

# Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores

Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini

**Abstract**—Single-issue processor cores are very energy efficient but suffer from the von Neumann bottleneck, in that they must explicitly fetch and issue the loads/stores necessary to feed their ALU/FPU. Each instruction spent on moving data is a cycle not spent on computation, limiting ALU/FPU utilization to 33% on reductions. We propose "Stream Semantic Registers" to boost utilization and increase energy efficiency. SSR is a lightweight, non-invasive RISC-V ISA extension which implicitly encodes memory accesses as register reads/writes, eliminating a large number of loads/stores. We implement the proposed extension in the RTL of an existing multi-core cluster and synthesize the design for a modern 22nm technology. Our extension provides a significant, 2x to 5x, architectural speedup across different kernels at a small 11% increase in core area. Sequential code runs 3x faster on a single core, and 3x fewer cores are needed in a cluster to achieve the same performance. The utilization increase to almost 100% in leads to a 2x energy efficiency improvement in a multi-core cluster. The extension reduces instruction fetches by up to 3.5x and instruction cache power consumption by up to 5.6x. Compilers can automatically map loop nests to SSRs, making the changes transparent to the programmer.

**Index Terms**—Parallel architectures, micro-architecture implementation considerations, energy-aware systems



## 1 INTRODUCTION

THE BREAKDOWN of Dennard scaling in modern silicon manufacturing has prompted a paradigm shift in the way we approach computer architecture design. Cutting-edge processing systems such as today's CPUs and GPUs are hitting the utilization wall [1]. Research effort is moving away from manufacturing technologies towards technology-aware computer architectures with a focus on energy efficiency. Performance at low power is a key ingredient in achieving high utilization of available hardware in order to mitigate the effect of limited frequency and overcome dark silicon [2].

In-order processor cores built around the load/store paradigm face an efficiency challenge in keeping their functional units busy. As an illustrative example, assume that we would like to compute the dot product over a long vector of Floating Point (FP) values. Consider the following snippet of RISC-V [3] assembly executed for every pair of values:

```
flw ft0, 4(a0!)
flw ft1, 4(a1!)
fmadd.s ft2, ft0, ft1, ft2
```

The processor loads two values from memory (`flw`), then multiplies them and accumulates the result (`fmadd.s`). We assume that iteration and pointer adjustment are performed via hardware loops and post-load increment, respectively, which are quite affordable microarchitectural enhancements

[4]. In a single-issue core this takes at least three cycles to execute. Since only every third instruction performs actual computation, the Floating Point Unit (FPU) is utilized at most 33% of the time. The fundamental problem is that in a load-store architecture, data transfers from and to memory have to be encoded explicitly as instructions. If the machine is only able to issue a single instruction per cycle, every load and store introduces at least one idle cycle in the FPU. The lower a kernel's operational intensity, the more pronounced this problem becomes.

One would like to keep a processor's functional units as busy as possible for various reasons. Consider the following two scenarios:

- 1) In the near-threshold regime or at the operating temperatures found in High Performance Computing (HPC) or the data center, leakage currents are a significant contributor to power consumption. We observe for example a 6.5× leakage increase from 5% at 25 °C to 25% at 85 °C in the 22 nm technology used throughout this paper. An idle unit still dissipates leakage power, unless it is power-gated at fine granularity, which is hard to do. This leakage power adds to the energy consumed per computation, decreasing overall energy-efficiency.
- 2) In an area-constrained setting such as an embedded application, idle cycles can be costly performance-wise. A core that keeps its FPU busy 50% of the time achieves half of the performance within a given area budget, and conversely requires twice the area for a given performance target, than a core which keeps its FPU busy 100% of the time.

Achieving high FPU/ALU utilization is the ultimate goal of a micro-architecture design. Hence, solutions to this problem exist. For example a core that can issue multiple instruc-

- F. Schuiki, F. Zaruba, and L. Benini are with the Integrated Systems Laboratory (IIS), Swiss Federal Institute of Technology, Zurich, Switzerland. E-mail: {fschuiki,zarubaf,lbenini}@iis.ee.ethz.ch
- T. Hoefler is with the Scalable Parallel Computing Laboratory (SPCL), Swiss Federal Institute of Technology, Zurich, Switzerland. E-mail: htor@inf.ethz.ch
- L. Benini is also with the Department of Electrical, Electronic, and Information Engineering (DEI), University of Bologna, Bologna, Italy.

tions per cycle does not have this fundamental limitation, as it can keep the Load-Store Unit (LSU) and the FPU busy at the same time. In our previous example, such a core would need to be able to issue two loads and one FP operation, hence fetching and decoding three instructions, per cycle. Accommodating such an increase in fetch bandwidth does not come for free [5]. In fact, moving to superscalar out-of-order execution requires the instruction fetch interface to at least triple in width, followed by three replicated instruction decoders. In addition, the parallel execution of LSU and FPU in non-VLIW cores requires at least some form of dependency tracking and reordering, and is likely to entail register renaming as well.

As a result, achieving high execution unit utilization with out-of-order superscalar execution is not energy efficient [6]. This is true even for moderately complex multiple-issue cores. Complex Instruction Set Computer (CISC) machines can allow for memory accesses to be encoded in a compute instruction directly, thus potentially reducing the bandwidth required to issue three instructions in parallel. Decoding CISC instructions is highly non-trivial however and compiler and micro-architecture become more complex. As a result, even CISC Instruction Set Architectures (ISAs) such as x86 internally unpack these complex instructions into multiple RISC-like micro-operations. Other approaches such as vector processors and Very Long Instruction Word (VLIW) require moving to a much more complex data-parallel micro-architecture and/or incur additional instruction bandwidth, with a major impact on the ISA or the compiler.

In this paper we propose Stream Semantic Registers (SSRs), a simple and lightweight extension to single-issue load-store architectures to remove its utilization bound without major impacts on the complexity of the micro-architecture and its silicon implementation. The key idea is to allow loads and stores to be encoded in any instruction for instruction sequences with regular data accesses, instead of explicit load/store instructions. We do this by giving a few registers stream semantics: reading from or writing to the register issues a read or write into the memory system, respectively. An Address Generation Unit (AGU) placed outside the processor core allows these memory accesses to follow a programmable affine address pattern which is very common for many compute-intensive workloads such as Digital Signal Processing (DSP), stencils, and machine learning. A key advantage of following such a predictable address pattern is the fact that data can be loaded proactively, implementing a form of prefetching. This allows our introductory example to be rewritten as

```
fmadd.s ft2, ft0, ft1, ft2
```

where `ft0` and `ft1` now have stream semantics. Intuitively the expected speedup is  $3\times$ . We will show that in general the achieved speedup is related to the operational intensity of a kernel, the size of the register file, and the size of the first level memory, and may in practice vary between  $2\times$  and  $5\times$ , with speedups on realistic kernels as high as  $3.7\times$ . To sum up, the key contributions of this paper are:

- 1) A register file extension for Reduced Instruction Set Computer (RISC) architectures that allows for implicit

encoding of data transfers within any instruction. We present the necessary architectural changes in an existing multi-core platform (Section 2).

- 2) Competitive experimental results for the proposed architecture applied to a low power multi-core system [4]. We provide a performance, power, and area analysis for an implementation in a 22 nm technology and compare against other systems (Section 4).
- 3) A discussion and preliminary results on how to extend compilers to directly emit code that leverages the proposed architectural changes (Section 4).

The remainder of this paper is organized as follows: Section 2 describes the proposed SSR extension, Section 4 presents an evaluation, experimental results, and comparison to other systems. The remaining sections describe related and future work, and offer concluding remarks.

## 2 ARCHITECTURE

We start with a description of the SSR extension in detail and show the architectural changes necessary. As an implementation we extend a RI5CY (“*riscy*”) core [4] and PULP cluster [7] with two SSR data movers. The RI5CY core has a single-issue in-order pipeline, but it features a rich set of micro-architectural enhancements that make it a challenging baseline when assessing efficiency in executing DSP workloads. This includes the support for hardware loops and post-increment load/store operations, which removes the need for branches and many address calculations in the innermost hot loops of a computation: hence RI5CY actually achieves the single-issue performance bound for many kernels. We first provide an overview of the SSR architecture and then outline the necessary changes inside the core (Section 2.2), at the core interfaces (Section 2.3), and in the memory system (Section 2.5).

### 2.1 Overview

The key idea of SSR is to intercept accesses to certain registers at the register file and route those accesses out of the core and into the memory system. A separate address generator assigns an address to each such access. Since the address generator is configured up front, this can be seen as proactively prefetching the next accesses, rather than reactively hiding access latency. In our implementation we assign stream semantics to the `t0` and `t1` integer and the `ft0` and `ft1` floating point registers. Reads from and writes to these four registers will be diverted out of the processor. These are the first two caller-saved integer and floating point registers in RISC-V.

The architectural changes can be subdivided into two main parts: the mapping from register accesses to transactions on a stream interface, and the mapping from those transactions to memory accesses. In our case we perform the mapping to the stream interface inside the RI5CY core and the mapping to memory accesses using a data mover outside the core. This minimizes the changes to RI5CY and exposes a clean interface.

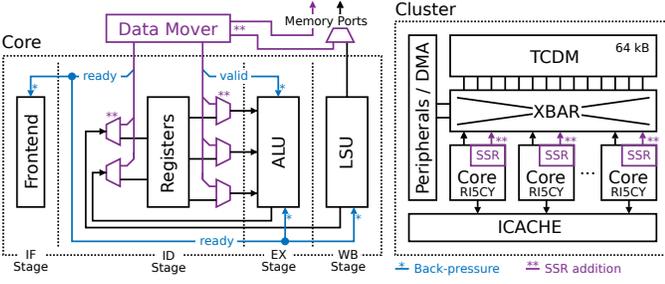


Figure 1. On the left: High-level data flow of the SSR extension in a single core. Read and write accesses to certain registers are filtered and diverted out of the processor core. The data mover then assigns memory addresses to the accesses and forwards them into the memory system. On the right: cluster of multiple cores, attached to TCDM and peripherals via logarithmic interconnect (XBAR). Pipeline additions to control back-pressure marked in blue (\*); SSR data path additions marked in purple (\*\*).

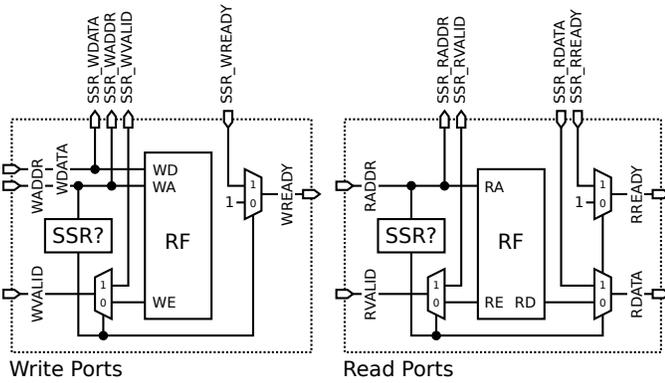


Figure 2. Additional circuitry required per register file write port (left) and read port (right) to accommodate stream semantics. The additions are implemented as a wrapper around the core’s existing register file (RF). The SSR access check “SSR?” is equivalent to evaluating  $A \in \{t0, t1, ft0, ft1\} \& E$ , where  $A$  is the register address and  $E$  the SSR enable CSR bit. See Section 2.2.1 for details.

## 2.2 Core

The following modifications to the processor core are necessary to map register accesses onto a stream interface:

- The register file must be extended to intercept and re-route accesses to a subset of registers.
- The stream interfaces, one per register file port, need to be exposed in the core’s port list.
- A Control and Status Register (CSR) is needed to enable or disable stream semantics.
- Additional stall conditions and back pressure paths introduced by the stream interface must be considered in the core controller.

In the following paragraphs we provide a detailed description of each of these modifications.

### 2.2.1 Register File

The fundamental architectural change of the SSR extension is in the processor’s register file. A generic vanilla register file consists of read ports which supply the operands for subsequent pipeline stages, and write ports which store back the result of an instruction. In our case the original register file has three read and two write ports. For SSR we would like to intercept accesses to a certain set of

registers and, instead of accessing the register file, perform a read or write transaction on the external stream interface. Figure 1 depicts the high-level data path implied by this architectural change. Instead of directly routing read and write accesses to the register file, we first determine if an accessed register has stream semantics enabled and if yes, re-route the access onto the corresponding stream interface. Each port into the register file has a corresponding stream interface. RISC-V with “IFD” extensions allocates 32 integer and 32 float registers, and uses 5bit to address them. The RISC-V core used in our implementation fuses these into a register file with 64 registers and 6 bit addresses, where the most significant address bit is set depending on whether the requesting instruction is an integer or floating point operation.

Figure 2 outlines the additional circuitry needed per write and read port. In both cases we determine if an access has stream semantics (“SSR?”) by checking the following conditions:

- 1) The register address WA or RA must be one of the registers with stream semantics ( $t0, t1, ft0, ft1$  in our implementation).
- 2) The stream semantics must be enabled in the core’s CSRs.

If both conditions hold the transaction is routed out of the core via a stream interface. These interfaces use a valid/ready handshake which allows for the data mover and memory system to assert back pressure into the core if a request cannot be serviced immediately. The additional hardware is implemented as a wrapper around the core’s existing register file.

### 2.2.2 Control and Status Registers

The SSR extension needs to be opt-in and disabled by default. This allows code that does not benefit from the use of streams to have the full set of registers available. Furthermore it maintains compatibility with existing code and code generated by a compiler that is not aware of the extension. To this end, we have added the `ssrcfg` CSR with address `0x7C0` to the core. It contains a single bit that enables or disables stream semantics in the core. The subset of registers with stream semantics is fixed in hardware and can only be enabled or disabled all at once. Sections of code using SSR are expected to set this bit at their beginning and clear it at their end, essentially defining an “SSR region” in the code. Special care must be taken to handle interrupts and exceptions; see Section 2.4.

### 2.2.3 Pipeline Considerations

Our proposed architectural extension has the following implications on the processor pipeline:

- 1) First and foremost, due to its newly-gained stream nature the register file loses its *idempotency*. A non-SSR processor may not contain any control signals to precisely enable or disable register accesses during stalls, since reading or writing the same register with identical data has no consequences in an idempotent register file. As an example, a multi-cycle instruction may apply bogus data to its destination register during all but the last cycle of execution, and only present valid data in its

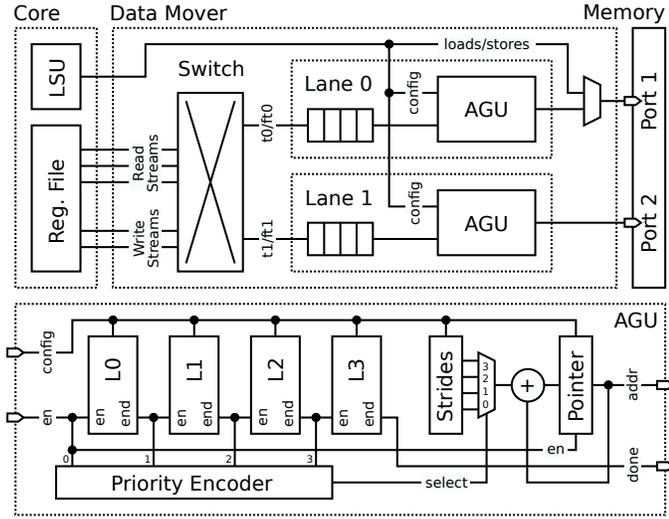


Figure 3. Above: Architecture of the data mover which translates from a stream of register accesses to the corresponding memory addresses and accesses. Below: Architecture of an individual AGU with four nested loops. “L0” to “L3” are loop counters. Patterns are configured via the “config” interface, which modifies the loop counters, strides, and pointer register.

last cycle. As an optimization the processor may choose to enable register writes during all cycles, knowing that the last cycle will eventually override bogus data with the valid result. Similarly, an instruction may keep reading registers even during a pipeline stall. With SSR such optimizations are not possible and additional signals must be added to ensure each instruction accesses its registers exactly once. Similarly, a compiler may not delete redundant writes to an SSR. All of the following considerations are a result of this loss of idempotency.

- 2) Instructions that enable or disable stream semantics, CSR writes in the RISC-V case, require pipeline bubbles to be inserted between them and subsequent instructions operating on stream registers. This is necessary since the register file access is likely to be located in a different pipeline stage than the CSR write. A non-idempotent access to a stream may therefore erroneously occur before a preceding SSR-disabling CSR write fully executes. Vice versa, a stream access may be skipped due to the preceding SSR-enabling write not having taken effect.
- 3) Instructions that operate on SSRs may not access the register file speculatively. In the case of RI5CY, this requires such instructions to be stalled in the decode stage when a branch is currently being resolved.
- 4) The register file must be able to exert back-pressure into the pipeline on reads and writes. A non-SSR processor may not provision for such a scenario. In the RI5CY case, additional stall control signals are needed from the register file to the instruction decode stage for reads, and to the write-back stage for writes.

### 2.3 Data Mover

The modifications outlined thus far transport SSR register accesses selectively out of the core. Figure 3 shows the data mover that is used to map these accesses into the memory

system. It consists of three general parts described in the following.

Each port into the register file is exposed as a separate stream at the boundary of the core, three read and two write streams in our case. A *switch* uses the register address to map each access on these streams to the targeted data mover lane. Our example contains two such lanes, one for the  $t_0/ft_0$  and one for the  $t_1/ft_1$  registers.

Each *lane* consists of a *First-In First-Out (FIFO)* queue to buffer read and write data. An *address generator* based on the one presented by Schuiki et al. [8] and Conti et al. [9] assigns memory addresses to the stream-based accesses performed by the core. The lane can be put into read mode, in which case the address generator is used to fetch data from memory and store it in the FIFO. If put into write mode, the address generator is used to tag each datum written into the FIFO with an address and send it off into the memory system. This requires a register to be exclusively used as source or destination operand until the complete address pattern has been exhausted; thus a stream cannot be used to interleave read and write operations.

A significant advantage of this architecture is the fact that the data mover can pro-actively perform memory reads. In this way, when the processor decides to read from an SSR, the datum is already present. This is in stark contrast to a regular load instruction which merely initiates a memory access. Subsequent instructions must wait for the accessed data to return, which amounts to at least one cycle of delay. SSR-based operations thus become significantly more memory latency tolerant than regular loads. This pro-active read introduces a source of incoherence in that a write to a memory location does not update the corresponding value in the read FIFO. Read streams started *after* a write will always see the written value. Thus to avoid data races write operations shall not be performed on a memory range that is currently used in a read stream. This limitation proves to be inconsequential in all kernels investigated in this paper.

### 2.4 Interrupts and Exceptions

The AGUs and enable-state of the SSRs add additional implicit state to the processor core that must be adequately saved/restored. Depending on the features of the processor in question, and the requirements of the target applications, this comes at varying complexity. In general we identify the following three options of handling exceptions:

- 1) **No exceptions:** The simplest approach is to not support exceptions at all. While not applicable to operating-system-capable cores, specialized number-crunching cores, e.g., the ones in a GPU, have very limited exception support [10]. Since control flow cannot be diverted in this case, no special treatment is required.
- 2) **Deferred exceptions:** A similar approach is to defer exception handling to SSR-disabled regions. While this may incur significant latency in the presence of long-running streams, it allows number-crunching cores to support a minimal amount of external interrupts. Applicability is limited to a similar scenario as the above approach, since the utility of deferred exceptions is very limited in practice [11, p. 261]. This is mainly due to difficult debugging and operating systems requiring

Table 1  
Levels of exception support required for different processor core features and applications. See Section 2.4 for details.

Exception Support	External Interrupts	Virtual Memory	Application
1) No exceptions	no	no	GPU
2) Deferred exceptions	yes	no	IoT
3) Precise exceptions	yes	yes	CPU

finer-grained exceptions. It is also necessary that the core cannot generate memory exceptions, since streaming over a sufficient number of pages could otherwise livelock the core.

- 3) **Precise exceptions:** It is also possible to fully support precise exceptions with SSRs. This is required for cores with virtual memory and/or memory protection, or cores that must give timing guarantees on exception handling. The SSRs expose their precise architectural state as registers, which become part of the saved/restored thread context. Streams can thus be interrupted and resumed at will. Such a scheme is easily implemented in an operating system, that already features a platform-specific exception handler. As an upper bound, precise exceptions double the area and power of the AGUs, since now in addition to the AGU running ahead as data is requested, a separate AGU is required which advances in synchrony with the instruction stream, and thus represents the “architectural state” of the stream. Upon an exception, this AGU reflects the state before the exception occurred, and is used to restart execution once the exception handler returns.

Consider Table 1 for an overview of processor features, applications, and required exception support. Our implementation is of the “deferred exceptions” kind: the RI5CY core supports external interrupts, but these remain disabled during SSR operation. Since we operate directly on local scratch pad memory without virtual addressing or memory protection, the only source of memory exceptions are accesses to unmapped addresses, after which the program is not expected to resume. Rather, the exception handler aborts the SSR streams and terminates the program. However, note that this does not preclude the use of virtual memory in the remainder of the system: data is moved explicitly via a DMA which can support efficient page fault handling [12].

## 2.5 Memory System

It is important to note that the SSR extension increases the peak memory bandwidth that a core can request from the memory system. In our implementation this translates into an increase in the number of ports into the local scratch pad memory, but may correspond to additional ports into the top-level cache in other systems. The following aspects are relevant for choosing a reasonable number of SSRs, streams, data movers, and ports into memory.

### 2.5.1 Impact of SSRs and Streams

On an architectural level, the memory traffic that a core may generate via SSRs is limited by two factors: (1) the number of ports into the register file, and (2) the number

of register operands in an instruction. The former is five in our implementation (three read and two write ports). The latter is four as defined by RISC-V [3] (three source and one destination registers). The smaller of the two is the upper bound on the memory traffic that can be generated, 4 word/cycle in our case.

### 2.5.2 Impact of Data Movers

The number of data movers determines the number of independent memory address patterns a core can keep track of. Since the data movers are tied to individual registers, there need to be at least the same number of registers with stream semantics as there are data movers. Multiple SSRs may address the same data mover, for example to use the data mover both in integer and FP instructions. This is the case in our implementation where the `t0` and `ft0` registers are bound to the same data mover lane. As such the number of data movers puts an additional upper bound on the memory traffic a core may generate, 2 word/cycle in our implementation.

In this paper we only consider a simple data mover that generates a single memory access per stream transaction. This is not a requirement. More elaborate data movers may issue more memory traffic. Consider indirect addressing for example: in such a setting a data mover would load an address from memory, then use that address to perform the load or store corresponding to a stream transaction, effectively accessing 2 word/cycle. Depending on the cost per memory port and the achieved speedup, such schemes may warrant the use of more than one port per data mover.

Not every data mover necessarily requires a separate port into memory, however. A system might provide, for example, eight data movers to independently track eight separate sequences of memory addresses, and instructions would pick a sequence via the stream registers they use. However such a system would still be limited to 4 word/cycle of traffic due to the limited number of operand registers (see Section 2.5.1). In this case it would be beneficial to multiplex the eight data movers onto four memory ports via an arbitration scheme.

### 2.5.3 Impact of Memory Ports

Due to the nature of single-issue load-store architectures, an instruction may either exercise the LSU or the data mover, but never both. (Technically an SSR could be used as the destination register for a load, but we do not consider `memcpy` a critical application.) Therefore the cumulative bandwidth generated by the LSU and one of the data movers never exceeds 1 word/cycle. Since memory ports are costly we thus suggest to always multiplex the core’s LSU and one of the data movers into a single port.

Reduced to their fundamental instruction, operations such as multiply-add have an intensity of 0.25 op/word, addition or multiplication one of 0.33 op/word, and multiply-accumulate of 0.5 op/word, meaning for every operation performed they consume and produce four, three, and two data words, respectively. In order to sustain one instruction per cycle, a core would require four, three, and two ports into the memory system, respectively. Our implementation allocates two memory system ports per core and uses two data movers, one of which is multiplexed with the core’s

	SSR Implementation:	Baseline Implementation:
Addr. Pattern Cfg	<code>la %0, ssr_registers</code> ①	<code>lp.setup L0, %N, +3</code>
	<code>addi %1, %N, -1</code>	<code>p.flw ft0, 4(%A!)</code> ③
	<code>sw %1, (DM0 BOUND_0)(%0)</code>	<code>fmadd.s %x, ft0, ft1, %x</code>
	<code>sw %1, (DM1 BOUND_0)(%0)</code>	<code>p.flw ft0, 4(%A!)</code>
Hot Loop Iterations	<code>li %2, sizeof(float)</code>	<code>p.flw ft1, 4(%B!)</code>
	<code>sw %2, (DM0 STRIDE_0)(%0)</code>	<code>fmadd.s %x, ft0, ft1, %x</code>
	<code>sw %2, (DM1 STRIDE_0)(%0)</code>	<code>[repeats 998x]</code>
	<code>sw %A, (DM0 READ_1D)(%0)</code>	<code>= 3001 instructions executed</code>
	<code>sw %B, (DM1 READ_1D)(%0)</code>	
	<code>csrwi ssrcfg, 1] Enable</code> ②	
	<code>lp.setup L0, %N, +1</code>	
	<code>fmadd.s %x, ft0, ft1, %x</code> ③	
	<code>fmadd.s %x, ft0, ft1, %x</code>	
	<code>[repeats 998x]</code>	
<code>csrwi ssrcfg, 0] Disable</code> ④		
<code>= 1012 instructions executed</code>		
	<b>Corresponding C Code:</b>	
	<code>for (i = 0; i &lt; N; i++) {</code>	
	<code>sum += A[i] * B[i];</code>	
	<code>}</code>	

Figure 4. Basic usage pattern of SSRs. Address patterns are configured by writing to the memory mapped address generator registers (1), enabling the stream semantics by writing to the `ssrcfg` CSR (2), performing the actual hot loop (3), and disabling the stream semantics again (4). Right-hand side shows baseline RISC-V implementation with RI5CY custom extensions. Note the two additional post-increment load operations per hot loop iteration absent from the SSR case. See Section 3.

LSU data port using a fixed priority arbitration scheme. This allows the core to sustain kernels with an operational intensity of 0.5op/word or higher, which covers the pervasive multiply-accumulate operations found in linear algebra and machine learning.

### 3 PROGRAMMING MODEL

The fundamental SSR usage follows the simple sequence outlined in Figure 4: address pattern configuration (1), enabling the stream semantics (2), computation (3), and disabling the stream semantics again (4). The configuration registers of the data mover are memory mapped and can be accessed by the processor via load and store instructions. An in-depth explanation of the available registers follows in Section 3.1. The region of code that makes use of the stream semantics (the “SSR region”) must be surrounded by writes to the `ssrcfg` CSR to enable SSRs upon entry and disable them again upon exit of the region. The SSR region itself can contain any sequence of assembly instructions.

#### 3.1 Pattern Configuration

Each address generator contains ten configuration registers and supports up to four nested loop dimensions. We have found four dimensions to cover all problems investigated later in Section 4.2. This number is a design parameter and can be changed. Looping over additional outer dimensions may be performed in software. The `status` register contains the address pointer, the number of enabled nested loop dimensions, stream direction (read or write), and a flag indicating whether the end of the pattern has been reached. A streaming operation is triggered by writing to this register. The `repeat` register allows each datum loaded from memory to be emitted into the core multiple times. This is useful if a value loaded from memory is used as an operand multiple times. Eight registers control the iteration behavior, two for each loop dimension. The `bound0–3` registers contain the number of iterations and the

`stride0–3` registers the address increment for each loop. Note that while the SSR extension allows for many address stepping and data transfer instructions to be removed from the instruction stream, the program must still issue the exact number of compute instructions (such as `fmadd`) to fully exhaust the pattern in the address generator. This means that the fundamental loop nest containing the compute instruction must still be present. As we will show this is most easily accomplished through the use of hardware loops.

#### 3.2 Automated Code Generation in LLVM

Mapping nested loops from an input language such as C to SSRs is straightforward if the loop bounds are constant for the duration of the loop and addresses are a linear function of the indices. This is the case for many data-oblivious kernels where the control flow does not depend on the data values [13]. We propose the following recipe for a pass to map loops to SSRs in the LLVM compiler framework [14]. The pass operates on the Machine IR (MIR) and is executed after instruction selection and before register allocation takes place. At this stage trivial loop induction variables have been identified and mapped to simple address increments as far as possible. Our pass is divided into the following phases:

- 1) Identify loops in the MIR data and control flow graph. This information is already provided by the LLVM infrastructure.
- 2) Visit all load and store instructions within the identified loops. Check if the address expression is a simple counter which increments by a constant amount. Since the MIR is in Single Static Assignment (SSA) form this can be done as a simple pattern match: We check if the address is determined by a `phi` and `add` node loop in the graph, and whether the input to the `add` is constant. This check is done recursively across multiple nested loop levels. Loads and stores which pass are marked as candidates for SSRs replacement.
- 3) Allocate the candidates to the available data movers (two in our case). We start with the deepest candidates first, in terms of nesting level, which is a simple heuristic for the number of loop iterations.
- 4) Emit instructions to configure the SSR before the loop header.
- 5) Remove the load/store instruction from the MIR and replace any uses with the corresponding stream register.
- 6) Block registers with stream semantics in the register allocation pass.

A limitation of following this recipe without further considerations is that all loops will be mapped to SSRs as far as possible. However, as we will show later, not every loop benefits from SSRs. Especially very short loops may take longer to execute with SSRs due to the increased setup overhead. The decision whether to “SSR-ify” a loop should thus be made either at compile time based on the expected number of iterations via an execution trace or heuristic. Or at runtime based on the actual number of iterations, in which case both an SSR and non-SSR implementation must be provided.

## 4 PERFORMANCE ANALYSIS

In this section we evaluate the impact and benefits of SSRs on the ISA (Section 4.1), a single core (Section 5.2), and an entire cluster (Section 5.3) in terms of performance, and area and energy efficiency.

### 4.1 ISA-level Impact

In this section we provide an evaluation of the performance and unit utilization impact of SSRs at the ISA level. We assume an ideal memory system with a constant access latency of one cycle. Consider the assembly code for a reduction operation in Figure 5a as an example. In a standard RISC-V implementation the hot loop consists of six instructions: two loads, two pointer increments, one Fused Multiply-Add (FMA), and a branch. Of these only the FMA is executed on the FPU and performs actual work towards the result, putting the upper bound of FPU utilization at 17%. The proposed SSR extension allows the loads and pointer increments to be implicitly encoded in the use of `ft0` and `ft1` as input registers, as shown in Figure 5b. This reduces the number of instructions in the hot loop to three: one counter decrement, one FMA, and a branch; putting the FPU utilization bound at 33%. Thus in a standard RISC-V ISA core the SSRs bring an architectural speedup of  $2\times$ .

SSRs interoperate well with hardware loop extensions such as those described by Gautschi *et al.* [4]. The use of hardware loops removes the back-branch from the hot loop and alleviates the need to explicitly track a loop counter. The baseline code shown in Figure 5c now has five instructions in the hot loop, with an FPU utilization bound of 20%. Using SSRs the hot loop reduces to a single instruction: the FMA, as shown in Figure 5e. Since loads and pointer adjustments are handled by SSR, and the loop iteration is handled by the hardware loop, the only thing left is the actual computation. This puts the FPU utilization to 100%. The hardware loops introduce an additional `lp.setup` instruction in the setup code. Thus in the presence of hardware loops the proposed SSR extension brings an architectural speedup of  $5\times$  with respect to a vanilla micro-architecture.

The RI5CY core [4] used in our evaluation provides additional instruction set extensions such as load/store post-increment instructions. These allow for the pointer increments in the baseline code to be elided as shown in Figure 5d, reducing the hot loop to three instructions: two post-increment loads and one FMA. Thus the upper bound on FPU utilization is at 33%, and the SSR extension still provides a speedup of  $3\times$ .

#### 4.1.1 Setup Amortization Analysis

The SSR extension introduces additional setup instructions at the beginning of a loop. This overhead must be amortized via the speedup gained due to elision of memory transfer instructions in the loop body in order for SSR to be beneficial. For our analysis we assume a loop nest of dimension  $d \in \mathbb{N}$  (and corresponding number of hardware loops),  $s \in \mathbb{N}$  data movers, and  $L \in \mathbb{N}^d$  loop iterations and  $I \in \mathbb{N}^d$  instructions per nesting level. This yields the following model for the

Table 2

Number of instructions  $N$ , useful ALU/FPU utilization  $\eta$ , and associated SSR-induced speedup  $S$  in the hot loop of a reduction, for integer and floating-point arithmetic, and different numbers of unrolled loop iterations  $U$ .

Kernel	Arith.	$U$	no SSR		with SSR		$S$
			$N$	$\eta$	$N$	$\eta$	
Standard RV32	int32	1	6	17%	3	33%	$2\times$
+ Hardware Loops	int32	1	5	20%	1	100%	$5\times$
+ Post-Increment	int32	2	6	33%	2	100%	$3\times$
Standard RV32	fp32	1	6	17%	3	33%	$2\times$
+ Hardware Loops	fp32	3	11	27%	3	100%	$3.7\times$
+ Post-Increment	fp32	3	9	33%	3	100%	$3\times$

total number of executed instructions in the SSR and non-SSR case:

$$N_{\text{ssr}} = \underbrace{4ds + s + 2}_{(a)} + \sum_{i=1}^d (I_i + 1) \prod_{n=1}^i L_n - \prod_{i=1}^d L_i \quad (1)$$

$$N_{\text{base}} = 1 + \sum_{i=1}^d (I_i + \underbrace{1 + s}_{(b)}) \prod_{n=1}^i L_n - \prod_{i=1}^d L_i \quad (2)$$

In the above, (a) describes the one time setup overhead for the SSR data movers before the loop nest; and (b) captures the explicit data movement instructions necessary in the non-SSR case (for example one load/store per data mover,  $s$  in total). We are now interested in finding the break even point where  $N_{\text{ssr}}$  becomes smaller than  $N_{\text{base}}$  and thus SSRs become advantageous. Algebraic transformation yields:

$$N_{\text{ssr}} \leq N_{\text{base}} \Rightarrow 4d + 2 \leq \sum_{i=1}^d \prod_{n=1}^i L_n \quad (3)$$

Interestingly, the number of iterations  $I_i$  per nesting level  $i$  does not affect the amortization behavior, and neither does the data mover count  $s$ . As an instructive example, let us assume that each loop in the nest performs the same number of iterations  $l$ , and the overall loop nest thus performs  $l^d$  iterations. The SSR implementation outperforms the baseline on loop nests with more than 5, 4, 1, or 1 overall iterations  $l^d$ , for 1D, 2D, 3D, or 4D loop nests, respectively. For the same scenario, Figure 6 outlines the achieved useful ALU/FPU utilization  $\eta$  with SSRs for a reduction over a  $d$ -dimensional hypercube with side length  $l$ , and thus overall number of iterations  $l^d$ . Each additional loop level in the nest introduces an instruction overhead for loop configuration, therefore requiring exponentially more iterations to achieve the same useful utilization as for lower-dimensional loops.

#### 4.1.2 Data Dependency Hazards

Until now we have assumed no data dependency stalls and all instructions to have one cycle latency, which holds for an integer reduction. Loads and floating point FMAs have two and three cycles of latency in RI5CY however, which requires some amount of loop unrolling to avoid any stalls. Table 2 shows the necessary unrolling and corresponding speedup  $S$  provided by SSRs. We define the useful utilization  $\eta$  as the fraction of cycles where the computation in the ALU/FPU contributes directly towards

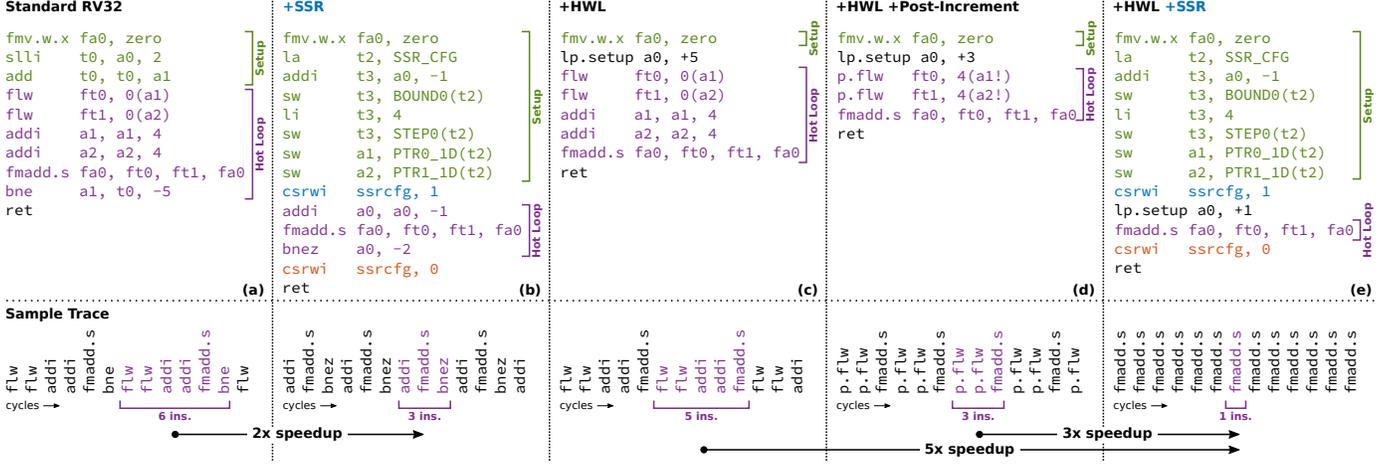


Figure 5. Comparison of assembly kernels with different ISA extensions (above). Sample execution trace for each kernel with a single hot loop iteration highlighted (below). From left to right: (a) baseline RV32 processor; (b) RV32 with SSR extension; (c) RV32 with hardware loops (HWL); (d) RV32 with hardware loops and post-increment loads; (e) RV32 with hardware loops and SSR extension. See Section 4.1 for details.

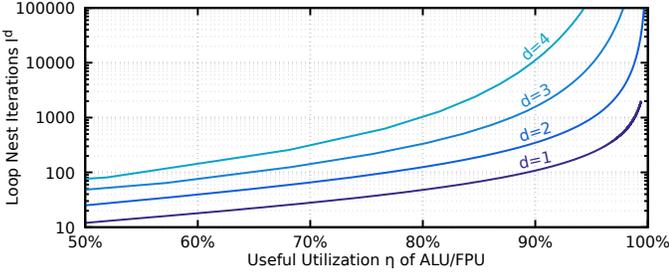


Figure 6. Useful ALU/FPU utilization with SSRs for a reduction over a  $d$ -dimensional hypercube with side length  $l$ , and thus overall number of iterations  $l^d$ . Deeper loop nests contain more overall loop configuration overhead, thus requiring longer-running loops to achieve the same useful utilization as lower-dimensional loops. See Section 4.1.1.

the result, as opposed to address or branch calculations. The integer reduction with post-increment loads requires two-fold loop unrolling in the baseline case to avoid data dependency stalls on the loads. The FP reduction additionally requires three-fold loop unrolling in the SSR case to avoid data dependency stalls on the accumulation register due to the FMA latency. If all loop iterations are fully unrolled (no data dependency stalls and all branch and pointer increment instructions fully amortized) the SSR-induced speedup reaches  $3\times$  in the limit in all cases.

We conclude that our proposed extension provides a significant  $2\times$  to  $5\times$  speedup, specifically  $3\times$  in our implementation in a DSP-optimized core. Furthermore it raises the upper bound for useful compute unit utilization to  $>95\%$  for reasonably-sized loop nests. As we will show in the following sections, these gains are not purely theoretical but translate well into real-life speedups in single- and multi-core settings.

## 4.2 Kernels

We have implemented the following kernels on our architecture to evaluate the performance benefits of SSRs:

- a *reduction* (dot product) over 2048 values;
- a *scan* (all prefix sums) over 4096 values;

- star-shaped *stencils* as they are found in the discrete Laplace operators (stencil diameter 11) in 1D (1024 points) and 2D ( $64\times 64$  points);
- the dense matrix-vector product (*GEMV*) of a  $64\times 64$  matrix and vector of 64 elements;
- the dense matrix-matrix product (*GEMM*) of two  $32\times 32$  matrices;
- the *ReLU* operation found in deep learning ( $\max(0, x)$ ) over 1024 values;
- the *fast Fourier transform* over 2048 values; and
- a *bitonic sort network* over 1024 values.

Problem sizes were chosen to fit into the first-level memory (TCDM) to remove effects of DMA transfers to other parts of the memory hierarchy from the analysis.

## 5 DESIGN AND IMPLEMENTATION RESULTS

### 5.1 Methodology

We based our implementation on RI5CY and the PULP platform [4], a silicon-proven open-source RISC-V many-core platform written in SystemVerilog. Our extensions were directly implemented in the core’s RTL description. We use the Synopsys Design Compiler to synthesize individual RI5CY cores and entire PULP clusters with and without the proposed hardware extension. Synthesis is performed under worst-case conditions at 0.72 V and  $-40^\circ\text{C}$  for GLOBAL-FOUNDRIES’ 22FDX technology, a 22 nm FD-SOI node. The design is constrained for a 500 MHz worst-case performance target. To estimate power consumption of the kernels we run simulations of the synthesized netlists in QuestaSim and perform a power estimation on the resulting trace using Synopsys PrimeTime. Architectural performance improvements in terms of cycles or unit utilization are technology independent.

### 5.2 Single-Core Comparison

#### 5.2.1 Performance

Figure 7 shows the speedup achieved by the proposed SSR extension in a RI5CY core over a range of data-oblivious

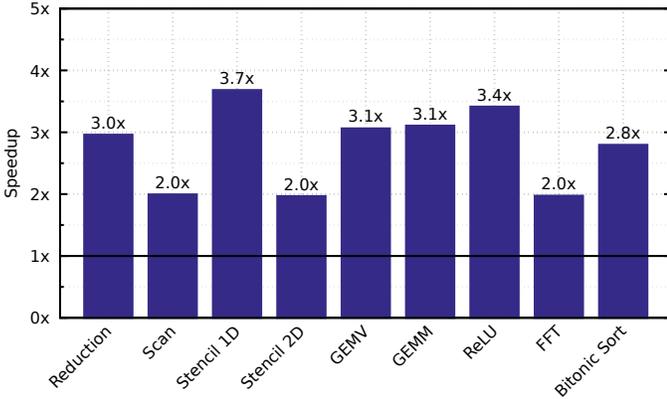
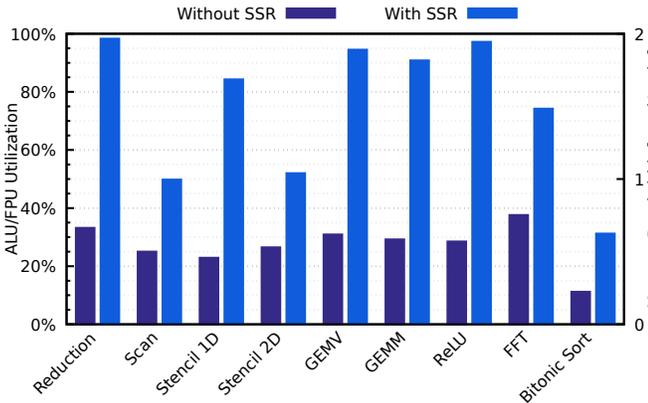


Figure 7. Speedup in a single RI5CY core extended with SSRs.

Figure 8. Useful ALU/FPU utilization  $\eta$  achieved by a single RI5CY core without and with SSR extension, including all overheads. Note that for the investigated kernels the utilization shown on the left axis is directly proportional to the memory bandwidth shown on the right axis.

kernels. Again we assume an ideal memory system with a constant access latency of one cycle. The implementations are fully optimized such that the loop bodies only consists of mandatory non-amortizable instructions. SSRs provide a speedup between  $2.0\times$  and  $3.7\times$  across the kernels, and generally at or above  $2\times$ .

Figure 8 shows the useful ALU/FPU utilization  $\eta$  of a RI5CY core before and after adding the SSR extension when executing the kernels outlined in Section 4.2. As outlined in Section 4.1, we consider  $\eta$  to be the fraction of instructions executed that contribute to the result. Without SSRs, the utilization is generally around 33%, heavily bounded by the number of load/store instructions during which the corresponding computational units do not perform any useful work. With SSRs, the hot loop can generally be reduced to the instructions essential for computing the result, such that the utilization during the hot loop reaches close to 100% in many cases.

Not only does the proposed architecture extension allow for almost perfect utilization of the functional units for many kernels, it also shifts the execution bottleneck from the instruction stream to the data interface. In the case of, for example, the hot loop of a reduction our implementation without SSRs would issue two loads over three cycles (66% data memory bandwidth utilization with one port), one

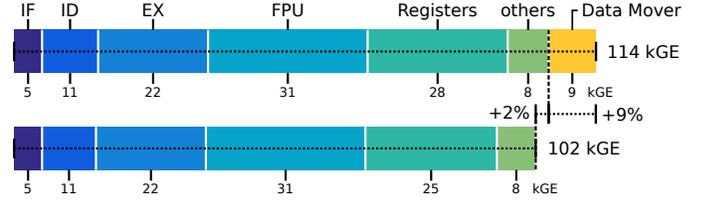


Figure 9. Area breakdown of a RI5CY core with (top) and without (bottom) the SSR extension in 22 nm technology. Shown are the area of the instruction fetch (IF), instruction decode (ID), and execute stages (EX), the FPU, register file, and other components inside the core; as well as the data mover outside of the core. Introduction of SSRs amount to a core-internal overhead of 2%, while the addition of a data mover and memory port multiplexer outside of the core increases area by another 9%. In total the addition of SSR increases area by 11%.

computation over three cycles (33% compute utilization), yet three instruction over three cycles (100% instruction bandwidth utilization). With SSRs, the core issues two loads over one cycle (200% data memory bandwidth utilization with one port, 100% with two ports), one computation over one cycle (100% compute utilization), and one instruction over one cycle (100% instruction bandwidth utilization). As indicated in Figure 8 the achieved memory bandwidth is proportional to the ALU/FPU utilization. The SSRs allow the core to ingest around  $3\times$  more data because a single instruction can consume three registers, which directly corresponds to the speedup and utilization gain achieved.

### 5.2.2 Critical Path

In our implementation the SSR extension adds up to 11 levels of logic to read and write timing paths surrounding the register file. This includes register address comparison and data multiplexing inside the core, and arbitration and FIFO access outside of the core. The critical path of the processor lies within the FPU and multiplier in the “execute” stage and measures 86 levels of logic. For comparison the “instruction decode” stage where the register file accesses occur covers merely 31 levels of logic. The addition of 11 levels of logic there has no effect on the critical path. Additional timing arcs exist from the “execute” stage to the register file’s write port. SSRs add 4 levels of logic to these paths, an increase of 4.7% in terms of logic depth or 85 ps (4.3%) in terms of delay. We thus conclude that the SSR extension increases the critical path by less than 5%.

### 5.2.3 Area

Figure 9 shows the area breakdown of a RI5CY core. First and foremost, the introduction of SSRs increases the area of the RI5CY core by 11.6 kGE or 11%, of which 2% are due to core-internal changes around the register file, and 9% are due to the data mover and memory port multiplexer around the core. This is a moderate cost considering that a core with SSR-equivalent capability would have to issue three instructions per cycle. For comparison, the SweRV dual-issue core [15] has an area of 237 kGE<sup>1</sup> at 902 MHz. This is due to a significant increase in complexity in the instruction fetch and decoding stages, and additional ports into the register file. Note that this does not yet include the impact

1. Synthesized by us for the same technology; performance for SSG 0.72 V –40/125 °C corner. Excludes caches.

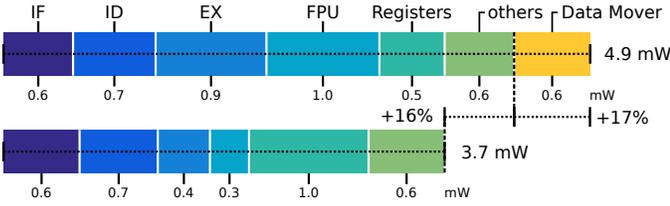


Figure 10. Power breakdown of a RI5CY core with (top) and without (bottom) SSRs executing a dot product. SSR accounts for a power consumption increase of 16% inside and 17% outside of the core, for a total increase of 33%. This is due to increased overall activity of the core. Note how FPU power increases due to higher utilization, and how register file power decreases due to the SSRs directly operating on memory. Power scaled to 1 GHz.

on the instruction cache that comes with a tripled instruction bandwidth. The SweRV core achieves  $38.3 \text{ Gop/s mm}^2$ , which is  $1.7\times$  less than the  $66.1 \text{ Gop/s mm}^2$  of RI5CY with SSRs. Note that SweRV is only dual-issue and has no FP support; a core with matching triple-issue and FPUs would further increase hardware complexity and overhead.

### 5.2.4 Energy Efficiency

Figure 10 shows the power breakdown of a RI5CY core. The introduction of SSRs increases the power consumption of the core by 33%, of which 16% are due to core-internal changes and 17% outside of the core. Note that the increase in power consumption is more significant than the increase in area, since the addition of SSRs significantly increases the computational throughput of the core. This can be seen in the increased FPU power (0.3mW to 1.0mW). Since most data is now directly streamed from memory, the power consumed by the register file is reduced from 1.0mW to 0.5mW, while the new data mover accounts for a corresponding additional 0.6mW. Together with the  $2\times$  to  $3\times$  speedup this translates into an energy efficiency improvement of  $1.5\times$  to  $2.3\times$  for the various kernels. Note that this does not yet account for energy saved in the instruction cache due to a reduced instruction bandwidth. This effect will be explored in the next sections.

## 5.3 Multi-Core Comparison

To evaluate the impact in a multi-core setting, we deploy the RI5CY core enhanced with SSRs described in Section 5.2 in a PULP cluster [7]. The cores operate on a shared TCDM which provides single-cycle access latency (see Figure 1). The memory is separated into multiple banks that are individually arbitrated. Multiple cores accessing the same bank will see one core succeed and the other cores stall for a cycle.

### 5.3.1 Performance

The results presented for a single core in Section 5.2 translate well into a cluster setting with multiple cores, with the speedup remaining largely the same. This multi-core environment introduces two new sources of overhead compared to a single-core environment:

- 1) Contention in the memory system. While the number of banks in the TCDM is chosen to provide approximately twice the bandwidth that the processors can request,

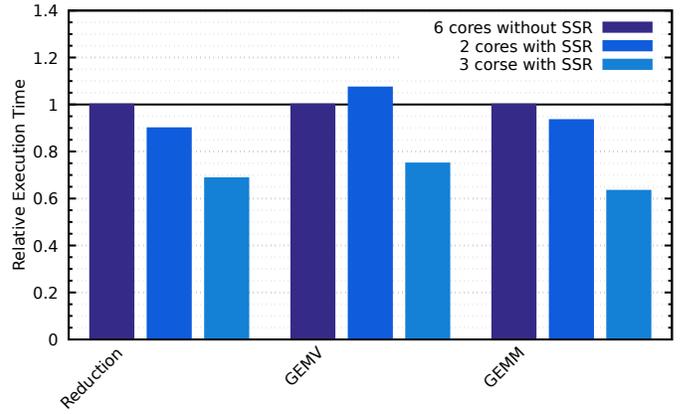


Figure 11. Execution time of a cluster with 3 cores and 3 FPUs with SSRs, a cluster with 2 cores and 2 FPUs with SSRs, both relative to a cluster with 6 cores and 3 FPUs without SSRs.

accesses into the same memory bank still conflict. Depending on the memory access pattern of a kernel, this leads to the cores observing stall cycles on the memory interface. In practice we observe that by employing data placement such as non-power-of-two data dimensions and offsets,  $>80\%$  of memory accesses are served immediately.

- 2) Parallelization and coordination overhead. The kernels require additional instructions to subdivide the problem space across the available cores. These occur generally outside the hot loop of the computation and are easily amortized over the problem size. Some kernels such as the FFT operate in stages, with the cores requiring synchronization after each stage. This synchronization is achieved with a dedicated event unit that provides efficient hardware barriers [16]. In practice the overhead of this synchronization is negligible.

Given the two- to three-fold speedup across kernels, we now evaluate by how much the number of cores and FPUs in the SSR-augmented cluster can be reduced while maintaining the performance of the non-SSR cluster. To this end we use a six core cluster with three FPUs, each shared by two cores, without SSRs as performance baseline. We then evaluate the performance of a two and a three core SSR-enabled cluster with core-private FPUs to this baseline. Figure 11 shows the kernel execution times for the two SSR clusters normalized to the execution times in the baseline cluster. We are interested in the cases where the ratio of execution times is close to one, at which point the reduced cluster perfectly matches the performance of the baseline. In order to match performance, kernels with an SSR-induced speedup around  $2\times$  are ideally executed on the three core cluster, while kernels around  $3\times$  speedup shall run on the two core cluster. The next sections evaluate how this reduction in size and complexity translates into area and energy savings.

### 5.3.2 Area

Figure 12 shows a detailed area breakdown for the three different cluster configurations described in Section 5.3.1. As elaborated above the smaller clusters offer equivalent performance across the kernels due to the addition of SSRs.

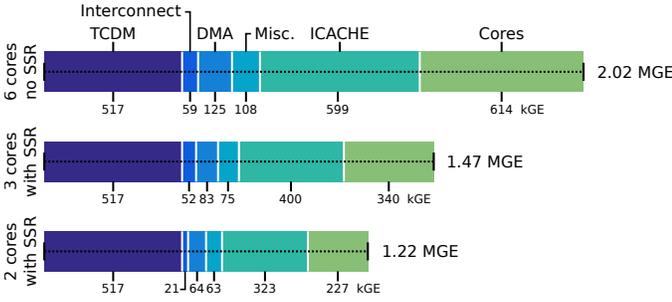


Figure 12. Area breakdown of three clusters: 6 cores/FPUs without SSRs (top), 3 cores/FPUs with SSRs (middle), and 2 cores/FPUs with SSRs (bottom). See Section 5.3.2.

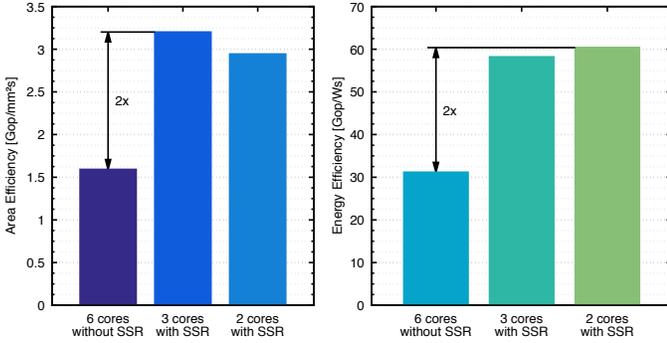


Figure 13. Area efficiency (left) and energy efficiency (right) of the three investigated cluster configurations, given as the number of operations per second achieved on a reduction workload, per area and power consumption of the cluster. See Section 5.3.2 and Section 5.3.3.

An interesting observation is the fact that the reduction in the number of cores provides additional savings in the cluster infrastructure. Fewer cores require less instruction bandwidth, which allows us to reduce the size and parallelism of the instruction cache. Furthermore the DMA and event unit need to provide fewer control ports. As such the 270 kGE to 390 kGE saved by removing cores translate into an overall saving of 550 kGE to 800 kGE when considering all secondary effects on the cluster.

This reduction in area at no loss in performance translates into a significant efficiency gain in terms of operations per second and area. Figure 13 shows the area efficiency achieved for the different cluster configurations. The introduction of SSRs yields an area efficiency improvement of  $2\times$ .

### 5.3.3 Energy Efficiency

Figure 14 shows a power breakdown for the three different cluster configurations. We use a trace of the reduction kernel to evaluate the power consumption. Power was estimated for typical silicon at 0.8 V and 25 °C. The smaller clusters with SSRs provide a  $1.3\times$  and  $1.7\times$  reduction in power consumption while maintaining a performance equivalent to the baseline cluster without SSRs. Note again that the reduction in cores saves more than just the core power itself, due to a reduction in cluster infrastructure, as described in Section 5.3.2. Figure 13 shows how the reduction in cluster power influences the per-operation energy efficiency. The proposed SSR extension provides an increase of  $2\times$  in energy efficiency across the investigated kernels.

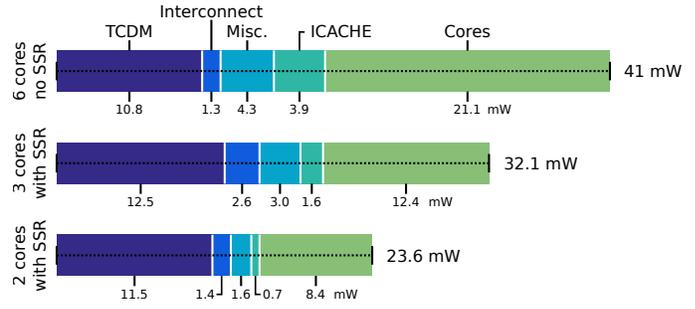


Figure 14. Power breakdown of three clusters: 6 cores/FPUs without SSRs (top), 3 cores/FPUs with SSRs (middle), and 2 cores/FPUs with SSRs (bottom). The clusters are running a reduction kernel; estimated for 1 GHz. See Section 5.3.3.

### 5.3.4 Instruction Pressure

One of the main benefits of SSRs is that they reduce the number of instruction fetches. The two-core cluster with SSRs executes  $3\times$ ,  $3.2\times$ , and  $3.5\times$  fewer instructions on the kernels in Figure 11 than the six-core cluster without SSRs. Power consumption of the instruction cache is reduced by  $5.6\times$ , further facilitated by the reduced number of cores.

## 5.4 Amdahl's Law and Strong Scaling

We observe an interesting effect of SSRs on parallel kernels. A dot product executed on a single core experiences a speedup of  $3\times$  through the use of SSRs. In a six-core cluster the speedup drops to  $2.2\times$ . This is due to additional overhead such as work splitting and synchronization that is purely sequential and cannot be parallelized, as per Amdahl's law. As shown in Figure 11, a two-core cluster with SSRs matches the performance of a six-core cluster without SSRs. This is thanks to the proposed architecture providing a speedup to sequential code by improving how well a single instruction stream can utilize computational resources. Since this reduces the number of cores — and thus parallelism — required to maintain a given compute performance by  $2\text{--}3\times$ , the implications of Amdahl's law and associated strong scaling are significantly mitigated. A system with SSRs requires  $2\text{--}3\times$  less compute resources and separate instruction streams to achieve a given performance, reducing the parallelization overhead correspondingly.

## 5.5 Automated Code Generation

The compiler additions outlined in Section 3.2 allow kernels to leverage the SSR extension without explicit instruction by the programmer. Applied to a reduction kernel, our LLVM pass achieves a  $2.0\times$  speedup over the baseline, compared to the  $2.1\times$  speedup of the manual implementation. In both cases manual replacement of hardware loop and DSP instructions were necessary since LLVM does not yet fully support all instruction set extensions of the RI5CY core. The remaining 5% of performance gap are due to sub-optimal instruction selection during SSR configuration in our prototype pass. This result shows the promise of using automated passes in later stages of a compiler to leverage SSRs in a manner transparent to the programmer. Existing software would be able to make use of our architectural extension via simple recompilation without the need for re-engineering

work. Further work includes improved handling of nested loops and only selectively mapping loops to SSRs based on compile-time heuristics or runtime knowledge.

## 5.6 Comparison to Other Cores

Table 3 outlines the impact of our architectural extension in comparison to other processor cores. We count FMA as 1 op and normalize silicon area to our 22nm technology node. Where otherwise unavailable we have estimated power consumption as proportional to area relative to the Ariane core [17] listed in Table 3. The kernel in question is a reduction with no data reuse, such as a dot product. For our comparison we consider an entire cluster of RI5CY cores as described in Section 5.3 to also capture the contributions of the instruction cache and TCDM as an equivalent to the L1 caches in other processors. In general we observe that due to factors such as bit width, Single Instruction Multiple Data (SIMD) vectorization, cache sizes, and target speed, area and energy efficiency do not necessarily correlate.

### 5.6.1 Peak Utilization

Table 3 lists the maximum theoretical utilization each core can achieve (“Util. Limit”). We define this metric as

$$\eta_{\max} = \lim_{N \rightarrow \infty} \eta(N) \quad (4)$$

for a problem size  $N$ . The case where  $N$  tends to infinity is interesting because it captures inherent inefficiencies that remain even when setup and loop overheads have asymptotically disappeared. For example, a dot product on RI5CY without SSRs has a constant setup overhead of two and a loop body of three instructions (see Section 4.1.1). With SSRs the overhead is seven and the loop body one instruction. Each loop iteration contains one “useful” operation. The utilization limit thus is:

$$\lim_{N \rightarrow \infty} \frac{N}{2 + 3N} = 33\% \quad (\text{without SSR}) \quad (5)$$

$$\lim_{N \rightarrow \infty} \frac{N}{7 + N} = 100\% \quad (\text{with SSR}) \quad (6)$$

As shown in Figure 6, RI5CY with SSRs already reaches close-to-peak utilizations at practical problem sizes, for example 93% at  $N = 100$ , and 99.3% at  $N = 1000$ . This groups processors into “efficiency classes” according to the peak utilization they can theoretically achieve. Single-issue in-order cores are limited to 33% as elaborated throughout this paper. Dual-issue cores reach 50% due to the ability to overlap instructions. Finally vector processors reach up to 100% due to the ability to overlap long-running instructions.

### 5.6.2 Single-Issue Cores

The Rocket core [18], [23] is a single-issue, in-order, 32 bit core. It suffers from the same bottleneck as RI5CY *without* SSRs. RI5CY *with* SSRs provides a 1.1× and 6.2× increase in area and energy efficiency, respectively.

### 5.6.3 Multiple-Issue Cores

BOOM [19] and SweRV [20] are both dual-issue, out-of-order, 32 bit cores. The ability to issue a load/store in parallel with a computation allows for a peak utilization  $\eta$  of 50% on long reductions. SSRs outperform BOOM by 6× in

terms of energy efficiency, and are roughly on par in terms of area efficiency. We have synthesized the SweRV core in 22nm ourselves, where it reaches 1 GHz in a typical process corner. Our extension achieves a 1.9× and 3.8× higher area and energy efficiency, respectively. Note that SweRV does not support FP instructions in contrast to the other cores, which would further decrease efficiency.

### 5.6.4 Vector Processors

In terms of vector processors we consider Ara [21] in a 16 lane configuration and Hwacha [18] in a four lane configuration. The design of the RISC-V vector extension allows the vector computation and loads/stores to run in parallel. This is a similar benefit as with our architecture, allowing the cores to reach a peak utilization  $\eta$  of 100%. Since both processors operate on 64bit data paths, we assume two-way SIMD vectorization of 32 bit operations by doubling performance and efficiencies. Our approach provides a 1.4× and 3.2× higher energy efficiency. SSRs have a 4.9× and 3.5× lower area efficiency, respectively. The vector processors achieve this with a highly regular data path which entails 32-way and 8-way SIMD vectorization. This comes at a significant reduction in programmability due to the need for special data placement and shuffling. SSRs on the other hand have no such requirements.

More specifically, vector processors are optimized to perform element-wise or reduction operations on long vectors. This is usually achieved by tightly coupling FPU lanes to individual memory banks of the Vector Register File (VRF), yielding excellent scaling behaviour. However this also strictly limits which vector elements can form an operand pair for an instruction. Many interesting problems, e.g., matrix multiplication, FFT, convolution, or stencils, exhibit data reuse. In the presence of data reuse, a vector element is paired with more than one other elements, in possibly many different VRF banks. Leveraging this reuse in a vector processor requires inter-bank data exchange, and is essential to avoid additional memory accesses, which unnecessarily push a kernel towards the memory-bound region. Such data exchange is commonly implemented by shuffling instructions and units. In order to keep up with the FPUs, and the likely need to shuffle before every computation in a realistic problem, the shuffle unit must match the bandwidth of the FPUs. Since instructions operate in a register-to-register fashion, this doubles the bandwidth load of the VRF and requires the vector processor to sustain simultaneous FPU and shuffling operations.

Intuitively, a vector processor *without* shuffling unit is comparable to our system (see Section 5.3) without the all-to-all memory interconnect. In this scenario our SSR-based cluster has a small power and area overhead of 3% to 8% due to the interconnect. A more realistic vector processor *with* shuffling unit requires additional VRF bandwidth and all-to-all bank connectivity, leading to a power overhead of 26% to 48% compared to the equivalent SSR-based system (see Figure 14). This does not account for additional load on the integer core when preparing index arrays for the shuffle, or the additional bandwidth required for the index array to be streamed into the shuffling unit. SSRs essentially provide a *free shuffle* for every data word.

Table 3  
Comparison of the SSR architectural extension with other processor cores for a reduction kernel. See Section 5.6 for details.

Core	Width [bit]	Order	Issue	Peak Perf. [op/cycle]	Util. Limit	Freq. [MHz]	Tech. [nm]	Area <sup>†</sup> [mm <sup>2</sup> ]	Power [mW]	Area Eff. <sup>‡</sup> [Gop/s mm <sup>2</sup> ]	Energy Eff. [Gop/s W]
2x RI5CY & SSR [us]	32	in	1x	2	100%	625	22	0.243	23.6	3.20	60.4
6x RI5CY [4]	32	in	1x	6	33%	625	22	0.402	41.0	1.59	31.2
Ariane [17]	64	in	1x	2	33%	1700	22	0.239	88.1	13.4	6.37
Rocket [18]	32	in	1x	1	33%	1000	40	0.118	34.0	2.80	9.71
BOOM [19]	32	out	2x	2	50%	1200	28	0.321	118 *	3.74	10.1
SweRV [20] †	32	out	2x	2	50%	1000	22	0.168	62.0 *	5.95	16.1
Ara (16 lanes) [21]	64	in	1x	32	100%	1040	22	1.99	794	15.6	41.9
Hwacha (4 lanes) [18]	64	in	1x	8	100%	1000	45	0.717	430	11.2	18.6

<sup>†</sup> Normalized to 22 nm [22]    <sup>‡</sup> Our synthesis in 22 nm    \* Area-proportional estimate relative to similar core

Table 4  
Peak efficiency comparison of the SSR architectural extension with commercial processor cores. See Section 5.7 for details.

Core	Tech. [nm]	SIMD	Area Eff. [kop/s GE]	Energy Eff. [Gop/s W]
3x RI5CY & SSR [us]	22	1	1.28	58.2
Cortex A5 [18]	40	1	1.24	12.5
Core i9-9900K [24]	14	16	1.14	26.2
Volta V100 [25]	12	32	1.32	61.2

## 5.7 Comparison to Commercial Processors

Another avenue to improve compute unit utilization is explored by superscalar processors [26] as they are built by Intel or AMD, for example. Architectural extensions to the simple single-issue core enable multiple instructions to operate in parallel and out of order [15], [27]. This enables address calculation and data movement to occur in parallel to the actual computation, ideally to an extent where the FPU utilization reaches 100%. Here the von Neumann bottleneck itself is not addressed directly, requiring significant instruction bandwidth and hardware resources to sustain computation. We note that all these schemes involve a significant *per FPU* increase of hardware resources dedicated to fetching and decoding of instructions, while providing capabilities to compute addresses and perform memory accesses in parallel. These changes come at a considerable area and energy cost and must be further complemented by the memory system and other parts of the core infrastructure. SSRs offer a more lightweight approach to improve FPU utilization which directly addresses the von Neumann bottleneck. Furthermore SSRs may have applicability within superscalar processors as well, acting as alternative or complementary approach to SIMD parallelization.

Table 4 shows a high level comparison of SSRs with commercial processor cores considering only peak metrics. In contrast to the open source cores in Section 5.6, detailed micro-architectural analyses of commercial cores are not readily available. We therefore perform our own estimates of area and energy efficiency based on public information on technology, die area, and area ratios evident in die shots. The area efficiency normalized to gate equivalents is roughly equal among all four cores. A cluster with three SSR-enabled RI5CY cores outperforms an ARM Cortex A5 and an Intel Core i9-9900K by 4.7 $\times$  and 2.2 $\times$  in terms of energy efficiency, and is roughly on par with an Nvidia Volta V100. Note that the larger Intel and NVIDIA cores have a

significant technology-driven energy efficiency advantage. They also require substantial SIMD/SIMT vectorization to operate efficiently, which is not needed in our architecture.

## 6 RELATED WORK

### 6.1 Streaming Acceleration

#### 6.1.1 NTX

The SSR extension bears similarity with the approach taken in the “Network Training Accelerator” [8], [28]. It leverages the regularity of nested loops and affine addressing to operate directly on multi-dimensional data in a scratch pad memory. NTX is designed as a co-processor tightly coupled to but positioned outside of a RISC-V core, requiring explicit operation offloading. Since it can only execute a single fundamental operation in its innermost loop body, kernels that consist of more than one such operation have to be executed in multiple passes. Extending NTX to allow for multiple operations would require significant hardware additions: Each instruction in the loop body would require another configuration cycle by the control processor, or an instruction fetch unit to read instructions from memory directly. The ability to perform different operations would require an instruction decoding stage, local registers, and an ALU. In essence such an extension would be akin to developing another instruction set processor. Rather, our architecture is a generalization of this scheme as we pull the essential address generation and data streaming into a RISC-V core to leverage the already existing infrastructure. Not only does our approach boost the utilization of a RISC-V core significantly, it does so while *decreasing* the number of fetched instructions at a very small area overhead. The seamless integration into the register file leaves the ISA untouched, except for the addition of a single CSR.

#### 6.1.2 Out-of-Order Processors

Wang et al. [29] have explored the applicability of stream-based memory accesses to out-of-order processors based on high-level architectural simulations. They conclude that a significant fraction of loop nests in their investigated benchmarks is amenable to representation as streams, which supports the wide applicability of our architectural extension. We provide a more hardware-centric view and show that stream semantics can be embedded into an existing ISA in a very lightweight and non-invasive way. Furthermore we find that such a streaming extension applies especially

well to small, single-issue, in-order cores, where it provides much more significant speedups and energy savings than in out-of-order cores.

### 6.1.3 WM Machine

Wulf et al. [30] have presented a machine architecture where loads and stores act on FIFOs rather than directly on a register file. The FIFOs are exposed in the ISA as a special “register zero” ( $r0/f0$ ). This is similar to our approach. Yet it requires explicit loads/stores, address calculation, and branching due to the lack of hardware loops, which reduces the positive impact on tight loop performance.

## 6.2 Loop Acceleration

Hayenga et al. [31] have targeted loop execution on large-scale, superscalar, out-of-order x86-style processors. The authors modify such a processor such that loops in the instruction stream pass through the processor frontend only once. Instructions are then re-issued multiple times in the backend, which is given self-iteration capabilities. The approach thus reduces frontend overheads which arise in such large-scale processors. The loop execution and potential to pre-execute loads is conceptually similar to SSRs. Our proposed SSR architecture does not focus on large-scale processors, but offers a solution across a potentially wide range of processor sizes. In contrast to this architecture, SSRs support nested loops and allow for loads/stores to be completely elided. We show that the benefits are significant especially in small-scale in-order processors.

## 6.3 Data Address Generator

Data address generators have been thoroughly studied in literature [32], [33], [34], and are the subject of a wide range of patents [35], [36], [37]. Their complexity ranges from simple pointer incrementing functionality to extensions made to accommodate address patterns commonly found in Fourier and discrete cosine transforms [38].

### 6.3.1 DSPs

Address generators are well-established in DSPs [39]. Specialized load/store instructions on designated address registers allow for pointer arithmetic to be subsumed. These AGUs generally support a wide range of patterns, modes, and modulo operations, which makes them costly in terms of area and energy footprint [32]. Our work extends this by four essential insights:

- 1) streaming semantics on registers allow for elision of load/store instructions in addition to pointer arithmetic;
- 2) SSRs remove the need for specialized instructions, reducing the ISA impact to almost zero by allowing all existing instructions to leverage streams and AGUs;
- 3) this allows for streaming operations and AGUs to be integrated into small, energy-efficient, general-purpose cores; and
- 4) a lean and efficient dedicated AGU for affine address patterns and loop nests is sufficient to provide significant 2x to 3x gains in such cores, while leaving more complex and irregular patterns to be handled by the

instruction stream. As we have shown many kernels can be arranged such that their innermost hot loops are of such a regular structure, and handling of further complexity in the outer loops is amortized well over the computation.

### 6.3.2 Superscalar Processors

Modern superscalar processors such as those built by Intel and AMD include multiple AGUs closely coupled to the load/store units to perform address calculation. These AGUs may be triggered as part of the indirection in a memory operand as part of a CISC instruction, or by using explicit string instructions such as REP with LODS/STOS [40]. Our proposed SSR extension has less impact on the ISA as it does not require additional, dedicated instructions. Provisioning for explicit streaming instructions would require a 16x replication of each instruction in a four-operand ISA such as RISC-V, and 9x in a three-operand ISA such as x86. Furthermore we show that it is possible to encode memory accesses in compute instructions even in a RISC architecture, without the need for complex operand addressing modes.

## 6.4 Vector Processors

### 6.4.1 ORCA LVE

An approach similar to ours has been proposed in the “Lightweight Vector Extension” of the ORCA processor [41], [42]. This LVE streams data directly from a scratch pad memory through the ALU of a processor and stores the result back in memory. One fundamental operation is performed on the data as it is in flight, making the approach similar to NTX [8]. It suffers from the same limitation, requiring multiple “ping-pong” passes from and to memory to compute more complex kernels. In its current implementation, the LVE only supports one-dimensional data. The SSR extension is a superset of the LVE and given the similarity of the approaches should be comparable in hardware complexity.

### 6.4.2 RISC-V “V” Extension

Other approaches to improve FPU utilization exist. Vector processors have seen an increase in popularity recently with new architectures such as Hwacha [18] and Ara [21] being proposed, and vector semantics being integrated into ISAs such as RISC-V. These processors amortize instruction overheads by encoding identical operations on multiple different operands in a single instruction, allowing a single instruction to trigger hundreds of operations. Such a scheme requires high operational and spatial regularity as is commonly found in SIMD data paths, but in an amplified fashion due to the increased and often dynamic vector length. This requires powerful scatter/gather accesses into the memory system and shuffle operations to reorganize data within the register file, which add complexity and do not contribute directly to the computation result. Our proposed architectural allows instructions to operate directly on the scratch pad memory at word granularity, essentially providing a “free” shuffle operation before and after each computation. The freedom in terms of data access patterns provided by the data mover in our architecture allows for a significantly more relaxed programming model, with less

emphasis on data placement and no need for shuffling instructions. In a sense, an SSR machine is a superset of a vector machine in that instructions for the latter can readily be translated into the former, but not vice versa.

## 7 CONCLUSION

In this work we have presented the “Stream Semantic Register” ISA extension for single-issue in-order processor cores. It introduces stream semantics on a subset of the processor’s registers which can be enabled and disabled at the software’s discretion via an additional CSR. This leaves all existing instructions in the ISA virtually untouched and allows them to leverage data streams. SSRs bring a  $2\times$  to  $5\times$  speedup at the ISA level by allowing load/store instructions to be elided. We have implemented the SSR extension into RI5CY, a highly DSP-optimized RISC-V core. In this setting, it provides a  $2\times$  to  $3.7\times$  speedup and  $1.5\times$  to  $2.3\times$  energy efficiency gain across a wide range of data-oblivious kernels. This comes at a modest cost of a 5% increase in critical path delay and 11% increase in area. In a multi-core cluster the number of cores can be reduced by  $2\times$  to  $3\times$  while maintaining the same performance, which improves area and energy efficiency by  $2\times$ . SSRs reduce the number of fetched instructions by  $3.5\times$  and instruction cache power by  $5.6\times$  on realistic kernels.

## ACKNOWLEDGMENTS

This work has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement number 732631, project “OPRECOMP”.

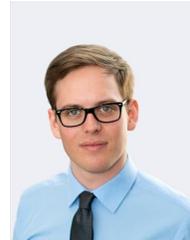
## REFERENCES

- [1] M. B. Taylor, “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse,” in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1131–1136.
- [2] S. Pagani, H. Khdr, J.-J. Chen, M. Shafique, M. Li, and J. Henkel, “Thermal safe power (TSP): Efficient power budgeting for heterogeneous manycore systems in dark silicon,” *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 147–162, 2016.
- [3] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2,” RISC-V Foundation, May 2017.
- [4] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, “Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.
- [5] G. S. Sohi, “Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers,” *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 349–359, 1990.
- [6] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, “Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis,” in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 26–36.
- [7] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beigné *et al.*, “Energy-efficient near-threshold parallel computing: The pulpv2 cluster,” *Ieee Micro*, vol. 37, no. 5, pp. 20–31, 2017.
- [8] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, “A scalable near-memory architecture for training deep neural networks on large in-memory datasets,” *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 484–497, 2019.
- [9] F. Conti, P. D. Schiavone, and L. Benini, “XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2940–2951, 2018.
- [10] I. Tanasic, I. Gelado, M. Jorda, E. Ayguade, and N. Navarro, “Efficient exception handling support for GPUs,” in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 109–122.
- [11] *Cell Broadband Engine Architecture*. IBM, 2007.
- [12] A. Kurth, P. Vogel, A. Marongiu, and L. Benini, “Scalable and efficient virtual memory sharing in heterogeneous SoCs with TLB prefetching and MMU-aware DMA engine,” in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 292–300.
- [13] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious RAMs,” *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.
- [14] C. Latner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.
- [15] Western Digital, “SweRV RISC-V Core,” [https://github.com/westerndigitalcorporation/swerv\\_gh1](https://github.com/westerndigitalcorporation/swerv_gh1), December 2018, accessed: February 2019.
- [16] F. Glaser, G. Haugou, D. Rossi, Q. Huang, and L. Benini, “Hardware-Accelerated Energy-Efficient Synchronization and Communication for Ultra-Low-Power Tightly Coupled Clusters,” in *Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition*, 2019.
- [17] F. Zaruba and L. Benini, “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.
- [18] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, “A 45nm 1.3 GHz 16.7 double-precision GFLOP-S/W RISC-V processor with vector accelerators,” in *ESSCIRC 2014-40th European Solid State Circuits Conference (ESSCIRC)*. IEEE, 2014, pp. 199–202.
- [19] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic, “BOOMv2: an open-source out-of-order RISC-V core,” in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.
- [20] T. Marena, “RISC-V: high performance embedded SweRV™ core microarchitecture, performance and CHIPS Alliance,” Western Digital Corporation, April 2019.
- [21] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, “Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.
- [22] R. Brain, “14 nm Technology Leadership,” *Technology and Manufacturing Day*, 2017.
- [23] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelvitz *et al.*, “The rocket chip generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
- [24] D. Schor, “Coffee Lake - Microarchitectures - Intel,” May 2019, accessed: September 2019. [Online]. Available: [https://en.wikichip.org/wiki/intel/microarchitectures/coffee\\_lake](https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake)
- [25] NVIDIA, “Tesla V100 GPU Architecture Whitepaper,” August 2017, accessed: September 2019. [Online]. Available: <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
- [26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.
- [27] K. Asanovic, D. A. Patterson, and C. Celio, “The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor,” University of California at Berkeley Berkeley United States, Tech. Rep., 2015.
- [28] F. Schuiki, M. Schaffner, and L. Benini, “NTX: An Energy-efficient Streaming Accelerator for Floating-point Generalized Reduction Workloads in 22 nm FD-SOI,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 662–667.
- [29] Z. Wang and T. Nowatzki, “Stream-based memory access specialization for general purpose processors,” in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 736–749.
- [30] W. A. Wulf, “The WM computer architecture,” *ACM SIGARCH Computer Architecture News*, vol. 16, no. 1, pp. 70–84, 1988.

- [31] M. Hayenga, V. R. K. Naresh, and M. H. Lipasti, "Revolver: Processor architecture for power efficient loop execution," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 591–602.
- [32] G. Talavera, M. Jayapala, J. Carrabina, and F. Catthoor, "Address generation optimization for embedded high-performance processors: A survey," *Journal of Signal Processing Systems*, vol. 53, no. 3, pp. 271–284, 2008.
- [33] A. Sudarsanam, S. Liao, and S. Devadas, "Analysis and evaluation of address arithmetic capabilities in custom DSP architectures," *Design Automation for Embedded Systems*, vol. 4, no. 1, pp. 5–22, 1999.
- [34] M. Kuulusa, J. Nurmi, J. Takala, P. Ojala, and H. Herranen, "A flexible DSP core for embedded systems," *IEEE Design & Test of Computers*, vol. 14, no. 4, pp. 60–68, 1997.
- [35] D. M. Pfeiffer, D. T. Stoner, J. P. Norsworthy, D. D. Dipert, J. A. Thompson, J. A. Fontaine, and M. K. Corry, "High speed image processing system using separate data processor and address generator," Jan. 15 1991, uS Patent 4,985,848.
- [36] R. J. Gove, K. M. Gutttag, K. Balmer, and N. K. Ing-Simmons, "Address generator with controllable modulo power of two addressing capability," Feb. 25 1997, uS Patent 5,606,520.
- [37] S. P. Pekarich and X.-a. Wang, "Address generator for interleaving data," Apr. 15 2003, uS Patent 6,549,998.
- [38] W. Luo and J. Xu, "Fast Fourier transform address generator," Feb. 13 1996, uS Patent 5,491,652.
- [39] M. Ilić and M. Stojčev, "Address generation unit as accelerator block in DSP," in *2011 10th International Conference on Telecommunication in Modern Satellite Cable and Broadcasting Services (TELSIKS)*, vol. 2. IEEE, 2011, pp. 563–566.
- [40] C. H. Van Berkel and P. P. E. Meuwissen, "Address generation unit for a processor," Jun. 3 2008, uS Patent 7,383,419.
- [41] G. Lemieux, "ORCA-LVE: Embedded RISC-V with Lightweight Vector Extensions," in *Proceedings of the 4th RISC-V Workshop*, July 2016.
- [42] G. G. F. Lemieux, J. Edwards, J. Vandergriendt, A. Severance, R. D. Iaco, A. Raouf, H. Osman, T. Watzka, and S. Singh, "TinBiNN: Tiny Binarized Neural Network Overlay in about 5,000 4-LUTs and 5mW," *CoRR*, vol. abs/1903.06630, 2019. [Online]. Available: <http://arxiv.org/abs/1903.06630>



**Fabian Schuiki** received the B.Sc. and M.Sc. degree in electrical engineering from ETH Zürich, in 2014 and 2016, respectively. He is currently pursuing a Ph.D. degree with the Digital Circuits and Systems group of Luca Benini. His research interests include computer architecture, transprecision computing, as well as near- and in-memory processing.



**Florian Zaruba** received his BSc degree from TU Wien in 2014 and his MSc from the Swiss Federal Institute of Technology Zurich in 2017. He is currently pursuing a PhD degree at the Integrated Systems Laboratory. His research interests include design of very large scale integration circuits and high performance computer architectures.



**Torsten Hoefler** is a Professor of Computer Science at ETH Zürich, Switzerland. He is also a key member of the Message Passing Interface (MPI) Forum where he chairs the "Collective Operations and Topologies" working group. His research interests revolve around the central topic of "Performance-centric System Design" and include scalable networks, parallel programming techniques, and performance modeling. Torsten won best paper awards at the ACM/IEEE Supercomputing Conference SC10, SC13, SC14, EuroMPI'13, HPDC'15, HPDC'16, IPDPS'15, and other conferences. He published numerous peer-reviewed scientific conference and journal articles and authored chapters of the MPI-2.2 and MPI-3.0 standards. He received the Latsis prize of ETH Zurich as well as an ERC starting grant in 2015.



**Luca Benini** holds the chair of digital Circuits and systems at ETHZ and is Full Professor at the Università di Bologna. Dr. Benini's research interests are in energy-efficient computing systems design, from embedded to high-performance. He has published more than 1000 peer-reviewed papers and five books. He is a Fellow of the ACM and a member of the Academia Europaea. He is the recipient of the 2016 IEEE CAS Mac Van Valkenburg award.