Performance Counter Design Variation in Rocket Chip via Feature-Oriented Programming

Justin Deters
Ron Cytron
j.deters@wustl.edu
cytron@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

ABSTRACT

Performance counters provide critical information to developers about how well their applications work on a given platform. Currently, the Rocket Chip generator includes a performance counter system that allows variation only of the *number* of counting registers. For RISC-V to be attractive across a wide range of applications, other variations should be possible. For example, a developer may be interested only in microarchitecture events. Perhaps the events of interest may yield more information if counted only within a specific function.

We reformulate the performance counter subsystem into separate and orthogonal *feature units* that can be applied to Rocket Chip either individually or in combination. We developed a tool that applies features by manipulating Scala abstract syntax trees and automatically determines feature dependencies. We also designed a simple domain specific language to construct such features.

By *feature-orienting* the implementation of the performance counters, we offer Rocket Chip developers a much larger range of possible implementations. Developers can select any subset of RISC-V events for monitoring. New events can be easily introduced into attached processors for monitoring. If desired, events can be counted only within certain ranges of the program counter.

We reconstruct the current performance counters using our features as well as other interesting design endpoints. We present results showing the resources needed for those configurations.

1 INTRODUCTION

The use of a common architecture has many advantages, and RISC-V has been proposed as such an architecture [16]. RISC-V's success in that regard will depend on its ability to adapt to diverse applications, ranging from low power, embedded devices to supercomputers. Across that spectrum, some features will be necessary and some features can be eliminated, both at the architecture and micro-architecture levels. A monolithic implementation of RISC-V may contain the union of all possible features, but achieving a desired subset becomes a tedious and error-prone process. Moreover, some features may have variations that are optimized for certain applications, such as networking or graphics processors.

We have been experimenting with a feature-oriented approach to construct RISC-V-based architectures using aspect-oriented inspired programming to weave in the features of interest. Our work begins with the Rocket Chip [2] characterization of RISC-V, and we leverage the rich type system and tree-editing capabilities of Chisel [3] and Scala [12] to weave features.

Using this approach, only those features of interest are included, with absolutely no trace of excluded features for a given application. Moreover, our approach opens up a marketplace of feature implementations that can easily be incorporated into a RISC-V platform by meeting the feature specification.

In this paper we focus on one feature and its variations, namely performance counters, which are defined in the RISC-V architecture specification. However, some applications may have no need of performance counters, and their omission can result in smaller footprint and less power. Some applications may need only a subset of the counters, while others may need more counters that are provisioned in the architecture.

We reformulate performance counters into the following orthogonal features.

Which events are counted? Applications can specify which events in RISC-V are of interest, including architected instructions, microarchitecture events, and cache events. Moreover, our approach allows the introduction of events and their counters into coprocessors and other logic deployed alongside RISC-V.

When are events counted? As designed, RISC-V counters *always* count events when they occur. However, developers are often interested in events only within certain methods or regions of program execution [4]. Our approach allows the introduction of address-range specifications that restrict when events are counted.

We describe the process by which we rendered performance counters to be feature-oriented and present the results of useful endpoints in the counters' feature-design space.

2 PRIOR WORK

In prior work [5–7, 10] have shown feature-oriented design to be useful for feature management and resource reduction for software in the context of the CORBA [11] *Event Channel*, which is responsible for publishing events to subscribers. The nature and disposition of the publishers and subscribers dictate the features needed in the Event Channel. In its monolithic form, unnecessary features avoid execution, but their presence still consumes memory. Moreover, even a simple runtime check concerning a feature's inclusion can adversely affect latency for time-critical applications.

Instead of removing features they did not want, they took a compositional approach. By stripping down the system and then adding features only when needed. They were able to get significant savings in memory area and increases in performance.

A feature-specific Event Channel was evaluated and compared with a monolithic implementation in terms of its area and throughput [10]. For area, the feature-specific version was 1.4–3 times smaller than the monolithic version: an appreciable savings for an embedded systems application. The throughput for the feature-specific version was approximately 131,000 events per second as compared to approximately 1,600 events per second in the monolithic version, with unnecessary features present but disabled.

They accomplished this through aspect-oriented programming [9]. Aspect-oriented programming is useful when implementation concerns do not fit cleanly into a single module within a program. An **aspect** is a unit that contains **advice** about how implementation information should be applied to a codebase. A piece of advice is applied to a **pointcut** which is a set of **join points** which are individual points in the codebase. Typically, the act of applying the implementation information is called "weaving" as the aspect applies *across* the different modules. In the event channel, they use aspects to capture features and then conditionally apply them to produce different feature sets.

Chisel Aspects. Chisel does contain an aspect library [8]. However, in our preliminary work, we found this library to be too restrictive to accomplish the types of transformations we wished. As such, we have built our own tool discussed in Section 4.

3 CURRENT MONOLITHIC DESIGN

The current performance counter system design follows the structure encouraged by traditional hardware definition languages rather than the modularity of an object oriented language like Chisel. Regardless of whether the user actually requires performance counters, the *entire* infrastructure for creating the system remains in the generator. This includes all the routing infrastructure needed to get the signals to the counters themselves. In total there are 9 separate classes where the performance counter system is implemented, with 3 of those just routing signals. Furthermore, most of the infrastructure is hard coded into the constructors of the classes, obscuring where the performance counter implementation actually is.

A monolithic approach like this not only makes it harder to understand and customize the performance counter system, but also the whole Rocket Chip system itself. Hard coding and entangling features unintentionally complicates the maintenance and extension of existing features. These unintended side effects are exemplified in the performance counter system[1].

Both removing functionality and extending functionality of the performance counters completely changes the configuration values for the resulting system. This is true for both individual events and event sets provided. The bitmask value that configures a counter to capture an event is determined by the *order* in which it appears in the generator. Thus, adding or subtracting events changes the bit patterns for all the other events that follow it in the code. This can become very complicated if users only want a subset of the provided events with the bit patterns changing in unexpected ways.

[1]Referenced code can be found at github.com/chipsalliance/rocket-chip/blob/master/src/main/scala/rocket/RocketCore.scala

4 FEATURE APPLICATION USING SCALA TREES

Instead of monolithically including all possible hardware features in a generator, our approach is to separate out the generation infrastructure into *feature units* (i.e. aspects) which then can be *applied* into the generator code base.

In order to automate the process of applying features, we have created a tool in Scala called **Faust** (feature application using Scala trees) [2]. Faust is still in the early phases of development, but already provides functionality for capturing features in a custom DSL and automatic collection of feature dependencies.

4.1 Scala Trees

Faust uses Scalameta [14] to apply features by manipulating Scala abstract syntax trees (ASTs). A feature unit may *crosscut* many different parts of the code base with implementation information that changes many different code modules. For example, a feature that implements a event to be monitored may add IO and connections to many different modules when routing the signal.

To apply features, Faust takes in a directory of Scala code, the feature units themselves, and dependency information. Each file is parsed into its corresponding AST. The AST is then traversed to find a point where feature implementation information needs to be applied and then the tree is transformed producing a modified AST. This process is repeated until all implementation information is existed. Finally, a copy of the original file is saved and a new one with the resulting AST is produced.

Opposed to the aspect library in Chisel [8] which operates upon the produced FIRRTL, we are directly manipulating the Scala tree giving us the ability to influence *any* part of the generator along with retaining the full Chisel build system.

4.2 Feature DSL

In order to ease the creation of features, we provide a small DSL embedded in Scala. Although Faust is not a proper aspect-oriented compiler [9] we do borrow the general syntax found in aspect-oriented programming languages such as AspectJ [15]. Listing 1 shows a feature and its syntax.

Keywords. `before (join point)` and `after (join point)` indicate where the advice information will transform the AST. Currently that is either directly before or directly after the join point. `extend (class)` allows a class to be extended with new type information. `insert (code)` tells Faust what new implementation information needs to be inserted into the AST. Sometimes a user might wish to refine the join point by using a specific type context `in (context)` provides this functionality. Finally `register` adds the advice to the system.

To add a new feature to the system, users need only to extend the `Feature` class and fill the body with implementation advice. In order to make sure that all features can be properly implemented, users must also provide dependency information for new features.

[2]Code at github.com/jdeters/faust

```
class CounterSystemFeature () extends Feature {
  val numPerfCounters = 4

  //modifying Rocket Core
  val RocketCoreContext = q"class RocketImpl"
  after (q"hookUpCore()") insert (q"csr.io.counters
     foreach { c => c.inc := RegNext(perfEvents.evaluate(
     c.eventSel)) }") in RocketCoreContext register

  //modifying CSR
  val CSRContext = q"class CSRFile"

  val stat = q"${mod"override"} val counters = Vec(${
     numPerfCounters}, new PerfCounterIO)"
  extend (init"CSRFileIO") insert (q"{ $stat }") in
     CSRContext register

  before (q"buildMappings()") insert (q"val
     numRealCounters = ${numPerfCounters}") in CSRContext
      register

  after(q"buildMappings()") insert q"performanceCounters.
     buildMappings()" in CSRContext register

  before (q"buildDecode()") insert (q"performanceCounters
     .buildDecode()") in CSRContext register
}
```

Figure 1: A feature unit for implementing the base of the counter system.

4.3 Dependency Management

Faust automatically determines and applies the parent features that any child feature depends upon. A JSON file contains a graph of all the features and their dependency relations. In a separate JSON file, the end user lists out the features they would like Faust to apply to the code base.

For each feature in the requested features file, Faust determines what other features need to be implemented doing a depth first traversal of the feature graph. Features are added until either the root node has been reached or a feature where all the parents have already been explored. Thus, we also avoid having to re-traverse portions of the graph already included by other features.

4.4 Current Limitations

Due to the prototype nature of Faust, it currently has a few limitations. First, all of the join points, implementation code, and contexts must be captured as quasiquotes. These are strings that represent ASTs, making implementation of Faust easier. Second, new features must be manually added to Faust's management system. In the future, we would like to have this be an automatic process. Finally, Faust can only recognize pure dependencies. More advanced types of dependencies, discussed in Section 5, must be manually handled by the user.

4.5 Modifications to Rocket Chip

Faust can hook onto individual statements, thus could apply features inside of unmodified Rocket Chip. However, this approach is very fragile as any change to a statement would break the feature application. In order to facilitate robust feature application, we have made several changes to Rocket Chip that make the generator designs more modular.

```
trait CSRHardware {
  def buildDecode(): Unit
  def buildMappings(): Unit
}

class CSRFile(perfEventSets: EventSets = new EventSets(),
  customCSRs: Seq[CustomCSR] = Nil) with CSRHardware {

  buildMappings()

  buildDecode()

  def buildMappings() = {
    //mapping code
  }

  def buildDecode() = {
    //decode code
  }
}

abstract class PerformanceCounters(perfEventSets:
     EventSets = new EventSets(),
  csrFile: CSRFile, nPerfCounters: Int) extends
     CSRHardware {

  def buildMappings() = {
    //mapping code
  }

  def buildDecode() = {
    //decode code
  }
}
```

Figure 2: A more modular CSR structure.

The current design of Rocket Chip treats `Module` classes as direct analogs for Verilog modules. Instead, we propose thinking of classes as *hardware types* that perform various *generation tasks*. A class that implements a hardware type should only extend the `Module` class when appropriate. Listing 2 shows this technique in the context of the CSR structure and how we have utilized it in the implementation of our Performance Counter system[3].

As of writing, the `CSRFile` class in Rocket Chip contains 907 lines of code. By introducing this structure, generator users do not need to sift through all the code to find exactly where new features can be inserted giving us a robust set of join points in which to easily integrate features into the system.

5 FEATURE-ORIENTED DESIGN

Using Faust, we have completely refactored the Rocket Chip performance counter system to be feature-oriented. In addition, we have also enhanced the functionality of the current system and provided features that extend or modify the system that chip designers might use.

Our *Base System* is the modified version of Rocket Chip discussed in Section 4. The base system does not contain any generation infrastructure for the performance counters.

5.1 Feature Decomposition

We are careful to manage the dependencies between the features in our system. Other than the *Base System* itself, none of the other features can produce a functional system on their own. Figure 3 maps

[3]We have also completed similar refactoring on several other classes, but we will not cover them here.

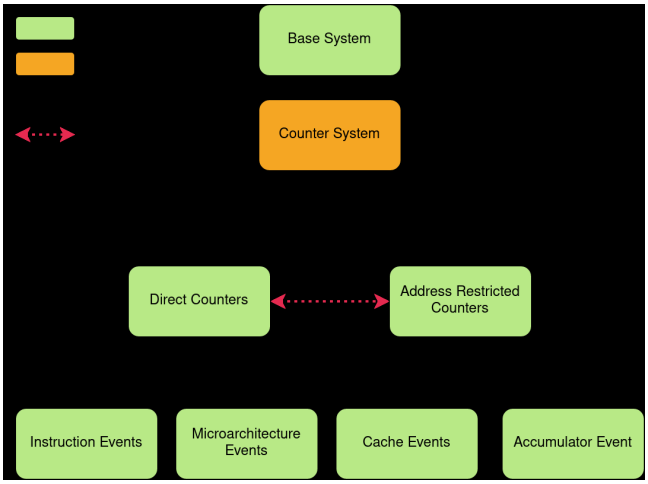

Figure 3: The dependency relationships between the features.

out all the current features in our system and their dependencies. The majority of the current dependencies are pure dependencies. However, two of our features are mutually exclusive and cannot exist in the system at the same time. Right now, mutually exclusive features must be handled manually, but we plan to make this automatic in the future.

We borrow our feature schema from prior work on feature-oriented design by Hunleth and Cytron [7]. Following their schema, we have two types of features in our system. **Abstract Features** are features that provide infrastructure, but do not complete primary functionality on their own. Abstract Features must always be augmented by a Concrete Feature. A **Concrete Feature** provides primary functionality given that its dependencies are satisfied.

Counter System. The *Counter System* feature is the only abstract feature that exists in the system. This provides the necessary infrastructure into Rocket Chip to realize the performance counter system, but leaves out creation of the performance counter registers.

Direct Counters. The *Direct Counters* build out the standard performance counters that are found in Rocket Chip. End users can still choose how many performance counters they wish to include in the system.

Address Restricted Counters. Instead of collecting event information over the whole address range, the *Address Restricted Counters* produce counters that are inhibited unless the program counter is within a certain address range. This feature allows users to customize the address range. Furthermore, this is managed completely outside of the architecture. The architected interface does not change.

Instruction, Microarchitectural, & Cache Events. These three features reimplement the standard event sets from Rocket Chip. However, unlike the original events and event sets, the order in which they appear in the code does not change the bit values to configure the events. Now, each event and event must be configured with a bit value to distinguish itself. The advantage here is that the events and event sets can be applied to the codebase *in any order* and events can be added or subtracted *without affecting any other events.* In fact, the way in which we have split the events into these sets is completely arbitrary and could be atomized even further.

In order to ensure that correct interfaces have been created, we provide correctness checking at generation time. If one or more events share the same bit mask, then the generator will throw an exception telling the end user they have created an invalid configuration.

Accumulator Event. We recognize that this feature is very limited, but we include it to demonstrate extra functionality that can quickly be included via our feature-oriented approach. The *Accumulator Event* feature routes a signal out of an RoCC [13] accumulator accelerator (included in Rocket Chip) and into the core for performance counting. This feature provides a template for others to include performance counter information from their accelerators as well.

6 RESULTS

Here we characterize the behavior of design endpoints in terms of area utilization. Our simulation and testing was completed using Chipyard [1] and our system is realized within a design based on the `TinyRocketChip` provided by Chipyard. We targeted the XC7A35T-1CPG236C FPGA which is in the Atrix-7 family of FPGAs from Xilinx. All implementation was done using Vivado 2019.2.

6.1 Rocket Chip vs. Our System

	Rocket Chip	Our System
Zero Counters	24056 (1.00)	**24025 (1.00)**
All Counters	26542 (1.10)	26467 (1.10)

Figure 4: A comparison of Rocket Chip and our system in area. The parenthesis show the normalized results in relation to the bold result.

In Figure 4 we compare the zero counter configuration of Rocket Chip to our system (in bold) as well as the recreation of the full counter system in Rocket Chip to our system. We also show all results normalized to our base system in parenthesis. The resulting zero counters systems are within 0.1% of each other in LUT usage. Furthermore, when recreating the full performance counter system from Rocket Chip we find similar results. These two systems are within 0.3% of the same area usage. Thus, we find that feature-orienting the performance counter system does not add any space penalty. While the feature set we are examining here is small, we are still effecting a large portion of the design with our features. This result demonstrates efficacy for further applications of feature-oriented design for hardware.

6.2 Design Endpoint Variations

Although our system can produce numerous design endpoints, in Figure 5 we show the interaction of features through four different design endpoints. In all of these endpoints, we have the system create 29 performance counters, limiting the variation to just the features listed. All numbers in parenthesis are the results normalized to our base system which is bold in Figure 4.

	Direct	Address Restricted
Instruction Events	26253 (1.09)	26378 (1.10)
All Events	26553 (1.11)	26872 (1.12)

Figure 5: The area in LUTs of four design variations. This table shows direct versus address restricted counters and just instruction events versus including all the possible events. The results normalised to our base system are shown in parenthesis.

Instruction Events vs. All Events. Regardless of the type of counters being used, there is about a 2% increase in area between just the *Instruction Events* and *All Events* endpoints. This shows a clear advantage for letting end users choose what features should be included in the performance counter system. Easy savings in area are not realized when all features are included by default. By taking this additive approach, Rocket Chip could provide not only more flexibility in the design, but automatically save designers space that would take manual effort to do currently.

Direct vs. Address Restricted Counters. Regardless of the type of events in the system, the *Address Restricted Counters* feature only uses 1% more space than the *Direct Counters*. A feature like this is extremely useful for programmers as many times they only wish to performance monitor sections of their code. By completing this action exclusively in hardware, programmers do not have to perturb the system, leading to more accurate results. Our feature-oriented approach allows Faust to insert this using a single line of code which creates a new class that is only 7 lines. By having an agreed upon way to add features to the system, we help guide the creation of new features to affect the rest of the system as little as possible.

Comparing Features. A prime advantage of this approach is shown by this analysis. Since *elided features do not exist in the system*, when we add a feature we can be more certain about what how each feature effects the system. Furthermore, paring this approach with Faust means that we can quickly compare different design endpoints with accuracy.

7 CONCLUSION AND FUTURE WORK

Overall, our feature-oriented performance counters provide a composable, extendable, and easy to understand system all while incurring no significant area penalty in the design. It is our hope that others will contribute their own features to this system and continue to build up the catalogue of features that can be integrated into the system.

One such feature of interest may be deployment of performance measurements *outside of* the ISA, allowing cycle-accurate measurement of events that are perturbed by the execution of instructions that establish and sample the counters [4].

In this work, we demonstrated the power of feature-oriented programming to simplify the customization and understanding of hardware through the isolation of hardware generator features. Currently, we consider this to be a proof-of-concept for feature-oriented hardware design. In our continuing work we wish to bring this sort of design to more portions of Rocket Chip and would encourage others to consider designing their hardware generators this way as well.

In order for RISC-V to become the universal ISA that it set out to be, we must provide hardware designs that are not only tailored to specific use cases, but are also easy to understand and extend. Feature-oriented design presents a viable path forward to do just that.

REFERENCES

[1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[2] 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

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[4] Richard Hough, Phillip Jones, Scott Friedman, Roger Chamberlain, Jason Fritts, John Lockwood, and Ron Cytron. 2006. Cycle-Accurate Microarchitecture Performance Evaluation. In *Proceedings of Workshop on Introspective Architecture.*

[5] Frank Hunleth. 2002. *Building Customizable Middleware using Aspect-Oriented Programming.* Master's thesis. Washington University in Saint Louis.

[6] Frank Hunleth, Ron Cytron, and Christopher Gill. 2001. Building Customizable Middleware using Aspect Oriented Programming. In *The OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems.* ACM, Tampa Bay, FL. www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html.

[7] 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

[8] Adam Izraelevitz. 2019. *Unlocking Design Reuse with Hardware Compiler Frameworks.* Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-168.html

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[10] Ravi Pratap M., Ron K. Cytron, David Sharp, and Edward Pla. 2003. Transport Layer Abstractions in Event Channels for Embedded Systems. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems* (San Diego, California). 144–152.

[11] Object Management Group 2001. *Event Service Specification Version 1.1* (OMG Document formal/01-03-01 ed.). Object Management Group.

[12] Martin Odersky and Tiark Rompf. 2014. Unifying Functional and Object-Oriented Programming with Scala. *Commun. ACM* 57, 4 (April 2014), 76–86. https://doi.org/10.1145/2591013

[13] Berkeley Architecture Research. [n.d.]. 6.3. RoCC vs MMIO — Chipyard documentation. https://chipyard.readthedocs.io/en/latest/Customization/RoCC-or-MMIO.html

[14] Scalameta. [n.d.]. Scalameta · Library to read, analyze, transform and generate Scala programs. https://scalameta.org/index.html

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

[16] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html