SPEX: A Programming Language for Software Defined Radio

Yuan Lin[1], Robert Mullenix[1], Mark Woh[1], Scott Mahlke[1], Trevor Mudge[1],
Alastair Reid[2], and Krisztián Flautner[2]

[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{linyz, rbmullen, mwoh, mahlke, tnm}@umich.edu

[2]ARM, Ltd.
Cambridge, United Kingdom
{alastair.reid, krisztian.flautner}@arm.com

Abstract

High-throughput, low-power Software Defined Radio(SDR) solutions require multi-core SIMD DSP processors to meet real-time performance requirements. Given the difficulty in programming traditional DSPs, these new multi-core signal processors provide even greater challenges for programmers and compilers. In this paper, we describe SPEX, a programming language which is aimed at narrowing the semantic gap between the description of complex SDR systems and their implementations. SPEX supports three different types of programming semantics, allowing SDR solutions to be developed with a divide-and-conquer approach. For DSP algorithm kernels, SPEX is able to support DSP arithmetics and first-class vector and matrix variables with sequential language semantics. From wireless protocol channels, it is able to support sequences of data-processing computations with dataflow language semantics. And for protocol systems, it is able to support real-time deadlines and concurrent executions with synchronous language semantics. The design choices are motivated by our experience implementing W-CDMA protocol on a reprogrammable substrate. In the paper, we also briefly explain SPEX's compilation strategies.

1 Introduction

There is a growing trend for using multi-core DSP processor architectures to support high-throughput, low-power mobile Software Defined Radio(SDR) solutions [5] [13] [12]. Unlike traditional uniprocessor DSPs, these multi-core DSPs have non-uniform memory access latencies, wide SIMD units, narrow data width, and the requirement to support dynamically changing workloads with real-time deadlines. Traditional DSP programming models are designed for stand-alone algorithms and uniprocessor architectures. Even for uniprocessors, developers are often forced to program in assembly language to achieve maximum performance. The new multi-core signal processors provide even greater challenges for programmers and compilers. To make matters worse, the software implementations of wireless protocols require a large set of operations that are not natively supported in general purpose languages. These include fixed-point DSP arithmetics, streaming and concurrent computations, and real-time process scheduling. Clearly, there is a need for better programming models to bridge the widening gap between high-level software developments and efficient DSP hardware utilization.

In Section 2, we first describe the software implementation of the W-CDMA protocol, and its behaviors and requirements. We then describe SPEX, a programming language for SDR in Section 3. SPEX includes three different types of programming semantics – Kernel SPEX, Stream SPEX, and Synchronous SPEX. This allows SDR solutions to be developed with a divide-and-conquer approach, where each component can be implemented and verified independently. Kernel SPEX is an imperative language supporting native DSP arithmetics and first-class vector and matrix variables. It is best used to describe DSP algorithm kernels. Stream SPEX is a dataflow language for streaming computations. It is best used to describe wireless protocol's communication channels. Synchronous SPEX is a concurrent language with real-time support. It is best used to describe the overall protocol's system operations. And finally, in Section 4, SPEX's compilation strategies are described.

2 SDR Case Study: W-CDMA

In order to design a language suited for SDR, we need to first examine the implementation requirements of wireless protocols. In our study, we use W-CDMA as a case study because it has many of the characteristics found in a typical SDR system, including complex DSP algorithms, real time computational requirements, and dynamically changing workloads.

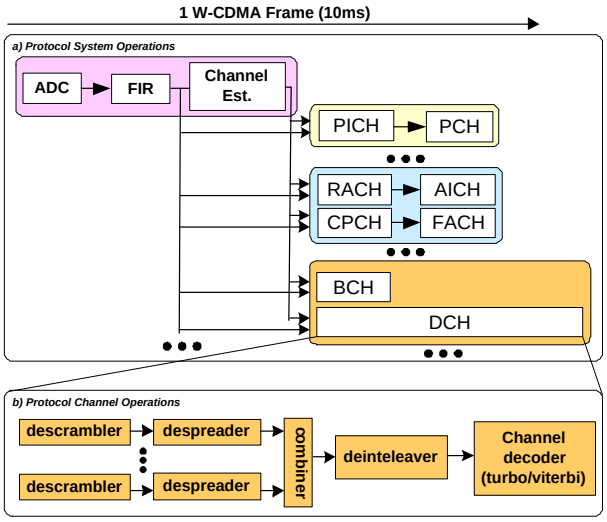

Figure 1: W-CDMA Protocol Diagram

Protocol System. The top diagram in Figure 1 shows the W-CDMA wireless protocol's system execution patterns. W-CDMA's data are divided up into periodic 10msec frames. Some channels, such as the frontend ADC, FIR, and Channel Estimation, are needed for every frame. Other channels are executed based on the protocol operation conditions. Some channels are periodically executed for every frame, whereas others are aperiodic. Some channels have hard real-time deadlines, whereas others have soft real-time deadlines, or no real-time deadlines. For mobile terminals, there may only be one channel active at a time, but base stations needs to support multiple users. This requires most of the channels to be executed concurrently, with some channels having data dependencies. This level of SDR description can be modeled as a concurrent system with multiple real-time deadlines.

Protocol Channels. The bottom diagram in Figure 1 shows the W-CDMA DCH channel. Data first travels through multiple descramblers and despreaders. They are then combined together with the combiner, sent to the deinterleaver, and then to the channel decoder. This level of SDR description does not have the notion of real-time. Data is processed in a pipelined sequence of DSP kernels. This level of SDR description can be modeled as stream computation.

Protocol Algorithms. Each protocol channel is made up of a number of DSP algorithms. Our previous study [11] has examined the algorithm characteristics of W-CDMA, and we found that the majority of the computations are done on vector variables with 8- to 16-bit fixed point precisions. Therefore, in SDR DSP algorithms, it is important to support fixed-point vector variables and operations.

3 SPEX Programming Model

SPEX supports three different levels of programming semantics: Synchronous SPEX, Stream SPEX, and Kernel SPEX. An implementation of W-CDMA using SPEX is shown in Figure 2. At the top level, the programmers use Synchronous SPEX to model the wireless protocol as a synchronous real-time system. Protocols channels are treated as concurrent nodes with a set of execution patterns and deadlines. Inter-channel data dependencies are described with communication and synchronization primitives. Synchronous SPEX nodes are allowed to execute only if both of the input data dependencies and timing constraints have been met. In the next level, each protocol channel is modeled as a dataflow stream using the Stream SPEX. This level of abstraction still allows concurrent execution, but has no notion of real-time. Each channel is constructed as a sequence of DSP kernels connected together with dataflow communication primitives. Each kernel is allowed to execute after its input data dependencies are met. At the lowest level, Kernel SPEX can be used to model DSP kernels. Each DSP algorithm is described as an sequential kernel consisting of a set of functions and local states.

Static Language. SPEX is a static object-oriented language modeled after C++. Like C++, local variables and member functions are supported as part of class declarations. However, unlike C++, objects cannot be created dynamically. All variables and functions have to be statically declared and allocated during compile time. This means that C++ features, such as late-binding virtual functions, are not supported. This is enforced to increase the efficiency of the compiled code. Unlike general purpose computing, a SDR platform has very high computation requirements and low power budget, with little overhead to support dynamic task scheduling and memory managements. Therefore, it is desirable to offload these tasks to the compiler. SPEX is designed to trade off the convenience of dynamism for compilation efficiency.

3.1 DSP Data Types

DSP arithmetics have many intrinsic characteristics that are not supported by traditional general purpose programming languages, such as C++. For SPEX, we have introduced a set of special DSP data types made for DSP computations. These include a set of type modifiers on top of existing types, and a set of first class array types for DSP computations.

DSP Attribute Modifiers. SPEX supports DSP variables with additional type attributes that are not supported by general purpose languages. These are modeled after System C variable attributes, which include: 1) fixed-point arithmetics, 2) data precision, 3) overflow mode, and 4) rounding mode. Many embedded DSP processors do not sup-

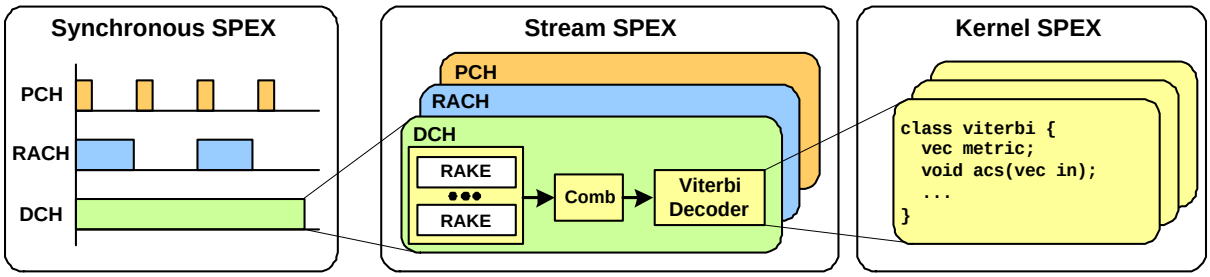

Figure 2: Implementation of W-CDMA using SPEX's three different levels of programming abstraction

port 32-bit floating point, but simpler 8- or 16-bit fixed-point arithmetic operations instead. Therefore, in addition to integer and floating-point numbers, SPEX supports fixed-point numbers with user-defined data precisions. In C, the result of overflowing fixed-point operations are implicitly wrapped-around. In DSP computation, saturation is needed for some operations. In fixed-point and floating point computations, different rounding modes also need to be explicitly defined in order to preserve the accuracy of the results.

Vectors and Matrices. SPEX supports first class arithmetic operations on vector, matrix and complex numbers, modeled after Matlab data operations. For example, programmers can use an add operator on two vectors, instead of writing a loop to iterate through each vector element. SPEX also supports predication and permutation operations on vector and matrix variables, which allows programmers to express algorithms more efficiently.

3.2 Communication Primitives

Wireless protocols are concurrent real-time embedded systems that can be modeled as a set of concurrent nodes connected together with communication primitives. In SPEX, the communication primitives are implicitly defined as the function arguments passed between functions. In order to distinguish between concurrent and imperative code, SPEX provides a set of communication primitives as a method for the programmers to explicitly define concurrency.

Channels – Message Passing Objects. SPEX channels are objects build on top of the traditional message passing concepts. Like the message passing protocols, channels stall on reading from an empty channels and writing to a full channel. Each channel is required to have only one writer, but can have potentially many readers. However, unlike message passing protocols, channels are not strictly FIFO buffers, and the exact size of data transmission packets and communication patterns can be determined by the compiler.

Channels support streaming computations through implicit communication. Between functions, the sender and the receiver of a channel do not have to enforce strict sequential execution order. A channel consists of an array of communicating elements. As long as the element-wise data consistency is maintained, any execution order between the 2 functions are allowed. Within a function, channels can be used in the same way as vectors with one restriction – all channel variables are either read-only or write-only, not both. This relaxed form of guarantee allows the compiler to optimize the data communications between functions, as streams can be established between the caller and callee functions running concurrently on two different processors.

```
template<class T, TAPS, S>
class FIR {
 private:
  vector<T, TAPS> coeff;

 public:
  ...

  kernel run(channel<T,S+TAPS-1> in, channel<T,S> &out)
  {
    vector<T, TAPS> v;
    for (i = 0; i < S; i++) {
      v = in(i,i+TAPS-1);
      out[i] = sum(v * coeff);
    }
  }
}
```

Figure 3: Kernel SPEX example – FIR filter

Signals – Shared Memory Objects. SPEX signals are objects built on top of the shared memory concepts with some restrictions. Like shared memory variables, a signal can change its value at any time during execution. To provide efficient compilation, signals are restricted to have only one writer, but can have potentially many readers. Because many embedded multi-core processors do not have cache structures or support cache coherent protocols, supporting software managed shared memory variables is not very efficient. Signals should only be used for control-related code and synchronization mechanisms. Therefore, SPEX restricts the declarations of signals to be scalar variables only.

3.3 SPEX Objects

SPEX is an object-oriented language based on C++ semantics. The three different levels of programming abstractions are expressed as three different types of class function declarations. Three additional keywords: **kernel**, **stream**, and

```
01 template<spex_type T, S>
02 class DCH
03 {
04  private:
05   ...
06
07   Rake<T,S> rake[max_rake_fingers];
08   Viterbi<256,S> viterbi;
09   Combiner<T,S> combiner;
10
11   channel<T,S> ch1[max_rake_fingers];
12   channel<T,S> ch2;
13
14  public:
15   ...
16
17   stream run(channel<T,S> ADC_in,
18              channel<T,S> searcher_in,
19              channel<T,S> & out_MAC,
20              signal<int> & done)
21   {
22     for (int i = 0; i < num_rake_fingers; i++)
23       rake.run(ADC_in, searcher_in, ch1[i]);
24     combiner.run(ch1, ch2);
25     viterbi.run(ch2, out_MAC);
26
27     barrier;
28     done = true;
29   }
30 }
```

Figure 4: Stream SPEX example – DCH

synchronous are added to distinguish the three types of functions. The following describe these three types of semantics.

Kernel SPEX. Kernel SPEX functions assume a sequential execution order, following the semantics of C. They are declared with the **kernel** keyword, as shown in Figure 3. Formally, for each instruction i inside a kernel function, the results of instruction i can only be updated after all of its preceding instructions have finished their update, and must be updated before all of its succeeding instructions start their updates. Because of this, channels and signals can be used within a kernel function, but they cannot be declared as local variables, or passed as an argument to other functions. In addition, kernel functions are only allowed to call other functions that are also kernel functions.

The strict sequential ordering means that the kernel functions are best fitted for SIMD and VLIW compilations. We envision kernel functions being used to describe individual DSP algorithms. This provides a clean interface between the system engineers that design the protocol channels using Stream and Synchronous SPEX, and the DSP algorithm engineers that design the algorithms with Kernel SPEX.

Stream SPEX. Stream SPEX functions are used to support concurrent data streaming with a dataflow computation model. Functions are declared with the **stream** keyword. With the streaming semantics, the strict sequential execution order is not enforced between instructions, only data consistency is required. A pair of communicating instructions can execute in any order, as long as the correct data values are consumed by the receiving function. Stream functions are not allowed to declare signals as local variables, nor pass them as arguments in function calls.

```
01 class WCDMA
02 {
03   ...
04   AdcFir<int16, wcdma_frame> adcfir;
05   DCH<int16, wcdma_frame> dch;
06   Clock<slot, 15> frame_clock;
07   Signal<bool> bch_done, dch_done;
08
09   ...
10   synchronous run()
11   {
12     when (clock == 0) {
13       adcfir(ch1);
14       bch(ch1,bch_done);
15       chan_est(ch1,ch2,num_fingers);
16       if (wcdma_data_mode)
17         dch(ch1,ch2,num_fingers,dch_done);
18     }
19
20     when (clock == bch_deadline)
21       assert(bch_done == true);
22
23     when (clock == dch_deadline)
24       assert(dch_done == true);
25   }
26 }
```

Figure 5: Synchronous SPEX example – W-CDMA

Figure 4 shows a simplified implementation of a W-CDMA DCH channel. In the function run, the instructions between line 22 and 25 represent a data stream going from the rake receiver to the combiner to the viterbi decoder. Each algorithm can potentially execute on a different processor with the data streamed between them. The **barrier** keyword on line 27 is used in a streaming function to enforce a sequential execution order between instructions. For example, in function run, the statement $done = true$ on line 28 is defined after the barrier instruction. If it is defined before the barrier, then the value of $done$ may be set to true before the end of the streaming execution. This is because operations in a stream function are not guaranteed to start after its predecessor completes. Therefore, line 28 may start execution before the $viterbi$ function call on line 26 has finished its computation. If there is no barrier defined, then we assume that the entire function is under one streaming scope.

Synchronous SPEX. Synchronous SPEX is used to support discrete real-time computations, with functions declared with **synchronous** keyword. It is modeled after Synchronous Languages, such as Esterel [3] and Signal [6]. A partial implementation of the W-CDMA real-time system is implemented as shown in Figure 5. Synchronous functions are allowed to declare and call stream and kernel functions, and channel and signal communication primitives. In this programming abstraction level, instructions are treated as concurrent nodes that can execute in parallel. In the W-CDMA example, the three **when** scopes on line 12, 20, and 23 can be executed concurrently. In addition, the body of the scope on line 12 can also be executed concurrently. The expressions on line 12, 20, and 23 stall until the conditions evaluate to true.

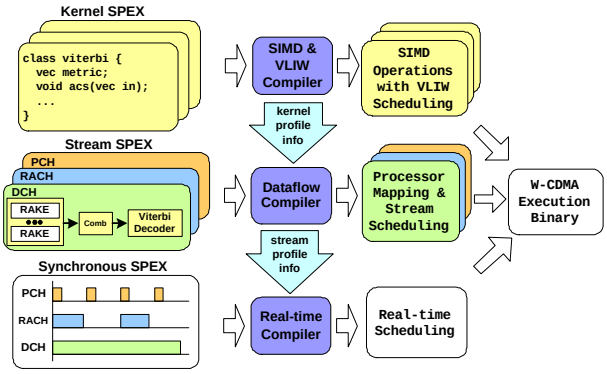

Figure 6: SPEX Compilation Flow – Multi-tier Compilation

Synchronous SPEX also provides support for periodic clock variables. In this example, a clock that tracks W-CDMA frames is declared on line 6, and used on line 12, 20, 23. Clock variables are declared with two parameters: time step and clock period. Time step is the smallest time increment for the clock variable, and it has to be a multiple of the clock period of the processor. In this example, it is declared to be the length of one W-CDMA slot. Clock variable's internal time counter is set back to zero at every clock period. In this example, the clock period is set to the length of one W-CDMA frame, which contains 15 slots. The main purpose of the clock is to provide synchronous execution, and to describe deadlines.

4 SPEX Compilation

SPEX's three different programming semantics require three different compilation techniques. Kernel SPEX functions can generally be compiled onto a single DSP processor, which requires VLIW and SIMD compilation techniques. Stream SPEX functions should be mapped across multiple processors for concurrent stream executions, which requires dataflow compilation with multi-processor assignments and DMA scheduling. Synchronous SPEX functions have timing constraints, requiring real-time scheduling algorithms.

Ideally, we should compile these three different types of function independently. However, in wireless protocols, these three types of functions are inter-connected together through channels, signals and other types of variables. Furthermore, the compilation efficiencies for these types of functions are inter-dependent on each other. For example, the quality of the stream scheduling may depend on the predictability of its kernel functions' execution times. However, the optimal scheduling of a sequential DSP kernel may depend on the inter-processor network traffic, which is determined by the stream compilation. To solve this problem, we propose an iterative multi-tier compilation process. We first compile the kernel functions, and profile their execution behaviors. We then pass these profile information to compile stream functions, treating each kernel function as an atomic instruction. The stream function's execution behaviors are then extracted to compile the synchronous functions. If all of the timing conditions are met, then we are done. Otherwise, we take this more realistic system assumptions, and recompile the kernel functions to reiterate the compilation process. This iterative process halts when the system deadlines and met, or if the compiler cannot converge onto a feasible solution. SPEX's multi-tier compilation is shown in Figure 6. The following sections describe the compilation process for each of the three types of functions.

Kernel SPEX Compilation. The process of kernel compilation aims to map the algorithms onto a specific DSP platform efficiently. The kernel SPEX representation is passed through the compiler that has a specification of the pertinent characteristics of the DSP. These include SIMD width, arithmetic and logical operations, and vector permutation primitives. The more esoteric data types such as complex should be expressed in the simpler constructs of the architecture. Vector permutation operations are broken into a series of assembly operations that perform the equivalent task. Vectors specified at the SPEX level that are larger than the native SIMD width must also be broken up into appropriately sized chunks.

Stream SPEX Compilation. For stream compilation, the problem can be broken down into three steps. In the first step, each channel is analyzed to determine its usage pattern and its streaming rate. In the second step, functions that are connected together through channels are match in the streaming rate. In the third step, functions are partitioned onto multiple processors based on the workloads, memory are allocated, and DMA instructions generated, using a dataflow compilation algorithm.

Synchronous SPEX Compilation. For synchronous compilation, the problem involves real-time scheduling. According to previous work [9], multiprocessor real-time scheduling can be broken into three steps: 1) processor assignments, 2) task ordering, and 3) execution timing. All three steps can be done either during compile time or during run time. For SDR, we find that processor assignments and partial task ordering can be done during the compile time. But execution timing can not be determined during compile time. This is because W-CDMA protocols can execute different types of channels with different execution conditions. Idle mode and data transferring mode use different protocol channels. Because the channels have data-dependencies, a relative order can be determined during the compile-time.

5 Related Work

There have been many research studies on efficient DSP languages and programming models. Many of them have focused on a single programming semantic. DSP-C [1] extends the C language to include types with different bitwidths, support for saturation mode, and circular buffers. While these are useful extensions for DSP programming, DSP-C is missing the SIMD-centric data structures as well as concurrency support, which are essential for high-through multi-core DSP architectures.

The Kahn process network [7] is a model of computation that has been popular in the DSP community. In Kahn's model, network nodes communicate concurrently with each other through unidirectional infinite-capacity FIFO queues. Each network node contains its own internal state. Reading from FIFO queues is blocking, and writing to the FIFO queues is non-blocking. Because of the blocking-read operation, the context switching overhead is high. Researchers have later proposed dataflow process networks [10], which is a special case of a Kahn network. In dataflow process network, the communication between network nodes (called actors) are explicitly defined as firing rules. Many variations of dataflow processor networks have been proposed, including quasi-static dataflow [14], and dynamic dataflow [4]. We think a variation of these dataflow models can be used for Stream SPEX.

Stream-C [8] is a C-extensions developed as part of the Imagine project [2]. It expresses DSP concurrency through streams, very similar to SPEX's stream objects. However, it is designed for uniprocessor architecture with no explicit real-time constraints. In addition, explicit SIMD object definitions and data attribute information are also not supported. StreamIt [15] was a dataflow language developed for the RAW project. Like Stream-C, it lacks explicit SIMD object definitions or data attribute information, making it unsuitable for multi-core SIMD or VLIW DSP architectures. In addition, it is also very hard to express real-time constraints in dataflow. There are also other non-C based approaches; SPL [16] is a good example. Although SPL contains SIMD-centric vector and matrix data structures, it does not expose any concurrency information. In addition, it generates code for C or Fortran, which does not ease the compilation efforts.

6 Conclusion

We have presented a programming language designed for embedded multi-core DSP system for SDR. SPEX allows programmers to efficiently express DSP algorithms as well as concurrent real-time systems. SPEX support these vastly different programming requirements by offering three different levels of programming semantics — kernel, stream and synchronous functions. Each one is design for a specific requirement of the software descriptions of wireless protocols.

7 Acknowledgment

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

References

[1] http://www.dsp-c.org.

[2] J. H. Ahn et al. Evaluating the Imagine Stream Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[3] F. Boussinot and R. de Simone. The Esterel Language. In *Proc. IEEE*, volume 79, pages 1293–1304, Sept. 1991.

[4] J. Buck. Static scheduling of code generation from dynamic dataflow graphs with integer valued control signals. In *Asilomar Conf. on Signals, Systems and Computers*, 1994.

[5] J. Glossner, E. Hokenek, and M. Moudgill. The Sandbridge Sandblaster Communications Processor. In *3rd Workshop on Application Specific Processors*, pages 53–58, Sept. 2004.

[6] P. L. Guernic, T. Gautier, and M. L. Borgne. Programming Real-time Applications with SIGNAL. In *Proc. IEEE*, volume 79, Sept.

[7] G. Kahn. *The semantics of a simple language for parallel programming*. J.L. Rosenfeld, Ed. North-Holland Publishing Co., 1974.

[8] U. J. Kapasi et al. Programmable stream processors. *IEEE Computer*, August, 54-62 2003.

[9] E. Lee and S. Ha. Scheduling Strategies for Multiprocessor Real-Time DSP. In *Proc. GLOBECOM*, Nov. 1989.

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

[11] H. Lee et al. Software Defined Radio - A High Performance Embedded Challenge. In *Proc. 2005 Intl. Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, Nov. 2005.

[12] Y. Lin et al. SODA: A Low-power Architecture For Software Radio. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[13] B. Mohebbi and F. Kurdahi. Reconfigurable parallel dsp - rdsp. In *Software Defined Radio*, 2004.

[14] 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.

[15] W. Thies. Streamit: A compiler infrastructure for stream programs. *IBM Programming Language Day*, 2004.

[16] J. Xiong et al. SPL: A Language and Compiler for DSP Algorithms. In *PLDI*, June 2001.