International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation.
July, 2008. Greece. pp. 10-17

A Parameterized Dataflow Language Extension for Embedded Streaming Systems

Yuan Lin, Yoonseo Choi, Scott Mahlke, and Trevor Mudge
Advanced Computer Architecture Laboratory
University of Michigan at Ann Arbor

Chaitali Chakrabarti
Department of Electrical Engineering
Arizona State University

Abstract—**Many embedded DSP systems can be characterized as streaming applications. Imperative programming languages are ill suited for describing the concurrency within these DSP systems. SPEX is a language extension designed to let the programmers describe the inherent parallelism within DSP systems. In this paper, we highlight SPEX's language features for describing the streaming computation and communication patterns of DSP systems, and allowing the compiler to generate efficient code for embedded DSP architectures. This language extension is based on the parameterized dataflow computation model, with modifications to better describe DSP systems' complex streaming patterns. SPEX is applied as an extension onto the C++ programming language. It consists of a set of language constructs for describing the semantics of parameterized dataflow computations, and a set of language restrictions for helping the embedded compilation process. In this paper, the W-CDMA wireless protocol is used as our case study.**

I. Introduction

Within the past twenty years, engineers have designed increasingly complex multimedia DSP systems in order to satisfy our growing appetite for faster and more sophisticated multimedia content. These increasing computational requirements have motivated the design of embedded SoC DSP architectures. Software development for uniprocessor DSPs is hard, and SoC DSP architectures make this hard problem even harder. There is a clear need for better language support to help manage the complexity of mapping DSP systems onto DSP hardware. SPEX (Signal Processing EXtension) is a language extension designed to address these problems for embedded streaming DSP systems. It has two design objectives: allowing programmers to express the inherent parallelism within streaming DSP systems, and providing an efficient interface for the compiler to generate code for embedded DSP hardware. It is designed to support all aspects of embedded DSP computations. This includes vector arithmetic operations to describe DSP computations, dataflow constructs to describe streaming computations, and real-time constructs to describe real-time operations and deadlines. A summary of SPEX's language features can be found in [12]. This paper is focused on SPEX's dataflow extension for supporting streaming computations and communications.

Parameterized Dataflow Computation Model. Dataflow computation models have been proposed to describe streaming computations. However, many streaming DSP systems also have critical control flow operations. In between long episodes of streaming computation, DSP systems intermittently reconfigure their streaming patterns to account for changes from the users and the environments. SPEX is based on the parameterized dataflow (PDF) computation model, where the dataflow is described with a set of parameters. Each parameter is a variable with a finite set of possible values, describing a set of possible dataflow configurations. We propose a three-stage run-time execution model to provide efficient computation on embedded multi-core hardware. During the first stage, a static dataflow is initialized by assigning a constant value to each parameter variable. The second stage is the stream computation using a compiler-generated static synchronous dataflow schedule. The third stage finalizes the stream computation with updates to the dataflow variables and states.

Streaming Communication Model. Although parameterized dataflow model is good for describing reconfigurable streaming computation, its First-In First-Out (FIFO) communication pattern is inadequate for describing DSP system's complex streaming communication patterns. Therefore, we propose a modified pseudo-dataflow computation model where data can be shared among dataflow actors. Complex streaming patterns can be constructed using these actors as basic building blocks.

SPEX Language Extension. Parameterized computation models have been proposed before for modeling DSP systems [1]. However, given the popularity of existing languages such as C and C++, it is challenging for programmers to adopt a completely new concurrent programming paradigm. SPEX aims to reduce this challenge by implementing the parameterized dataflow model as a language extension to the familiar C++ language syntax. To provide efficient code generation for embedded DSP architectures, we also find that some of C++'s language features cannot be supported or must adopt different semantic meanings consequently. Even though SPEX is applied to C++ in this study, it is general enough to be applied to any programming language. The dataflow extension consists of two parts: a set of language primitives and constructs for describing the parameterized dataflow model and a set of language restrictions to limit the expressiveness of the host language.

The remainder of this paper is organized as follows. Section II provides our analysis of the operation characteristics of DSP systems, our rationale for using the parameterized dataflow computation model, and our modifications to the

978-1-4244-1985-2/08/$25.00 ©2008 IEEE

Authorized licensed use limited to: University of Michigan Library. Downloaded on January 13, 2009 at 11:26 from IEEE Xplore. Restrictions apply.

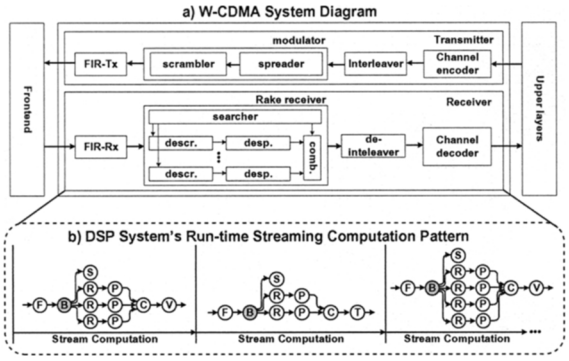


Fig. 1. Part a: W-CDMA System Level Diagram. W-CDMA is used as the ongoing example for SPEX in this study. Part b: DSP system run-time streaming computation pattern. The receiver may use different number of rake fingers (denoted by the R and P nodes) and different channel decoding algorithms (denoted by the T and V nodes). Shaded B nodes are memory buffers.

dataflow model for supporting streaming communication patterns. Section III describes SPEX, a language extension based on our modified parameterized dataflow computation model.

II. Modeling DSP Systems

In this section, we first describe our rationale for using the parameterized dataflow model for streaming computation. We then describe our rationale for modifying the dataflow model for streaming communication. We illustrate the features of SPEX through the W-CDMA wireless protocol's physical layer processing [8]. Figure 1a shows the system-level diagram of W-CDMA. The receiver consists of a FIR filter, rake receiver, interleaver, and channel decoder. These algorithm kernels are connected in a feed-forward pipeline.

A. Modeling Streaming Computation

Streaming Computation in DSP systems. In streaming applications, functions are organized in pipeline-like computation chains, and data is streamed through the pipeline in a sequential order. In between long episodes of streaming computation, DSP systems intermittently reconfigure their streaming patterns to account for changes from the users, the environment, and received inputs. Most DSP systems support multiple operation modes that are optimized for different services. These include changes in streaming rates, dataflow configurations, and algorithm kernels. For example, W-CDMA supports multiple data transmission rates ranging from 15Kbps to 2Mbps. The lower data rates are used for voice communications, and the 2Mbps is used for high-speed data communications. These different data communication rates also require different DSP algorithms and different stream configurations. Figure 1b describes this periodic reconfiguration in the streaming computation. During run-time streaming computation, the receiver may use different numbers of rake fingers (denoted in the figures by the R and P nodes) and different channel decoding algorithms (denoted in the figure by the T and V nodes).

Parameterized Dataflow Computation Models. The concurrent dataflow model has been used to describe streaming computations. A dataflow graph consists of a set of actors (or nodes) interconnected together with edges. Each edge contains both input and output stream rates for the source and destination actors. An actor's stream rates correspond to the amount of data consumed and produced per invocation. In particular, synchronous dataflow (SDF) has received considerable attention as the computation model for compilation onto multicore architectures [6]. SDF is a type of dataflow model where the dataflow properties are defined statically. This allows the run-time execution schedule to be generated statically during compile-time [10]. Embedded systems usually have tight performance constraints and limited run-time scheduling support. Many of these embedded systems also use scratchpad memories instead of cache, where memory management is a software problem. Compiler-generated execution schedules are favorable because they require less run-time scheduling and memory management overhead. However, because of its statically defined dataflow properties, SDF is too restrictive to describe the run-time reconfigurations of complex DSP systems. An ideal computation model for embedded systems should have the run-time efficiency of the SDF, while also providing enough flexibility to describe run-time reconfigurations.

SPEX is based on a more dynamic dataflow, namely the parameterized dataflow (PDF) computation model [1]. In PDF, dataflow attributes are described with parameters instead of constants. A parameter is a variable with a finite set of discrete values. Our choice of using the PDF is motivated by the fact that most DSP systems only have a finite set of discrete operating modes. These configurations in the dataflow can be adequately captured by a set of parameters with discrete values. We find that the following set of four dataflow properties should be parameterized to describe DSP systems' streaming computation.

- **Variable Dataflow Rates:** The input and output stream rates of dataflow actors may take on a range of values.
- **Conditional Dataflow:** Conditional dataflow is supported by using parameters to describe the branching conditions.
- **Number of Dataflow Actors:** Parameters can be used to fire a subset of the actors in a dataflow graph during run-time.
- **Streaming Size:** The number of data elements streamed per invocation should also be defined with parameters.

Run-time Execution Model and Compilation Support. Dataflow graphs are executed on hardware through run-time schedules. The schedule may be statically determined by a compiler or dynamically generated by a run-time scheduler. As mentioned before, SDF is good for embedded architectures because compiler-generated execution schedules require less run-time resources. With SPEX's PDF computation model, we propose a three stage run-time execution model: 1) dataflow initialization, 2) dataflow execution, and 3) dataflow finalization, as shown in Figure 2. These three stages are executed for

Authorized licensed use limited to: University of Michigan Library. Downloaded on January 13, 2009 at 11:26 from IEEE Xplore. Restrictions apply.

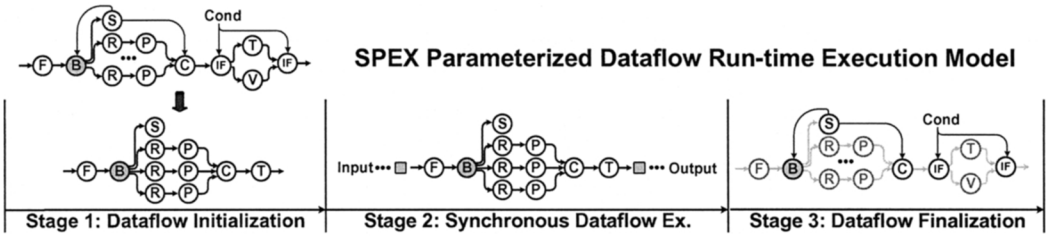


Fig. 2. PDF execution model consists of three steps. Step 1, the parameterized dataflow graph is constrained into a synchronous dataflow graph. Step 2, the dataflow is executed following a static compile-time schedule. Step 3, PDF graph's data and states are updated with the most recent computed values.

every PDF graph invocation. During the initialization stage, the parameters are set to constant values, effectively constraining a PDF graph into a SDF graph. The dataflow execution stage follows a compiler-generated schedule for this SDF graph. The finalization stage updates the dataflow variables and states with the results of the dataflow computation. This three stage PDF execution model provides the best of both worlds; it still maintains the efficiency of SDF execution schedules, while also provides the flexibility to reconfigure the dataflow through initialization and finalization stages.

The PDF compilation process can be divided into three steps: identifying SDF configurations, scheduling each SDF configuration, and generating run-time control code to select one of the SDF schedules. Because PDF graphs are parameterized, the compiler can identify all possible run-time synchronous dataflow configurations by iterating through the parameters. Each of these synchronous dataflow configuration can be compiled and scheduled using existing dataflow scheduling algorithms. Finally, control code is generated within PDF graphs' initialization and finalization stages to select the appropriate synchronous dataflow schedule during run-time.

B. Modeling Streaming Communication

Even though the parameterized dataflow model is good at describing streaming computation, many DSP systems have complex streaming communication patterns that cannot be accurately described with the dataflow graph's one-dimensional FIFO communication edges. The following is a list of communication patterns that are needed.

- **Multi-dimensional Streaming Patterns:** Many DSP systems operate on vectors and matrices, which require memory buffers with multi-dimensional streaming patterns. For example, a vector FIFO buffer may have two different streaming attributes: the streaming pattern within each vector element and the streaming pattern among the buffer vector elements.
- **Non-sequential Streaming Patterns:** Many DSP algorithms do not follow strict FIFO streaming order. For example, a complex filter may access the real or the imaginary components of an array of complex numbers in strided sequential order. An interleaver may access an array in pre-computed random order.
- **Decoupled Streaming:** Many DSP systems consist of multiple decoupled dataflow computations. These decoupled dataflow computations may still be connected through buffers, but they may operate asynchronously from each other. For example, in a W-CDMA receiver, the front-end filter must operate under the periodic real-time deadline. The data gets down-converted into a lower data rate ranging from 15Kbps for voice communication up to 2Mbps for data communication. The output is then run through a backend error decoder that does not have strict real-time deadline requirements. Many decoder implementations do not operate in sync with the front-end filter.
- **Shared Memory Buffers:** DSP systems have memory buffers that are shared between multiple readers and writers. Dataflow edges are FIFO queues that support queue push and pop. Push and pop couple two separate operations for each data element: memory allocation/deallocation and memory read/write. Shared memory buffers requires decoupled operations for memory allocation/deallocation and memory read/write.

Previous works have attempted to address these different streaming patterns by proposing different dataflow computation models. For example, multi-dimensional dataflow was proposed for supporting streaming vectors and matrices [13]. Cyclo-static dataflow can be used to model strided streaming patterns [15]. However, these are point solutions that only address a specific streaming pattern. In SPEX, we propose a different design approach: relax the dataflow computation model to let the programmers construct the appropriate streaming patterns. Instead of attempting to describe a streaming pattern with one dataflow actor or edge, SPEX allows the programmer to use a set of dataflow actors and non-dataflow functions. This set of actors and functions are not explicitly connected, but are allowed to share the same data. The dataflow actors are used to describe dataflow streaming patterns, non-dataflow functions are used for infrequent variable updates. In SPEX, these special dataflow actors are called memory actors, and these non-dataflow functions are called memory functions. This is different from a traditional dataflow model where actors cannot share data. By sharing data, each memory actor or function can be used as a building block to model one aspect of the streaming pattern. The combination of a set of actors can

Authorized licensed use limited to: University of Michigan Library. Downloaded on January 13, 2009 at 11:26 from IEEE Xplore. Restrictions apply.

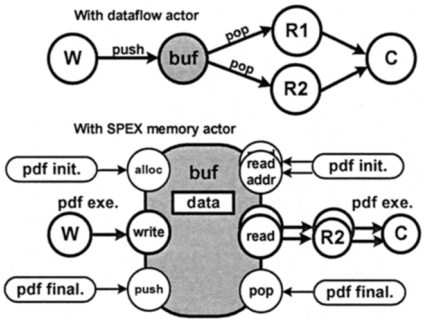


Fig. 3. Example of a vector stream buffer with 1 writer and 2 readers. This buffer's communication pattern has all four streaming properties. This is a vector buffer, which requires multi-dimensional streaming patterns. Its has non-sequential streaming patterns because its readers must periodically reconfigure their streaming addresses. The writer and readers are decoupled because they have different real-time deadlines. This is also a shared memory buffer because the readers share the same data, but have different streaming patterns.

be used to model complex stream patterns. The disadvantage for our approach is the data consistency problem. Traditional dataflow computation models do not have to deal with this problem because there is no shared data between the actors. In our PDF model, programmers must use locking mechanisms to access shared data. However, because our execution model enforces static SDF run-time execution schedules, the compiler has complete knowledge of the data access pattern for all actors and functions. Implementing locking mechanisms for a static schedule is more deterministic than for a multi-threading run-time environment.

The four streaming patterns listed previously can all be described using multiple memory actors and functions. A W-CDMA vector stream buffer is shown in Figure 3 with 2 readers and 1 writer. This buffer requires all four streaming patterns. 1) This is a vector buffer, which requires multi-dimensional streaming pattern. Read and write operations access scalar data elements within a vector element. Push and pop operations manage the vector queue by accessing across vector elements. 2) This buffer has a non-sequential streaming pattern because its readers' streaming address must be periodically reconfigured. This is implemented with a memory function that sets up the reading address during the PDF initialization stage. 3) The writer and reader of this buffer are decoupled because they operate with different real-time deadlines. 4) This is also a shared buffer because there are two readers with different streaming patterns. The buffer is allowed to pop the data only after the data is read by both readers. Pop can be implemented as a memory function that runs during the PDF finalization stage. In comparison, a traditional dataflow actor can only define this complex streaming communication with vector push and pop.

III. SPEX Overview

SPEX is a concurrent language extension that can be applied to any programming language. In our study, it is applied onto the C++ programming language due to its popularity and large user base. The SPEX dataflow extension consists of two parts: a set of language constructs for describing the modified PDF computation model and a set of language restrictions for supporting embedded multi-core compilation. The rest of this section is organized as follows: Section III-A describes a set of new language primitives that are added to describe a PDF graph. Section III-B describes the semantics of constructing a PDF actor in SPEX. Section III-C describes the semantics of constructing a PDF graph in SPEX. In each of the sections, we first discuss the language constructs, and then the language restrictions.

A. SPEX Primitives

Parameters. A parameter, denoted by the keyword `param`, is a variable that may only take on a finite set of discrete values specified by the programmer. Parameters are used to describe various dataflow properties of a PDF graph. One may declare a variable of parameter subtype with the following syntax: `param<base_type> var_name`. Currently, only integer-based parameter subtypes are supported in SPEX. The range of a parameter variable can be defined using the following two functions: `void param<T>::range(T min, T max)` and `void param<T>::values(int num_vals, T val1, ...)`. The two functions are used to declare either a range of values or a set of discrete values for each parameter. Arithmetic and comparison operations are supported for parameter variables.

Channels. SPEX channels, denoted by the keyword `channel`, are FIFO queues that are used to model dataflow edges. One may declare a variable of channel subtype with the following syntax: `channel<base_type> chan_name`. Both integer and floating-point channel subtypes are supported. In PDF, a dataflow edge may have parameterized input and output rates. The rates are specified through channel access functions. There are two types of functions supported for accessing channel variables: push and pop. Reader stalls on reading from an empty channel, and writer stalls on writing to a full channel. The size of the channel variables cannot be specified by the programmer. SPEX requires the compiler to determine the optimal sizing for each channel variable during compile-time.

PDF Functions. SPEX PDF functions are used to construct parameterized dataflow graphs. There are two types of PDF functions: `pdf_node` for describing a PDF actor, and `pdf_graph` for describing a PDF graph. PDF function arguments are communication channels. Each channel is either an input edge or an output edge. Therefore, each PDF function argument is either read-only or write-only. Read-only function arguments follow the pass-by-value syntax in C++: `func(arg_type arg_val)`. Write-only function arguments follow the pass-by-reference syntax in C++: `func(arg_type & arg_val)`. `pdf_node` functions only allow channel and parameter variables as function arguments. `pdf_graph` functions also allow memory actors as function arguments. PDF function calls also take on a different language semantic than in traditional imperative languages:

Authorized licensed use limited to: University of Michigan Library. Downloaded on January 13, 2009 at 11:26 from IEEE Xplore. Restrictions apply.

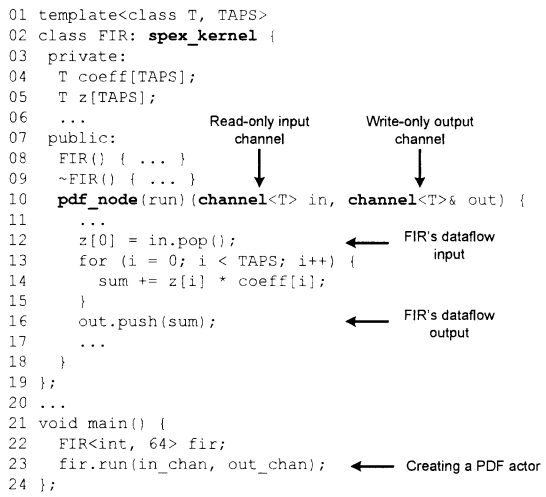


Fig. 4. DSP kernel object example – FIR filter. The keyword `spex_kernel` is on line 2 to indicate that this is a kernel class object.

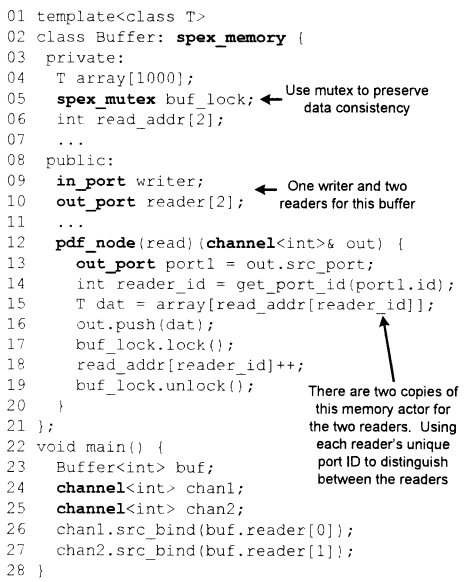


Fig. 5. A vector stream buffer with 2 readers and 1 writer. Data objects are declared with the keyword `spex_memory` (on line 2). This example implements the same buffer shown in Figure 3

each PDF function call creates an explicit PDF object. Multiple function calls to the same `pdf_node` function create multiple copies of the same PDF actor.

SPEX Language Restrictions. All variables and functions must be statically declared. This means that C++'s dynamic language features, such as late-binding virtual functions and run-time memory allocations, are not supported. Because the dynamism in DSP systems is described through parameters, dynamic language features do not add any benefits in describing these systems. Static variable and function declarations also produce more efficient code because they require less run-time management.

B. SPEX PDF Actors

PDF actors are described as C++ classes inherited from `spex_kernel` and `spex_memory`. `spex_kernel` is used to describe DSP algorithms, and `spex_memory` is used to describe streaming communication patterns. These are the only classes in SPEX that are allowed to declare and use PDF functions. Although PDF functions are sufficient for describing the dataflow computation, an object-oriented description has numerous advantages. Besides the dataflow computation, DSP algorithms also require initialization operations, infrequent coefficient updates, and other miscellaneous functionalities. Object-oriented programming groups gather together a DSP algorithm's related functions under one class. Not only does this provide a clearer conceptual programming interface, it also helps the compiler to determine the code placement for these functions on an embedded DSP architecture.

SPEX Kernel Class. SPEX kernel class is designed to describe DSP algorithm kernels as dataflow actors. Because dataflow actors do not share states, SPEX does not allow declaration of public variables in a kernel class. PDF functions can be declared as public member functions inside a kernel class. Non-PDF functions can also be declared in a kernel class following C++'s sequential execution model. Figure 4 shows an implementation of FIR filter as a SPEX kernel class. In this example, a PDF actor is declared on line 10. It has one input channel and one output channel. The filter's internal states are declared as private variables on lines 4 and 5.

SPEX Kernel Class Language Restrictions. Because dataflow computation model does not allow shared variables among actors, each kernel class may only declare one PDF function as its member function. Because PDF functions describe dataflow actors, a PDF function can call a non-PDF function within its function body, but a non-PDF member function may not call a PDF function. The input and output data rates of a PDF actor are derived from its channel variables' push and pop operations. Therefore, push and pop operations may only be called once within a PDF function. They may not be called within loop bodies or conditional branches, and they must only push/pop a constant or parameterized number of data elements.

SPEX Memory Class. Memory class, declared with keyword `spex_memory`, is used to describe a set of memory actors and functions that share the same data and states. A vector stream buffer is shown in Figure 3, its SPEX implementation is shown in Figure 5. Memory actors are defined as PDF node functions, and memory functions are defined as normal C++ member functions. This buffer is shared between two readers, which means that there are two callers to the `Buffer::read` function. The PDF function must be able to distinguish between the two callers. This is done by defining two output ports using the `out_port` keyword, as shown on line 10. Each read port explicitly binds to a reader channel, as shown on lines 26 and 27. Because each port has an unique ID. the two readers are distinguished through their ports.

Authorized licensed use limited to: University of Michigan Library. Downloaded on January 13, 2009 at 11:26 from IEEE Xplore. Restrictions apply.

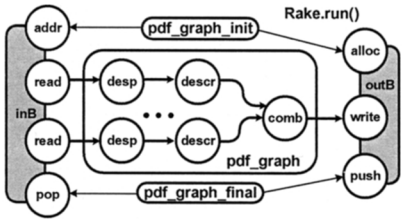
Traditional dataflow computation models do not have the data consistency problem because data cannot be shared among actors. Because PDF's memory actors and functions are allowed to share the same data, there is no inherent guarantee for their data consistency. It is up to the programmers to implement memory consistency through locks. Conceptually, the PDF actors are similar to threads in POSIX. Mutex variables are provided to enforce memory consistency between the PDF functions. An example is shown in Figure 5 on line 5 as a variable is declared of type `spex_mutex`.

SPEX Memory Class Language Restrictions. SPEX memory class may include only PDF actor functions, not PDF graph functions. Because memory actors can share data, multiple PDF actor functions can be declared within the class body. In addition, a set of input and output ports must be declared. These ports represent the number of writers and readers that may connect to the class' memory actors. The input ports are declared with keyword `in_port`. The output ports are declared with keyword `out_port`, as shown on line 9 and 10 in Figure 5. As mentioned previously, SPEX is a static programming language. Therefore, the number of readers and writers for each memory object must be statically defined, and each port must also explicitly bind to either a writing or reading channel.

C. SPEX PDF Graphs

As mentioned in Section II-A, we propose a three stage PDF run-time execution model. In SPEX, these three stages are described as three separate PDF graph functions. `pdf_graph_init` is used to describe the PDF initialization stage; `pdf_graph` is used to describe the PDF computation stage; and `pdf_graph_final` is used to describe the PDF finalization stage. For a given PDF graph, these three functions must have the same name and arguments. Figure 6 shows the W-CDMA rake receiver implemented as a PDF graph. These PDF graph functions are defined on lines 15, 29, and 39. Rake receiver is composed of three DSP algorithms: despreader, descrambler, and combiner. Each despreader and descrambler pair is called a rake finger. There are three run-time reconfigurations modeled in this simplified version of the rake receiver: 1) the number of rake fingers; 2) the number of elements streamed per function invocation; 3) the streaming read address for each rake finger. Because the number of rake fingers and the number of streaming elements both affect the dataflow configuration, they are described with parameter variables `fingers` and `stream_size`. The streaming read address is determined by initializing the input stream buffer during the PDF initialization stage, shown on line 36.

PDF Initialization & Finalization. The purpose of the PDF initialization stage is to setup the PDF graph for execution. Because the dataflow computation must use a synchronous dataflow schedule, the initialization stage is responsible for setting all of the parameter variables to constant values. Setting up the stream communication patterns is also done in this step. In the rake receive example shown in Figure 6, the initialization function for the rake receiver is shown between



a) Rake receiver PDF diagram

```
01 class Rake: spex_kernel {
02  private:
03   channel<int> chan1[MAX_FINGERS];
04   channel<int> chan2[MAX_FINGERS];
05   channel<int> chan3[MAX_FINGERS];
06   channel<int> chan4;
07   Despreader<int> despreaders[MAX_FINGERS];
08   Descrambler<int> descramblers[MAX_FINGERS];
09   Combiner<int> combiner;
10   param<int> stream_size;
11   ...
12  public:
13   Rake() { stream_size.values(2, 1000, 2000); }
14   ...
15   pdf_graph(run)(Buffer inb, Buffer& outb,
16                  param<int> r, param<int> fingers) {
17     pdf {
18       for (int j = 0; j < stream_size; j++) {
19         ll_for(int i = 0; i < fingers; i++) {
20           inb.read(chan1[i]);
21           despreader[i].run(chan1[i], chan2[i]);
22           descrambler[i].run(chan2[i], chan3[i]);
23         }
24         combiner.run(chan3, chan4);
25         outb.write(chan4);
26       }
27     }
28   }
29   pdf_graph_init(run)(Buffer inb, Buffer& outb,
30                       param<int> r, param<int> fingers) {
31     if (r == 4)      stream_size = 1000;
32     else if (r == 8) stream_size = 2000;
33     inb.num_readers(fingers);
34     combiner.num_rake_fingers(fingers);
35     for (int i = 0; i < fingers; i++)
36       inb.reader_start_addr(i, peak[i]);
37     outb.write_alloc(250);
38   }
39   pdf_graph_final(run)(Buffer inb, Buffer& outb,
40                        param<int> r, param<int> fingers) {
41     if (r == 4)      inb.pop(1000);
42     else if (r == 8) inb.pop(2000);
43     outb.push(250);
44   }
45 };
```

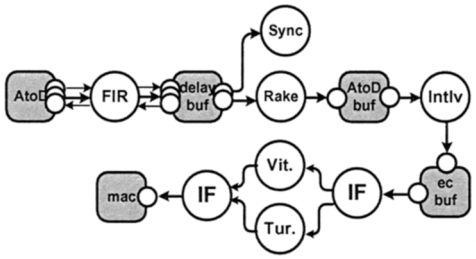
b) Rake receiver SPEX implementation

Fig. 6. Rake receiver implemented with PDF graph functions: `pdf_graph_init`, `pdf_graph`, and `pdf_graph_final`. These three PDF functions are used to describe the three stages in a PDF's run-time execution. `pdf_graph_init` is used to describe the PDF graph initialization; `pdf_graph` is used to describe the PDF graph execution; and `pdf_graph_final` is used to describe the PDF graph finalization.

lines 29 and line 38. Lines 31 and 32 set the parameter variable `stream_size` to a constant value based on the input spreading factor. The input stream buffer is initialized for each finger by setting the starting memory read address for each finger on line 35 and 36. To setup the streaming write buffer, we allocate memory space for the output on line 37.

The purpose of the PDF finalization stage is to update the PDF graph's internal states with the computed dataflow results. It is also used to perform memory management operations for the input and output buffers, as shown in Figure 6 on lines 41, 42, and 43. After the data is read out of the input buffer by all of the rake fingers, its memory is deallocated. The output buffer performs a push operation to make the written output

Authorized licensed use limited to: University of Michigan Library. Downloaded on January 13, 2009 at 11:26 from IEEE Xplore. Restrictions apply.



a) W-CDMA receiver PDF diagram

```
01 void WCDMA_Receiver(int* AtoD, int* mac)
02 {
03   int delay_buf[slot_size];
04   int itlv_buf[slot_size];
05   int ec_buf[slot_size];
06
07   for (int i=0; i<15; i++) {
08     addr = 0;
09     for (int j=0; j<slot_size; j++) {
10       int data = AtoD[addr];
11       data = fir_run(data);
12       delay_buf[addr++] = data;
13     }
14     fingers = sync_run(delay_buf);
15     rake_run(delay_buf, itlv_buf, fingers);
16     interleaver_run(itlv_buf, ec_buf);
17     if (mode == voice)
18       viterbi_run(ec_buf, mac);
19     else if (mode == data)
20       turbo_run(ec_buf, mac);
21   }
22 }
```

Must allocate large buffers for the entire stream

Imperative computation descriptions

b) W-CDMA receiver C implementation

```
01 class WCDMA_Receiver: spex_kernel {
02  private:
03   FIR<int> fir;
04   Rake rake;
05   // ... other DSP kernel declarations
06
07  public:
08   pdf_graph(run)(Buffer AtoD, Buffer& mac) {
09     pdf {
10       for (j = 0; j < 15; j++) {
11         pdf {
12           AtoD.read_addr(0);
13           delay_buf.push_alloc(slot_size);
14           chan1.src_bind(AtoD.reader);
15           chan2.dst_bind(delay_buf.writer);
16
17           for (i = 0; i < slot_size; i++) {
18             AtoD.read(chan1);
19             fir.run(chan1, chan2);
20             delay_buf.write(chan2);
21           }
22
23           AtoD.pop(slot_size);
24           delay_buf.push(slot_size);
25         }
26         sync.run(delay_buf, fingers);
27         rake.run(delay_buf, itlv_buf, rate, fingers);
28         Interleaver.run(itlv_buf, ec_buf);
29         if (mode == voice)
30           viterbi.run(ec_buf, mac);
31         else
32           turbo.run(ec_buf, mac);
33       }
34     }
35   }
36
37   pdf_graph_init(run)(Buffer AtoD, Buffer& mac) { ... }
38   pdf_graph_final(run)(Buffer AtoD, Buffer& mac) { ... }
39 };
```

Implicit PDF initialization

Nested PDF graph description with implicit initialization and finalization

Implicit PDF finalization

c) W-CDMA receiver SPEX implementation

Fig. 8. W-CDMA receiver implementation. The example shows both C and SPEX implementations of the receiver.

SPEX concurrent language constructs	Explanations

`pdf {`
`  ... // code for PDF initialization`
`  for (...) {...} // PDF dataflow`
`  ... // code for PDF finalization`
`}`
	A PDF graph construct. It must contain only one for-loop construct for describing the dataflow. While and do-while loops are not allowed in pdf construct.

`ll_for (init;cond;incr)`
`{`
`  ... // loop body`
`}`
	Parallel for-loop construct. The different iterations of the loop body are executed in parallel. There can not be data dependencies across loop iterations.

Fig. 7. SPEX language constructs for describing dataflow operations.

visible to other PDF actors that are reading or writing to this buffer.

Figure 7 lists the set of parallel language constructs that are supported in SPEX for describing a concurrent dataflow graph. `pdf` construct is used to describe a PDF graph. SPEX requires each dataflow to be described as a for-loop construct. Therefore, each PDF construct must contain one for-loop. The initialization and finalization stages can either be explicitly defined through PDF functions as shown in Figure 6b, or implicitly defined as shown in Figure 8c from lines 11 to 25. Parallel operations can be described using the `ll_for` construct. An example of this construct is shown in Figure 6 from lines 20 to 24.

SPEX PDF Graph Language Restrictions. The following set of language restrictions are defined to enforce synchronous dataflow computation during run-time. All of the PDF actors, channels, and memory buffers that are used in a PDF graph must be declared as private variables within the PDF graph class. Array of objects can be created using the same syntax as C++, but must be declared statically. Parameter variables must also be declared as private variables. They are not allowed to be declared as local variables within a PDF function. The value of parameter variables must be defined in the `pdf_graph_init` function or before the for-loop construct in `pdf`. And they are not allowed to be redefined in the `pdf` for-loop body.

W-CDMA Receiver Implementation. Figure 8 shows a simplified W-CDMA receiver implementation, in both C and SPEX. SPEX implementation requires larger code size than the C implementation. The stream computation itself requires 26 lines of SPEX code and 15 lines of C code. However, all of the streaming characteristics are lost in the C implementation. Because the algorithms are executed in sequential order, large buffers are allocated to pass entire streams of intermediate results. It is possible to reduce the buffer size by manually rewriting the C code. However, because optimal buffer sizes are dependent on the size of hardware's physical memory, programmers are forced to write machine-dependent code. In SPEX, because the streaming patterns are exposed in the language, the compiler can automatically pick the optimal buffer size. Programmers do not have to be aware of the

Authorized licensed use limited to: University of Michigan Library. Downloaded on January 13, 2009 at 11:26 from IEEE Xplore. Restrictions apply.

underlying hardware.

The W-CDMA standard divides the receiving data into TTI (Transmission Time Interval) blocks. Each TTI block contains a maximum of 5 W-CDMA frames, and each frame is further divided into 15 W-CDMA slots. Dataflow reconfigurations occur at multiple data block granularity. Because each reconfiguration requires its own PDF initialization and finalization stages, the W-CDMA receiver is implemented with nested PDF graphs, as shown in Figure 8c from line 11 to 25.

IV. Related Work

Dataflow Computation Models. There has been considerable work in reconfigurable dataflow models. These include less restrictive dataflow models, hybrid SDF with finite-state-machines(FSMs) [16], and parameterized SDF (PSDF) [1]. Examples of less restrictive dataflow models include the cyclo-static dataflow model (CSDF) [15], Boolean dataflow model (BDF) [3], and Synchronous piggybacked dataflow (SPDF) [14]. CSDF supports cyclic dataflow rates. BDF includes conditional split and merge actors on top of the SDF. SPDF supports reconfigurations by coupling infrequent control updates with the synchronous dataflow. In the hybrid SDF+FSM models, the different dataflow configurations are expressed as the different states of the FSM. SPEX's PDF model is very similar to the PSDF model. One noticeable difference is that our model supports memory actors that share data. Hierarchical dataflow models have also been proposed before to model multi-rate DSP applications with constraints [4].

Dataflow Languages. There have been many dataflow languages proposed for modeling DSP systems. Some of these are frameworks that are designed for a wide range of application by supporting multiple dataflow models, such as the Ptolemy project [11], the DIF format [9], and the PeaCE design flow [7]. There also have been languages that are designed explicitly for a processor architecture. StreamIt [17] was proposed for mapping streaming computations onto tiled processor architectures. The original StreamIt was designed based on the SDF computation model. Recent updates have also introduced parameterized variables, allowing the description of variable rate dataflow. StreamIt supports stream reconfigurations and updates through teleporting messages [18], which has similar functionality to the SPDF model.

Other Streaming Languages. There are also other streaming languages that are not based on dataflow computation models, such as Brook [2] and Sequoia [5]. Both are imperative languages with explicit constructs for streaming array structures. Sequoia is also designed to expose an application's memory hierarchy to the programmers.

V. Conclusion & Acknowledgments

In this paper, we describe SPEX's language features for supporting streaming DSP computations and communications. SPEX's streaming semantics are based on a parameterized dataflow computation model. We have modified this dataflow model to introduce special dataflow actors that are allowed to share data. This allows complex streaming communication patterns to be described with a set of dataflow actors. SPEX is applied onto the C++ programming language. It consists of a set of language constructs for describing the semantics of parameterized dataflow computations, and a set of language restrictions for helping the embedded compilation process.

Acknowledgements. Yuan Lin is supported by a Motorola University Partnership in Research Grant. This research is also supported by ARM Ltd. and the National Science Foundation grants NSF-ITR CCR-0325898, CCR-EHS 0615135, and CCR-0325761.

References

[1] B. Bhattacharya and S. S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. pages 2408–2421. IEEE Transactions on Signal Processing, Oct. 2001.

[2] I. Buck. Brook Language Specification. In *http://merrimac.stanford.edu/brook*, Oct. 2003.

[3] J. T. Buck and E. A. Lee. Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model. Proc. Int. Conf. Acoust., Speech, Signal Processing, April 1993.

[4] N. Chandrachoodan and S. S. Bhattacharyya. The Hierarchical Timing Pair Model for Multirate DSP Applications. In *IEEE Transactions on Signal Processing*, volume 52, no. 5, May 2004.

[5] K. Fatahalian et al. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Nov. 2006.

[6] M. Gordon et al. A Stream Compiler for Communication-Exposed Architecture. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004.

[7] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. Joo. PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems. In *ACM Transactions on Design Automation of Electronic Systems*, Aug. 2007.

[8] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access For Third Generation Mobile Communications*. John Wiley and Sons, LTD, New York, New York, 2001.

[9] C. Hsu, I. Corretjer, M. Ko, W. Plishker, and S. S. Bhattacharyya. Dataflow interchange format: Language reference for DIF language version 1.0, users guide for DIF package version 1.0. In *Technical Report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park*, June 2007.

[10] E. Lee and D. Messerschmidt. Synchronous Data Flow. In *Proc IEEE*, 75, 1235-1245 1987.

[11] E. A. Lee. Overview of the Ptolemy Project. In *Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley*, July 2003.

[12] Y. Lin et al. SPEX: A Programming Language for Software Defined Radio. In *Software Defined Radio Technical Conference and Product Exposition*, 2006.

[13] P. Murthy and E. A. Lee. Multidimensional Synchronous Dataflow. In *IEEE Transactions on Signal Processing*, August 2002.

[14] C. Park, J. Chung, and S. Ha. Extended Synchronous Dataflow for Efficient DSP System Prototyping. pages 295–322. Design Automation for Embedded Systems, Kluwer Academic Publishers, March 2002.

[15] T. M. Parks, J. L. Pino, and E. A. Lee. A Comparison of Synchronous and Cyclo-Static Dataflow. In *Asiolmar Conference on Signals, Systems and Computers*, October 1995.

[16] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. FunState — An Internal Representation for Codesign. Proc. International Conference on Computer Aided Design, Nov. 1999.

[17] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the 2002 International Conference on Compiler Construction*, June 2002.

[18] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport Messaging for Distributed Stream Programs. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.

Authorized licensed use limited to: University of Michigan Library. Downloaded on January 13, 2009 at 11:26 from IEEE Xplore. Restrictions apply.