Feature-Oriented Cache Designs

Justin Deters* and Ron K. Cytron
j.deters@wustl.edu*,justin@simplerose.com*,cytron@wustl.edu
SimpleRose* and Washington University
St. Louis, Missouri, USA

ABSTRACT

We propose a novel methodology for designing and implementing caches. Instead of a monolithic specification of all features, we compose features to arrive at a desired implementation.

To achieve this goal, we leverage programming language types and finite-state machine states as hooks for feature specification and deployment. Our approach accommodates immediate integration of cache components and their variations, as each feature is woven automatically into a base implementation.

We designed and implemented such a cache using the RISC-V-mini implementation, presenting resource usage for 10 distinct endpoints across both the instruction and data caches.

1 INTRODUCTION

In their 2018 Turing award lecture [12], John Hennessey and David Patterson advocated for the adoption of the RISC-V architecture by industry and research. Experiment metrics (*e.g.*, instruction counts, cycles per instruction) published using RISC-V can be more meaningfully compared because they are based on a common instruction set. The design of the RISC-V ISA allows for omission of features not needed, resulting in smaller footprint for embedded systems. Ideas developed in research can more easily be adopted by industry when both use a common architecture. These are all arguments in favor of the adoption of RISC-V.

While these goals are certainly attractive, it is nonetheless a daunting task to modify a characterization of RISC-V to add, modify, or remove features. A large number of implementations of RISC-V currently exist [22]. Each is generally a separate characterization intended to support applications of interest. The following implementations are of the greatest relevance to our work:

- RISC-V-mini [16] is a 3-stage pipelined implementation of 32-bit RISC-V processor.
- Rocket Chip [3] is a robust implementation of 64-bit RISC-V using a 5-stage pipeline.
- BOOM [6] is an out-of-order execution implementation of RISC-V that shares some of the same libraries of Rocket Chip.

From the RISC-V-mini to BOOM, these implementations add features to create an increasingly sophisticated implementation of RISC-V, yet most hardware features are unique to each characterization.

In our approach, features are defined so that they can be *woven* into an implementation (a base characterization along with features already chosen and deployed) using well-defined type and state information. A feature-oriented approach does not come for free and it requires the following development discipline. A base implementation and its features must contain sufficient types and states so that other features can use those as hooks for deployment. Newly conceived features may require refactoring both the base implementation and affected features, to provide necessary hooks for a new feature's deployment. Continual refactoring is seen as an advantage in the software world, as it raises the abstraction level of a project and tends to clarify logic and purpose of code. We provide examples of such refactoring in Section 4.3.

Section 2 describes work upon which our ideas and implementation are based. Our contributions in this paper are as follows: Section 3 describes how we use aspects to modify finite-state machines (FSMs), borrowing from software aspect-oriented languages. Section 3.2 describes a library we have written for Chisel in Scala that supports straightforward specification of features that modify FSM behavior. Section 4 describes our feature-oriented cache characterization. We provide examples of features, the advice needed to realize the features, and pictures of the cache's FSM before and after feature inclusion. The caches we generate using this approach have been deployed and tested in RISC-V-mini. Section 5 describes our experiments and results that show the savings in area for various endpoints in the cache-design feature space.

We show in this paper that our approach allows cache features that we have written to be easily incorporated into RISC-V-mini. Furthermore, our methodology should allow and encourage others similarly to develop features and implementations that compete with or augment what we have accomplished. We aspire to create a marketplace for publishing and using all manner of cache-design features.

2 PRIOR WORK

Chisel, Scala, FIRRTL. Most hardware designs, including caches, are specified using hardware *characterization* languages such as Verilog [4] or VHDL [5]. While those languages offer some abstraction in the form of parameters and restrictive loops, they lack high-level concepts such as classes, traits, and types that allow cleaner and more robust specifications of software systems.

Hardware generation languages like Chisel, on the other hand, allow the designer to write a program whose execution generates a design. Chisel, based on Scala, has abstractions such as types, classes, and traits that allow a richer specification of a cache.

Transformations of the hardware design, for example to weave in features of interest, can be accomplished at the Scala level (using Scala meta) or FIRRTL level (using tree operations on Chisel's intermediate form). While, we chose to target Chisel due to its maturity and level of adoption, this technique could be suited for other hardware generation languages as well. For example, our tools could be directly used with SpinalHDL [1] as a Scala based language.

Aspect-Oriented Programming. Large software systems are subject to the "tyranny of the dominant decomposition" in that the objects and methods that form the nouns and verbs of the system

are those that are most prominent in the design. Such systems, however, often have cross-cutting concerns. For example, the logging of class and method activity can require action at each method's call and return. Aspect-oriented programming (AOP) [15] allows the specification of *join-points* to model method calls and returns, along with *advice* that specifies what should be done at those points of execution. A general specification of a set of join-points is called a *pointcut*. AspectJ [21] is a robust implementation of an AOP using Java.

For our work, AOP provides the approach we need to modifying finite-state machines, such as the one controlling a cache, to incorporate features as desired. Internally, the cache's operation is governed by a finite-state machine. Features are woven into the design as modifications of the finite-state machine (a base specification along with other features of interest).

Feature-oriented Programming in Software Systems. Our approach to cache design is influenced by the success of this idea in the software world. For example, a CORBA event channel [7] offers many features in a complete implementation, but a given application may need only a few of those features. Using an aspect-oriented approach, features were incorporated as desired, yielding implementations whose footprint and delay are affected only by necessary features [13]. That work shows that by omitting unnecessary features, implementations can be generated that take much less area (code) and that have significantly better performance (latency) than the full-featured, monolithic version. That work used the aspect compiler AspectJ [21] and a code base for an event channel and its features written in Java [11].

3 FEATURE-ORIENTED FINITE STATE MACHINES

FSMs serve as the basis of control for many components of an architecture implementation, including cache control and multicache coherence, bus arbitration, and network protocols. We first describe generally how aspects can transform such machines to implement features of interest. We then describe a library we have implemented in Scala/Chisel to simplify expression of aspects for FSMs. When run as a Chisel program, the resulting FSMs synthesize Verilog with the features of interest.

3.1 Finite-State Machine Aspects

We follow [19] in the treatment of aspects for FSMs. Essentially, a state is like a method and a transition between states is like a method call. We next describe the specific formulation of aspects for FSMs.

As an example to show the advantages of an AOP approach for cache designs, consider the inclusion of a write-back feature for a cache. The feature requires modification of the cache design in multiple places:

- Each cache line must contain a dirty bit.
- A line's dirty bit must be set if the program modifies any byte in that line.
- When a line is replaced, the dirty bit must be consulted to determine if the line needs to be written to the backing storage.

Without an AOP approach, the FSM is modified by hand in various portions of its specification to realize the above behaviors. If subsequently a write-through approach is desired, the write-back logic must be edited out and the write-through logic deployed. This is an error-prone and tedious activity. With an AOP approach, the designer can easily change from write-back to write-through by simply selecting the desired feature for inclusion. For write-back, the above FSM modifications are expressed together in a single feature, whose inclusion affects the FSM appropriately as described above. The elements of the feature are:

- A *pointcut* specifies where in the FSM changes should occur. The pointcut then yields a set of states.
- *Join-points*, which are specific states or transitions in the FSM, at which modifications to the FSM occur.
- *Advice* in the form of FSM modifications is applied.

3.1.1 Pointcuts. Pointcuts denote a set of states or a set of transitions in an FSM, typically conditioned on some predicate p. Evaluated at run-time when a design is generated, p can be any Boolean-valued function that selects items of interest, typically based on types, traits, or values of instance variables.

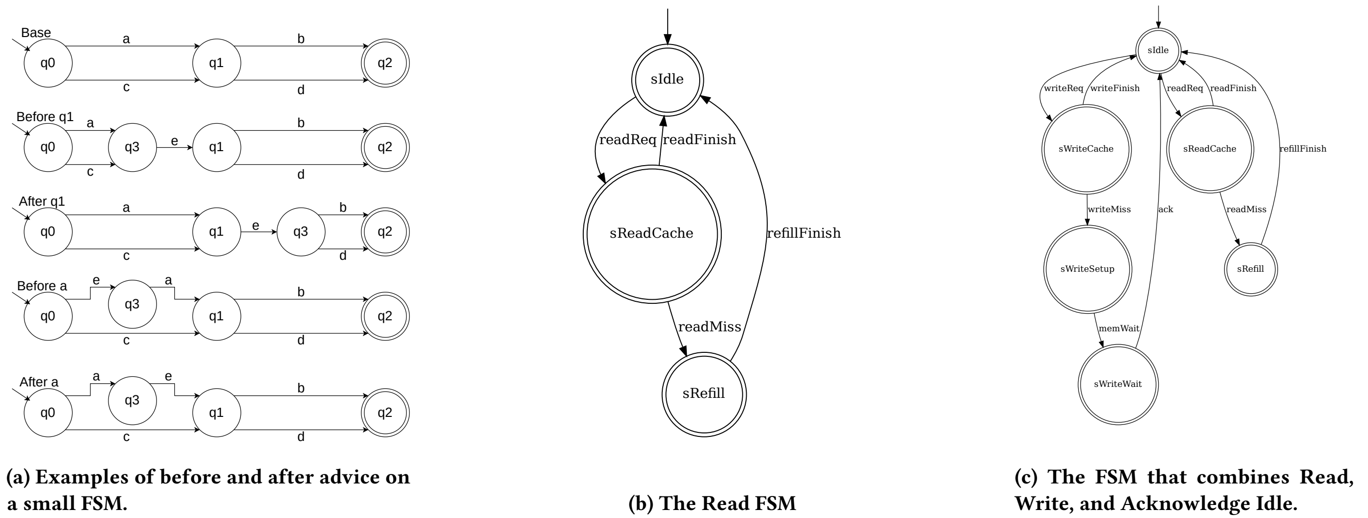
State pointcuts are analogous to a set of function bodies in a program. Just as a function body is unique within a program, each state join-point within the pointcut will also be unique. Pointcuts that specify transitions on tokens are analogous to a set of function calls. Unlike function bodies, calls to a given function may appear multiple times within a program. This is also true for a token within a FSM. Thus, the resulting pointcut will contain all the *transitions* where the predicate p was satisfied.

3.1.2 Advice. Continuing the analogy, an FSM is essentially a set of function bodies (states) where function calls (transitions) occur at the end of each body. For our purposes, advice is inserted into the execution flow of an FSM to implement a given feature. The advice affects the actions performed before, during, or after a state, including modification of transitions to target new or other states.

We consider the usual forms of *around*, *before*, and *after* advice (*cf.* AspectJ [21]). Around advice takes the join-point and replaces it with something else. States replace states and transitions replace transitions. Before and after advice come in the form of token-state or state-token pairs in order to add a new path through the FSM. Figure 1a shows the effect of different types of advice on a simple FSM. Before advice (q3, e) on q1 takes all paths into q1 and directs them into q3, on e the machine transitions to q1. The reverse is true for after advice (e, q3) on q1. All paths leaving q1 now leave q3 and the transition from q1 to q3 takes place on e. Similarly, before advice (e, q3) on a places e going into q3 and a out of q3. The reverse is true for after advice (q3, e) on a. In each of these cases, e can be the empty string λ if the behavior of the FSM should change but not require additional input to do so.

3.2 Software Library

We have incorporated all of the above mechanics into a software library. Because we targeted Chisel for hardware generation, this library is also built in Scala (https://github.com/wustl-frisc/foam). As is, it can be dropped into any Chisel hardware generator to construct and generate hardware FSMs. The software library contains



(a) Examples of before and after advice on a small FSM.

(b) The Read FSM

(c) The FSM that combines Read, Write, and Acknowledge Idle.

Figure 1: Example FSMs.

a set of base classes, `FSM`, `state`, and `token` that can be arbitrarily extended by hardware designers to suit their applications.

3.2.1 Writing Aspects for Finite-State Machines. We have modeled the programming interface using the well established aspect language for Java, AspectJ [21]. This provides a familiar interface for aspect practitioners. No new syntax is needed to implement our library; everything is specified using ordinary Scala/Chisel.

Pointcuts. Figure 2 demonstrates the creation of a pointcut from our implementation. Here, the predicate is written using a Scala `match` statement. The `Pointcutter` will iterate over all the states in the FSM and add them to the pointcut if the predicate evaluates to true. A predicate can be written as any arbitrary Scala code as long as it follows the interface of consuming a state (or token) and returns true if it satisfies the properties defined in the predicate. The result of this example would create a pointcut of states where the type of the state is `WriteWaitState`. This the state in cache where it waits for acknowledgement from the backing store in the AXI protocol.

```
val waitPointcut = Pointcutter[State, WriteWaitState](
    nfa.states,
    state => state match {
      case s: WriteWaitState => true
      case _ => false
    }
)
```

Figure 2: A pointcut from our implementation.

Advice. Figure 3 shows the construction of advice using the `AfterState` class. Any arbitrary Scala code can be executed inside the advice body as long as it returns advice. `StateJoinpoint` provides reflexive access to the join-point as well as its context. In this advice, in the `match` statement, we test to see a transition already exists coming out of the state using the join-point context. If we do not do this, the advice would apply again. The result of this advice would be to insert a new `ack` edge between every `WriteWaitState` and the `Idle` state. This transition represents the completion of the write transaction. Once the the cache receives acknowledgement from the backing store, it returns to the idle state to wait for another transaction.

```
AfterState[WriteWaitState](waitPointcut, nfa)((
     thisJoinpoint: StateJoinpoint[WriteWaitState],
     thisNFA: NFA) => {
    thisJoinpoint.out match {
        case Some(t) => (None, thisNFA)
        case _       => (Some(WriteFSM.ack, ReadFSM.sIdle
    ), thisNFA)
    }
})
```

Figure 3: Advice using the pointcut from Figure 2 from our implementation.

3.2.2 Hardware Generation. Our software library uses Chisel constructs to generate hardware FSMs. These are the exact same hardware constructs that Chisel uses under the hood to generate their hard-coded FSMs. Before generation, end users associate each state and token with a string ID. Then, each state is given a conditional block and the transitions placed inside with their own conditional blocks. The end user is returned a handle to the FSM. The handle has one wire associated with each state. This signal is asserted when the state becomes active. The handle also has one assignable wire for each token. When this signal is asserted and it is associated with current active state, the transition occurs.

In Figure 4 we show how an FSM handle is used. Since the FSM interacts with the outside world via a handle, the *implementation* of

what happens when a state is asserted is completely decoupled from the FSM itself. This is extremely useful from a feature-orientation standpoint. New implementation information can be added as features are added. As long as the FSM's handle remains in scope and a `module` barrier is not crossed, the signals in the handle can be used anywhere.

```
when(fsmHandle("sReadCache")) {
  when(hit) {
    io.resp.valid := true.B
  }
}
fsmHandle("readFinish") := !io.req.valid && hit

```

Figure 4: Using an FSM handle in our software library.

4 FEATURE-ORIENTED CACHE

We next use our AOP FSM library to create a feature-oriented cache design. We present the base machine in terms of the hooks it exposes for features, and then we define features using those hooks. While the result is presented in its final form here, the process is one of refactoring and evolution as new features are included.

4.1 Feature Decomposition

The stateful nature of the cache requires feature decomposition across both the internal hardware and the FSM that controls it. However, the instruction and data cache designs are not mutually exclusive. They both share the same base design for both the FSM and the cache overall. All endpoints for either cache are the result of applying features to the same base designs. All our designs are built for the RISC-V-mini processor [16].

4.1.1 Finite-State Machine. We have divided the FSM's mechanisms into five separate features.

- **Read** provides all the functionality for a read-only cache.
- **Write** provides the functionality to write to memory. This is an *abstract feature*. By itself it does not have enough to complete cache transactions. One of the two following features must also be included.
- **Acknowledge Idle** returns the cache back to the idle state after the backing store acknowledges a write.
- **Acknowledge Read** returns the cache back to the read state after the backing store acknowledges a write.
- **Dirty Accounting** provides the necessary transitions for a write-back cache.

The evolution of our FSM is shown in Figures 1b and 1c. Note, we reuse the **Read** feature all of our FSMs without any modification. Furthermore, the FSM in Figure 1c and the write-back FSM[1] largely share the same structures. They share all the same states and nearly all the same transitions. Our feature-oriented approach to FSMs allows us to take the FSM in Figure 1c suitable for write-through and simply *add* new edges to make it suitable for write-back. In practice, we have found that extending this design was relatively easy using our tool, taking only about an hour. Since the generator itself calls into our software library, all of our FSM features are subsumed into our hardware generator features (discussed in Section 4.2.1).

[1] Omitted due to space constriants.

4.2 Compositional Correctness

Currently, neither of our tools check for compositional correctness. Even in AspectJ [9] it is possible to write to aspects which their composition results in infinite execution of the program. These tools, for instance, would allow the end user to apply the features for both a direct-mapped cache and fully-associative cache at the same time. Bluespec [2] accomplishes compositional correctness through the guarded atomic actions. This approach focuses on *how* the circuit functions, rather than *what* hardware is being included. Compositional correctness for our tools, however, is beyond the scope of this paper and is left for future work.

4.2.1 Hardware Generator. Below are the 9 separate hardware generation features for our cache. In all cases, the cache implements the AXI [18] interface for communication with the backing store and follows the request-response interface already present in the RISC-V-mini datapath.

To show the extensibility of our feature-oriented approach we have also included an esoteric **Dusty** [10, 17] feature, an idea proposed to reduce unnecessary write-backs, particularly for reference counts. When the cache is about to write a dirty line back to memory, a dusty cache will first consult a secondary "image" of the line to see if the line has *actually* changed since it was first brought into cache. This is useful for fetched values that change temporarily but return soon to their originally fetched value.

- **Base System** provides the structure to which all other features can be applied.
- **HasBufferBookeeping** enables just enough bookkeeping to hold one buffered memory transaction.
- **HasMiddleAllocate** builds out the internal memory for the cache and allows cache lines to be allocated.
- **HasWriteStub** stubs off the write channel for read-only memory.
- **HasWriteFSM** introduces the write FSM with both the **write** and **Acknowledge Idle** features applied.
- **HasSimpleWrite** provides the hardware necessary to write to memory.
- **HasInvalidOnWrite** invalidates the buffer or cache line if the tag matches.
- **HasMiddleUpdate** allows the internal memory of the cache to be updated by writes from the datapath.
- **Dirty Accounting** modifies the FSM with both the **Acknowledge Read** and **Dirty Accounting** features. As well as providing all the hardware necessary for handling dirty cache lines.
- **Dusty** creates a second internal memory for the cache that acts as an image of main memory. A cache line is only considered dirty if the internal memory and image of the line are not equal.

4.2.2 Endpoints. As stated earlier, our features combine to form endpoints for both an instruction and data cache. Endpoints for both the instruction and data caches can exist in the same system at the same time (how this is achieved is discussed in Section 4.3).

This means that our cache system can realize 10 separate overall endpoints, two choices for instruction cache and five choices for the data cache. Figure 5 lists all of the endpoints for the instruction and data caches, as well as the features combined to achieve each of them.

The **Read-Channel** and **Write-Channel** endpoints implement the hardware necessary for memory transactions as well as a small buffer for use with the AXI interface. In all other cases our generator creates a direct mapped cache with 256 sets and 16 byte cache lines. Set-associative and fully-associative caches are embraced well by our approach, but they are beyond the scope of the experiments in this paper.

Endpoint	Features
Read-Channel	HasWriteStub, HasBufferBookeeping
Read-Only	HasWriteStub, HasMiddleAllocate
Write-Channel	HasWriteFSM, HasSimpleWrite, HasBufferBookeeping, HasInvalidOnWrite
WriteBypass	HasWriteFSM, HasSimpleWrite, HasMiddleAllocate, HasInvalidOnWrite
WriteThrough	HasWriteFSM, HasSimpleWrite, HasMiddleAllocate, HasMiddleUpdate
WriteBack	HasWriteFSM, HasSimpleWrite, HasMiddleAllocate, HasMiddleUpdate, Dirty Accounting
Dusty	HasWriteFSM, HasSimpleWrite, HasMiddleAllocate, HasMiddleUpdate, Dirty Accounting, Dusty

Figure 5: Endpoints for the Instruction and Data Caches.

4.3 Feature Implementation

Our implementation combines Chisel, our library, and another aspect-oriented tool, Faust [8] that inserts code into Scala programs via abstract-syntax tree modification. We use Faust to apply all of our features *automatically* to the RISC-V-mini codebase. As such, since Faust directly modifies Scala code, there are no restrictions on what features and where we can apply them, as long as they are written in legal Chisel code.

By taking a feature-oriented approach we were able to achieve a high level code leverage and design reuse between features. On average each feature is only 43 lines of generation code. Our largest feature, **Dirty Accounting** is just 104 lines of code. This feature, by itself, adds all the necessary hardware to build a write-back cache out of a write-through cache.

4.3.1 Hardware Generation Through Types. Rather than tying the structure of the hardware generator code to the hardware hierarchy, we propose separating hardware into types with specific generation tasks. Our base `Cache` class is responsible for creating and initializing the hardware needed for optional cache features.

We have further subdivided the cache into a “frontend” and “backend” with classes backing each of these. These two classes are responsible for generating the hardware needed for communication with the datapath and backing store respectively. In the future, if the design moved away from AXI to a different protocol, the backend could be swapped out for a different class that has the same interface, with appropriate chances to the FSM and IO for the module.

Hardware generation types give us a way to *optionally* generate hardware through class methods. Instead of writing one implementation for a read-only cache and another for a read-write, we can call different methods to generate the read and write components separately while still sharing the same overall design. In addition, the hardware design my easily be extend by adding new hardware generation methods or overriding old ones.

Critically, hardware as types creates “hooks” for us to grab onto, to modify the abstract syntax trees with Faust. Withing our generator, the instruction cache and data cache are *subtypes* of `Cache`. We direct Faust to modify the cache of a specific type with the desired features. This enables us to easily create endpoints modifying both caches without having to worry about cross contamination of features between the two.

4.3.2 Features as Traits. Traits in Scala function similarly to Java interfaces. Unlike Java, Scala traits support multiple inheritance and Scala code can be called from within a trait. RISC-V-mini, Rocket Chip, and BOOM all have traits that implement some functionality. However, our approach differs significantly in that we package *whole features* into traits. Thus, to add a feature to a design, the type can just be extended with the trait containing the feature.

Consider the **HasMiddleAllocate** feature in Figure 6. By extending the `InstructionCache` with **HasMiddleUpdate** instead of **HasBufferBookeeping** the read-channel is transformed into a read-only cache. By combining this with our type encapsulation technique (Section 4.3.1), zero hand rewriting of the design is required to make this extensive change.

Since Faust modifies the abstract syntax tree of the hardware generator, creating different endpoints only requires instructing Faust to extend the cache classes with different traits. All of our endpoints except for **WriteBack** and **Dusty** are created this way.

4.3.3 Inserting Hardware into Traits. Sometimes, there are features that cannot be succinctly captured in a single trait, but instead crosscut the hardware generator. To implement a Write-Back cache, new features must be added to the FSM, hardware must be created for storing the dirty bit and transmitting that information, as well as crating or changing conditions for cache updates and memory transactions.

In this instance, we heavily utilize the aspectual nature of Faust. The majority of the implementation of **Dirty Accounting** is written as an aspect in Faust. However, **Dirty Accounting** requires more than just the extension of traits with new traits. We use Faust's ability to weave new code into the hardware generator to modify and extend the designs contained in traits. **HasMiddleUpdate** is modified to hold the dirty bits, as well as connect with the new mechanisms in the FSM for write-back. **HasWriteFSM** receives new features to account for dirty cache lines and writing back to memory. **HasSimpleWrite** is updated with new conditions for writing to memory. Conveniently, this can all be contained in one selectable aspect within Faust.

This is further exemplified by the **Dusty** feature. The whole implementation of **Dusty** is captured in a single aspect within Faust. Figure 7 shows the relatively small amount of work needed to implement **Dusty**. First, once again utilizing the type isolation technique (Section 4.3.1), we create our reference image of memory by instantiating another `Middleend` class. Then, we provide advice *around* the dirty signal to judge a cache line dirty only if it does not equal what exists in the reference image of memory.

5 PERFORMANCE AND AREA

We have taken our feature-oriented designs all the way up through simulation and synthesis. The hardware generator is implemented

```
trait HasMiddleAllocate extends Cache {
    val middle = new Middleend(fsmHandle, p, address, tag
     , index, valids)
    val (oldTag, readData) = middle.read(buffer,
     nextAddress, offset, Some(hit), Some(cpu))
    middle.allocate(Cat(mainMem.r.bits.data, Cat(buffer.
     init.reverse)), readDone)
}
```

Figure 6: HasMiddleAllocate feature.

```
class Dusty extends Aspect {
  Before ("HasMiddleUpdate", "dirty", "Bool") {
    val dusty = new Middleend(fsmHandle, p, address, tag,
      index, valids)
    val (_, dustyData) = dusty.read(buffer, nextAddress,
     offset)
    dusty.allocate(Cat(mainMem.r.bits.data, Cat(buffer.
     init.reverse)), readDone)
  }
  Around ("HasMiddleUpdate", "dirty", "Bool") {
    proceed() && (dustyData =/= readData)
  }
}
```

Figure 7: Dusty feature in Faust.

	Read-Channel			
benchmark	**Write-Channel**	**Write Bypass**	**Write Through**	**Write Back**
median	3.59	3.11	3.11	3.02
multiply	2.81	2.78	2.78	2.77
qsort	3.48	3.38	3.19	3.00
towers	3.78	3.69	3.36	2.46
vvadd	3.71	3.07	3.07	3.00
Average CPI	3.47	3.21	3.10	2.85
	Read Only			
benchmark	**Write-Channel**	**Write Bypass**	**Write Through**	**Write Back**
median	2.41	1.93	1.93	1.87
multiply	1.37	1.33	1.33	1.33
qsort	2.11	2.01	1.83	1.64
towers	2.84	2.76	2.42	1.61
vvadd	2.64	2.00	2.00	1.93
Average CPI	2.27	2.01	1.90	1.68

Figure 8: Cycles per instruction for each RISC-V benchmark, by endpoint.

Endpoints	**LUTs (normalized)**
readChannel-writeChannel	1.00
readChannel-writeBypass	1.25
readChannel-writeThrough	1.35
readChannel-writeBack	1.52
readChannel-dusty	1.57
readOnly-writeChannel	1.25
readOnly-writeBypass	1.49
readOnly-writeThrough	1.56
readOnly-writeBack	1.76
readOnly-dusty	1.81

Figure 9: The area in LUTs of the endpoints.

in Chisel 3.5.1. All designs were emitted as Verilog. We simulated the large benchmarks from the RISC-V tests repository [14] using Verilator 4.214 [20]. Designs were synthesized for an xc7a100tcsg324-1 FPGA using Vivado 2022.1 at 50 MHz with the Vivado synthesis defaults. Figure 8 shows the cycles per instruction for each benchmark with the average CPI of all the benchmarks displayed at the bottom. Figure 9 shows the synthesized area of the whole synthesized chip design in LUTs.

The smallest endpoint, **readChannel-writeChannel** takes only 55.2% of the area of our largest endpoint, **readOnly-dusty**, which reduces CPI by 47.5%. By feature-orienting our design, we have enabled a fine grain of design space exploration. We can see the CPI drop nearly linearly as more features are added to the design. Coarse grained analysis is not lost in this technique either. Comparing the two instruction cache endpoint groups we see that the addition of an instruction cache has an average decrease in CPI of 1.19 cycles and an average increase in LUTs of 438.

While this result is not surprising, adding an instruction cache should improve performance at the cost of increasing design area, we speculate that this technique can help to quickly identify the properties of new designs and save designers time. For instance, the **readOnly-writeChanel** has a lower CPI than any of the **readChannel** endpoints and takes less area than all of them except **readChannel-writeChannel**. This analysis tells us going down the path of more advanced data cache features is not worth the area cost if there is no data cache. Again, this is not surprising, but we posit that there are other opportunities in hardware designs for this sort of analysis.

A feature-oriented approach can help hardware designers better balance trade-offs while maintaining high levels of design reuse. Without the need to start hardware designs from scratch, designers can accomplish quicker prototyping while maintaining a high level of analysis and code reuse.

6 CONCLUSION

We have presented a new methodology for authoring cache designs. In place of a monolithic design, capabilities and implementation choices for the cache are captured as aspectual features, which are woven into a base design to obtain an implementation. The result is a more structured code base, with features authored separately from the base implementation. We have described a library we have authored in support of writing and weaving aspects into Chisel hardware designs. We have presented results of using this approach to generate various cache-design endpoints.

With this approach, caches with specific feature choices, many more than the ones we have presented here, can be generated as quickly as features can be chosen, with Verilog produced (quickly, in seconds) as the output of executing the resulting Chisel program. That Verilog characterization can then undergo synthesis using typical toolchains.

ACKNOWLEDGEMENTS

We would like to acknowledge SimpleRose for seeing the value of this work and continuing to fund the main author after the publication of this paper.

REFERENCES

[1] Sallar Ahmadi-Pour, Vladimir Herdt, and Rolf Drechsler. 2022. The MicroRV32 framework: An accessible and configurable open source RISC-V cross-level

platform for education and research. *Journal of Systems Architecture* 133 (2022), 102757. https://doi.org/10.1016/j.sysarc.2022.102757

[2] Arvind. 2013. Bluespec and Haskell. In *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-Specific Languages* (Boston, Massachusetts, USA) *(FPCDSL '13)*. Association for Computing Machinery, New York, NY, USA, 1–2. https://doi.org/10.1145/2505351.2508149

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[4] IEEE Standards Association et al. 2006. IEEE standard for Verilog hardware description language (IEEE 1364-2005). *http://standards. ieee. org/* (2006).

[5] Jean-Michel Berge. 1992. *VHDL Designer's Reference*. Kluwer Academic Publishers, USA.

[6] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Technical Report UCB/EECS-2015-167. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[7] X. Defago, P. Felber, and R. Guerraoui. 1997. *Reliable CORBA Event Channels*. Technical Report 97/229. Departement d'Informatique, EPFL.

[8] Justin Deters and Ron K. Cytron. 2021. Performance Counter Design Variation in Rocket Chip via Feature-Oriented Programming. In *Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021)*.

[9] Eclipse Foundation. 2022. *AspectJ*. https://www.eclipse.org/aspectj/

[10] Scott Friedman, Praveen Krishnamurthy, Roger Chamberlain, Ron K. Cytron, and Jason E. Fritts. 2005. Dusty Caches for Reference Counting Garbage Collection. In *Proc. of Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*.

[11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2022. *The Java Language Specification Java SE 19 Edition*. Technical Report. Oracle. https://docs.oracle.com/javase/specs/jls/se19/html/index.html

[12] John Hennessy and David Patterson. 2018. John Hennessy and David Patterson Deliver Turing Lecture at ISCA 2018. https://www.acm.org/hennessy-patterson-turing-lecture.

[13] Frank Hunleth and Ron K. Cytron. 2002. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems* (Berlin, Germany). ACM Press, 38–45. https://doi.org/doi.acm.org/10.1145/513829.513838

[14] RISC-V International. 2022. riscv-tests. https://github.com/riscv-software-src/riscv-tests.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. *Proceedings of ECOOP '97* (1997).

[16] Donggyu Kim. 2022. riscv-mini. https://github.com/ucb-bar/riscv-mini.

[17] Praveen Krishnamurthy, Roger D. Chamberlain, Ron K. Cytron, and Jason E Fritts. 2006. Evaluating Dusty Caches on General Workloads. In *Proceedings of the Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Boston, Massachussetts.

[18] ARM Limited. 2021. *AMBA AXI and ACE Protocol Specification*. Technical Report. ARM Limited.

[19] Roy Rønmo, Ragnhild Kobro Runde, and Birger Møller-Pedersen. 2013. Confluence of aspects for sequence diagrams. *Software & Systems Modeling* 12 (2013), 729–824.

[20] Wilson Snyder. 2004. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*.

[21] The AspectJ Organization. 2001. Aspect-Oriented Programming for Java. www.aspectj.org.

[22] Wikipedia. 2022. *Existing RISC-V Implementations*. https://en.wikipedia.org/wiki/RISC-V#Existing