Procedia Computer Science

Volume 29, 2014, Pages 1289–1298

ICCS 2014. 14th International Conference on Computational Science

A Dataflow Programming Language and Its Compiler for Streaming Systems

Haitao Wei, Stéphane Zuckerman, Xiaoming Li, and Guang R. Gao

University of Delaware, Newark, DE, U.S.A
hwei@udel.edu, szuckerm@udel.edu, xli@udel.edu, ggao.capsl@gmail.com

Abstract
The dataflow programming paradigm shows an important way to improve programming productivity for streaming systems. In this paper we propose COStream, a programming language based on synchronous data flow execution model for data-driven application. We also propose a compiler framework for COStream on general-purpose multi-core architectures. It features an inter-thread software pipelining scheduler to exploit the parallelism among the cores. We implemented the COStream compiler framework on x86 multi-core architecture and performed experiments to evaluate the system.

Keywords: Dataflow Programming, COStream, Compiler, Streaming

1 Introduction

As streaming systems based on multi-core processors have become ubiquitous, there is an urgent demand to design parallel programming models and compiler techniques to exploit parallelism on these systems. Parallel programming models like MPI and OpenMP provide a good way to perform parallel programming. But they still require the programmer to have the parallel model in mind and be careful to avoid data races, which adds to the burden of the programmer—especially for domain experts. Parallelizing compilers translate sequential programs to multi-thread ones automatically, but have only achieved limited success.

Recently, Fresh Breeze [3, 4] and the Codelet Model [17], both rooted in dataflow, have proposed a promising execution and architecture model for dataflow programming to exploit pervasive models such as data, task and pipelining parallelism. SWARM [9] and DARTS [13] are two implementations of Codelet model, the former from and industrial, and the latter from an academic standpoint. They use a dynamic dataflow model (DARTS uses a hybrid static-dynamic dataflow model) to implement a fine-grained parallelism at runtime for the large scale parallel system. The data dependency is determined and scheduled at runtime.

A significant amount of critical applications require a streaming model. They can be naturally supported by dataflow-inspired program execution models such as Fresh Breeze and

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.116

the Codelet Model. In the dataflow graph of streaming applications, each node is an autonomous computational unit. It has an independent instruction stream and address space and the data flow between nodes can be made through communication channels, implemented as FIFO queues. The dataflow model of computation exposes communications and an abundance of parallelism which offers the compiler many opportunities to lead to an efficient execution. The interesting thing is it uses the size of the data in each receiver/sender to control the ready signal to drive the computation. This can be used as a schema of rate control in the application.

In this paper we propose COStream, a programming language as a implementation based on Flesh Breeze and Coldlet model. Compared with previous dataflow language like StreamIt [6], COStream adopts some grammar structure from IBM SPL [7] to improve the programbility and code reuse. It takes a multiple input/output actor instead of single input/output to support common computation. It uses explicit varible passing to make dataflow graph construction easy. It uses a special structure to support sub-graph construction which can be reused for larger graph. We also proposed a compiler framework for COStream. We implemented the compiler framework and evaluate it on x86 multi-core architecture.

Section 2 presents the two program execution models we use as a fundation for COStream. Sections 3 and 4 respectively describe the COStream programming language and compilation framework. Section 5 presents our experimental results. Section 6 presents our vision to apply COStream to DDDAS. Finally, we conclude in Section 7.

2 Fresh Breeze and the Codelet Model

2.1 Codelet Model

The Codelet Execution Model [17], is a hybrid model that incorporates the advantages of macro-dataflow [8, 5, 11] and the Von Neumann model. The Codelet Execution Model can be used to describe programs in massively parallel systems, which are expected to display a certain level of hierarchical or heterogeneity. The Codelet Execution Model extends traditional macro-dataflow models in the way shared system resources are managed. As in macro-dataflow, computation is done through units of small serial code known as codelets. The codelets are similar in their intent and behavior to the actors of macro-dataflow. However, where traditional macro-dataflow is only concerned with data availability, codelets can depend on other events, which refer to the availability of a (shared) resource, such as bandwidth availability, a specific processing element, a given power envelope, etc. Codelets are tagged with resource requirements and linked together by data dependencies to form a graph (analogous to a dataflow graph [2]). This graph is further partitioned into asynchronous procedures which are invoked in a control flow manner. This type of threading/synchronization model enables fine-grain execution.

2.2 The Fresh Breeze Memory Model

The Fresh Breeze [3, 4] model also uses the concept of codelet as the core computational unit (i.e. event-driven non-preemptible tasks which are *fired* only when all dependencies are satisfied). In addition, its memory model uses trees of fixed-size chunks of memory to represent all data objects. Chunks are fixed size memory buffers (for now 128 bytes). Each chunk has a unique identifier, its handle, that serves as a globally valid means to locate the chunk within the storage system. Chunks are created and filled with data, but are frozen before being shared with concurrent tasks. This write-once policy eliminates data consistency issues and simplifies memory management by precluding the creation of cycles in the heap of chunks. Another

benefit is that low-cost, reference-count based garbage collection may be used to reuse memory chunks for which no references exist in the system. Such a memory model provides a global addressing environment, a virtual one-level store, shared by all user jobs and all cores of a many-core multi-user computing system. It may be further extended to cover the entire online storage, replacing the separate means of accessing files and databases in conventional systems.

2.3 Discussion: Toward a Streaming Codelet Execution Model

The stream programming model mentioned in this paper can be seen as a data-driven or stream implementation of Fresh Breeze/Codelet Model. In stream programming, the program is represented as a dataflow or stream graph. Nodes represent computation, edges represent communication implemented as FIFO queues. Following the Codelet model, computation is done by a codelet or an actor. Events in the stream programming model are triggered by the data items and FIFO buffer resources availability. The producer writes the data into the FIFO queue only when the computation is finished. The consumer reads the data from its FIFO queue at the beginning of the computation. Therefore, the write-once policy of Fresh Breeze Model is guaranteed by the single-writer / single-reader attributes of FIFO queues.

3 The COStream Programming Language

3.1 Operator and Stream

Adopted from SPL [7], an *operator* in COStream is the basic computation unit. An operator can be seen as a procedure which consumes data from its input channel and produces data to its output channel. The data channel is called a *stream*. Different operators are connected by streams to construct a dataflow graph which represents the whole COStream program. An example of operator in Figure 1(a) is `Aver`, a component of the moving average application. Each operator contains three sections: *init*, *work* and *window*.

Init is called at initialization time. In this case, `Aver` calculates the weights for the moving average. *Work* describes the most fine-grained execution step of the operator in the steady state schedule. Operators can be either *stateful* or *stateless*. A stateful operator depends on the local state of last execution and modifies the state at each execution. A stateless operator does not depend on the state of previous executions. The source in the example is a stateful operator, because the value of x in this execution depends on x's value in the previous execution.

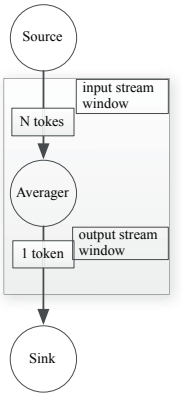
3.2 Window

The only way an operator accesses the data from a stream is using *windows*. A window is bound to a stream. Within a work section, an operator can fetch data from the input stream window via a subscript in an array fashion, and put the result to the output stream window. As with SPL [7], there are two kinds of windows: *sliding* and *tumbling* windows. A sliding window has two parameters: the window size ws and the sliding size ss. A tumbling window only has one parameter: the window size. Each time a work section executes, the operator can read ws data items in the sliding window of input stream and slide out ss data items when the execution is done. The tumbling window is a special case of the sliding window: $ss = ws$. The output stream window is always a tumbling window. If not indicated, the default window type is a tumbling window and the size is 1 by default. The *stream* `Source` and `Sink` operators have a default tumbling window of size 1.

```
composite Main(){
  int N=10;
  graph
    stream<float x> S=Source(){
      int x;
      init{x=0;}
      work{
        S[0].x=x;
        x++;
      }
//initialize an instance of composite
MovAve
  stream<float x> P=MovAver(S)(N);
    Sink(P){
      work{
        print(P[0].x);
      }
    }
}
```

```
composite MovAver(output O,
input rawIn){
  param
    int N;
  graph
    stream<float x> O=Aver(rawIn){
      float w[N];
      init{
        for(i=0;i<N;i++)
          w[i]=i;}
      work{
          int sum=0, i;
          for(i=0;i<N;i++)
            sum += rawIn[i].x*w[i];
          O[0].i=sum/N;
      }
      window{
          rawIn sliding(N,1)
          O tumbling(1);
      }}}
```

(a) An Example of a COStream program



(b) The corresponding dataflow graph

Figure 1: An Example of COStream program and its Graph Representation.

The window and sliding sizes show a way for the programmer to control the computation rate of a operator. Operators are executed in a data driven fashion. As long as an operator has enough data (which is equated to window size) to consume in its input stream window, it runs repeatedly and produces data to its output stream window. As illustrated in Figure 1(b), the input stream window accumulates N items. For each execution, the operator consumes 1 item (called consuming rate) from the input stream (dumped by the sliding window) and produces 1 item (called producing rate) to the output stream. The operator can read N tokens in the input stream window without consuming them. It does not alter the state of the input stream.

3.3 Component—Constructing the Hierarchical Dataflow Graph

The different operators are connected in the graph section of a composite body to construct the stream graph, passing parameters through input and output streams. *Composites* [7] provide a flexible mechanism to construct a structural dataflow program. A composite can be instantiated within other composites to build a larger dataflow graph. The program in Figure 1(a) shows an instance of `MovAver` that the input stream comes from operator `Source` and the output stream flows to operator `Sink`. Figure 2 shows a common example of composite and instantiation. The program defines a composite `M`, and instantiates two composite `M` using different streams and parameters. There will be two copies of the composite `M` in the expanded graph without any conflict. Composites can also be constructed in a nested fashion. This gives the programmer a flexible way to construct a large dataflow graph by code reuse.

4 Compiler Framework and Implementation on Multi-core

4.1 The Compiler Framework

Figure 3 illustrates the COStream compiler framework. The compiler front-end first translates a COStream program into an abstract syntax tree which describes the hierarchical structure of its composite. Since the composite structure can be parameterized, the compiler propagates constants and instantiates the composite hierarchically to a static flattened structure of op-

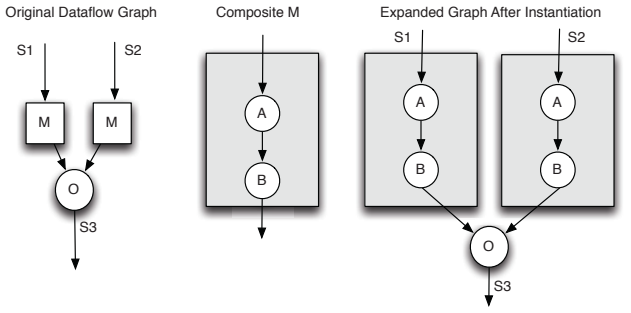


Figure 2: An example of Composite and Instantiation.

erators. Then, the compiler analyzes the dependencies of the operators by stream parameter passing to build a static stream graph. Each operator corresponds to a node of the stream graph. At this point, we can calculate the periodic schedule for the nodes of the stream graph.

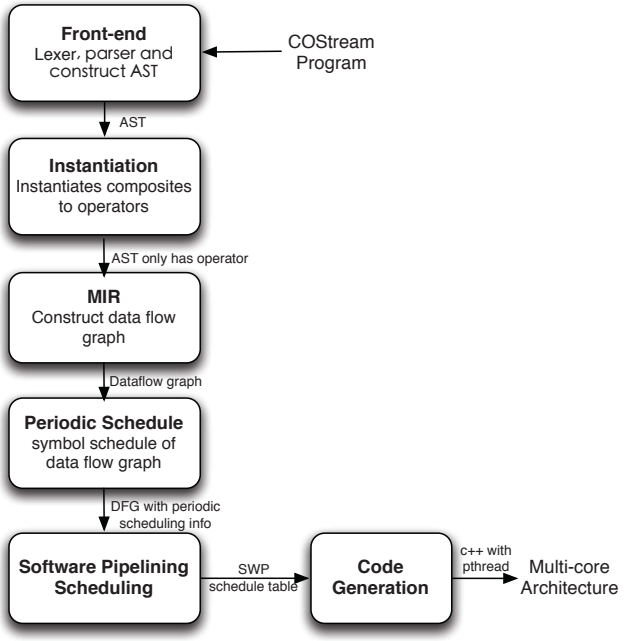


Figure 3: The COStream Compiler Framework.

The periodic schedule calculates the initialization and steady-state schedule for the data graph. A separate initialization schedule is needed because the sliding windows size may be larger than the sliding size (number of dumped items) and the stream graph must execute to fill the window before the steady-state execution. During initialization, each operator with a sliding window will execute until it leaves the window size-sliding items in its window. The steady-state schedule is periodic, and its execution must preserve the same number of live items on each channel as in the previous execution of the graph. In the compiler, the initialization

schedule is built by the symbolic execution of the stream graph. For the steady-state schedule, we use a Single Appearance Schedule (SAS) [1] for each operator. In the SAS schedule, each node appears exactly once and is surrounded by a loop denoting the number of times that it executes in one steady state execution of the stream graph.

After the periodic scheduling, the compiler performs a software pipelining schedule to exploit the parallelism of the stream program. It tries to divide the stream graph into partitions to balance the workload between them then schedule them to different cores and using pipeline to execute. Finally, the compiler generates C++ code with Pthread calls, which is compiled by a low-level C++ compiler to executable code for the multi-core architecture.

4.2 Software Pipelining Schedule

In theory, a synchronous dataflow dataflow program has infinite data inputs. The dataflow program can be seen as a set of connected operators wrapped by a loop. Thus, the loop optimization techniques like software pipelining scheduling can be used on the dataflow program. The problem is: How to schedule the dataflow graph to the different cores to parallelize the execution of the program and get the best performance? We use classic modulo scheduling technique [12] to deal with this problem. In modulo scheduling, there are two constraints we need to obey:

- `Resource constraints` For multi-core architecture, the resource is the core. Unlike instruction-level software pipelining, resources for each operator are homogeneous: Each operator of the data flow program can be scheduled to any one of the cores. The occupied duration of the resource is equated to the execution time of the operator that is scheduled to the core. Therefore, the resource constraints can be seen as a partition problem which targets the best load balance of different cores.

- `Dependency constraints` In a SDF program, there are two kinds of edges: normal dataflow edges and feedback loop edges, which correspond to the local dependencies and loop carried dependences. The feedback loop edge shows a great performance penalty in the SDF program and is rarely used [14]. Therefore, in this paper we only consider SDF programs without feedback loop edges.

We use the optimization framework in [16, 15] to get the software pipelining schedule. First, we use a multilevel k-way graph partitioning algorithm [10] to assign the operator to the core, and then we use a stage assignment algorithm to assign the stage to each operator. The details are explained in [16]. Figure 4 shows an example of software pipelining schedule. In Figure 4(a), after the scheduling, operator `A` is assigned to core 0, operators `B1` and split are assigned to core 1, and operators `B2`, `C` and `Join` are assigned to core 2. And then each operator is assigned with a stage number. Figure 4(b) shows the software pipelining schedule table. At the prologue of software pipelining, stages are activated sequentially, thus filling up the pipeline and enabling executions of data dependent operators belonging to earlier iterations concurrently with operators from later iterations. At the kernel of software pipelining, all stages are active, thus the pipeline is at steady state. At the epilogue, the stages are deactivating and the pipeline is drained.

4.3 Code Generation

In the backend, the compiler generates the code in three part for the COStream program: the computation node, the buffer allocation and the software pipelining codes. The compiler gener-

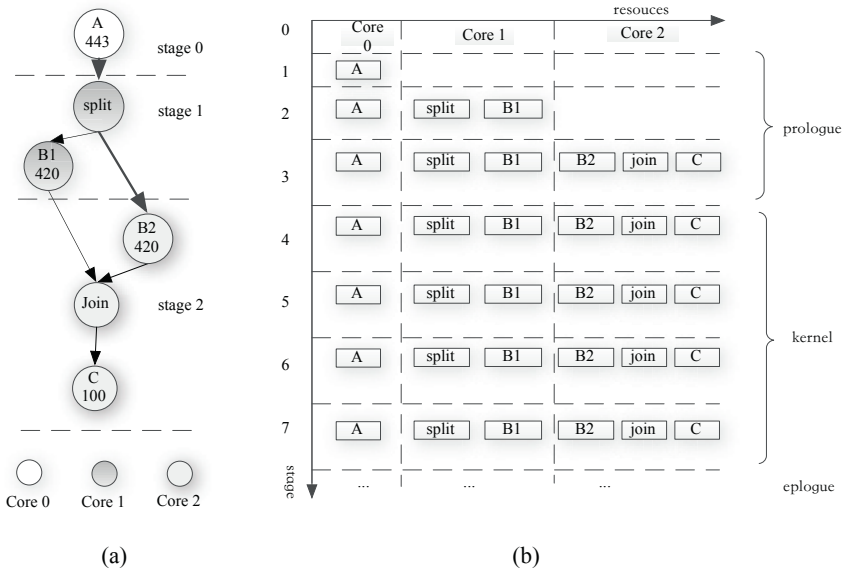


Figure 4: An example of sofware pipeling schedule (a) resouce allocation and stage assignmnet (b) software pipelining schedule table

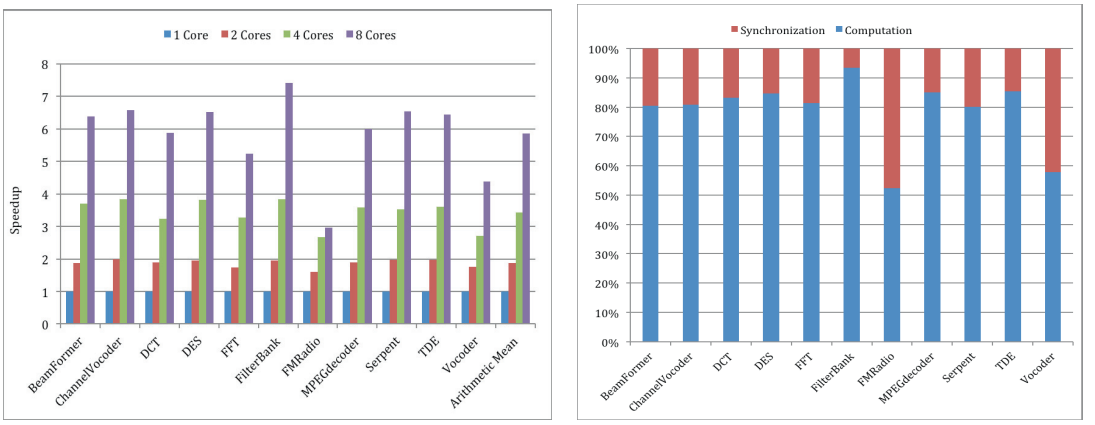
ates a separate function for each of the operators in the program. The kernel-only scheme [12] is used to generate the software pipelining code. At each stage, there will be a function call to the operator code which is scheduled at that stage in the software pipelining schedule table. Rotating buffer queues are used to allocate the data for software pipelining schedule, similar to the Modulo Variable Expansion [12] technique in traditional software pipelining. The rotating buffer queues make sure that different iterations can access the different data without any conflicts.

5 Experimental Results

We performed our experiments on a 2.4 GHz 8cores (2 way 4 cores) Intel Xeon processor, equipped with 4GB of DRAM. The low level compiler we used in our compiler backend is gcc v4.1. We ran our experiments on a Linux kernel v2.6.18 using the Pthread library.

The benchmarks tested in the experiments are from the StreamIt benchmarks [10]. Most of them are from the signal processing domain. We rewrote eleven benchmarks using the COStream language and used them as the inputs of our compiler. These benchmarks cover both computation intensive as well as data access intensive applications . In these benchmarks, the number of nodes in the dataflow graph scales from 17 to 116, and the number of edges varies from 5 to 63.

Figure 5(a) illustrates the speedup of the benchmarks running on 8 cores. For most benchmarks, the compiler achieves near linear speedup. There are six benchmarks obtaining over 6.4x speedup. The best one is achieving 7.4x speedup. The mean speedup for all the benchmarks is 5.9x. The performance of the program is mainly determined by three facts: workload balance, locality and the characteristics of the program itself (number of edges and workload of the node). We used the multilevel k-way partitioning [10] algorithm to perform load balancing while keeping connectivity. Therefore, most benchmarks can achieve consistent performance



(a) Benchmarks Speedup on 8 cores

(b) Benchmarks Computation and Synchronization Ratio

Figure 5: Results of the StreamIt Benchmarks Using COStream.

gains. The *FilterBank* benchmark obtains a 7.4× speedup. The reason is that the amount of work for each operator is almost the same and the graph has simple dataflow edges which provide the practitioner a good load balance and locality opportunities. For the *FMRadio* benchmark, the speedup obtained is only 3.0×: Although there are 31 operators, there is very little work to be performed for each operator, while there are lots of memory accesses. The *Vocoder* benchmark is the biggest benchmark which has 116 nodes and 63 edges, and several nodes have extra larger workload than others. It gives the partitioner a good challenge and makes the speedup only 4.4×.

Figure 5(b) shows the computation and synchronization ratio of the benchmarks. The compiler uses modulo software pipelining to exploit parallelism. After each stage, all the cores need to synchronize with each other which makes sure the data produced at this stage is written to the buffer. The compiler uses a sense-reversing barrier to implement the synchronization between different cores. The ratio reflects the performance difference. From the figure, we can see that *FMRadio* and *Vocoder* have a large part of synchronization overhead which is caused by the unbalanced workload and bad locality. *FilterBank* keeps about 5% sychronization overhead which achives the beast performance.

We also performed cache optimization for stream programs where applicable. Because stream programs execute their steady state schedule as a basis, they can repeat the steady state execution any time without change the semantics of the program. Hence we can keep as much data in cache as possible by repeating the steady state execution. For now, our cache optimization is implemented in a very simple way: We compute the biggest multiple that repeats the steady state execution such that the amount of data of the actor will not spilled out of cache. When an actor is running on a core, it almost fully uses the cache before switching to run another actor. Figure 6 shows results of cache optimization. Using this technique, we achieve up to 38.7% performance improvement. This is because repeating execution improves instruction and data reuse. Another reason is by executing multiple times it increases the computation and synchronization ratio and decreases the number of barriers in the pipelining schedule. *FMRadio* features the lowest computation and synchronization ratio in the benchmarks, achieving a 69.6% improvement.

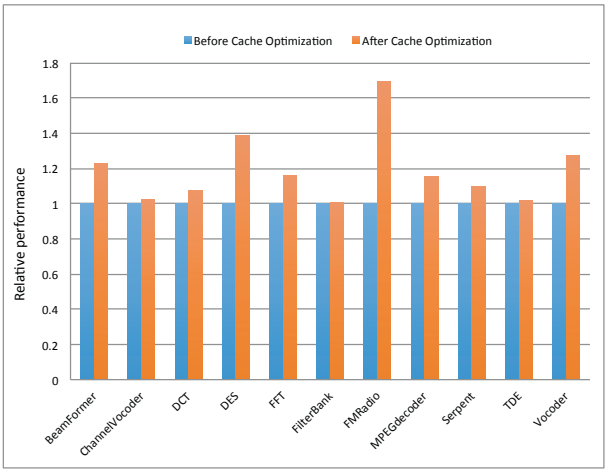


Figure 6: Cache Optimization for the StreamIt Benchmarks

6 Perspective—Applying Stream Programming Model to DDDAS

As we explained in Section 3, our stream programming model is based on dataflow. Data driven execution, the definition of events and the explicit expression of system resources in the parallel execution model make the stream programming model a promising model capable of addressing the many small-spontaneous-tasks-on-a-large-system challenge faced by Dynamic Data Driven Application Systems (DDDAS) applications.

In order to apply our stream programming model to DDDAS, we need to extend our system in the following ways. First, extending the stream programming model to abstract the interaction/interface with external devices such environment sensors and control elements. We can design a special component to describe the external devices and create an instance when each external device connects to the stream system. Second, we need to extend the stream programming model to support spontaneous attributes. This can be achieved by extending the programming model semantics in the COStream language. Third, a new software pipelining technology design must be performed to support spontaneous data flow execution.

7 Conclusion

In this paper we propose COStream, a dataflow programming language and compiler framework for multi-core architectures. Compared with previous dataflow languages, COStream introduces some new characteristics like windows, multiple inputs/outputs, stream variable passing and composites to improve programmability and data reuse. Our compiler uses the modulo software pipelining schedule to expoit parallelism. However, the barrier operations between each stage bring extra overhead for nodes with dependences. Our future work will mainly focus on extending the stream programming model to dynamic data-driven application.

Acknowledgments
The work is supported by U.S Air Force Office of Scientific Research under Grant No. FA9550-13-1-0213. Moreover, the work was supported by This research was supported by the NSF

through grants CCF-0833122, CCF-0925863, CCF-0937907 and OCI-0904534, the European FP7 project TERAFLUX, id. 249013, Department of Energy [Office of Science] under Award Number DE-SC0008717. The first author also thanks Prof. Junqing Yu for his support on the experimental design.

References

[1] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *VLSI Signal Processing*, 21(2):151–166, 1999.

[2] Alan L. Davis and Robert M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, 1982.

[3] Jack B. Dennis, Guang R. Gao, and Xiao X. Meng. Experiments with the fresh breeze tree-based memory model. *Computer Science - R&D*, 26(3-4):325–337, 2011.

[4] Jack B. Dennis, Guang R. Gao, Xiao X. Meng, Brian Lucas, and Joshua Slocum. The fresh breeze program execution model. In *PARCO*, pages 335–342, 2011.

[5] Guang R. Gao, Herbert H. J. Hum, and Jean-Marc Monti. Towards an efficient hybrid dataflow architecture model. In *PARLE (1)*, pages 355–371, 1991.

[6] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman P. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS*, pages 291–303, 2002.

[7] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark P. Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. Ibm streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7, 2013.

[8] R. A. Iannucci. Toward a dataflow/von neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 131–140, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[9] ETI Inc. Swarm: Scalable performance optimization for multi-core/multi-node. [online], 2011. White paper.

[10] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *SC*, page 28, 1998.

[11] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, 1999.

[12] B. Ramakrishna Rau, Michael S. Schlansker, and Parthasarathy P. Tirumalai. Code generation schema for modulo scheduled loops. In *MICRO*, pages 158–169, 1992.

[13] Joshua Suetterlein, Stéphane Zuckerman, and Guang R. Gao. An implementation of the codelet model. In *Euro-Par*, pages 633–644, 2013.

[14] William Thies and Saman P. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT*, pages 365–376, 2010.

[15] Haitao Wei, Junqing Yu, Huafei Yu, and Guang R. Gao. Minimizing communication in rate-optimal software pipelining for stream programs. In *CGO*, pages 210–217, 2010.

[16] Haitao Wei, Junqing Yu, Huafei Yu, Mingkang Qin, and Guang R. Gao. Software pipelining for stream programs on resource constrained multicore architectures. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2338–2350, 2012.

[17] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.