Automatic Deployment of Streaming Applications on Hybrid Architectures

Roger D. Chamberlain
Mark A. Franklin

Roger D. Chamberlain and Mark A. Franklin, "Automatic Deployment of Streaming Applications on Hybrid Architectures," in *Proc. of 11th High Performance Embedded Computing Workshop*, September 2007.

Dept. of Computer Science and Engineering
Washington University
Campus Box 1045
One Brookings Dr.
St. Louis, MO 63130-4899

Automatic Deployment of Streaming Applications on Hybrid Architectures

Roger D. Chamberlain and Mark A. Franklin
Dept. of Computer Science and Engineering, Washington University in St. Louis
{roger,jbf}@wustl.edu

Introduction

Streaming computation models have received considerable attention recently as a convenient way to reason about and develop data intensive applications. Example applications include signal processing, cryptography, biosequence analysis and graphics. Associated streaming languages have been proposed (e.g., [1,2]) and some are now in production use.

Embedded applications are often streaming in nature and are frequently deployed on *hybrid* architectures that include multiple types of computing resources, such as chip multiprocessors and FPGAs. Such architectures can potentially exploit the unique features of a resource and can exploit pipelining and parallelism to achieve higher overall performance. However, applications are difficult to design, analyze and deploy on real hybrid systems.

Lee [3] has argued that coordination languages represent a good mechanism for reasoning about concurrency and representing dependence between computational components. In the hybrid system domain, Franklin et al. [4] have proposed, and subsequently implemented, the **X** coordination language. Common to both of the above is the use of data flow semantics between “kernels” or “blocks” that specify undecomposable computations that are to be mapped to individual computing resources (e.g., processor, FPGA, DSP, etc.). Unlike some streaming languages, the computations associated with an **X** block may be expressed in one of a number of languages, including standard languages such as C/C++ and hardware description languages such as VHDL or Verilog. This removes the necessity for learning an entirely new language for computational kernels.

Once blocks are defined and their interactions expressed in **X**, the development environment permits exploration of performance issues [5] and easy redefinition of block interactions. Once an **X** description has been completed, the blocks and interconnections must be mapped to appropriate and available hardware resources and, finally, deployed (e.g., downloading the block binaries and bitfiles to the actual resources). Here, we describe the capabilities of X-Dep, the deployment tool for applications authored in the **X** language. X-Dep is a part of the Auto-Pipe application development environment [4].

Example Application: Sorting

As an example application, we will use sorting to illustrate the capabilities of the X-Dep tool. A common approach to sorting is to first sort groups of records that are subsequently merged in a later step. This approach exploits record locality, both on chip multiprocessors and on FPGAs. Sorting can be expressed as a streaming application as shown in Figure 1.

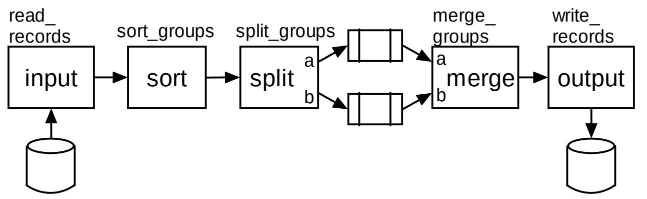

Figure 1: Sorting application expressed as a stream.

The `sort` block sorts fixed size groups of records and the `split` block routes successive groups out of `sort` into distinct inputs to `merge`, which performs a merge sort. Queues are shown between `split` and `merge`, however, they actually exist along all of the arcs. The **X** language source code for the sorting application of Figure 1 follows:

```
block app_1 {
    read_records  input;
    sort_groups   sort;
    split_groups  split;
    merge_groups  merge;
    write_records output;

    input -> sort -> split;
    split.a -> merge.a;
    split.b -> merge.b;
    merge -> output;
};
```

Once the block labels have been declared, the topology of the streaming application is described. In addition to the above **X** code, one must also implement each of the block types (`read_records`, `sort_groups`, etc.) in a language supported by the resources on which it is a candidate to be deployed (e.g., C/C++ for processors, VHDL or Verilog for FPGAs).

Alternative Streaming Sort Application

If multiple resources are available so that separate copies of the `sort` block can execute in parallel, an alternative streaming topology might be considered (Figure 2).

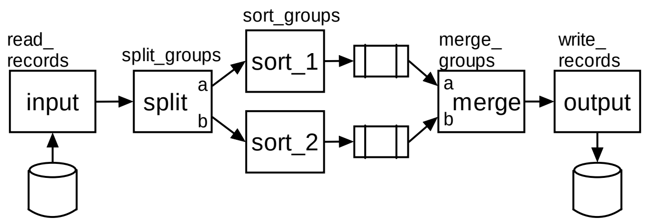

Figure 2: Alternative sort application.

This research is supported by NSF grant CCF-0427794.

Note that in this case, blocks `sort_1` and `sort_2` are operating on independent groups of records. The **X** source code for this alternative approach is shown below.

```
block app_2 {
    ...
    sort_groups   sort_1, sort_2;
    ...
    input -> split;
    split.a -> sort_1 -> merge.a;
    split.b -> sort_2 -> merge.b;
    merge -> output;
};
```

Note the limited differences in the expressions of these two applications. Moving from one application to the other with **X** is straightforward (as it should be for this simple a transformation). When the topology is not expressed in a high-level coordination language, this type of application transformation is often complicated and error-prone.

Deployment on Hybrid Architectures

To illustrate deployment of an **X** described system on a hybrid architecture, we use a system available in our laboratory. The system consists of 4 processor cores (2 dual-core AMD Opterons) and a Xilinx Virtex-II 6000 attached via the PCI-X bus (on an Avnet board). These resources are declared as follows:

```
resource proc[4] is C_x86;
resource FPGA is HDL_Avnet;
```

The resource type `C_x86` indicates that the blocks mapped to this resource type are expressed in C/C++ for an x86 processor core. The resource type `HDL_Avnet` similarly indicates that blocks mapped to this resource type are expressed in Verilog or VHDL for our Avnet board.

At this point, blocks from the application can be mapped to the available resources. In the following mapping (for the application of Figure 1), the blocks are divided across the available processors, leaving the FPGA unused.

```
map proc[1] = {app_1.input};
map proc[2] = {app_1.sort};
map proc[3] = {app_1.split,
               app_1.merge};
map proc[4] = {app_1.output};
```

A second candidate mapping is to put both file I/O blocks on one processor core, the `sort` block on the FPGA, and the `split` and `merge` blocks on a second processor core. This leaves two processor cores available for other tasks.

```
map proc[1]={app_1.input, app_1.output};
map proc[2]={app_1.split, app_1.merge};
map FPGA   ={app_1.sort};
```

With the system we have developed, the deployment of compiled block computational code and all data delivery across block interfaces is automatically handled by the development environment. In the first mapping above, records to be sorted are delivered from processor core 1 to processor core 2 without any additional specification required on the part of the developer (here using the shared memory system). If the two processors were connected via a network, the necessary communications protocols would be invoked for data delivery across the network.

In moving from the first to the second mapping, the developer has indicated that records to be sorted must now be delivered from processor core 1 to the FPGA. Here, the data is moved across the PCI-X bus to the FPGA board, again without any additional specification required on the part of the developer.

A third example mapping is for the alternative sort topology of Figure 2. Here, I/O is mapped to one processor core, each sort block is assigned to a separate processor core, and `split` and `merge` are assigned to the FPGA.

```
map proc[1]={app_2.input, app_2.output};
map proc[2]={app_2.sort_1};
map proc[3]={app_2.sort_2};
map FPGA   ={app_2.split, app_2.merge};
```

Again, data delivery between processor cores and to/from the FPGA is automatically handled by the development environment.

Conclusions

We have presented an illustrative example of the capabilities of the **X** language and the Auto-Pipe development environment. The system is capable of describing, mapping, and deploying streaming applications to hybrid architectures. In addition to the sorting application described, several additional applications have been and are being authored in **X**. These include encryption (3DES), an astrophysics signal processing pipeline (from the VERITAS project [6]), and an N-body simulation application. While the current system is limited to processors and FPGAs as compute resources, we are planning extensions to support graphics processors, DSPs, heterogeneous chip multi-processors (e.g., Cell), etc. in the future.

References

[1] W. Thies, M. Karczmarek, and S.P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proc. of 11th Int'l Conf. on Compiler Construction*, pp. 179-196, 2002.

[2] M.B. Gokhale, J.M. Stone, J. Arnold, and M. Kalinowski, "Stream-Oriented FPGA Computing in the Streams-C High Level Language," in *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, 2000.

[3] E.A. Lee. "The problem with threads," *IEEE Computer*, **39**(5):33-42, May 2006.

[4] M.A. Franklin, E.J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer, "Auto-Pipe and the X Language: A Pipeline Design Tool and Description Language," in *Proc. of Int'l Parallel and Distributed Processing Symp.*, April 2006.

[5] S. Gayen, E.J. Tyson, M.A. Franklin, and R.D. Chamberlain, "A Federated Simulation Environment for Hybrid Systems," in *Proc. of Principles of Advanced and Distributed Simulation Workshop*, June 2007.

[6] T. Weekes et al. "VERITAS: the very energetic radiation imaging telescope array system," *Astroparticle Physics*, **17**(2):221-243, May 2002.