Adding Support for Staged Functions in Spatial

David Koeplinger
dkoeplin@stanford.edu

Tushar Swamy
tswamy@stanford.edu

1 SUMMARY

In this project, we add a limited subset of staged functions to the Spatial hardware accelerator design language. This support includes implementation of language support for staged functions, analyses for optimizing performance and area, and generating synthesizable implementations of these staged functions. We evaluate these additions by comparing the runtime and area of a set of benchmarks compiled with the current Spatial compiler and with the modified compiler with staged functions. We show that the addition of staged functions allows us to reduce resource utilization by 1× to 2×, depending on reuse opportunities, with no performance overhead.

2 BACKGROUND

2.1 Spatial

Spatial (previously called "DHDL" [6]) is a language and corresponding compiler being developed in our research group at Stanford [2]. Spatial is designed to serve as a higher level language for the development of application accelerators on reconfigurable architectures, including FPGAs and supported coarse-grain reconfigurable architectures (CGRAs).

The frontend of Spatial is implemented as a domain-specific language (DSL) embedded in Scala and built on top of a modified version of the Lightweight Modular Staging (LMS) project [7]. Like many other embedded DSLs, operations in Spatial are defined such that they "staged". In this scheme, running an operation does not immediately execute it, but instead adds nodes to a graph on the fly. The final graph represents the the entire program being executed, which can then be optimized by the Spatial compiler.

Spatial provides a mix of control structures and scheduling directives which help users to more succinctly express their programs while also allowing the compiler to identify parallelization opportunities. Control structures can be arbitrarily nested without restriction. Table 1a provides a list of the relevant control structures in the language for this project.

Spatial programs are internally represented in the compiler as a hierarchical dataflow graph (DFG) intermediate representation (IR). Nodes in this graph represent control structures, data operations, and memory allocations, while edges represent data and effect dependencies. Nesting of controllers directly translates to the hierarchy in the intermediate representation graph. Scheduling of control is kept as metadata in the compiler. Scheduling of each node is derived from compiler analyses, but can be overriden by the optional user directives shown in Table 1b.

CS242, Fall 2017, Stanford, CA

(a) Control Structures

`min until max by stride* par factor*`
A counter over the range [`min`,`max`).
`stride`: optional counter stride, default is 1
`factor`: optional counter parallelization, default is 1

`Foreach(counter+){body}`
A parallelizable *for* loop.
`counter`: counter(s) defining the loop's iteration domain
`body`: arbitrary expression, executed each loop iteration

`Reduce(accum)(counter+){func}{reduce}`
A scalar reduction loop, parallelized as a tree.
`accum`: the reduction's accumulator register
`counter`: counter(s) defining the loop's iteration domain
`func`: arbitrary expression which produces a scalar value
`reduce`: associative reduction between two scalar values

`Parallel{body}`
Overrides normal compiler scheduling. All statements in the body are instead scheduled in a *fork-join* fashion.
`body`: arbitrary sequence of controllers

`Pipe{body}`
A "loop" with exactly one iteration.
Inserted by the compiler, generally not written explicitly.
`body`: arbitrary expression

(b) Optional Scheduling Directives

`Sequential.(Foreach|Reduce)`
Sets loop to run sequentially.

`Pipe(ii*).(Foreach|Reduce)`
Sets loop to be pipelined.
`ii`: optional overriding initiation interval

`Parallel.(Foreach|Reduce)`
Informs the compiler that the loop is parallelizable.

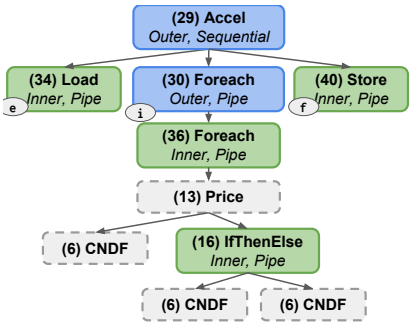
Table 1: A subset of Spatial's control syntax. Parameters followed by a '+' denote arguments which can be given one or more times, while a '*' denotes that an argument is optional.

When discussing DFG transformations and optimizations, it is often useful to abstract the program as a tree of control nodes. This tree abstracts away primitive operations, leaving only relevant controller hierarchy. In this representation, a node which contains another node is termed its "parent." Within this tree, the least common ancestor (LCA) between any two control nodes is defined as the last node that is a common parent to both nodes prior to an uncommon parent.

Figure 1 shows an example application written in Spatial and its corresponding control tree. The section of the code which will run on the FPGA is in the `Accel` scope. This program loads chunks of an array of records onto the FPGA from main memory, performs a long series of computations on each element in the `Price` and `CNDF` function calls, and stores the results back to another on-chip scratchpad. We

```
@struct class Elem(
  sptprice: Float, strike: Float, rate: Float,
  volatility: Float, time: Float, otype: Int
)

def CNDF(x: Float): Float = {
  val ax = abs(x)
  val x2 = exp((ax**2) * -0.05f) * 1/sqrt(2*Pi)
  ... // Long taylor series approximation
  mux(x < 0, xLocal, xLocalInv)
}

def Price(elem: Elem): Float = {
   val a = CNDF(...) // Supporting logic
   val b = ... // Supporting logic
   val x = if (...) CNDF(a) else CNDF(b)
   ... // Supporting logic
}

def ExampleApp(args: Array[String]) {
  val B = 1024 // Block size
  val data = loadData[Elem]("data.csv")
  val dram = DRAM[Elem](data.length)
  val result = DRAM[Float](data.length)

  // Transfer data from host to accelerator
  sendArray(data, dram)

  Accel {
    Foreach(0 until dram.length by B){i =>
      val block = SRAM[Elem](B)
      val output = SRAM[Elem](B)
      // Load data DRAM into on-chip memory
      block load dram(0::width)

      Foreach(0 until B){ii =>
        output(ii) = Price(block(ii))
      }

      // Store result back to DRAM
      block store dram(0::width)
    }
  }
}
```

(a) Spatial syntax.



(b) Controller tree.

Figure 1: An example program implemented in Spatial with corresponding controller tree. The program is inspired by BlackScholes option pricing, but has additional control flow at line 16.

elide some of the details of this computation for the sake of brevity.

Note that the control tree also includes schedule information. Both `Foreach` loops, for example, are set to run in a pipelined fashion. Operations in one branch of the `IfThenElse` are statically known not to run at the same time as those in the other branch.

2.2 Hardware Modules

Applications are laid out in space on reconfigurable architectures. This means that an application's achievable performance is limited by the number of resources it takes up on the target architecture. A typical FPGA, for example has two main compute resources: lookup tables, grouped as "slices" in Xilinx devices and dedicated multipliers ("DSPs"). This limitation creates a fundamental tradeoff space between an application's utilized target resources and its achievable performance.

In addition to their traditional role in reducing code duplication, functions ("modules" in hardware) are a key concept for analyzing the tradeoff between resource utilization and application performance for hardware accelerators. Since module instantiations represent sections of hardware with identical functionality, they give designers coarse-grained blocks which can either be duplicated to achieve a higher throughput or time-multiplexed to achiever lower resource utilization.

In hardware description languages (HDLs), each function "call", or module instantiation, is equivalent to duplicating the module ("inlining the function") at the call site, albeit in two-dimensional space rather than the instruction stream. Reuse of a single instantiation is always explicitly managed by the hardware designer. High level synthesis (HLS) tools like Vivado HLS [1] allow a function's resources to be time-multiplexed rather than inlined, but this choice is left entirely up to the user through pragmas. This pragma can only be specified per function, meaning more complicated usage patterns with a mix of duplication and time multiplexing cannot be captured by this annotation. Neither HDLs nor HLS tools support recursive modules/functions.

Spatial supports non-recursive functions, but like HDLs, currently inlines all function calls. This is the result of Spatial being an embedded language in Scala. Each function call occurs during Scala program execution (staging) and does not register anything within the Spatial dataflow graph. This limitation is in part due to a lack of support for first-class syntax for function declarations in LMS.

In Spatial, inlining functions corresponds to duplicating the hardware resources for each function call. Duplication may take up area resources unnecessarily, especially when two calls are guaranteed not to occur concurrently. This also has the side effect of exploding the size of the program's dataflow graph, thus slowing down the Spatial compiler.

The program in Figure 1 is an example of a design where inlining every function leads to suboptimal resource utilization. The calls to the `CNDF` function on line 16 occur in two branches of an `IfThenElse`, meaning they are statically known to never occur simultaneously for a single input. These two calls are most efficiently implemented as a multiplexer on inputs `a` and `b` to a single function module. However, this transformation is impossible without a representation of the function within the IR.

Additionally, a solution which always time multiplexes calls to a single `CNDF` instance, like the HLS function pragma

approach, is also not the right solution, as we would ideally like to pipeline the calls on line 16 with the call on line 14. Instead, the ideal design is one with two function instances created for three function calls, one for the call on line 14 and one for the two calls on line 16.

In this report, we outline the steps we took to add staged functions/modules and function calls/instantiations to Spatial. To avoid placing time multiplexing decisions on the user and the inefficiencies of pragma-based approaches, we place the burden of function duplication and inlining on analysis passes in the Spatial compiler. This required the following additions:

- We add staging support for function declarations and calls in Spatial's Scala frontend (Section 3.1).
- We add corresponding function nodes in the Spatial compiler.
- We add a compiler pass in Spatial which optimizes for module reuse opportunities (Section 3.2).
- We add code generation rules for multiplexed (reused) modules in Spatial (Section 3.3).

Our primary goal in adding this support was to minimize the area required for a given application without affecting application performance. As we show in Section 4, our additions allow us to maximally exploit module reuse opportunities with no performance degradation. Resource utilization improvements on our benchmarks was between 1× to 2×, but in practice achievable improvements are highly application-dependent.

```
1  @module
2  def functionName(arg1: T1, ... argN: TN): R = {
3    stm1; ...; stmN
4  }
```

(a) Syntax for a staged function in Spatial.

```
1  object functionName {
2    // This creates and saves the staged function.
3    private val __function = {
4      // Bound edges representing inputs to the function
           .
5      val arg1 = boundVar[T1]
6      ...
7      val argN = boundVar[TN]
8
9      // Create a scope and run all body statements.
10     val body = stageBlock{ stm1; ...; stmN }
11
12     // Stage the function declaration.
13     // This returns an edge representing this function
14     // which can be used in function calls.
15     Func.declare(List(arg1,...,argN), body)
16   }
17
18   // Enables staging of calls to the staged function.
19   // In Scala, the "apply" method allows syntax sugar
20   // for, e.g.  functionName(a, b)
21   def apply(arg1: T1, ..., argN: TN): R = {
22     // Stage the function call with given arguments.
23     // Returns an edge representing this staged call.
24     Func.call(__function, List(arg1,...,argN))
25   }
26 }
```

(b) The AST representing this syntax after the `@module` macro has been expanded.

Figure 2: Pseudocode for the `@module` macro AST expansion.

3 APPROACH

3.1 Function Staging

To support simple syntax for staged functions in Spatial, we first need to capture function declarations and function calls made in Scala. We found that the best way to do this was through Scala blackbox macros [4]. Blackbox macros are an experimental feature in the Scala language which allows the Scala AST to be transformed, with very few restrictions, after language parsing and prior to AST typing. We added function staging to the Spatial language via an optional, user-facing macro annotation [5] called `@module`.

Figure 2 shows pseudocode for the input and output of our `@module` macro annotation. The macro converts its corresponding function declaration to a Scala object. This object has one private field, a symbol representing the staged function, and one method, the method to call the function. Note that the "apply" method in Scala creates syntax sugar which allows an instance to be called as if it was a method. This is, for example, how `Function` objects in Scala work [3].

During application staging (Scala execution), the staged function is created immediately and added to the graph like any other node. Function calls are created on demand and added to the DFG at the call site through the apply method.

3.2 Compiler Analysis

After controller scheduling during Spatial compilation, we analyze each staged function and its corresponding calls in order to determine the number of physical function instances. In this analysis, we aim to minimize the amount of device resources required for all instances of the function while maintaining the current control schedule defined by the DFG and its scheduling metadata.

Given the program DFG, we first collect all calls to each function and record the full control trace, including function calls, that contain that call. This trace uniquely distinguishes each function call. For each function f, we then group calls based on whether they are statically known to occur at the same time. Calls which are known to never at the same time can be multiplexed to a single function instance, and are therefore grouped together. The pseudocode for performing this grouping is shown in Figure 3.

Following function call groupings, we then transform the DFG. Each function declaration is duplicated by the total number of groups found in the grouping analysis. Each function call is then updated to call its corresponding duplicate. Function instances with only one function call are inlined at their call site.

```
1  function Compatible(c, I):
2     for all calls c' in I:
3        lca = LCA(c', c)
4        if lca ∈ Parallel or lca ∈ Pipeline:
5           return false
6      end for
7  end for
8
9  C → set of sets of multiplexable calls
10 C = ∅
11 for all calls c to function f:
12    for all set of calls I in C:
13       if Compatible(I, c) then
14          add c to I
15          break
16       else add {c} to I
17    end for
18 end for
19 return C
```

Figure 3: Pseudocode for the compiler pass determining function duplicates for a given function f. *Parallel* is the set of `Parallel` control nodes. *Pipeline* is the set of control nodes which have a pipelined execution schedule.

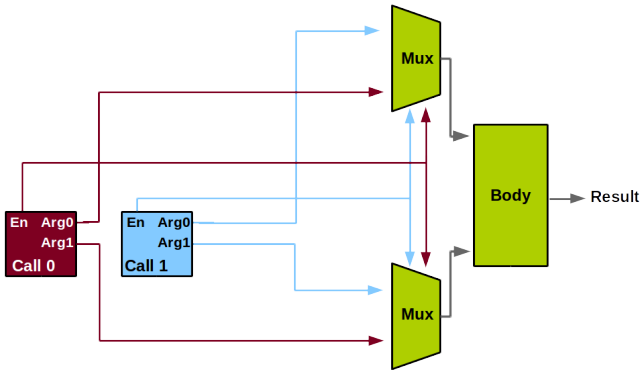


Figure 4: Generated Function Hardware for a reused function with two inputs

3.3 Code Generation

Once the compiler has duplicated and updated the IR nodes for function declaration instances and their corresponding calls, it can begin emitting code in Chisel. Chisel is a high level hardware construction language that is embedded in Scala. The Spatial compiler uses Chisel as a backend language which is then compiled down to Verilog and then finally synthesized for a target FPGA board using Xilinx tools such as the Vivado toolchain.

The hardware for the function declaration is first emitted by creating a one hot multiplexer for each input argument as well as each standard control signal used by Spatial controllers. Finally, the body of the function is emitted as a standard Spatial block. If the output of the function body is a value that can be represented in bits, it is tied as the value of all calling nodes in the dataflow graph.

Next, the function call hardware is created by tying the enable signal from the call's parent node directly to a bit in the selector of each multiplexer created by the associated function declaration. The arguments and control signals passed from the function call's controller parent are attached to the inputs on the appropriate multiplexers. While this configuration adds minor overhead in the form of the multiplexers that select arguments, the cost of this should be far less than duplication of the function body in most cases. A block diagram of the resulting hardware can be seen in Figure 5.

3.4 Limitations

We conclude our description of our language and compiler additions with a discussion of the limitations of our prototype implementation of staged functions.

Memory Aliasing. We currently assume that only types with a statically known number of bits can be passed as inputs or outputs of functions. This includes primitive values, as well as custom structs of primitives like the `Elem` struct shown in Figure 1.

It should be possible to extend our logic for function duplication to include passing local and off-chip memories as arguments and outputs of functions. This requires managing memory aliases within the compiler. Aliases should always be statically determinable, since aliases between on-chip memories ultimately correspond to multiplexers. In practice, multiplexers will be needed for the address, data, and enables for loads and stores.

Unfortunately, the Spatial compiler currently assumes that memory aliases do not occur within the program IR. While this assumption is not fundamental to the correct operation of the compiler, it is pervasive across most of the compiler's analysis passes. Updating these passes to allow memory aliasing in the context of function calls proved to be beyond the scope of this project, but should be possible in the future.

Recursive Functions. Our staged function implementation and analyses currently assume that there are no recursive function calls. The `@module` macro assumes that a function will not be called while staging that function's declaration, and our function call analysis assumes that there are no cycles in the call trace.

In order to support recursive functions, the staged function macro will have to be modified to create two separate IR nodes/edges: a placeholder for the function name declaration and a function body declaration. This will allow functions to be called prior to completing the staging of the function body. Additionally, the call analysis will have to include specific logic to account for cycles. We may be able to translate special cases such as tail recursive calls to loops, but other cases will require multiplexing of recursive calls to avoid exponential duplication of the function.

4 RESULTS

In this section, we examine the effects of our compiler changes on design runtime and area across four benchmarks. Each

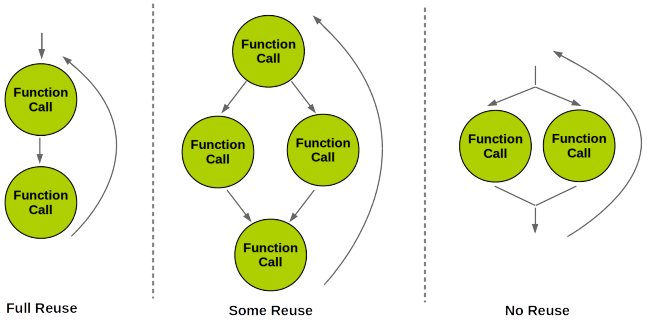


Figure 5: Reuse Variants for Benchmarks

benchmark has three variants exposing different levels of module reuse.

4.1 Methodology

Each of the four applications in our benchmark suite was designed to target a specific aspect of a typical Spatial program.

Nested-1. The first application in our suite was a simple primitive test where the program's only function performed a single multiplication operation. The simplicity of operations and lack of added control logic allowed us to test the basic functionality of our compiler modifications.

Nested-2. Our second benchmark demonstrates our improvements on doubly-nested function calls with a large amount of computation. The basis for the compute in this application is the calculation of the Black-Scholes financial model.

Nested-3. Our second benchmark adds another layer of function call nesting to the Nested-2 test to demonstrate our compiler's analysis on deeper function call traces like those found in real applications.

Looped. Our last benchmark demonstrates the ability of our approach to account for control logic within function bodies. In this test, each functional call performs a reduction that sums across values in a dataset

Figure 5 shows the three types of control flow in our benchmarks for calling a function multiple times. Each is inspired by a common control pattern we see in real applications, and each has different levels of function module reusability.

In the first variant, we test the ability of a single time multiplexed module to handle all function calls made throughout the program. Function calls are scheduled such that they are never called concurrently. If the function calls are the dominating logic in the program, we would expect to see a 2× improvement in area compared to the inlined baseline.

In the second variant, we test the ability of the compiler to handle programs where there is partial availability for module reuse. Function invocations made concurrently result in duplicated hardware, while sequential calls should reuse hardware. In the case where function calls dominate the design area, we again expect to see a maximum of 2× area improvement over the baseline.

Finally, the third variant tests the scenario where there is no reuse possible because all calls are made concurrently. All function calls will be inlined, so we expect to see areas similar to our baseline tests.

We synthesized each benchmark with and without our compiler modifications using the Xilinx Vivado toolchain to target the Zynq ZC706 SoC board. We then ran the benchmarks on the board in order to verify correctness and measure the effect our modifications had on the runtime of the application. The inner body of our benchmarks are wrapped in a loop that runs one million times in order to simulate runtimes typically seen on real datasets, and to produce a measurable runtime that is not dominated by data transfer to the FPGA.

4.2 Discussion

The results of these comparisons are summarized in Table 2. We see that in the best case, the Nested-3 benchmark, we achieve a 1.95× improvement in slice utilization and a 2× improvement on DSP utilization when all function calls in the program can be reused. This is because Nested-3 has extremely compute-heavy function calls which use between 25 - 95% of the FPGA's DSPs. In the case of the Nested-3-*Some* benchmark, the inlined version of the program requires more DSPs than the FPGA has available, and therefore could not be synthesized or run. However, after applying our staging modifications, the program only required about half of the available DSPs and easily fit on the board. In general, the more complex the compute logic in our application, the better results we see in terms of slice and DSP utilization. The overhead of adding multiplexers at function inputs is negligible compared to the total size of the function.

In the *None* reuse cases, where no function reuse was possible, all function calls were inlined after staging, effectively making the baseline and staged versions of the program the same. The variation in slice utilization in these cases is negligible and occurs as a result of minor variations in the synthesis tool.

In the Looped benchmarks, benefits in DSP utilization are due to reuse of control logic. Accordingly, we see the largest reduction in DSP in this application due to additional optimizations done by the synthesis tool.

Finally, examining the runtime of each benchmark, we see that, in all cases, we have the same runtime of the staged version of the application as the inlined version. In fact, because our benchmarks have statically determinable runtimes, we see that the runtimes are consistently identical. This happens because, although we have duplicated compute logic, we have not changed the behavior of control logic in any way. The compiler modifications that we made therefore have no detrimental performance effects.

			Slices	DSPs	Runtime (ms)
Benchmark	**Reuse**	***Function***	54650	900	—
		Inlined	3881	6	269
Nested-1	Full	Staged	3877	3	269
		Change	**1.001**×	**2**×	**1**×
		Inlined	3965	12	409
	Some	Staged	3932	6	409
		Change	**1.008**×	**2**×	**1**×
		Inlined	3881	6	149
	None	Staged	3814	6	149
		Change	**1.018**×	**1**×	**1**×
		Inlined	16453	330	3389
Nested-2	Full	Staged	9069	165	3389
		Change	**1.814**×	**2**×	**1**×
		Inlined	28598	660	5089
	Some	Staged	15606	330	5089
		Change	**1.833**×	**2**×	**1**×
		Inlined	16043	330	1709
	None	Staged	16211	330	1709
		Change	**0.9896**×	**1**×	**1**×
		Inlined	25548	456	5009
Nested-3	Full	Staged	13103	228	5009
		Change	**1.950**×	**2**×	**1**×
		Inlined	*DNF*	*DNF*	*DNF*
	Some	Staged	24036	456	7519
		Change	**NA**	**2**×	**NA**
		Inlined	25492	456	2519
	None	Staged	25346	456	2519
		Change	**1.005**×	**1**×	**1**×
		Inlined	4007	6	2419
Looped	Full	Staged	3956	3	2419
		Change	**1.013**×	**2**×	**1**×
		Inlined	4397	12	3629
	Some	Staged	4180	3	3629
		Change	**1.052**×	**4**×	**1**×
		Inlined	3980	6	1219
	None	Staged	4014	6	1219
		Change	**0.9915**×	**1**×	**1**×

Table 2: Area utilization and runtime comparisons between inlined and staged functions in Spatial. The first row lists the total number of FPGA resources available. DNF: Design did not fit on the FPGA.

5 CONCLUSION

In this report, we present modifications to the Spatial compiler that allow for the staging of a function module that can be time-multiplexed across calls. We first enable the declaration of staged functions in the Spatial syntax by making use of Scala blackbox macros. We next add compiler analyses and transformation passes that are able to determine which modules can be reused and perform function duplications which minimize area given these reuse opportunities. Finally, we add code generation rules to the Spatial compiler that allow it to output Chisel code for time-multiplexed function calls. We show the effectiveness of our modifications by synthesizing and running 12 benchmarks that cover a number of common control scenarios with varying levels of function reuse. This demonstrated that resource utilization was improved by 1× to 2× while having no impact on the application's runtime.

6 WORK DISTRIBUTION

Equal work was performed by both project members.

REFERENCES

[1] 2016. Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. (2016).

[2] 2017. Spatial-Lang Repository. https://github.com/stanford-ppl/spatial-lang. (Dec 2017).

[3] 2017. Twitter Scala School: Functions are Objects. https://twitter.github.io/scala_school/basics2.html. (Dec 2017).

[4] Eugene Burmako. 2017. Blackbox vs. Whitebox. https://docs.scala-lang.org/overviews/macros/blackbox-whitebox.html. (Dec 2017).

[5] Eugene Burmako. 2017. Macro Annotations. https://docs.scala-lang.org/overviews/macros/annotations.html. (Dec 2017).

[6] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *International Symposium in Computer Architecture (ISCA)*.

[7] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314