Atomix: A Framework for Deploying Signal Processing Applications on Wireless Infrastructure

Manu Bansal, Aaron Schulman, and Sachin Katti, ***Stanford University***

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/bansal

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

Atomix: A Framework for Deploying Signal Processing Applications on Wireless Infrastructure

Manu Bansal, Aaron Schulman, and Sachin Katti
Stanford University

Abstract

Multi-processor DSPs have become the platform of choice for wireless infrastructure. This trend presents an opportunity to enable faster and wider scale deployment of signal processing applications at scale. However, achieving the hardware-like performance required by signal processing applications requires interacting with bare metal features on the DSP. This makes it challenging to build modular applications.

We present Atomix, a modular software framework for building applications on wireless infrastructure. We demonstrate that it is feasible to build modular DSP software by building the application entirely out of fixed-timing computations that we call *atoms*. We show that applications built in Atomix achieve hardware-like performance by building an 802.11a receiver that operates at high bandwidth and low latency. We also demonstrate that the modular structure of software built with Atomix makes it easy for programmers to deploy new signal processing applications. We demonstrate this by tailoring the 802.11a receiver to long-distance environments and adding RF localization to it.

1 Introduction

Programmable Digital Signal Processors (DSPs) are increasingly replacing ASICs as the platform of choice for performing the heavyweight signal processing in our wireless infrastructure. The primary driver for the shift toward this programmable infrastructure is the increased rate of wireless standard updates: the 3GPP releases a new LTE standard roughly once every 18 months [3]. DSPs enable such quick upgrade cycles since their software can be updated with the push of a button.DSPs in the infrastructure strike balance between high performance, low power, (only 5-8 Watts to run a 20MHz LTE basestation), and programmability. They do so by combining multiple processing cores for programmability with hardware accelerators for performance and power efficiency.

The trend toward building programmable DSPs into wireless infrastructure presents a valuable opportunity: programmers can build new signal processing applications and quickly deploy them on wireless infrastructure at scale. Such applications encompass both communication and non-communication signal processing. Communication applications include standard wireless protocols (e.g., LTE and WiFi), as well as customized protocols for particular environments (e.g., long distance rural broadband [14, 6]). Non-communication applications include using radio waves for localization [28, 8].

There are three basic primitives that programmers need from the software framework of DSPs to build and deploy signal processing applications: *tapping* into a signal processing chain, *tweaking* a signal processing block, and *inserting* or *deleting* a signal processing block. However, the DSP software framework must present a modular interface to the programmer that supports these primitives. Such modularity is essential to add new applications without needing to modify or understand the full software base which could uptake excessive effort. For example, tweaking a specific block to improve its function or efficiency should not affect other blocks, nor should it require the programmer to tweak or understand other parts of the software.

Designing a modular software framework for DSPs is challenging because of the demanding requirements of the communication applications that it must support. Communication applications are both high throughput and latency sensitive. This combination requires hardware-like performance from the DSP software; it must be highly efficient and have predictable execution timing. For example, to process a typical 20MHz WiFi channel, the DSP must process a sample of the signal every 25ns (assuming Nyquist sampling). Assuming the DSP is running at a 1GHz (this is a typical DSP clock rate), there are only 25 cycles available on average per

DSP core to process each sample. Additionally, the processing has to finish within a short time because of ARQ. For instance, WiFi needs to decode and send an ACK within 16μs of receiving the last sample of a packet.

Building modular DSP software that has hardware-like efficiency and predictability is challenging. The primary reason is, programmers must use several bare metal features to achieve hardware-like performance. DSPs such as the TI 6670 [23] are highly parallel processors with multiple cores and hardware accelerators. There is no rich operating system support to manage those resources with adequate performance. As a result, the programmer must manually parallelize software. DSPs typically lack cache-coherent shared memory. This forces the programmer to explicitly move data across cores and hardware accelerators. Further, DSPs tend to have shallow memory hierarchies with software-addressable SRAM that the programmer has to explicitly manage.

In this paper, we present Atomix, a modular software framework for developing high bandwidth and low latency apps for DSPs. The key idea behind Atomix is the *atom* abstraction, which is defined as a unit of execution that takes a fixed, known amount of time to run every time it is invoked. In Atomix, programmers can express every module in a signal processing application as an atom. Atoms can be an algorithmic blocks such as FFT, or compositions of blocks such as OFDM, or different packet processing modes in protocols such as LTE or WiFi, or even low-level plumbing primitives such as data transfer across cores.

We evaluate the Atomix framework by using it to build a standard 802.11a WiFi receiver called Atomix11a. WiFi is an ideal app to evaluate the performance capability of Atomix because it uses the same bandwidth as LTE with similar processing complexity and spectral efficiency of an OFDMA protocol. Additionally, WiFi's decode latency constraints of tens of microseconds are two orders of magnitude tighter than LTE's[1]. For a 10MHz WiFi channel, Atomix11a achieves a frame decode latency of 36.4μs with a variance of 1.5μs.

We also evaluate the modularity of apps developed in Atomix by developing two apps on top of Atomix11a: (1) we customize Atomix11a to support long-distance links and (2) we add RF localization to Atomix11a.

2 Background and Design Goals

2.1 App Taxonomy

A typical wireless stack is naturally specified in terms of signal processing blocks, a data flowgraph that composes those blocks, and a state machine that selects appropriate flowgraphs to process incoming samples. For example, an 802.11a baseband receiver (fig. 1) has blocks for OFDM demodulation, channel equalization, constellation-to-bit slicing, and Viterbi decoding. Different blocks are composed into separate data flowgraphs for decoding a 54Mbps sample stream, a 6Mbps sample stream, or the PHY layer header, or for transmitting an ACK. Finally, the flowgraphs are assembled into a receiver state machine that transitions from the header decode state to one of the data decode states, optionally followed by the ACK transmit state.

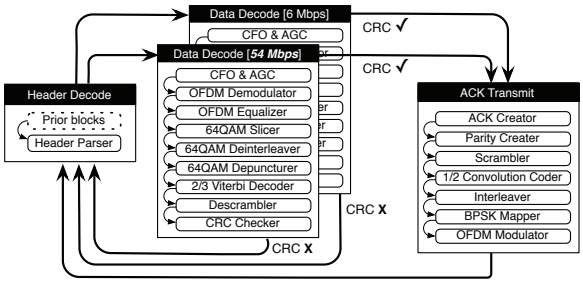


Figure 1: Modules of an 802.11a receiver: blocks, dataflowgraphs, and a state machine.

We classify apps according to the kind of modifications needed to the base wireless stack. (For a wireless stack app that is bootstrapping the base-station, the base stack is null.) To illustrate the classification, we use a simple OFDM receive chain (fig. 2) as the underlying stack that is modified. We classify applications based on modifications that fall into three main categories:

Tap. These modifications tap the signal at various points in the processing chain to implement their functionality. Fig. 2 shows an example of tapping the CSI output and sending it back for offline processing. Such tapping capability is needed for applications like indoor localization [28, 8]. Localization works by converting few samples from the right point in the signal processing chain into location estimates.

Tweak. These modifications tweak parameters of individual signal processing blocks that are already part of the wireless stack to implement tailored functionality. Fig. 2 shows an example of tweaking the channel equalization block. The ability to tweak parameters or modify the functionality of a block can enable applications like better receiver designs (e.g. [13, 25]) for existing transmission formats or adapting the signal chain to a different deployment setting such as long distance rural WiFi links as we show in sec. 5.

Insert (and Delete). These modifications insert new

[1]LTE's 3ms HARQ process turnaround requirement allows for approximately 1ms of decode latency.

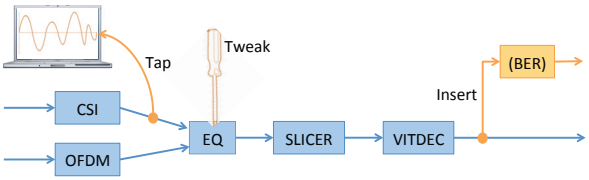
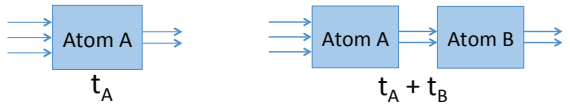


Figure 2: Three basic kinds of modifications



(a) An atom is a unit of execution with fixed timing

(b) Atom compositions have fixed timing

Figure 3: The atom abstraction

blocks into the signal processing chain, either in the critical path or as additional branches. By inserting new blocks, we can add new functionality to existing signal processing chains. For example, we can add a BER-estimation block for fine-grained channel quality indication to significantly improve rate-adaptation performance [26].

A new wireless stack app that bootstraps a basestation also requires insertion of new blocks.

2.2 Requirements and Design Goals

Our goal is to design a software framework for wireless infrastructure in which new signal processing applications can be easily created and deployed. This requires three properties from the framework:

1. Modularity — For a programmer to easily add new apps, the framework must provide a modular interface supporting tap, tweak, and insert primitives. In using those primitives, the programmer must be able to make local code changes without needing to understand or refactor unrelated parts of the existing implementation.

2. Predictable latency — When a new app is deployed, the underlying communication stack should exhibit hardware-like predictable latency. The programmer should be able to precisely estimate and control the latency down to the order of a microsecond. This will let the programmer ensure that the timing requirements specified by the communication protocol are still met (e.g., 16μs sample processing latency for 20MHz WiFi).

3. High computational throughput — As new apps are deployed, the underlying communication stack should remain computationally efficient. The programmer should have the tools needed to utilize the full computational power of the hardware platform to meet sample-rate processing requirements (e.g., 40Msps for a 20MHz WiFi channel). Modularity and predictable latency should not come at the expense of computational efficiency.

While is easy to meet any of these requirements in isolation, combining them generally leads to tradeoffs.

In a modular system designed for predictability, modules could always be provisioned for worst-case execution times. However, that would an inefficient choice if modules had large gaps between worst and average-case execution times. Another system that dynamically schedules modules with the objective of keeping processors busy would achieve high average processing efficiency. However, it could cause high variability in execution latencies of modules [15]. By programming an application using low-level instructions, a programmer could tightly control processing latencies; by optimizing it as a monolithic piece of software, she could extract the best possible performance from the hardware. However, the resulting implementation would lack modularity.

These tradeoffs make the design of Atomix challenging. However, as we discuss in the next section, our insights about the application structure of wireless dataplanes allows us to design for all of these goals simultaneously.

3 Design

3.1 The atom abstraction

The design of Atomix is based on the key abstraction of an *atom*:

Atom: A unit of execution with fixed timing.

In Atomix, every operation from signal processing to system handling can be implemented as an atom. Further, those atoms can be composed to form more complex atoms. When atoms are composed, their execution times add up, so that if A and B are atoms, $t(A \circ B) = t(A) + t(B)$ (fig. 3).

The Atomix framework provides the substrate to implement signal processing blocks as atoms (sec. 3.2). It also enables programmers to compose those simple atoms into more complex flowgraph atoms (sec. 3.3) and state atoms (sec. 3.4) to tie together an entire signal processing application. Finally, it provides the capability to map atoms onto multiple cores to create fine-grained pipelines exploiting parallelism (sec. 3.5).

The Atomix framework provides atoms for common platform handling operations such as transferring data between cores and interacting with accelerators. The

programmer only needs to implement atoms in C language for custom signal processing blocks. Then, she can tie them together into flowgraphs and a state machine using the language provided by Atomix. The framework provides a high-level declarative API to compose atoms and to map them to multiple cores. Atomix provides a compiler to translate the declarative code into C code. Translated application code and custom signal processing atoms are compiled with the target DSP's native C compiler. Atomix also provides an optimized runtime system (sec. 3.6) to execute compiled atoms efficiently with predictability. The full Atomix application development workflow is described in sec. 3.7.

Block, flowgraph, and state abstractions have one-to-one correspondence with the structure of baseband applications (i.e., the basestation stack and other signal processing applications). Consequently, baseband application software in Atomix has the modularity inherent in the application structure. Since the implementations are entirely made up of atoms, they have predictable timing at every granularity of the modular structure.

To create efficient parallel implementations that can process high sample bandwidths at low latency, the programmer simply ties together appropriate system-handling atoms with signal processing atoms. Then they use the high-level API to map flowgraphs and state machines onto multiple cores. Modularity and predictable timing of atoms simplifies the process of designing pipelines, since the programmer can design a schedule for different bandwidth or latency objectives by simply adding up timings of blocks under various layouts.

In the modular structure of Atomix, signal processing blocks are encapsulated into separate atoms. As a result, atom interfaces provide fine-grained signal tap points; blocks can be tweaked individually by tweaking the corresponding atoms; blocks can be inserted and deleted at fine granularity. Since atoms are tied together using a high-level API, inserting and deleting new atoms is easy.

The timings of atoms add up when they are composed. This simple additive relation allows the programmer to easily infer the effect of a modification on the end-to-end timing of the implementation. On tweaking an atom, she only needs to re-time the tweaked atom and substitute its new cost. On inserting an atom, she simply adds the execution time of the inserted atom to obtain the new end-to-end timing. After predicting the effect of modification on timing, the programmer can easily adjust the multicore layout in the high-level API, if needed to meet processing bandwidth or latency requirements.

We note that the atom abstraction — a unit of execution with *fixed* timing — is an idealized design goal. A software system will inherently have some variability in execution times. For example, the micro-architecture of a processor could affect timing due to bus contention

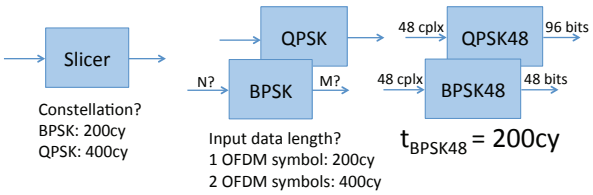


Figure 4: Signal processing blocks to atoms

or hardware instruction manipulation. Atomix strives to achieve sufficiently low variability in execution times of atoms for the purpose of building wireless signal processing applications.

3.2 Blocks decompose into atoms

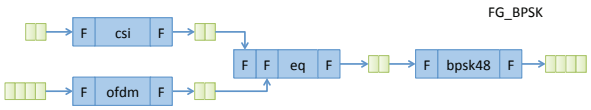
The basic unit of a signal processing application is a signal processing block. A block takes in an input signal and transforms it to another output signal. For example, a constellation bit-slicer block takes in constellation symbols like BPSK or QPSK symbols and produces data bits from them. The execution time of a block depends on the operation it performs, the length of data on which it operates, the processor type (i.e., DSP or accelerator) on which it operates, and the memory location of data buffers it operates on. For a block to be abstracted as an atom, it must execute in known, fixed amount of time. An Atomix signal processing block implements a fixed algorithmic function, operates on fixed data lengths, is associated with a specific processor type, and uses only the memory buffers passed to it during invocation.

Consider the example of a bit-slicing operation, as shown in fig. 4. A slicer block will be decomposed in Atomix as separate BPSK, QPSK, QAM blocks. Each of these blocks will be implemented to accept input and output buffers of fixed lengths, leading to BPSK48 for a BPSK block that takes in 48 complex symbols. The blocks must also be implemented for specific execution core types. If the BPSK48 block could run on both a DSP core and an ARM core, BPSK48DSP and BPSK48ARM would be separate blocks. They may share code internally through common subroutines. By following those decomposition rules, a programmer can implement blocks so that they always executes the same set of instructions on the same kind of processor.

Atomix blocks only make use of memory buffers passed in through function calls, making their memory accesses explicit. Further, the signature of the block implementation is annotated to indicate input and output ports. The Atomix compiler uses I/O port annotations to manage data flow between atoms, as discussed later in sec. 3.3. The Atomix framework provides high-level APIs to control memory placement so that a block al-

Atom	Operation
F_RG	buffer * readGet(fifo)
F_WG	buffer * writeGet(fifo)
F_RD	void readDone(fifo, buffer)
F_WD	void writeDone(fifo, buffer)

Table 1: Atomix FIFO atoms (denoted collectively by F).

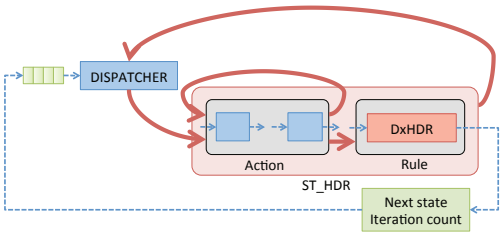


(a) Composing a flowgraph from atoms.

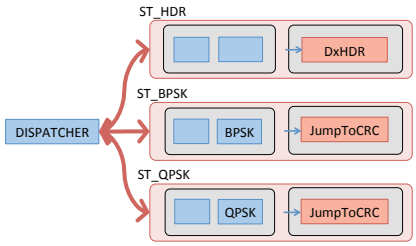
```
#atom:<atomname>:<atomtype>
atom :eq          :EQ
atom :bpsk48      :BPSK48
#fifo:<fifoname>:nbufs=<nbufs>:mem=<memoryid>
fifo :fEqDDSyms   :nbufs=2        :mem=L2
fifo :fDDBits     :nbufs=4        :mem=L2
#flowgraph:<fgname>:<atomname>;+
flowgraph:FG_BPSK:{csi; ofdm; eq; bpsk48;}
#wire:<atomname>:<fifoname>+
wire :eq          :fCSI,fFdDDSyms,fEqDDSyms
wire :bpsk48      :fEqDDSyms,fDDBits
```

(b) Code to compose the flowgraph

Figure 5: Composing a flowgraph from atoms



(a) State control flow. Blue: Data flow, Red: Control flow.



(b) State machine example. Blue: Data flow, Red: Control flow.

```
atom  :dispatcher  :Dispatcher
atom  :dxHDR       :HDRParser
atom  :jumpToCRC   :Jump
fifo  :fDispatcher :nbufs=2    :mem=L2
flowgraph:FG_HDR_AXN:{ ... }
flowgraph:FG_BPSK_AXN:{ ... eq; bpsk48; ... }
flowgraph:FG_QPSK_AXN:{ ... eq; qpsk48; ... }
flowgraph:FG_HDR_RULE:{ dxHDR; }
flowgraph:FG_DD_RULE:{ jumpToCRC; }
wire  :dispatcher  :fDispatcher
wire  :dxHDR       :fDispatcher
#state:<statename>:<FG_action>:<FG_rule>
state :ST_HDR      :FG_HDR_AXN:FG_HDR_RULE
state :ST_BPSK     :FG_BPSK_AXN:FG_DD_RULE
state :ST_QPSK     :FG_QPSK_AXN:FG_DD_RULE
```

(c) Code to compose a state machine

Figure 6: Composing a state machine

ways operates on the same memory types (L2 or DDR etc.). Further, in Atomix, blocks run to completion uninterrupted.

By following simple decomposition rules, Atomix enables the programmer to implement signal processing blocks as atoms. The blocks will run fixed sets of instructions executing uninterrupted on fixed resources using fixed memories. As a result, they will have fixed execution times.

3.3 Flowgraphs are expressed as atoms

The Atomix framework lets the user program application flowgraphs as atoms. For example, the 6Mbps data decoding flowgraph, shown previously in fig. 1, can be expressed as an atom. Flowgraphs are created by tying together signal processing blocks to FIFO queues. FIFOs provide intermediate storage and pass data between signal processing atoms. The framework provides operations to access FIFOs as atoms. This makes an Atomix flowgraph a composition of signal processing atoms and FIFO access atoms. Since a flowgraph is a composition of atoms, it is also an atom in itself.

To program a flowgraph in Atomix, the user declares atoms, FIFOs, wirings between atoms and FIFOs, and the execution sequence of atoms in the flowgraph. Consider the simple signal processing flowgraph shown previously in fig. 2. It shows a channel state information block (CSI) and an OFDM demodulator block (OFDM) both feeding data to a channel equalizer block (EQ) that feeds data to a slicer block (SLICER). A specific realization of this flowgraph with the BPSK48 block is shown in fig. 5a.

In the implementation, block-level atoms and FIFOs are declared with the `atom` and `fifo` API, as shown in 5b. In declaring FIFOs, the programmer is able to control the memory region in which the FIFO will be allocated, thus also controlling the execution time of the atoms that will operate on those FIFOs.

The programmer creates a named flowgraph construct that specifies the sequence in which atoms of the flowgraph will be executed (e.g., `FG_BPSK` in fig. 5b). FIFOs are wired to atoms using the `wire` API to specify the flow of data between atoms. FIFOs are wired to respective input and output ports of the atoms. Based on the wiring information, the Atomix compiler inserts ap-

propriate FIFO atoms (denoted by F, described in table 1) for each wired atom. FIFO atoms draw and return buffers from FIFO queues. These FIFO access atoms have known, fixed execution times.

Atomix implements a simple control flow model where a flowgraph is executed by executing each of its atoms in sequence without interruption. This straight-through execution model implies that the timing of a flowgraph is simply the sum of timings of its constituent atoms, namely, the signal processing atoms created by the user and the FIFO atoms inserted by the framework.

3.4 States are expressed as atoms

At the top level, Atomix applications are state machines. With blocks and flowgraphs implemented as atoms, the final application component that the programmer needs to create is the state machine. Atomix enables the user to program states as atoms.

The main components of a state machine are a dispatcher atom and named state structures, as shown in fig. 6a. A state is made up of two flowgraphs, the *action* flowgraph and the *rule* flowgraph. Control flow starts from the dispatcher atom which invokes the next state in the state machine's transition sequence with an iteration count `n`. When the framework invokes a state, it executes the action flowgraph `n` times followed by the rule flowgraph once. The action typically performs signal processing operations while the rule flowgraph uses the output of the action flowgraph to decide the next state to transition to.

As an example, consider the reference 802.11a receiver previously shown in fig. 1. An implementation of a subset of that state machine in Atomix is shown in fig. 6b. It shows a header decode state, two data decode states and an ACK transmit state. States are declared with the `state` API as shown in table 6c for reference state machine.

A block making state transition decision writes out a decision buffer into the dispatcher's queue, where the decision buffer indicates the next state and the corresponding iteration count `n`. In the example in fig. 6a, the decision atom `DxHDR` translates the header field decoded by the corresponding action flowgraph into a decision output of (< `ST_BPSK` | `ST_QPSK` >, `n`). When control returns to the dispatcher at the end of `ST_HDR`, the dispatcher reads the next-in-queue decision buffer to continue state transitions.

The components that make up the execution sequence of a state are the dispatcher atom and the action and rule flowgraphs. Since a state is composed of atoms, it is also an atom in itself.

3.5 Atoms generalize to multiple cores

DSPs are able to processing high-bandwidth communication applications at low power because of the parallelism of multiple DSP cores, and hardware acceleration. Atomix enables the programmer to easily parallelize and accelerate flowgraphs and states on multicore DSPs.

Multicore flowgraphs. Flowgraphs are parallelized by splitting into smaller flowgraphs, one for each core. Each sub-flowgraph contains a subset of the blocks in the original flowgraph. The programmer may choose any assignment of blocks to the sub-flowgraphs.

Consider the example of the BPSK flowgraph shown previously in fig. 5a. It is shown laid out as a pipeline on three DSP cores in fig. 7a. The single flowgraph (`FG_BPSK`) is split into three sub-flowgraphs, one for each core: `FGBa0` processes the `ofdm` block on dsp0, `FGBa1` processes the `csi` and `eq` blocks on dsp1, and `FGBa2` processes the `bpsk48` block (shown as `b`) on dsp2. Declarations of these flowgraphs are shown in fig. 7c.

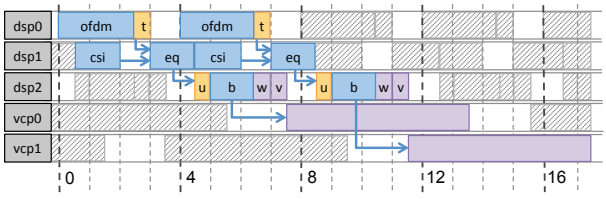
Multicore blocking FIFOs and transfer atoms. When flowgraphs are parallelized, FIFOs are assigned to the same cores as the blocks they are wired to. However, it is possible that a FIFO is wired to multiple blocks that are assigned to different cores. In our example, one such FIFO is at the output of `ofdm` and the input of `eq`. In such a case, the FIFO is replaced with multiple FIFOs, one for each core to which its wired blocks are assigned. The blocks are re-wired to the FIFOs on their own cores to prevent expensive remote FIFO access. This is necessary to preserve the timing of blocks.

When a FIFO is replaced by multiple FIFOs on different cores, data needs to be transferred between them for the flowgraph to compute the correct result. Atomix provides data transfer functionality as transfer atoms. By inserting transfer atoms in the sub-flowgraphs, continuity of the flowgraph is preserved. In the example of `ofdm` and `eq` blocks, the transfer atom `t` is inserted in sub-flowgraph `FGBa0` on dsp0. This atom *pushes* data from the output FIFO of `ofdm` on dsp0 to the input FIFO of `eq` on dsp1. This is shown in figs. 7a and 7b. (The atom `t` could have been inserted into sub-flowgraph `FGBa1` instead. In that case, it would *pull* data from dsp0 to dsp1, and the cost of data transfer operation would be added to `FGBa1`.)

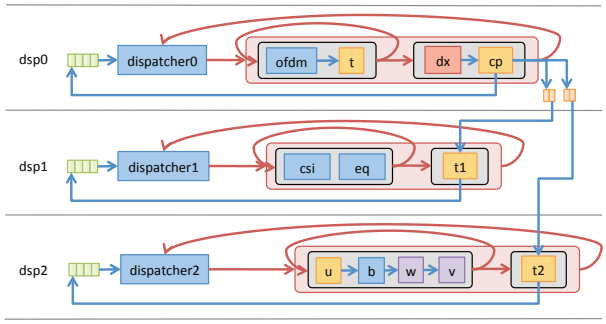
The FIFO read and write atoms are implemented as blocking operations; a read call on a FIFO returns only when it has a filled buffer and, similarly, a write call returns when the FIFO has an empty buffer. By being blocking, FIFO access atoms are able to synchronize ex-

ecution of multiple cores on data dependencies. For example, the `eq` atom on dsp1 is able to run only when the `ofdm` atom on dsp0 has produced a data buffer and `t` has finished transferring it to dsp1.

Simple extensions to the declarative API let the programmer specify multicore assignment of atoms and FIFOs, as shown fig. 7c. Transfer atoms are inserted into flowgraphs like any other signal processing atoms.



(a) Multicore pipeline example.



(b) Multicore state machine structure.

```
#atom :<name>   :<atomtype>    :core=<id>
atom  :dis0     :Dispatcher    :core=0
atom  :dis1     :Dispatcher    :core=1
atom  :t        :TR            :core=0
atom  :dx       :Jump          :core=0
atom  :v        :VCPIssue      :core=2
atom  :w        :VCPWait       :core=2
#fifo :<name>   :nbufs=<nbufs>:mem=<id>
fifo  :fDis0    :nbufs=4:mem=core0.L2
fifo  :fDis1    :nbufs=4:mem=core1.L2
flowgraph:FGBa0:{ofdm; t;}
flowgraph:FGBr0:{dx; cp;}
flowgraph:FGBa1:{csi; eq;}
flowgraph:FGBr1:{t1;}
flowgraph:FGBa2:{u; b; w; v;}
flowgraph:FGBr2:{t2;}
#state:<stname>:core=<id>:<FGaxn>:<FGrule>
state :ST_BPSK  :core=0:FGBa0:FGBr0
state :ST_BPSK  :core=1:FGBa1:FGBr1
state :ST_BPSK  :core=2:FGBa2:FGBr2
```

(c) Code to compose a multicore pipeline. Highlighted fields are multicore extensions to the API.

Figure 7: Parallelizing atoms on multiple cores.

Multicore states. Multicore states enable parallelized execution of the top-level application state machine. A multi-core state structure is made up of a pair of action and rule flowgraphs for each core. When the multicore system enters a state, each core executes its respective flowgraph pair for that state. By setting the core-specific actions of a multicore state to sub-flowgraphs of a parallelized flowgraph, the programmer is able to create efficient multicore processing pipelines.

A three-core version of the BPSK state `ST_BPSK` is shown in fig. 7b. For this state, the cores now use sub-flowgraphs `FGBa0`, `FGBa1`, and `FGBa2` as actions. Similarly, they use `FGBr0`, `FGBr1`, and `FGBr2` for rules. The code to declare the multicore state is shown in fig. 7c.

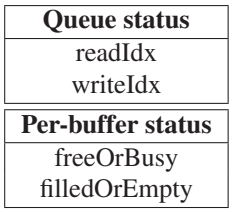
A multicore state machine executes like multiple parallel state machines executing asynchronously. Each core has its own dispatcher atom (fig. 7b. On any core, control flow starts at the dispatcher, passes through the next state to execute, and returns to the dispatcher. Cores synchronize on the state transition sequence by exchanging transition decisions computed in the rule flowgraphs.

In our example, once the system enters `ST_BPSK`, all cores start processing their respective action flowgraphs for the state. These sub-flowgraphs exchange data through the transfer atoms that the programmer inserted when parallelizing them. Collectively, they execute the parallelized BPSK processing pipeline.

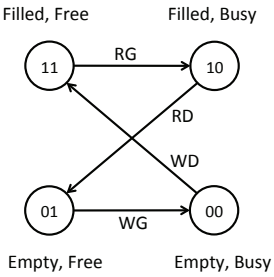
After finishing a state's action iterations, cores independently execute their respective rule flowgraphs for that state. In our configuration, only dsp0 executes the decision atom `dx` for the BPSK state. Its output decision is distributed to each dispatcher through transfer atoms `t1` and `t2`. By executing state-transition decision atoms on a single (though possibly different) core for each state, the programmer synchronizes state transitions across all cores. In this way, the entire system transitions states in lock-step.

Accelerator atoms. The pipeline shown in fig. 7a includes atoms for Viterbi-decoding on Viterbi Co-Processors (VCPs). To use the VCPs, Atomix provides VCPIssue (`v`) and VCPWait (`w`) atoms. A VCPIssue atom can configure a VCP and start its execution. A VCPWait atom can wait for VCP execution to terminate and thus synchronize a DSP core with a VCP core. A hardware accelerator takes a fixed amounts of time to execute a given workload. Consequently, issue and wait operations interacting with an accelerator finish executing in predictable amounts of time, making them atoms. This model extends to other accelerators.

Multicore execution model and timing. The multicore execution model of Atomix is similar to the single-core setting. Blocks access FIFOs for input and output buffers before they execute, they run as soon as buffers are available, and they always runs to completion. Cores communicate by transferring data between FIFOs, and syn-

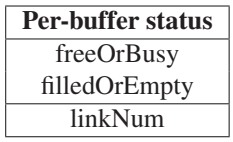


(a) Queue mgmt. data structures

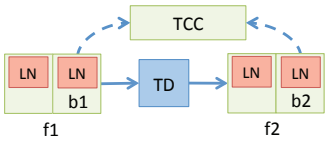


(b) Buffer state machine

Figure 8: Lock-free queue management



(a) Extended buffer state



(b) Transfer scenario

Figure 9: Link number field and Transfer Completion Code (TCC) for handling asynchronous transfers

chronize execution by polling FIFOs with blocking calls. When a block polls a FIFO, its wait duration is determined by execution times of upstream atoms, including data producers and transfer atoms.

In this manner, execution control flow simply mimics data flow in the system. This allows the timing of a multicore atom to be computed by adding up the execution times of atoms on the critical path of execution, i.e., the slowest path of data flow in the parallel execution.

To illustrate the timing model, we analyze the BPSK action flowgraph pipeline. We assume that data is arriving every 4 units of time. We also assume that the blocks have the following execution costs per iteration: `ofdm`:2.5, `csi,eq`:1.5, `b`:1.0, `t,u,w,v`:0.5, and VCP-processing:6.0. The total DSP computation load of this flowgraph is 6.0 units, which cannot be sustained at data arrival rate by a single DSP. When laid out on three DSP cores and two VCPs with transfer atoms, the pipeline is able to meet the processing throughput requirement, as shown in fig. 7a. The pipeline's latency is the timing of the critical path `ofdm-t-eq-u-b-w-v-VCP`, which is 13.5 units.

3.6 Efficient data-flow implementation

The runtime system of Atomix executes fine-grained pipelines with hardware-like efficiency using two main techniques: lock-free FIFOs, and asynchronous data transfers.

Lock-free FIFO implementation. The FIFO implementation in Atomix provides a simple API with four functions (table 1). We design the functions to execute extremely efficiently: the `Get` functions take 40 cycles each, and the `Done` functions take 8 cycles each. Typically, FIFO queues in multicore environments are implemented with locks to serialize access. Atomix does use them because locks are expensive, and they can create timing variability by serializing concurrent accesses in arbitrary order. Atomix FIFO API implementation is entirely lock-free.

In order to operate correctly without locks, Atomix requires every FIFO to have a single-reader and a single-writer (SRSW) at any point in multicore execution. In addition to common readIdx and writeIdx status for the queue, a per-buffer 2-bit (freeOrBusy, filledOrEmpty) status tuple is maintained (fig. 8a). As FIFO API calls are made, the buffer transitions through those states (fig. 8b). If a call cannot succeed (e.g., RG in state 01), it blocks until the buffer reaches the required state for the call to proceed. Only one of the four possible API calls can succeed and hence, modify the data structures in any given state. By the SRSW property, only one FIFO accessor will ever be allowed to write to the FIFO data structure, ensuring race-free operation without locks.

The SRSW constraint may seem too restrictive compared to typical FIFO APIs. However, in our experience, most FIFOs wireless applications naturally have single readers and single writers. Multiple readers or writers from different states could still be wired to the same FIFO since at any given time, the system is in exactly one state.

Handling asynchronous transfers. The FIFO functions `read/writeDone` cannot be used with asynchronous DMA transfer. In order to run them upon DMA transfer completion, DSP cores must be interrupted, which would cause variability in atom execution. To deal with this issue, we introduce the buffer state forwarding mechanism, where the FIFO manager is able to deduce `readDone`s and `writeDone`s without those calls being made explicitly.

To implement buffer state forwarding, we extend per-buffer status field with a linkNum field (LN). On our prototyping platform, TI KeyStone DSPs, DMA channels have associated event registers to indicate transfer completion. We denote this register by TCC (fig. 9). When the DMA transfer atom `TD` issues a DMA request, it sets up the LN field of source and destination buffers to point to the TCC register of the DMA channel used for the transfer. `TD` issues the request and returns. The core that executed `TD` moves on to other atoms. The transfer would finish later asynchronously but the buffers will

continue to be marked busy. However, when the buffers are accessed again in FIFO order, the queue manager is able to identify from non-zero LN field that they were marked busy for an asynchronous transfer. The queue manager then polls the TCC flag pointed by LN. When the TCC indicates transfer completion, the queue manager forwards the state of the buffer as if the corresponding `readDone` or `writeDone` was called on the buffer.

3.7 Full app development workflow

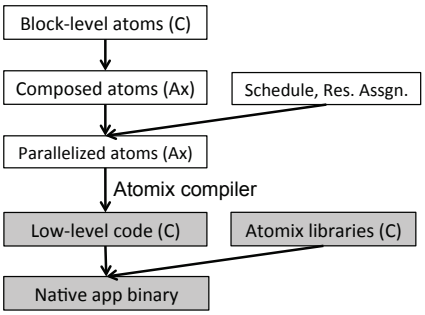


Figure 10: Full Atomix app development workflow

Stages in Atomix app development. To summarize the Atomix framework design, the following is an overview of the application development workflow (fig. 10). First, the user implements signal processing blocks as atoms in C language. Next, she composes blocks into flowgraphs and states using the declarative interface of Atomix. Then, she computes a parallelized schedule and resource assignment that will meet the latency and throughput requirements of the application, and incorporates it in the declarative app code. This high-level app code is then compiled down to low-level C code by the Atomix compiler. Finally, the low-level application C code is compiled with the platform's native C compiler and linked against Atomix runtime libraries into a binary. The modular structure of Atomix applications and the streamlined development flow let the user rapidly iterate designs and optimize performance.

Algorithmic scheduling. The execution model of Atomix makes it possible to algorithmically compute the best parallelized schedule for an application. Blocks can be profiled individually for execution times. Then, finding the optimal schedule for a flowgraph can be expressed as a resource assignment problem with dependency constraints. FIFO access costs, data transfer costs and accelerator access costs can all be modeled as additional constraints. We have been able to formulate the flowgraph scheduling problem as an Integer Linear Program (ILP). The flowgraph scheduling problem in Atomix is similar to the instruction-loop scheduling problem solved by VLIW compilers [17, 18]. Incorporating algorithmic scheduling in the Atomix compiler can simplify the application development flow even further.

4 Building an 802.11a Receiver in Atomix

We put Atomix framework's capability of providing low latency and high throughput to test by using it to implement a 10MHz 802.11a receiver called Atomix11a. We implement 802.11a as a benchmark due to its particularly demanding requirements. It uses the same bandwidth modes as LTE (5, 10, 20 MHz), imposing similar processing complexity. However, it places much more stringent frame-decoding latency constraints than LTE - 64μs for 5MHz and 32μs for 10MHz compared to more than 1ms for LTE. To stress the system, we implement the highest throughput Modulation Coding Scheme (MCS) in 802.11a, MCS 7: 64-QAM with 3/4 coding rate, which operates at 27Mbps on a 10MHz channel.

We first evaluate Atomix11a and show that (1) Atomix11a can decode over-the-air frames in real-time on the TI 6670 a 4-core, 4-Viterbi accelerator 1.25GHz DSP, (2) Atomix11a can achieve 36μs processing latency with 1.5μs of variability, sufficient to meet requirements of a 5MHz 802.11a channel and close to meeting 10MHz requirements, and (3) most of the atoms we built for Atomix11a have predictable timing. Next, we describe the use of Atomix to implement a modular 802.11a receiver that has low latency and high throughput on the low power (7 Watt) TI 6670 DSP. We could implement the receiver in about 3000 lines of declarative application code (excluding individual block implementations in C language).

4.1 Evaluation of Atomix11a

We first demonstrate that Atomix11a is a robust, faithful implementation of 802.11a signal processing. Then we evaluate Atomix11a's latency and timing variability, and finally we evaluate the runtime of Atomix11a's atoms to show that they come close to Atomix's fixed runtime atom abstraction.

Atomix11a exceeds receiver sensitivity requirements. An 802.11a compliant receiver must be able to decode 1,000 byte packets with a Packet Error Rate (PER) of 0.10 at -65dBm receive power over an AWGN channel, as measured at the RF frontend's antenna port [7]. In our experiment, we feed signal over an RF co-axial cable to emulate an AWGN channel without multipath reflections. On the receiver, we use the high-quality RF frontend of an R&S FSW spectrum analyzer to digitize received signal into baseband I/Q samples, so the experiment focuses on the robustness of Atomix11a

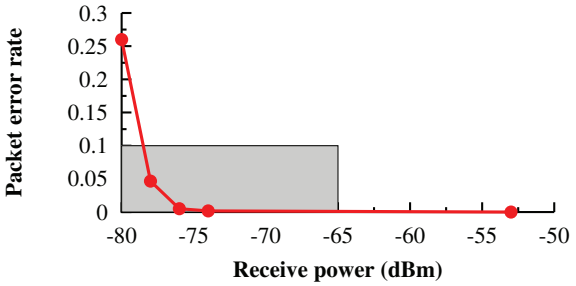


Figure 11: Atomix11a exceeds receiver sensitivity specifications (gray box).

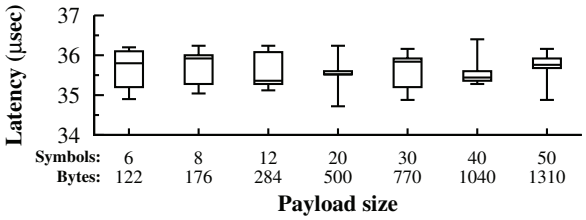


Figure 12: Atomix11a processing latency is at most 36.4μs and it varies at most by 1.5μs.

baseband running on the TI 6670 DSP, not the quality of the RF frontend. Fig. 11 shows the results of this experiment. Atomix11a exceeds 802.11a's requirement of 0.10 PER at -65dBm; it achieves a 0.046 PER at -78dBm.

Atomix11a operates robustly indoors. The second robustness test is to see if Atomix11a can decode packets sent over-the-air in challenging indoor multipath channels causing frequency selective fading. Robust 802.11a receivers use a computationally intensive zero-forcing channel equalizer to equalize the sub carriers.

We setup a 6 meter link across an office and transmitted 100,000 802.11a frames, each 1,000 bytes, at MCS 7 (64-QAM, 3/4 coding rate), and over the 10MHz channel at 2.479GHz. We used a USRP2 RF frontend connected to the TI 6670 DSP with gigabit ethernet. Both the transmitter and the receiver were connected with 3 dBi omnidirectional antennas. The receiver successfully decoded (verified CRC) for 99,999/100,000 frames. Therefore Atomix11a is capable of an extremely low PER of 0.001% in an indoor office environment, and is likely a faithful implementation of 802.11a.

Atomix11a has low processing latency. Next, we perform an end-to-end test of Atomix framework's ability to support a low latency, low timing variability, and high throughput signal processing chain. We transmitted 200 frames each of different sizes (6-50 symbols, 122-1310 bytes) to the USRP2 which is connected to the TI 6670 DSP. We compute the processing latency by subtracting the frame processing time of Atomix11a from the frame's airtime. We measure Atomix11a's maximum processing latency, as well as the range over which it varies.

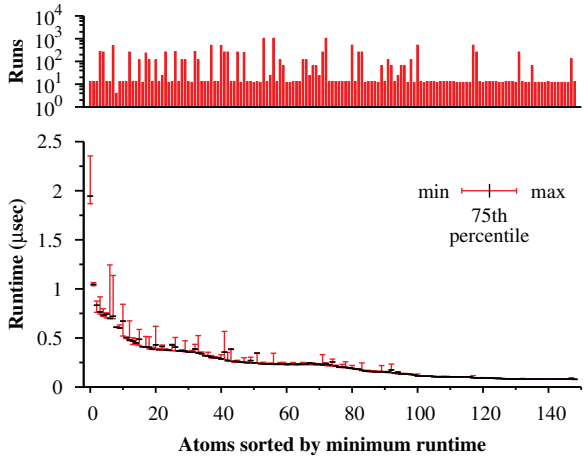


Figure 13: Atomix11a atoms have low timing variability.

Fig. 12 shows the results of this experiment. The whiskers are the minimum and max of each of the 200 frames, and the boxes show the 25th, 50th, 75th percentiles. For all the frame sizes tested, the minimum and maximum processing latency is similar, indicating that the composition of atoms adds minimal latency between frames. The max processing latency is 36.4μs, 1.14$\times$ the requirement of 32μs for 802.11a. Although 36.4μs latency is low, the Atomix11a implementation could be further optimized to meet protocol latency requirements. Specifically, there is room to optimize the implementations of individual atoms, and to algorithmically compute the parallelization schedule of atoms that minimizes decode latency.

Most Atomix11a atoms have low timing variability. In the final experiment, we observe the variability of the runtime of every atom in Atomix11a. We expect some variability due to L1 caching and micro-architectural sources of variability like bus contention (sec. 3.2). We instrumented each of the atoms in the Atomix11a implementation with a lightweight cycle counter that records the cycle count of each execution of the atom. We measured the runtime of every atom that executes while receiving 12 frames of 500 bytes (20 OFDM symbols at MCS 7). Most atoms run every OFDM symbol of the frame.

Fig. 13 shows the min, max, and 75th percentile runtime of all of the 150 atoms that were executed in Atomix11a (bottom), as well as the number of times each of those atoms were run in our experiment (top). Most

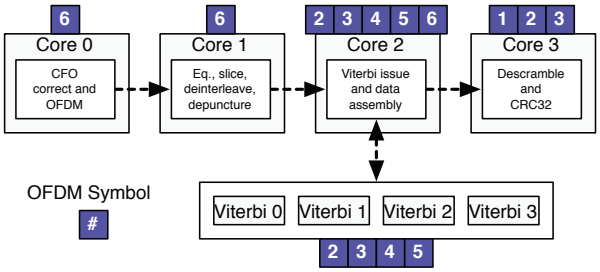


Figure 14: Fine-grained OFDM symbol pipelining in Atomix11a's data decode state.

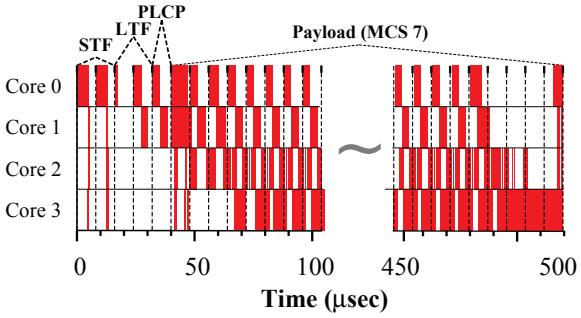


Figure 15: Core utilization of Atomix11a while it receives a 1500 byte MCS 7 frame at 10MHz.

atoms have a fixed or insignificant runtime (atoms 60-150). These atoms include both framework atoms such as memory transfers, as well as Atomix11a computation atoms such as the 64-QAM soft slicer and the PLCP's soft deinterleaver.

Atomix11a's atoms have at most a 0.486μs range between their max and min runtime due to L1 caching and micro-architecture. The atom with the largest runtime range (atom 0) is the channel estimator, which is the most computationally intensive atom in Atomix11a. Although the difference between the max and min runtime for these atoms can be significant, for all atoms the 75th percentile of runtime is close to the minimum. This shows that the atom abstraction holds well in practice. Further, execution times of different atoms have low co-variance. This explains the low end-to-end packet decoding variability of 1.5μs. This variability can be reduced further by disabling L1 caching on the TI 6670 and managing the L1 SRAM in software with Atomix transfer atoms.

4.2 Implementation of Atomix11a

Modularity resulting from the atom abstraction enabled us to implement Atomix11a with fine-grained pipeline parallelism, which was crucial in achieving high through and low latency on the TI 6670 DSP. We implemented signal processing blocks as fine-grained atoms that operate on one OFDM symbol (8μs worth of samples at 10MHz) every iteration, as discussed earlier. The Atomix API enabled us to compose the block-level atoms into flowgraphs and states and schedule them for high performance in a low power budget of 7W.

Precise timing of atoms allowed us to spread the pipeline over all the cores and accelerators so it would execute without stalls and achieve the required processing throughput of 10Msps reliably. Fine-grained atoms allowed us to pipeline data processing with data reception to achieve low decode latency of 36μs.

Fine-grained pipeline structure. The fine-grained pipeline structure of Atomix11a is shown in fig. 14. It depicts a snapshot of the pipelining in payload decoding state. At any point in the steady state, six OFDM symbols are being processed in the pipeline. With fine-grained resource allocation API of Atomix, we could lay out the pipeline so that each DSP core and Viterbi co-processor (see ahead) contributed to the processing of an OFDM symbol. This enabled the highest sample processing throughput achievable by the hardware resources.

The core utilization map resulting from our fine-grained pipeline is shown in fig. 15. It depicts the processor's active and idle time on all four DSP cores while it is receiving a 1500 byte 802.11a MCS 7 frame. The vertical grid indicates the arrival of a symbol to Atomix11a. As the figure shows, all cores are participating in a pipeline to process every arriving OFDM symbol.

Parallelizing Viterbi decoding for high throughput. Optimal Viterbi-decoding is a sequential algorithm. However, in practice, the decoded sequence of bits (codeword) can be partitioned into overlapping chunks that can be decoded in parallel with negligible performance penalty under certain overlap size constraints. We use this scheme to use all 4 VCPs for decoding the same WiFi frame, where each chunk is one-half OFDM symbol with appropriate padding of surrounding bits to satisfy overlap constraints. Such a scheme has also been used in BigStation [30] where it is described in more detail.

Optimizing latency with overlapping states. Atomix allows different cores to operate simultaneously in different states. This feature enables the implementer to use all cores to execute as many states at once as possible, and thus provide lower latency than if the states ran in serial. For example, fig. 15 shows as the LTF symbol arrives we split its processing across two cores, and same for the PLCP symbol[2]. The split point for the LTF

[2]Note that with a 10MHz signal, the LTF and PLCP processing ends before the arrival of the next symbol. However, with a 20MHz signal

state is after CFO estimation because the CFO estimates can immediately be applied on the PLCP symbol when it arrives. Then, when the channel estimates finish on core 1 in LTF, the rest of the PLCP is processed. Core 1 uses the channel estimates to complete the PLCP processing with the symbol transferred from core 0. In summary, Atomix enabled pipelined processing of two symbols in two states at the same time on two cores.

Easy to program. We implemented the Atomix11a receiver in 3000 lines of Atomix's high-level code. The Atomix compiler translated the high-level code into about 30,000 lines of low-level application-specific C code. Atomix reduces the LoC effort by an order-of-magnitude. Further, it makes the development process significantly easier through its abstractions compared to directly writing low-level code. All of the blocks in Atomix11a's signal processing library were implemented with less than 300 lines of C code each.

Low resource footprint. The Atomix atom abstraction requires each configuration of a signal processing block to be split out as a separate block, e.g. `BPSK48`, `BPSK96`, `QPSK48`. To avoid code size explosion, we implement similar blocks as wrappers on a common parameterized kernel (e.g., `BPSK<N>`). Atomix API also provides primitives to instantiate parameterized blocks in the atom declarations.

Using those techniques, Atomix11a has code size of 468 KByte per core. This comfortably fits in the 512 KByte of L2 SRAM per core that we allocated as program memory. The data size on any core is at most 145 KByte out of 512 KByte of L2 SRAM allocated for data memory. Code size could be further optimized by creating per-core binaries containing only code executed on each of the cores. Another approach is to write the blocks in C++ using templates. It is also possible to partition the SRAM in favor of program memory.

5 Modifiability

We demonstrate the modularity of the Atomix 802.11a receiver by adapting it to two new applications: long-distance links, and phase-array location signatures.

5.1 Adapting to long-distance links

Long-distance wireless links appear in settings like rural connectivity [6], point-to-point backhaul and community networks [4]. They are challenging because the *delay spread* of the multipath wireless channel also grows with link distance. However, WiFi is designed for shorter distance links. As a case-study of Atomix modifiability,

(4μs interval), the PLCP symbol will arrive during LTF processing.

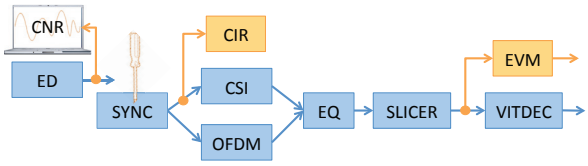
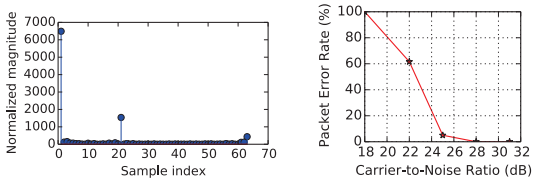


Figure 16: Modifications for long-distance app



(a) Measured CIR (b) PER with long-CP

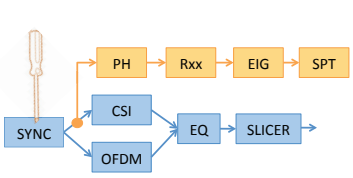
Figure 17: Performance on a two-tap multipath channel. Long cyclic-prefix (CP) is critical for long outdoor links.

we demonstrate the process of systematically diagnosing and adapting our base WiFi receiver to work well over an emulated long-distance link.

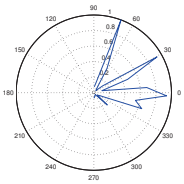
We emulate a long-distance channel that has two multipath components: the line-of-sight component, and a reflection delayed by 2us (20 samples) that is 12dB lower in strength. On this channel, the unmodified Atomix11a receiver resulted in 100% packet error rate (PER). To ascertain that the signal strength is not the limiting factor, we tap the baseband chain to read off sample energy values being computed by the energy-detection block (ED), as shown in fig. 16. On the simulated transmission with an average carrier-to-noise ratio (CNR) of 45dB, our tap revealed the CNR to be 45.4dB, confirming sufficient signal strength for successful decode.

Ruling out signal strength limitation, we suspected a high-distortion channel. To look deeper, we inserted an error-vector-magnitude (EVM) computation block after slicing the constellation. On a distortion channel, we would expect a high EVM value, or equivalently, a low SNR indicated by the EVM. The EVM block runs in 0.64us on core 1. We set it to compute on the output of the PLCP field. Core 1 had enough cycles (fig. 15) in the PLCP decode state to run EVM without adding latency to packet decode. Our EVM block indicated only 19.2dB SNR, much lower than 45.4dB CNR. This strengthened the suspicion of a poor channel.

To find the ground truth about the channel, we inserted a block to compute the time-domain channel impulse response (CIR) using the long training field (LTF) of the preamble, same as used for frequency-domain channel estimation. The CIR block runs in 3.4μs. We set it to run core 3, which has enough spare cycles after packet detection to run CIR without hurting packet decode. The



(a) Modifications for AoA app



(b) AoA spectrum

Figure 18: AoA spectrum app

CIR block showed the channel response to be as shown in fig. 17a. It accurately recovered the emulated channel and pointed us to the root problem: two strong components 20 samples apart, and a delay spread too high for the standard WiFi OFDM cyclic prefix length of 16 samples.

Armed with this knowledge, we implemented the solution of increasing the cyclic prefix (CP) length to 32 samples. To implement the longer CP communication mode, only the SYNC block needed tweaking. It is responsible for discarding the cyclic prefix from each OFDM symbol before passing it to rest of the processing chain. We tweaked it to discard 32 samples instead of 16. After implementing the longer CP, performance of the link came back to the expected CNR-PER quality on a benign channel. The observed performance of CP length 32 over the 2us two-tap channel is shown in fig. 17b. On the same reception as before, the EVM block now indicated SNR of 38.7dB, much closer to the CNR of 45.4dB, confirming that signal distortion had been effectively corrected.

By simply tapping existing signals, tweaking a block and inserting a few blocks, we could methodically adapt the receiver to a new application setting. The whole exercise took just a few hours of programmer-time to modify or add a total of 20 lines of declarative code (after we implemented the individual signal processing blocks).

5.2 Adding location signatures

Location signatures of a wireless transmitter can be computed using phase-array processing at a MIMO AP receiver. MUSIC [19] is a popular algorithm to compute angle-of-arrival (AoA) spectrum which is used in many location-based systems like [28, 29]. It is desirable to embed AoA spectrum computation on the AP to save bandwidth incurred in remote computation, and to make location estimates available at the AP at sub-millisecond latencies. For such scenarios, we demonstrate addition of an AoA computation application to our Atomix 802.11a receiver.

The set of blocks used to compute AoA spectrum is shown in fig. 18a. A flowgraph is composed with the phase-multiplier block (PH), correlation matrix computation block (Rxx), a complex Hermitian eigen-decomposition block (EIG), and a spectrum computation block (SPT). We leverage the baseband receiver to detect WiFi packets using preamble-detection. Preamble samples are tapped and copied on a FIFO for AoA computation. Once the packet decode finishes, AoA computation is invoked. A sample of AoA spectrum computed on the DSP processor is shown in fig. 18b.

The entire AoA computation chain takes 530us with room for further optimization. Its components are: PH: 1.5us, Rxx: 21us, EIG: 178us and SPT: 330us. For eigendecomposition, we use a FORTRAN-to-C translated routine from EISPACK [2]. Our simple implementation is already able to compute an AoA spectrum in about half of a millisecond at the base-station. We are able to reproduce the results from [28, 29], which we omit for brevity. To add the AoA app, we needed to add only about 30 lines of declarative code to the base Atomix11a receiver (in addition to blocks in C).

6 Related work

Existing frameworks suited to building baseband applications using abstractions of blocks and flowgraphs have only targeted GPPs and FPGAs. Also, operating systems for building realtime programs on DSPs do not provide the right abstractions for modular communication applications. To the best of our knowledge, Atomix is the first system to enable high-performance, modular communication applications on multi-core DSPs.

Modular frameworks for GPPs and FPGAs. GNU Radio [5] is a rapid prototyping toolkit for signal processing applications. It provides interfaces to connect blocks into flowgraphs on GPPs and DSPs [10]. However, unlike Atomix, GNU Radio's abstractions do not have guaranteed timing. They depend on abundant processing resources for real-time performance.

The SORA [21] software radio and its modular software architecture called Brick [11] provide real-time performance of wireless applications on GPPs. With similar modularity goals, SORA/Brick and Atomix share similarities in the programming interfaces. However, the differences in GPP and DSP architectures make for very different designs of the two systems. SORA's design includes mechanisms like binding threads to cores, masking interrupts from cores and special memory handling for optimization cache performance on signal processing apps. These apply well to the GPP environment with a general-purpose operating system but do not carry over to the DSP architecture.

Ziria [20] is a domain specific language targeted to GPPs for implementing efficient PHY designs. It facilitates concise and error-free expression of PHY control logic. It also provides support for automatic vectoriza-

tion of individual signal processing blocks. The advantages of programming signal processing apps in Ziria, currently limited to GPPs, are complementary to those of Atomix. They can be extended to DSPs by a compiler that translates Ziria programs to Atomix programs, using Atomix as a convenient abstraction layer.

The Click modular router [9] provides an abstraction of blocks and flowgraphs for building packet-based network infrastructure. The expressiveness of Click's simple elements and FIFOs was an inspiration for Atomix. However, Click is not suited to developing wireless applications on DSPs. For instance, Click's elements lack timing guarantees; the Click dynamic scheduler is a source of timing variability.

StreamIt [24] provides a flowgraph model and intuitive syntax to program stream processing applications. The StreamIt programming language explicitly reveals parallelization opportunities to the compiler so it can exploit task, data, and pipelining parallelism. In that sense, StreamIt's compiler complements Atomix. However, StreamIt's abstractions lack guaranteed execution times. Moreover, StreamIt encourages embedding branches within blocks that can cause variability. Atomix forbids this behavior, and requires all branches to be expressed as their own atoms.

AirBlue [12] simplifies design of cross-layer signal processing applications for FPGAs. Like Atomix, AirBlue also requires FIFOs between signal processing blocks to enable modular modifications. However, modular implementation of an application for predictable and efficient execution poses different challenges on multi-core DSPs and FPGAs.

Real-time operating systems for DSPs. Real-time operating systems such as QNX Neutrino [16], WindRiver VxWorks [27] are used in automotive, robotics and avionics systems. DSPs have their own real-time operating systems such as TI SYS/BIOS [22], and 3L Diamond [1]. In principle, the real-time guarantees provided by these systems (e.g., priority-scheduling, predictable interrupt timing) can be applied to real-time wireless infrastructure applications too. However, these abstractions can only be indirectly mapped to the blocks and flowgraphs that make up baseband applications. Real-time OSes generally provide some form of FIFO-based interprocess communication that can be used to create flowgraphs. However, such FIFOs are not designed for the fine-grained pipeline parallelism that Atomix's lock free FIFOs enable.

7 Discussion

Atomix in the development pipeline: Atomix aims to make signal processing applications easy to deploy on DSPs in the wireless infrastructure.However, Atomix can also enable at-scale prototyping and testing of wireless applications. Platforms like USRP E310 and E110 integrate DSPs and ARM cores with commodity RF frontends in embedded form-factors. These platforms are ideal for building wide-scale research testbeds. Atomix makes them easy to program for rapid prototyping.

Limitations of Atomix: Atomix pushes conditionals out of blocks to the top level construct of state machine. This makes stepping through conditionals much slower than having them embedded within blocks. Most baseband apps spend majority of their time in computationally heavy signal processing blocks, making the cost of stepping through states negligible. However, for applications dominated by data-dependent workloads like iterative computations like successive interference cancellation or searching/sorting, Atomix can be inefficient.

8 Conclusion

Atomix is a framework for building and widely deploying modular, predictable latency, high throughput baseband signal processing applications. The key abstraction that enables Atomix is that of an atom — a computation unit with fixed execution time. Atomix demonstrates that by abstracting operations as atoms, ease of programming and fine control for predictable timing can be provided simultaneously without sacrificing efficiency for signal processing applications. We show that it can be used to deploy apps in both WiFi and LTE infrastructures.

We plan to make Atomix available to the research community and enable the implementation and evaluation of new baseband apps at scale. We also plan to further develop Atomix to be a programmable dataplane for a software-defined radio access network. Specifically, we plan to investigate open APIs (analogous to OpenFlow) that programmable basestations should expose to enable networks to tightly manage packet processing on a per-flow basis, and enable software defined control over the wireless infrastructure.

9 Acknowledgments

We thank Kyle Jamieson and Jie Xiong for providing us with a reference implementation of SecureArray [29]. We thank Jeffrey Mehlman for help with implementation, and Yiannis Yiakoumis for suggestions on presenting the content. We also thank our shepherd Deepak Ganesan for his valuable feedback, and the anonymous reviewers for their insightful comments. This work was supported by the NSF POMI grant.

References

[1] 3L Diamond. http://www.3l.com/products/3l-diamond.

[2] EISPACK. http://www.netlib.org/eispack.

[3] 3GPP. Specification Release version matrix. http://www.3gpp.org/specifications/67-releases.

[4] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2005.

[5] E. Blossom. GNU radio: tools for exploring the radio frequency spectrum. *Linux J.*, 2004:4–, June 2004.

[6] K. Chebrolu, B. Raman, and S. Sen. Long-distance 802.11b links: Performance measurements and experience. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2006.

[7] IEEE. 802.11a-1999.

[8] K. Joshi, S. Hong, and S. Katti. Pinpoint: Localizing interfering radios. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 2000.

[10] S. Ma, V. Marojevic, P. Balister, and J. H. Reed. Porting GNU Radio to multicore DSP+ARM system-on-chip – a purely open-source approach. In *Karlsruhe Workshop on Software Radios*, 2014.

[11] Microsoft Research. Brick Specification. *The SORA Project*, 2011.

[12] A. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan. Airblue: A system for cross-layer wireless protocol development. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2010.

[13] K. Nikitopoulos, J. Zhou, B. Congdon, and K. Jamieson. Geosphere: Consistently turning MIMO capacity into throughput. In *Proc. ACM SIGCOMM*, 2014.

[14] R. Patra, S. Nedevschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer. Wildnet: Design and implementation of high performance WiFi based long distance networks. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[15] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 2000.

[16] QNX Software Systems, Ltd, Kanata, Ontatio, Canada. http://www.qnx.com.

[17] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. International Symposium on Microarchitecture (MICRO)*, 1994.

[18] J. Sánchez and A. González. The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures. In *Proc. International Conference on Parallel Processing (ICPP)*, 2000.

[19] R. O. Schmidt. Multiple emitter location and signal parameter estimation. *IEEE Transactions on Antennas and Propagation*, 1986.

[20] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agulló. Ziria: A DSL for wireless systems programming. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[21] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker. Sora: High performance software radio using general purpose multi-core processors. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[22] Texas Instruments. *SYS/BIOS (TI-RTOS Kernel)*.

[23] Texas Instruments. TMS320C6670 - Multicore Fixed and Floating-Point System-on-Chip. *http://www.ti.com/product/tms320c6670*.

[24] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proc. International Conference on Compiler Construction (CC)*, 2002.

[25] A. Vigato, S. Tomasin, L. Vangelista, V. Mignone, N. Benvenuto, and A. Morello. Coded decision directed demodulation for second generation digital video broadcasting standard. *IEEE Transactions on Broadcasting*, 2009.

[26] M. Vutukuru, H. Balakrishnan, and K. Jamieson. Cross-layer wireless bit rate adaptation. In *Proc. ACM SIGCOMM*, 2009.

[27] Wind River Systems, Inc, Alameda, CA, USA. `http://www.vxworks.com`.

[28] J. Xiong and K. Jamieson. Arraytrack: A fine-grained indoor location system. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[29] J. Xiong and K. Jamieson. Securearray: Improving WiFi security with fine-grained physical-layer information. In *Proc. ACM Conference on Mobile Computing and Networking (MobiCom)*, 2013.

[30] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, and Y. Zhang. BigStation: Enabling scalable real-time signal processingin large MU-MIMO systems. In *Proc. ACM SIGCOMM*, 2013.