# A Common Machine Language for Grid-Based Architectures

William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jeremy Wong, Henry Hoffmann, Matthew Brown and Saman Amarasinghe

{thies, karczma, mgordon, dmaze, jnwong, hank, morris, saman}@lcs.mit.edu

Laboratory for Computer Science, Massachusetts Institute of Technology

## 1. INTRODUCTION

A *common machine language* is an essential abstraction that allows programmers to express an algorithm in a way that can be efficiently executed on a variety of architectures. The key properties of a common machine language (CML) are: 1) it abstracts away the idiosyncratic differences between one architecture and another so that a programmer doesn't have to worry about them, and 2) it encapsulates the common properties of the architectures such that a compiler for any given target can still produce an efficient executable.

For von-Neumann architectures, the canonical CML is C: instructions consist of basic arithmetic operations, executed sequentially, which operate on either local variables or values drawn from a global block of memory. C has been implemented efficiently on a wide range of architectures, and it saves the programmer from having to adapt to each kind of register layout, cache configuration, and instruction set.

However, recent years have seen the emergence of a class of grid-based architectures [2, 3, 4] for which the von-Neumann model no longer holds, and for which C is no longer an adequate CML. The design of these processors is fundamentally different in that they are conscious of wire delays–instead of just arithmetic computations–as the barriers to performance. Accordingly, grid-based architectures support fine-grained, reconfigurable communication between replicated processing units. Rather than a single instruction stream with a monolithic memory, these machines contain multiple instruction streams with distributed memory banks.

Though C can still be used to write efficient programs on these machines, doing so either requires architecture-specific directives or a very smart compiler that can extract the parallelism and communication from the C semantics. Both of these options renders C obsolete as a CML, since it fails to hide the architectural details from the programmer and it imposes abstractions which are a mismatch for the domain.

To bridge this gap, we propose a new common machine language for grid-based processors: StreamIt. The StreamIt language makes explicit the large-scale parallelism and regular communication patterns that these architectures were designed to exploit. A program is represented not as a monolithic memory and instruction stream, but rather as a composition of autonomous filters, each of which contains its own memory and can only communicate with its immediate neighbors via high-bandwidth data channels. In addition, StreamIt provides a low-bandwidth messaging system that filters can use for non-local communication. We believe that StreamIt abstracts away the variations in grid-based processors while encapsulating their common properties, thereby enabling compilers to efficiently map a single source program to a variety of modern processors.

## 2. THE STREAMIT LANGUAGE

In this section we provide a brief overview of the StreamIt language; please see [5] for a more detailed description. The current version of StreamIt is legal Java syntax to simplify our presentation and implementation, and it is designed to support only streams with static input and output rates.

### 2.1 The Stream Graph

The basic unit of computation in StreamIt is the `Filter`. An example of a Filter is the `Adder`, a component of our software radio (see Figure 2). Each `Filter` contains an `init` function that is called at initialization time; in this case, the `Adder` records `N`, the number of items it should add at once. The `work` function describes the most fine grained execution step of the filter in the steady state. Within the `work` function, the filter can communicate with neighboring blocks using the `input` and `output` channels, which are typed FIFO queues declared within the `init` function. These high-volume channels support the intuitive operations of `push(value)`, `pop()`, and `peek(index)`, where `peek` returns the value at position *index* without dequeuing an item.

The basic construct for composing filters into a communicating network is a `Pipeline`, such as the `Radio` in Figure 2. Like a `Filter`, a `Pipeline` has an `init` function that is called upon its instantiation. However, there is no `work` function, and all input and output channels are implicit; the stream behaves as the sequential composition of filters that are specified with successive calls to `add` from within `init`.

There are two other stream constructors besides `Pipeline`: `SplitJoin` and `FeedbackLoop`. The former is used to specify independent parallel streams that diverge from a common *splitter* and merge into a common *joiner* (see the Equalizer in Figure 2). There are two kinds of splitters: 1) Duplicate, which replicates each data item and sends a copy to each parallel stream, and 2) RoundRobin($w_1, \ldots, w_n$), which sends the first $w_1$ items to the first stream, the next $w_2$ items to the second stream, and so on. RoundRobin is the only joiner type. The parallel streams are specified by successive calls to `add`; the $i$'th call sets the $i$'th stream in the SplitJoin.

StreamIt differs from other languages in that it imposes a well-defined structure on the streams: all stream graphs are built out of a hierarchical composition of Pipelines, SplitJoins, and FeedbackLoops. This structure enables the stream to be mapped efficiently onto a grid target, since all of the communication is between neighboring filters. Moreover, we are developing fission and fusion algorithms that can, for example, collapse a large Pipeline into a single filter for execution on a single processor, thereby allowing us to adjust the granularity of the stream graph to match the granularity of a given target.

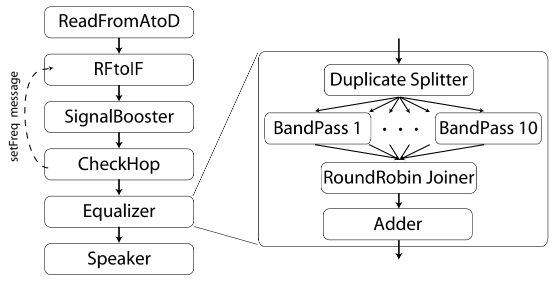

**Figure 1: Block diagram of a software radio.**

```
class Adder extends Filter {
  int N;

  void init(int N) {
    setInput(Float.TYPE); setOutput(Float.TYPE);
    setPush(1); setPop(N);
    this.N = N;
  }

  void work() {
    float sum = 0;
    for (int i=0; i<N; i++) {
      sum += input.pop();
    }
    output.push(sum);
  }
}

class Equalizer extends Pipeline {
  void init(int BANDS) {
    add(new SplitJoin() {
      void init() {
        int bottom = 2500;
        int top = 5000;
        setSplitter(Duplicate());
        for (int i=0; i<BANDS; i++, bottom*=2, top*=2) {
          add(new BandPassFilter(bottom, top));
        }
        setJoiner(RoundRobin());
    }});
    add(new Adder(BANDS));
  }
}

class CheckHop extends Filter {
   int DELAY;
   RFtoIFPortal portal;

   void init(RFtoIF rf2if, int DELAY) {
      setInput(Float.TYPE); setOutput(Float.TYPE);
      setPush(N); setPop(N);
      this.portal = new RFtoIFPortal(rf2if);
      this.DELAY = DELAY;
   }

   void work() {
      boolean hopped = /* calculate if hopped */
      float newFreq =  /* detect new frequency */
      if (hopped) {
         portal.setFreq(newFreq, DELAY);
      }
   }
}

class Radio extends Stream {
   void init() {
      add(new ReadFromAtoD());
      RFtoIF rf2if = add(new RFtoIF());
      add(new SignalBooster());
      add(new CheckHop(rf2if, 256));
      add(new Equalizer(10));
      add(new Speaker());
   }
}
```

**Figure 2: StreamIt code for a software radio.**

## 2.2 Messages

StreamIt provides a dynamic messaging system for passing irregular, low-volume control information between non-neighboring filters. Messages are sent from within the body of a filter's `work` function, perhaps to change a parameter in another filter. For example, in the `CheckHop` filter of our software radio example (Figure 2), a message is sent upstream to change the frequency of the receiver if the downstream component detects that the transmitter is about to change frequencies. The sender can continue to execute while the message is en route, and the `setFreq` method will be invoked in the receiver with argument `newFreq` when the message arrives. Since message delivery is asynchronous, there can be no return value; only void methods can be message targets.

The central aspect of the messaging system is a sophisticated timing mechanism that allows filters to specify when a message will be received relative to the flow of information between the sender and the receiver. This mechanism enables a consistent definition of message delivery timing across varying architectures that have no global clock. For example, the `CheckHop` filter sends a message with a latency of `DELAY`. This means that the target will receive the message when it is processing the data item that the the `CheckHop` filter sees in `DELAY` executions of its own work function. That is, the sender of a message specifies the delivery timing in terms of its own local time; the translation to the receiver's time is done relative to data items in the stream. This timing mechanism is important not only for portability, as it is independent of any architecture's clock or topology, but also for programmability, as it is often important to synchronize events with wavefronts of data in the stream.

## 3. STATUS AND CONCLUSIONS

We have implemented a fully-functional prototype of the StreamIt optimizing compiler that targets both uniprocessors and Raw [1]. The compiler exploits the structure of the stream graph to perform load-balancing transformations that can improve performance on Raw by over 300%.

We believe that StreamIt represents a viable common machine language for grid-based architectures. It abstracts away the target's granularity, memory layout, and network interconnect, while capturing the notion of replicated processors that communicate in regular patterns. With this representation, we believe that the StreamIt compiler will match the performance of C code that was hand-tailored for a given grid-based machine.

## 4. REFERENCES

[1] Michael Gordon *et al.* A Stream Compiler for Communication-Exposed Architectures. MIT Tech. Memo TM-627, Cambridge, MA, March, 2002.

[2] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular recongurable architecture. In *ISCA 2000*, Vancouver, BC, Canada.

[3] Elliot Waingold *et al.* Baring it all to software: The raw machine. MIT Tech. Report TR-709, Cambridge, MA, 1997.

[4] K. Sankaralingam, R. Nagarajan, S.W. Keckler, and D.C. Burger. A Technology-Scalable Architecture for Fast Clocks and High ILP. UT Austin, Tech. Report TR-01-02, 2001.

[5] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the Int. Conf. on Compiler Construction, to appear*, Grenoble, France, 2002.