How to make stream processing more mainstream

Shuvra S. Bhattacharyya, Gordon Brebner, Johan Eker, Jörn W. Janneck, Marco Mattavelli, Mickaël Raulet

How to make stream processing more mainstream

Shuvra S. Bhattacharyya* Gordon Brebner†, Johan Eker‡,
Jörn W. Janneck†, Marco Mattavelli§, Mickaël Raulet¶
* *Dept. of ECE and UMIACS, University of Maryland, College Park, MD 20742, ssb@umd.edu*
† *Xilinx Research Labs, San Jose, CA 95123, {gordon.brebner,jorn.janneck}@xilinx.com*
‡ *Ericsson Research, Mobile Platforms, SE-221 83 Lund, Sweden, johan.eker@ericsson.com*
§ *Microelectronic Systems Lab, EPFL, CH-1015 Lausanne, Switzerland, marco.mattavelli@epfl.ch*
¶ *IETR/INSA Rennes, F-35043 Rennes, France, mraulet@insa-rennes.fr*

1. What it takes

Stream processing has a long history as a way of describing and implementing specific kinds of computational processes. So far, however, it has largely remained an exotic field of endeavor, with relatively small momentum compared to traditional von Neumann computing, and a large variety of programming models, languages, tools, and hardware realizations. However, as sequential machines cease to become faster over time, and future growth in computational speed will clearly derive from an increase in parallelism, the time has come for a general parallel programming model to supplant or complement the von Neumann abstraction.

Many modern forms of computation are very well suited to a stream-based description and implementation, such as complex media coding [1], network processing [2], imaging and digital signal processing (e.g., see [3], [4]), as well as embedded control [5]. Together with the move toward parallelism, this represents a huge opportunity for stream processing.

This paper shortly introduces a simple stream-based model and discusses some of its properties in the light of requirements for a general parallel programming model.

2. Features of a (fairly) general stream-based programming model

We base this discussion on our experience with stream-based languages such as CAL [6], but it applies equally to other similar models such as UTL [7], and at least to some degree to many other stream-based models and languages, all of which emphasize communication (rather than computation) as the main structuring principle. CAL has been used in a wide variety of applications and has been compiled to hardware and software implementations, [8], [9], and work on mixed HW/SW implementations is under way.

The fundamental entity of this model is an *actor*, also called dataflow actor with firing or *transactor*. It consumes and processes streams of incoming data items (*tokens*), and produces streams of such items as a result. In addition, this model has the following properties:

Strong encapsulation. Every actor completely encapsulates its own state together with the code that operates on it. No two actors ever share state, which means that an actor cannot directly read or modify another actor's state variables. The only way actors can interact is through streams, directed connections they use to communicate data tokens.

Explicit concurrency. A system of actors connected by streams is explicitly concurrent, since every single actor operates independently from other actors in the system, subject to dependencies established by the streams mediating their interactions. In addition to whatever parallelism tools may extract from a program, the user has a lot of control over the parallelism within such a stream-based application.

Asynchrony, untimedness. The description of the actors as well as their interaction does not contain specific real-time constraints (although, of course, implementations may).

User-defined transactions. Actor execute by making discrete steps, or *transactions*, which are either executed completely, or not at all. During each step, an actor may consume tokens, produce tokens, and

modify its internal state. Users specify actors by listing the possible transactions it may perform. The description of each transaction includes the conditions that must be satisfied for its completion (including for instance the number of tokens consumed and produced during the transaction), and tools can use that information to analyze actor behavior at compile time (e.g., see [4], [10]).

3. Discussion

We can now discuss some of the implications of these features.

Scalable parallelism. In parallel programming, the number of things that are happening at the same time can scale in two ways: It can increase with the size of the problem or with the size of the program. Scaling a regular algorithm over larger amounts of data is a relatively well-understood problem, while building programs such that their parts execute concurrently without much interference is one of the key problems in scaling the von Neumann model. The explicit concurrency of the actor model provides a straightforward parallel composition mechanism that tends to lead to more parallelism as applications grow in size, and scheduling techniques permit scaling concurrent descriptions onto platforms with varying degrees of parallelism.

Modularity, reuse. The ability to create new abstractions by building reusable entities is a key element in every programming language. For instance, object-oriented programming has made huge contributions to the construction of von Neumann programs, and the strong encapsulation of actors along with their hierarchical composability offers an analog for parallel programs.

Scheduling. In contrast to procedural programming languages, where control flow is made explict, stream-based languages emphasize explicit specification of concurrency. Therefore, when mapping streaming programs onto programmable processors, it is necessary to *schedule* computation and communication in a way that is consistent with the original program while achieving the desired implementation constraints, e.g. on latency, throughput, and memory requirements. Scheduling is a hard problem in the general case [4], [10] which can be greatly facilitated by the explicit concurrency exposed by well-designed streaming representations as well as the use of effective scheduling heuristics that are matched to the application and the implementation target. Encapsulation and asynchrony provide the scheduler with the degrees of freedom it needs to operate, while user-defined transactions give the user control over the granularity of the schedule.

Portability. Rallying around the pivotal and unifying von Neumann abstraction has resulted in a long and very successful collaboration between processor architects, compiler writers, and programmers. Yet, for many highly concurrent programs, portability has remained an elusive goal, often due to their sensitivity to timing. The untimedness and asynchrony of stream-based programming offers a solution to this problem. The portability of stream-based programs is evidenced by the fact that programs of considerable complexity and size can be compiled to competitive hardware [8] as well as software [9], which suggests that stream-based programming might even be a solution to the old problem of flexibly co-synthesizing different mixes of hardware/software implementations from a single source.

Adaptivity. The success of a stream programming model will in part depend on its ability to configure dynamically and to virtualize, i.e. to map to collections of computing resources too small for the entire program at once. Moving parts of a program on and off a resource requires encapsulation, i.e. a clear distinction between those pieces that belong to the parts to be moved and those that do not. The absence of time from the specification and the reliance on asynchronous communication are helpful, too, since virtualization and dynamic configuration make it difficult to precisely predict timing properties. Finally, the transactional execution of actors generates points of *quiescence*, the moments between transactions, when the actor is in a defined and known state that can be safely transferred into or out of the computing resource.

4. Conclusion

We believe that the move towards parallelism in computing and the growth of application areas that lend themselves to stream-based processing present a huge opportunity for introducing a stream-based programming model that could supplant or at least complement von Neumann computing in many fields. We have discussed some properties that we believe a more mainstream stream processing model must possess, viz. strong encapsulation, explicit concurrency, asynchrony and untimedness, and user-defined transactions.

References

[1] J. Thomas-Kerr, J. W. Janneck, M. Mattavelli, I. Burnett, and C. Ritz, "Reconfigurable Media Coding: Self-describing multimedia bitstreams," in *Proceedings IEEE Workshop on Signal Processing Systems—SiPS 2007*, October 2007, pp. 319–324. 1

[2] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 217–231, 1999. 1

[3] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23. 1

[4] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for DSP," *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 849–875, September 2000. 1, 2

[5] S. S. Bhattacharyya and W. S. Levine, "Optimization of signal processing software for control system implementation," in *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design*, Munich, Germany, October 2006, pp. 1562–1567, invited paper. 1

[6] J. Eker and J. W. Janneck, "CAL language report," Electronics Research Lab, University of California at Berkeley, Technical Memo UCB/ERL M03/48, December 2003. 1

[7] K. Asanovic, "Transactors for parallel hardware and software co-design," in *International High Level Design Validation and Test Workshop*. IEEE, November 2007. 1

[8] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: an MPEG-4 simple profile decoder case study," in *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS)*, 2008. 1, 2

[9] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow programs: an MPEG-4 simple profile decoder case study," in *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS)*, 2008. 1, 2

[10] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000. 2