

SCORE on Millennium Cluster

A data-flow programming environment for a cluster of SMPs
Semester Project Fall 2001

CS262/267

Nan Zhou, Yatish Patel, Yury Markovskiy
{normzhou, yatish, yurym}@cs.berkeley.edu

May 16, 2002

Abstract

This work presents SCORE for Millennium, a data-flow programming environment adapted for a cluster of SMPs. Programs expressed as data-flow graphs expose inter-operator parallelism and dependencies, which permit a runtime system to analyze and improve their performance automatically, *without* human intervention. Here we describe the implementation of the infrastructure to support such a programming environment and discuss the critical issues and code placement methodology that affect application performance. We demonstrate that application performance automatically scales with resources and show the effectiveness of static (load-time) graph analysis in the form of improved performance of communication dominated applications.

1 Introduction

To efficiently exploit multi-processor and multi-node networked systems, the underlying hardware resource constraints are typically exposed to the application designer. While this eliminates potential inefficiencies by allowing the programmer to manually schedule and allocate resources for an application, development of applications for parallel systems requires intimate familiarity with underlying hardware and drastically increases the development time and complexity. To make matters worse, assuring automatic application performance scaling is not simple.

Traditional parallel programming environments, such as MPI, UPC and others, overwhelmingly support *data partitioning* between processors, and thus many developers choose to design their applications in such a manner (Figure 1a). These environments often provide an illusion of shared memory to simplify programming and the primitives to exchange data between the processors. A developer is given a great deal of control over application performance and ability to tune it to a particular computation target and/or

communication media parameters. However, a number of disadvantages exists in this approach.

First, a programmer must explicitly deal with communication and synchronization of threads executing on different nodes, which may lead to a deadlock and result in frequent barriers to avoid race conditions in data accesses. Second, a programmer is fully responsible for balancing computation and communication load between resources. Third, both the weakness and the strength, parameters of the underlying platform are exposed to the programmer, limiting application compatibility and automatic performance scaling across generations of systems, while potentially yielding better performance.

Data-flow programming paradigm attempts to address some of the disadvantages of traditional parallel programming systems. Instead of partitioning data among the processors, a data-flow program is expressed as a set of communicating operators, and the data streams (flows) through the processing elements executing the code in these operators. Each operator can be thought of as an independent flow of control, a thread. Contrast a traditional programming model shown schematically in Figure 1a with a data-flow program in Figure 1b, where the program is partitioned among processing elements, not the data.

Programs expressed as data-flow graphs possess several advantages over the traditional parallel programming paradigms. First, they cleanly expose of inter-operator (inter-thread) communication pattern, dependencies and execution precedence constraints. Second, they hide synchronization and communication from the programmer by making them implicit in the execution semantics (see Section 3 for more detail). These advantages allow the *runtime* environment (*e.g.* scheduler/OS) to efficiently allocate and manage computation and communication resources for an application. Although a user also can optimize and schedule an application for a specific target or architecture, however, moving to a different target or new generation of systems would require new optimizations and tuning. Since

data-flow programs expose inter-thread dependencies to the system, the runtime environment can automatically adapt the application execution to the target and improve/scale the performance without human intervention.

To demonstrate these ideas and feasibility of implementing a general data-flow programming environment on a cluster of SMPs, we have adapted SCORE to run on Millennium cluster. SCORE, a data-flow programming framework originally developed for embedded systems, provided a programming language and a set of concepts that form a powerful computation data-flow model (Section 3). Our goal for this project was to understand the challenges and issues that must be addressed to make such a system practical, to demonstrate *automatic* performance scaling with available resources, and take advantage of knowledge obtained from data-flow graphs to perform static operator placement. Our system is fully operational; it currently demonstrates reasonable performance improvements from investigated placement methodologies and scaling with resources.

The remainder of the paper is organized as follows. Section 2 summarizes work in data-flow computing that relates to SCORE on Millennium. Section 3 discusses the SCORE compute model and its application on the originally targeted domain: the embedded systems. Section 4 follows with discussion of implementation of SCORE on a cluster of SMPs, the critical performance issues and scheduling methodology. The results and future work are outlined in Sections 5 and 6.

2 Related Work

Data-flow programming has been used extensively in digital signal processing (DSP) and embedded systems domains. A DSP algorithm is traditionally represented as a signal-flow graph of operations to be performed on discrete signal samples [7]. In the embedded systems domain, data-flow paradigm is widely used to define system behavior and to generate software (firmware) automatically from a specified data-flow graph [3]. In both of these cases, the graph operators frequently define *primitive* operations supported by the underlying execution target, necessarily forcing fine-grain communication, *i.e.* a unit of communicated information is often as small as a bit. However, this work provides the infrastructure for data-flow programming for a cluster of SMPs, where both the granularity of computation executed by each graph operation and the granularity of a unit of communication is typically larger than in the case of embedded systems (*e.g.* operations of coarser grain are more appropriate for a large system such as Millennium).

Coarse grain data-flow programming, where each graph operation executes a *large* number of primitive instructions, has so far found only limited uses in network computer clusters. One example is Berkeley *River* project that of-

fers a data-flow programming environment that effectively minimizes performance impacts of heterogeneity and non-uniformity in a cluster of workstations [2]. *River* takes advantage of increased disk I/O bandwidth aggregated from all cluster members, and makes effective use of redundancy in data layout to balance work between all cluster nodes and maintain high application performance even in light of failures. Instead, SCORE strives to provide a general data-flow programming environment primarily suited for parallel computation and data processing. At this point, no attempt has been made to tolerate irregularity in cluster behavior that affects application performance.

3 SCORE Overview

Stream Computations Organized for Reconfigurable Execution (SCORE [4]), developed in Berkeley BRASS group under Professors Wawrzynek and DeHon¹, strives to eliminate existing barriers to efficient exploitation of reconfigurable devices by introducing a compute model based on paged virtual hardware (similar to virtual memory). Although the original intended underlying architecture is a *microprocessor/reconfigurable(FPGA) array* hybrid, SCORE principles apply to any interconnected collection of processing elements (*e.g.* computers, processors, custom logic), which can serve as a parallel programming target.

Under the SCORE compute model a program is expressed as a *data-flow* graph of operators executing in parallel with no shared global state. These operators communicate through streams, FIFO channels with logically unbounded buffering capacity. The semantics of communication operations include blocking reads and non-blocking writes, *i.e.* an operator blocks on its inputs (it executes when required input tokens are present) and never blocks on the output. In this respect, SCORE closely resembles Kahn process networks [5].

The use of streams accomplishes two goals in SCORE:

1. Structured as FIFO communication channels, streams effectively hide communication latency between operators.
2. Since operators only interact with the streams with no notion of timing, it is possible to execute the operators in any order while still obtaining deterministic results.

Under the current infrastructure, programs for SCORE are written in the language called TDF (Task Description Format), which is used to describe individual SCORE data-flow graph operators. In TDF each operator is a FSM (Finite State Machine), where the activation of each state depends on inputs token availability. Figure 2 shows a sample fragment of TDF code.

¹Professor at California Institute of Technology

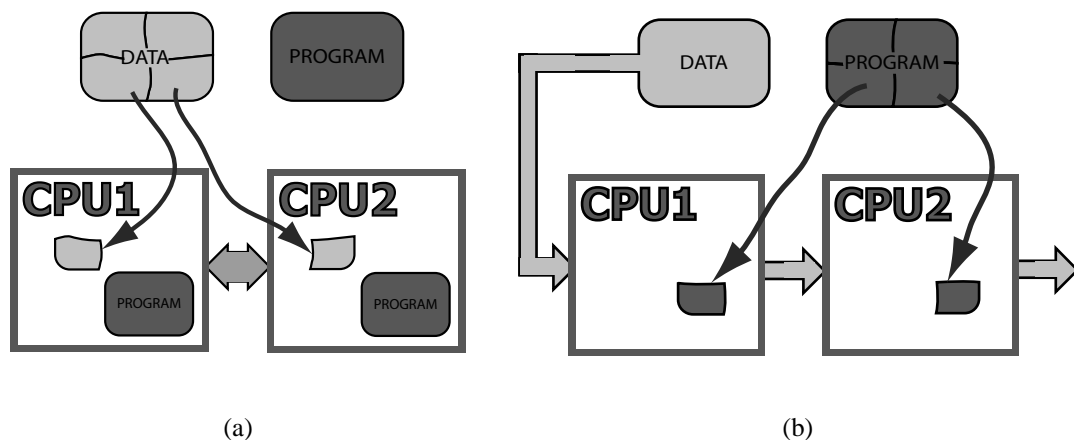


Figure 1: Kinds of parallelism that can be exploited in parallel programming paradigms: (a) Each processing node executes identical program operating only on a subset of data. (b) A program is partitioned between processing nodes, and the data streams (flows) through the composed pipeline.

```
operator B(input unsigned[32] in,
          output unsigned[32] out)
{
  unsigned[32] local_a;
  state one (in): {
    local_a = local_a + in; goto two;}
  state two (): {
    out = local_a; goto one;}
}
```

Figure 2: TDF description of operator B, which contains a small FSM with two states `one` and `two`. The operator is initially in state `one` and executes only when a token appears on `in`. The operator then transitions to state `two`.

In the current SCORE runtime environment, programs are written in TDF and compiled to functional and behavioral C++ code using a TDF compiler (`tdfc`). The functional code executes on a processor, while the behavioral code executes on a reconfigurable array simulator and is meant to represent RTL (register transfer language) description of the operator in hardware.

4 SCORE on Millennium

Millennium is a cluster of SMP machines (processing nodes) running Linux OS. These processing nodes are connected together via a high speed network. Although Section 3 described SCORE runtime environment targeted towards a very different processing architecture and network topology, the SCORE model does not limit itself from operating on

other platforms such as Millennium.

Operators in SCORE can easily be thought of as independent concurrent threads running on sequential processor(s). The functional code generated by TDF compiler (`tdfc`) can be spawned as independent threads or processes. As long as these threads communicate with each other through stream interface defined in SCORE, the result of a computation will remain the same. The order of execution or the assignment of operators to different nodes will only affect the performance of the program but not the correctness of the result.

To allow SCORE applications to execute on Millennium, we have modified TDF compiler back-end for a new target, built primitive stream operations on top of MPI and developed placement/scheduling algorithms to automatically map different operators to different SMP nodes in the cluster to take advantage of the parallelism in the cluster.

4.1 Implementation

The original TDF compiler provided with SCORE required modification to generate operator code compatible with the new infrastructure. Certain parts of the generated code only applied to the original SCORE environment and were removed. In addition, the compiler was modified to create operators with timing and profiling code. The statistics gathered were then used to improve the placement algorithms.

Each operator described in TDF runs as a single thread on a single Millennium node. At the start of an application an operator placement algorithm is executed to assign a SMP node to each operator, given the available resources (cluster nodes). A single node may run one or more operator threads.

Streams connect two operators that are placed on a single SMP node or two operators that are placed on separate

nodes. Streams connecting operators located on the same node are implemented as an unbounded queue in local memory. Stream write pushes a token into the queue, and stream read pops a token out of the queue. Operators located on separate nodes require remote streams capable of crossing node boundaries and accessing the network.

The remote stream primitives are implemented on top of Message Passing Interface (MPI) used on the Millennium cluster. MPI provides a method to pass messages in both shared memory and distributed memory systems. MPI functions such as MPI_Send and MPI_Recv are available to send and receive data blocks.

Unfortunately, the implementation of MPI used on the Millennium cluster is *not thread-safe*. Since our operators are implemented as threads, each operator thread cannot directly issue MPI calls. We introduced an I/O subsystem that runs as a separate thread and manages all inter-node communication. It is the only thread in the process making MPI calls. The I/O subsystem provides a layer on top of MPI that drains and fills all the streams that cross the network. The I/O subsystem polls all streams to gather data that needs to be sent and issues the MPI_Send command. Incoming messages are processed with asynchronous MPI_Irecv and tokens are directed into appropriate streams.

Figure 3 is an example of a data-flow graph running on two nodes. Streams that connects operator A to B, A to C, E to F, and D to F are local streams that use queues in local memory. Streams that connect B to E and C to D require the use of the I/O subsystem to transfer the data across the network.

The initial I/O implementation used a method where every MPI_Send only sent a single token. This proved to be an extremely inefficient use of the network resources causing very poor performance for remote streams. The final I/O implementation uses a system of stream token aggregation and batch sending. Any tokens available on all streams that are destined for the same node are collected and sent using a single MPI_Send call to improve network bandwidth utilization, especially for faster producers.

The initial I/O implementation of stream reads also required improvements. The first version required a mutex to be acquired for every operator `stream_read` to provide synchronization between the I/O thread and the operator. The overhead required for the mutex acquisition and contention with the I/O thread caused operators with low computation to communication ratios to perform poorly. The final implementation used a two level buffering system, where the operator issued `stream_read` fills a local buffer requiring only a single mutex acquisition to read all available token.

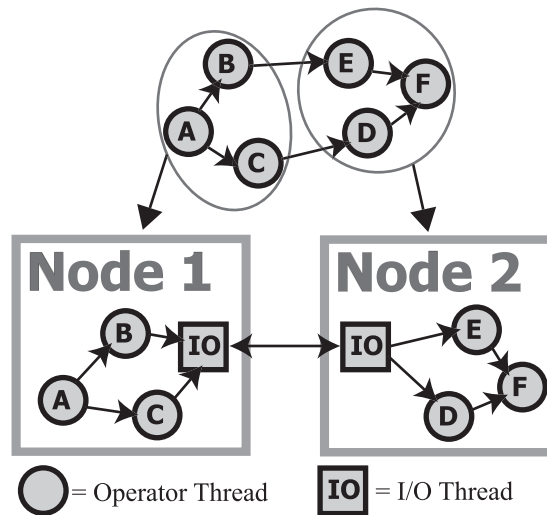


Figure 3: Example Mapping of Data-flow to Millennium

4.2 Load-time placement

Upon application start at its load-time, the scheduler is automatically invoked to compute assignment of each operator to a SMP node (operator placement). The assignment stays the same throughout the execution of a program and will only be recomputed when the program is run again. The choice of “static” scheduling as oppose to continuous monitoring of application execution by a dynamic scheduler is not accidental. In this project, we wanted to demonstrate that applications expressed as data-flow graphs expose critical inter-operator dependencies that permit analysis to efficiently allocate resources for optimal performance and thus reduce net run-time. In addition, many *practical* applications exhibit long term static data-flow behavior, even though in a short term data-flow (token flow) behavior can be very dynamic. Thus “static” scheduling is a viable alternative to dynamic, which makes scheduling decisions by discovering communication and resource utilization patterns in an application (see [6] for a more complete discussion).

In this project, we did not address the temporal scheduling of individual threads on a SMP node (time-multiplexing of threads), but left that responsibility to the OS scheduler. We were primarily concerned with the costs of communication in the data-flow environment, where communication (*flow*) is the key element. Therefore, we attempted to understand the behavior of the underlying platform and develop strategies for efficient placement of operators on SMP nodes, *i.e.* partitioning the graph between machines.

4.2.1 Cost Model

Our scheduler computes placement for each individual node based on two pieces of information: (1) data-flow graph

topology and historical token flow and operator execution behavior and (2) the cost model of the underlying platform.

Development of a practical analytical cost model for a complex system such as the Millennium cluster, a model that accounts for at least the basic components such as computation and communication throughput, turned out to be almost an impossible task. Performance of the Millennium cluster, *i.e.* its nodes and communication media, is subject to a large number of variables completely outside of our control: inability to reserve dedicated nodes to run experiments, congestion in the network and lack of control over the OS scheduler. However, without an accurate cost model, static scheduling can not be effectively performed, since observation of application behavior at run-time (*i.e.* dynamic scheduler) would be required to make scheduling decisions.

An alternative path was chosen to obtain a “perceived” cost model or a set of scheduling strategies that *should* lead to improved application performance. We have developed a set of small applications that can be manually placed (assigned to SMP nodes) to expose costs such as bandwidth, latency and processor utilization through their run-times (makespans). Although the cost model expressed in absolute quantities cannot easily be extracted from measured run-times, the *relative* costs can be understood. From these experiments, a placement strategy was developed:

1. co-place nodes that form a communication cycle in a data-flow graph;
2. co-place nodes that communicate frequently;
3. since load balancing on each SMP node was determined to have less of an effect on performance, prioritize communication sensitive partitioning over computation load balancing.

4.2.2 Partition Algorithms

Equipped with “perceived” relative cost model, several data-flow graph strategies were developed and evaluated to determine their effect on application performance.

- **Naive:** a simple pseudo-random assignment of operators to available SMP nodes, comparable to node assignment that `mpirun` provides.
- **Topological:** a combination of a loop clustering pass with a greedy breadth-first search packing of nodes into partitions. This scheduler guarantees that nodes that form communication cycles in data-flow graphs are co-placed and attempts to place neighbor nodes together (preserve topological locality in a data-flow graph).

- **Smart:** N-way graph partitioning algorithm based on recursive graph bisection. The algorithm attempts to minimize graph edge cuts and balance aggregate node weights of each partition. For this partitioner algorithm the node weights are set to 1, and edge capacities are set to 1. METIS graph partitioning library was used in our implementation [1].
- **Smart_min:** Equivalent algorithm to the one used in **Smart**, except that back annotations are used for node weights and edge capacities (discussed further below). The algorithm attempts to find mincuts in the partitioned graph while balancing individual partitions.
- **Smart_max:** Same as **Smart_min** except that the algorithm attempts to find max-cuts in the data-flow graph.

Our implementation makes use of profiling data to refine the scheduler’s understanding of graph behavior. After each run the following data is collected: (1) number of tokens transmitted through each stream, and (2) user execution time for each operator. When the application starts, the scheduler may choose to access profiling data from previous runs in order to back annotate the data-flow graph. Placement and partitioning can then be adjusted to account for previous application behavior.

5 Results

Two main applications were evaluated under SCORE on Millennium. The first application, Wavelet encoder, was borrowed from the original SCORE environment. The second application, Compute Heavy Application (CHA), was developed to demonstrate specific characteristics of our placement algorithms.

The Wavelet encode is an image compression program, consisting of thirty operators. Its original target was an embedded system with a reconfigurable array. This target led to the creation of operators with very low computational requirements, and hence application execution time is heavily dominated by I/O. The original Wavelet TDF code was recompiled and executed on Millennium without any modifications.

Figure 4 shows the Wavelet runtime versus the number of nodes, using each of our placement algorithms. With exception of the Naive, all our placement algorithms perform communication sensitive partitioning of a graph. Wavelet is a communication dominated application, and thus each of our intelligent placement algorithms displays significantly improved performance when compared to the Naive approach. The Naive algorithm’s inability to account for graph structure prevents it from scaling when more nodes are added. In fact, as more resources are added the Naive algorithm performs worse and worse. The noticeable sudden

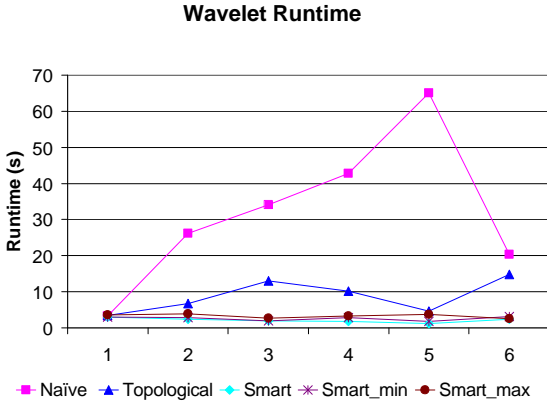


Figure 4: Wavelet runtime vs number of nodes

improvement in performance of the Naive algorithm when the number of machines was increased from five to six is simply explained as a stroke of luck. The Naive approach just happens to co-place some of the communication heavy nodes together, resulting in improved performance.

The graph in Figure 5 (a “zoomed-in” version of the Figure 4 without the Naive and Topological algorithms) shows that Smart algorithms that account for graph structure are generally able to scale application performance to a certain point as additional resources are added. However, graph behavior is *not* monotonic with available resources. We presently are not able to accurately explain this anomaly because of difficulty of constructing an accurate cost model that represents our target: networked cluster of SMPs. Advanced static placement techniques rely heavily on understanding of a cost model, and more analysis of our cost model and better measurements are required to explain the behavior of this program with different placement algorithms.

The Compute Heavy Application (CHA) is a twenty operator application specifically written to evaluate the performance of our placement algorithms. Each operator in the application has a much higher computation to communication ratio than Wavelet encoder, and thus this application represents better the types of programs that are suited for execution on Millennium cluster. Therefore, as Figure 6 shows, the CHA scales well with available hardware. Note, that all placement algorithms, including the Naive, result in approximately equal runtime for this application. Our placement algorithms are communication sensitive and thus do not have significant effect on the application whose runtime is compute dominated. In addition, as was noted in 4.2.1, the load balancing in data-flow graph partitioning has marginal effect on the performance, since this in this project we did not attempt to influence OS scheduling of operator threads in each SMP node.

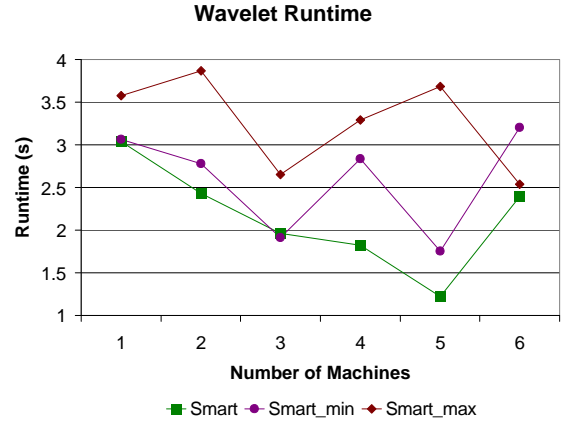


Figure 5: Wavelet runtime vs number of nodes (Zoomed)

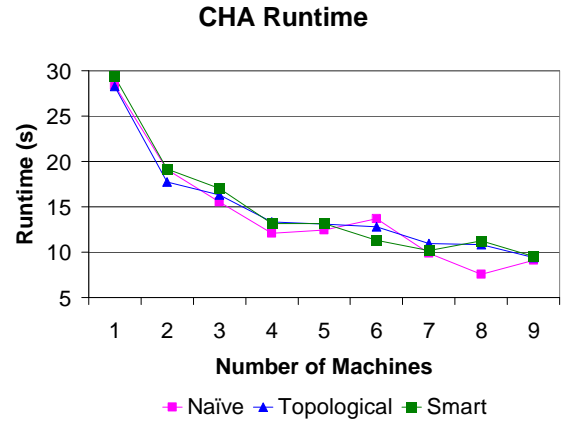


Figure 6: Compute Heavy Application runtime vs number of nodes

6 Future Work

As mentioned in subsection 4.2.1 an accurate, detailed cost model is required for the back-annotated placement algorithms. We were unable to obtain a good, clean mathematical representation to use in our algorithms. This caused the performance of the back-annotated versions of **Smart** to deviate from what was expected. Improved understanding and analysis of the dynamics of the Millennium cluster is required to establish an effective cost model.

In addition, the TDF language used describe the applications we tested was designed to be an intermediate language meant to be generated from a higher level language. The user was not expected to program directly in TDF, unfortunately no alternate language was proposed. The development or adaptation of a higher level more expressive language is required to represent data-flow applications and allow a compiler to partition data-flow graph nodes into operators

of appropriate computation granularity to better utilize the Millennium cluster's resources.

7 Conclusion

We have implemented the required infrastructure to allow SCORE, a data-flow programming environment originally designed for an embedded reconfigurable array, to run on Millennium, a cluster of networked SMPs. Applications originally written for execution on a reconfigurable array run unmodified on the Millennium nodes using our runtime environment. In doing so, we have demonstrated that SCORE is indeed a general purpose programming environment capable of operating on a wide range of platforms.

This project demonstrates that programs expressed as data-flow graphs expose inter-operator thread-level parallelism and dependencies that permit runtime environment to efficiently allocate resources (in our case, perform operator to node assignment) for an application and provide automatic scaling and adaptation to underlying execution platform. We have demonstrated several placement algorithms, limited performance scaling and identified the inability to obtain an accurate cost model of the underlying target as the greatest weakness of our approach. However, with more work and analysis this problem can be remedied, yielding a runtime system for data-flow programs that offers high performance and adaptability.

References

- [1] METIS: Family of Multilevel Partitioning Algorithms. <http://www-users.cs.umn.edu/~karypis/metis/>.
- [2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999. ACM Press.
- [3] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.
- [4] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (SCORE): Extended Abstract. In *Conference on Field Programmable Logic and Applications (FPL '2000)*, pages 605–614. Springer-Verlag, August 28–30 2000.
- [5] G. Kahn. Semantics of a Simple Language for Parallel Programming. *Info. Proc.*, pages 471–475, August 1974.
- [6] Yury Markovskiy, Eylon Caspi, Randy Huang, Joseph Yeh, Michael Chu, André DeHon, and John Wawrzynek. Analysis of Quasi-Static Scheduling Techniques in a Virtualized Reconfigurable Machine. In *Proceedings of the Tenth International Symposium on Field-Programmable Gate Arrays (FPGA 2002)*, February 2002.
- [7] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Prentice Hall, 1989.