coarser, loop or function level parallelism can be exploited across processors. In contrast: systolic arrays exploit only fine grain parallelism, SMP only coarse grain.
- At the semantic level, combinational logic circuits are naturally functional(their output depends only on their inputs) and are therefore easily synthesizable from dataflow graphs.
- With certain restrictions on actors, dataflow graphs do not have any implicit state. They allow easy and correct synthesis of sequential circuits where the storage elements are determined by the synthesis tool rather than by a compiler.

Various forms of dataflow graphs have been proposed and used for the programming of systolic and wavefront arrays [62, 105, 83] as well as for the synthesis, hardware/software co-design, of application specific hardware [51, 82].

The Cameron Project is a research effort that aims at leveraging the potentials of a dataflow graph program representation to provide high-level programming for RCMs [86]. The application domain that is targeted by the Cameron Project is image processing, however, the approach is general enough and could be extended to other areas. Cameron relies on Khoros as a graphical user interface. Khoros is a widely popular program development environment designed for image processing (IP) applications [92, 66].

Program modules that run on the RCM are written in a single-assignment subset of C called SA-C. SA-C programs are tightly integrated into Khoros. SA-C has been extended to support image processing constructs as well as compilation for a hardware implementation. The most salient features of SA-C are its support for true n-dimensional arrays, window-based operations on these, and variable precision fixed-point arithmetic. The first feature simplifies the expression of image processing algorithms while the second improves the efficiency of the mapping to hardware. Other C-like hardware definition languages have been developed (e.g., Handel-C [90]) with a large degree of success. However, they still require the manual translation of algorithms into logic circuit structures. The objective of Cameron is to automate that process while still providing the programmer with some control over the mapping to hardware. SA-C programs are compiled and optimized into a dataflow graph. This format is then compiled to VHDL for FRCMs or to native machine code for CRCMs.

The project described in [91] relies on the synchronous dataflow model, as implemented in Ptolemy, to develop mapping techniques for RCMs tailored to signal processing. It extends Ptolemy by developing a new reconfigurable computing domain that separates the interface specification from implementation for each signal processing functional block.

Reconfigurable computing is a very powerful paradigm with a great potential. It is, however, still in its infancy. The implementation of this model is enabled by the rapid advances in VLSI technology. Its greatest challenge lies in its programmability. Unless that challenge is effectively and efficiently addressed, reconfigurable computing runs the risk of remaining a potential paradigm.

5 Conclusion

As a model of computation, dataflow has a long history. It has been used as a principle for computer architecture, as a model of concurrency in software, as a high-level design model for hardware. It has demonstrated its flexibility and efficiency at representing computation by the wide variety of areas in which it has been used.

The astoundingly rapid evolution of computing machines and their phenomenal penetration of modern society in the last half century has been attributed, to a large degree, to the simplicity, constance and uniformity of the stored program model. It has provided the stable conceptual framework within which processors, operating systems, algorithms, languages and compilers were designed and implemented. However, as applications evolve and exploit all the computing power of modern machines, the demand for higher performance machines is rapidly increasing. It is in the implementations of high performance computing systems that the stored program model shows its limitations. These limitations are directly and elegantly addressed by the dataflow model. We believe that the dataflow model is not only still relevant in many computing research areas, but also that many emerging and future areas will benefit from this model.

Acknowledgments

The authors acknowledge the support from DARPA, NSA, NSF, and NASA.

The last author thanks members of his Computer Architecture and Parallel Systems Laboratory (CAPSL) group, Department of Electrical and Computer Engineering, University of Delaware, for their valuable input in several discussions. And finally, thanks to Dr. Ruppa Thulasiram for his careful reading of the manuscript and comments.

References

[1] J. N. Amaral and G. G. Gao. Implementation of I-structures as a library of functions in portable Threaded-C. CAPSL Tech. Note 04, Dept. of Elec. and Computer Eng ., U. of Delaware, Newark, Delaware, Jun. 1998.

[2] Annapolis Micro Systems, "Wildforce Board Web Page," *http://www.annapmicro.com*

[3] B. S. Ang, Arvind, and D. Chiou. StarT the next generation: Integrating global caches and dataflow architecture. In G. R. Gao, L. Bic, and J.-L. Gaudiot, editors, *Advanced Topics in Dataflow Computing and Multithreading*, pages 19–54 . IEEE Comp. Soc. Press, 1995.

[4] B. S. Ang, D. Chiou, D. Rosenband, M. Ehrlich, L. Rudolph, and Arvind. Start-voyager: A flexible platform for exploring scalable smp issues. CSG Memo 415, Computation Structures Group, MIT Lab. for Comp. Sci., Dec. 1998.

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

[6] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. CSG Memo 226, Computation Structures Group, MIT Lab. for Comp. Sci., 1987.

[7] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing . *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, Oct. 1989.

[8] P. M. Athanas and H. F. Silverman, "Processor Reconfiguration Through Instruction Set Metamorphosis," *Computer*, vol. 26, pp. 11-18, March 1993.

[9] P. S. Barth, R. S. Nikhil, and Arvind. M–structures: Extending a parallel, non-strict, functional language with state. CSG Memo 327, Computation Structures Group, MIT Lab. for Comp. Sci., Mar. 1991.

[10] M. J. Beckerle. Overview of the START(*T) multithreaded computer. In *Digest of Papers, COMPCON Spring '93*, pages 148–156, San Francisco, Calif., Feb. 1993.

[11] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1270-1282.

[12] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable Active Memories: A Performance Assesment," in G. Borriello and C. Ebeling, eds., *Research on Integrated Systems*, MIT Press, Cambridge, MA, 1993, pp. 88-102.

[13] P. Bertin and H. Touati, "PAM Programming Environments: Practice and Experience," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE CS Press, Los Alamitos, CA, 1994, pp. 133-139.

[14] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Static Scheduling of Multi-rate and Cyclo-Static DSP Applications", *Proc. 1994 Workshop on VLSI Signal Processing*, IEEE Press, 1994.

[15] J. T. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," *Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Minneapolis, MN, April, 1993, vol. I, pp. 429-432.

[16] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Tech. Report UCB/ERL 93/69, Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[17] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. Journal of Computer Simulation*, special issue on "Simulation Software Development," vol. 4, pp. 155-182, April, 1994. (http://ptolemy.eecs.berkeley.edu/papers/JEurSim).

[18] J. T. Buck, "Static Scheduling and Code Generation from Dynamic Dataflow Graphs with Integer-Valued Control Systems," Invited Paper, *Proc. of IEEE Asilomar Conf. on Signals, Systems, and Computers*, Oct. 31 - Nov. 2, Pacific Grove, CA, 1994.

[19] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE CS Press,1996.

[20] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas, "Managing Pipeline-Reconfigurable FPGAs," Sixth Int. Symp. on Field Programmable Gate Arrays, 1998.

[21] R. Carley, S. C. Goldstein, T. Mukherjee, R. Rutenbar, H. Schmit, and D. Thomas, "PipeRench Web Page," Carnegie-Mellon University, *http://www.ece.cmu.edu/research/piperench/*

[22] D. Chiou, B. S. Ang, Arvind, M. J. Beckerle, A. Boughton, R. Gre iner, J. E. Hicks, and J. C. Hoe. StarT-NG: Delivering seamless parallel computing. CSG Memo 371, Computation Structures Group, MIT Lab. for Comp. Sci., Feb. 1995.

[23] C. E. Cox and W. Ekkehard Blanz, "Ganglion – a fast hardware Implementation of a Connectionist Classifier," *Proc. IEEE Custom Integrated Circuits Conf.* IEEE Press, 1991, pp. 6.5.1-6.5.4.

[24] S. A. Cuccaro and C. F. Reese, "The CM-2X: A Hybrid CM-2/Xilinx Prototype," *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE CS Press, Los Alamitos, CA, 1993, pp. 121-131.

[25] A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, Feb. 1982.

[26] A. DeHon, "Comparing Computing Machines," in *Configurable Computing: Technology and Applications, Proceedings of SPIE 3526*, p. 124, November 1998.

[27] J. B. Dennis. First version of a data-flow procedure language. In *Proc. of the Colloque sur la Programmation*, number 19 in Lec. Notes in Comp. Sci., pages 362–376, Paris, France, Apr. 9–11, 1974. Springer-Verlag.

[28] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proc. of Supercomputing '88*, pages 368–373, Orlando, Flor., Nov. 1988.

[29] J. B. Dennis and G. R. Gao. Multithreaded architectures: Principles, projects, and issues. ACAPS Tech. Memo 29, Sch. of Comp. Sci., McGill U., Montréal, Qué., Feb. 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[30] J. B. Dennis and G. R. Gao. Multithreaded architectures: Principles, projects, and issues. In R. A. Iannucci, G. R. Gao, R. H. Halstead, Jr., and B. Smith, editors, *Multithreaded Computer Architecture: A Summary o f the State of the Art*, chapter 1, pages 1–72. Kluwer Academic Pub., Norwel l, Mass., 1994.

[31] J. B. Dennis and G. R. Gao. On memory models and cache management for shared-memory multiprocessors. ACAPS Tech. Memo 90, Sch. of Comp. Sci., McGill U., Montréal, Qué., Dec. 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[32] J. B. Dennis and G. R. Gao. On memory models and cache management for shared-memory multiprocessors. In *Parallel and Distributed Processing.* IEEE Comp. Soc., Oct. 1995.

[33] J. B. Dennis. A parallel program execution model supporting modular software construction. *In Proceedings of the Massively Parallel Programming Models (MPPM-97), IEEE Computer Society, pp 50–60*, 1997.

[34] J. B. Dennis. General parallel computation can be performed with a cycle-free heap. In *Proceedings of the International Conference on Parallel Architectures and Compiler Technology*, pages 96–103, Paris, France, Oct. 1996.

[35] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath," in *Proc. Field Programmable Logic*, 1996.

[36] G. Estrin, "Parallel Processing in a Restructurable Computer System," *IEEE Trans. on Electronic Computers*, December 1963.

[37] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipel ining.* Kluwer Academic Pub., Boston, Mass., Dec. 1990.

[38] G. R. Gao, R. Govindarajan, and P. Panangaden, "Well-Behaved Dataflow for DSP Computation," *Proc. of ICASSP-92*, San Francisco, March, 1992.

[39] G. R. Gao. An efficient hybrid dataflow architecture model. *J. of Parallel and Distrib. Computing*, 19(4):293–307, Dec. 1993.

[40] G. R. Gao and V. Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. ACAPS Tech. Memo 78, Sch. of Comp. Sci., McGill U., Montréal, Qué., Dec. 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[41] G. R. Gao, L. Bic, and J-L. Gaudiot, editors. *Advanced Topics in Dataflow Computing and Multithreading.* IEEE Computer Society Press, 1995. Book contains papers presented at the Second International Workshop on Dataflow Computers, Hamilton Island, Australia, May 1992.

[42] G. R. Gao and V. Sarkar. On the importance of an end-to-end view of memory cons istency in future computer systems. In *In* Proceedings of the International Sympo sium on High Performance Computing*, pages 30–41, Fukuoka, Japan*, 1997.

[43] G. R. Gao and V. Sarkar. Location consistency – a new memory model and cache consistency protocol. CAPSL Tech. Memo 16, Dept. of Elec. and Computer Eng ., U. of Delaware, Newark, Delaware, Feb. 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[44] K. Gharachorloo, A. Gupta, and J. Hennessy. Revision to 'memory consistency and event ordering in scalable shared-memory multiprocessors'. Tech. Rep. No. CSL-TR-93-568, Computer Systems Lab., Stanford U., Stanford, Calif., Apr. 1993.

[45] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of ISCA-17*, pages 15–26, Seattle, Wash., May 1990.

[46] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," to appear in *IEEE Trans. on CAD*, 1999. (Revised from Memorandum UCB/ERL M97/57, Electronics Research Laboratory, University of California, Berkeley, CA 94720, August 1997). (http://ptolemy.eecs.berkeley.edu/papers/98/starcharts)

[47] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and using a highly parallel programmable logic array," *IEEE Computer*, vol. 24, pp. 81 – 89, January 1991.

[48] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines*, April 1997.

[49] L. J. Hendren, G. R. Gao, X. Tang, Y. Zhu, X. Xue, H. Cai, and P . Ouellet. Compiling C for the EARTH multithreaded architectu re. ACAPS Tech. Memo 101, Sch. of Comp. Sci., McGill U., Montréal, Qué., Mar. 1996. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[50] M. D. Hill. Multiprocessors should support simple memory-consistency models. In *Computer*, volume 31, pages 28–34, August 1998.

[51] J. Horstmannshoff, T. Grötker, and H. Meyr, "Mapping Multirate Dataflow to Complex RT Level Hardware Models," 11th Int. Conf. on Application-specific Systems, Architectures and Processors 1997,(ASAP '97).

[52] H. H.-J. Hum. *The Super-Actor Machine: a Hybrid Dataflow/von N eumann Architecture*. PhD thesis, McGill U., Montréal, Qué., May 1992.

[53] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the EARTH-MANNA multithreaded system. *Intl. J. of Parallel Programming*, 24(4):319–347, Aug. 1996.

[54] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proc. of ISCA-15*, pages 131–140, Honolulu, Haw., May–Jun. 1988.

[55] R. A. Iannucci, G. R. Gao, R. H. Halstead, Jr., and B. Smith, editors. *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, Norwell, Massachusetts, 19 94. Book contains papers presented at the Workshop on Mult ithreaded Computers, Albuquerque, New Mexico, November 1991.

[56] S. Jenks and J.-L. Gaudiot. Exploiting locality and tolerating remote memory access latency using thread migration (in press). *International Journal of Parallel Programming*, 1996.

[57] S. Jenks and J.-L. Gaudiot. Nomadic threads: A migrating multithreaded approach to remote memory accesses in multiprocessors. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques*, pages 2–11, Boston, Mass., Oct. 1996.

[58] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305-1319.

[59] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol 8, pp. 231-274, 1987.

[60] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[61] C. Kim and J.-L. Gaudiot. *Data-flow and Multithreaded Architectures.* In Encyclopedia of Electrical and Electronics Engineering, Wiley Publishing Co.,, 1997.

[62] I. Koren, B. Mendelson, I. Peled and G. M. Silberman, "A Data-Driven VLSI Array for Arbitrary Algorithms," *Computer*, Vol. 21, No. 10, October 1988, pp. 30-43.

[63] J.-L. Gaudiot and L. Bic, Editors, *Advanced Topics in Data-Flow Computing*, Prentice-Hall, 1991.

[64] Y. Kodama, S. Sakai, and Y. Yamaguchi. A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation. In *Proc. of InfoJapan 90*, pages 291–298, Oct. 1990.

[65] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamag uchi. The EM-X parallel computer: Architecture and basic performance. In *Proc. of ISCA-22*, pages 14–23, Santa Margher ita Ligure, Italy, Jun. 1995.

[66] KRI, "Khoral Research Inc. Web Page," *http://www.kri.com*

[67] F. Kurdahi and N. Bagherzadeh, "The Morphosys Project," University of California, Irvine 1998, *http://www.eng.uci.edu/morphosys*

[68] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, 28(9):690–691, Sep. 1979.

[69] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals Architectures and Features*, IEEE Press, New York, 1997.

[70] P. Le Guernic, T. Gauthier, M. Le Borgne, C. Le Maire, Programming Real-Time Applications with SIGNAL, Proceedings of the IEEE, Vol. 79, No. 9, September 1991.

[71] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, January, 1987.

[72] E. A. Lee, "Consistency in Dataflow Graphs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 2, April 1991.

[73] E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995.

[74] E. A. Lee, "A Denotational Semantics for Dataflow with Firing," Memorandum UCB/ERL M97/3, Electronics Research Laboratory, U.C. Berkeley, January 1997.

[75] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proc. of ISCA-19*, pages 92–103, Gold Coast, Australia, May 1992.

[76] W.-Y. Lin and J.-L. Gaudiot. I-structure software cache: A split-phase transaction run-time cache system. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques*, pages 122–126, Boston, Mass., Oct. 1 996.

[77] W. H. Mangione-Smith, "Seeking Solutions in Configurable Computing," *IEEE Computer*, vol. 30, pp. 38 - 43, December 1997.

[78] W. H. Mangione-Smith, "Application Design For Configurable Computing," *Computer*, vol. 30, pp. 115-117, October 1997.

[79] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.

[80] A. Márquez, K. B. Theobald, X. Tang, T. Sterling, and G. R. Gao. A superstrand architecture and its compilation. CAPSL Tech. Memo 18, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Delaware, Mar. 1998.

[81] A. Marquez, K. B. Theobald, X. Tang, and G. R. Gao. A superstrand architecture and its compilation. In *In Proceedings of the MTEAC99 Workshop held in conjunction with HPCA-5*, Orlando, FL, Jan. 1999.

[82] B. Mendelson, B. Patel, and I. Koren, "Designing Special Purpose Co-Processors Using the Data-Flow Paradigm," in *Advanced Topics in Data-Flow Computing*, J.-L. Gaudiot and L. Bic, Ed.: Prentice-Hall, 1991, pp. 547-570.

[83] B. Mendelson and G. M. Silberman, "Mapping Data Flow Programs on a VLSI Array of Processors," in *Proc. Int. Symp. on Computer Architecture*, Pittsburgh, Pennsylvania, June 1987, pp. 72-80.

[84] W. A. Najjar, W. M. Miller, and A. P. W. Böhm. An analysis of loop latency in dataflow execution. In *Proc. of ISCA-19*, pages 352–360, Gold Coast, Australia, May 1992.

[85] W. Najjar and J.-L. Gaudiot. Multi-level execution in data-flow architectures. In *Proc. of ICPP '87*, pages 32–39, St. Charles, Ill., Aug. 1987.

[86] W. Najjar, A. P. W. Bohm, B. Draper, and R. Beveridge, "The Cameron Project," Colorado State University, Fort Collins, CO 1998, *http://www.cs.colostate.edu/cameron*

[87] S. S. Nemawarkar. *Performance Modeling and Analysis of Multithreaded Architectures.* PhD thesis, Montréal, Qué., Aug. 1996.

[88] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. of ISCA-16*, pages 262–272, Jerusalem, Israel, May–Jun. 1989.

[89] K. Okamoto, S. Sakai, H. Matsuoka, T. Yokota, and H. Hirono. Multithread execution mechanisms on RICA-1 for massi vely parallel computation. In *Proc. of the 1996 Conf. on Parallel Architectu res and Compilation Techniques*, pages 116–121, Boston, Mass., Oct. 1 996.

[90] Oxford Hardware Compilation Group, *The Handel Language*, 1997.

[91] E. K. Pauer, C. S. Myers, P. D. Fiore, J. M. Smith, C. M. Crawford, E. A. Lee, J. Lundblad and C. X. Hylands, "Algorithm Analysis and Mapping Environment for Adaptive Computing Systems," *Second Annual Workshop on High Performance Embedded Computing*, MIT Lincoln Labs, Lexington, MA, September 1998 (http://ptolemy.eecs.berkeley.edu/papers/98/ACSmapping/).

[92] J. Rasure and S. Kubica, "The KHOROS Application Development Environment," in *Experimental Environments for Computer Vision and Image Processing*, H. I. C. a. J. L. Crowley, Ed. New Jersey, 1994.

[93] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of ISCA-21*, pages 325–336, Chicago, Ill., Apr. 1994.

[94] X. Tang. *Compiling for Multithreaded Architectures.* PhD thesis, University of Delaware, Newark, DE, Apr. 1999.

[95] S. Sakai, K. Okamoto, H. Matsuoka, H. Hirono, Y. Kodama, and M. Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Conf. Proc., 1993 Intl. Conf. on Supercomputin g*, pages 251–260, Tokyo, Japan, Jul. 1993.

[96] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 6(26):42–50, 1993.

[97] A. Sohn, M. Sato, N. Yoo, and J.-L. Gaudiot. Data and workload distribution in a multithreaded architecture. *Journal of Parallel and Distributed Processing*, pages 256–264, 1997.

[98] X. Tang, J. Wang, K. B. Theobald, and G. R. Gao. Thread partitioning and scheduling based on cost model . In *Proc. of SPAA '97*, pages 272–281, Newport, R hode Island, Jun. 1997.

[99] X. Tang and G. R. Gao. How "hard" is thread partitioning and how "bad" is a list scheduling based partitioning algorithm? In *the Proceedings of Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 1998.

[100] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, Montréal, Qué., Jan. 1999.

[101] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of ISCA-19*, pages 256–266, Gold Coast, Australia, May 1992.

[102] J. Vuillemin et al., "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Trans. on VLSI Systems*, March 1996.

[103] E. Waingold, M. Taylor, D. Srikrishna, V. Srakar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarsinghe, and A. Agrawal, "Baring it all to Software: RAW Machines," *IEEE Computer*, vol. 30, pp. 86 - 93, September 1997.

[104] M. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the Supercomputing '93*, pages 12–21, Portland, Oregon, Nov. 1993.

[105] S. Weiss, I. Spillinger and G. M. Silberman, "Architectural Improvements for A Data-Driven VLSI Processing Array," *J. of Parallel and Distributed Computing*, Vol. 19, No. 4, December 1993, pp. 308-322.