Stream-based Memory Access Specialization for General Purpose Processors

Zhengrong Wang
University of California, Los Angeles
seanzw@ucla.edu

Tony Nowatzki
University of California, Los Angeles
tjn@cs.ucla.edu

ABSTRACT

Because of severe limitations in technology scaling, architects have innovated in specializing general purpose processors for computation primitives (e.g. vector instructions, loop accelerators). The general principle is exposing rich semantics to the ISA. An opportunity to explore is whether richer semantics of memory access patterns could also be used to improve the efficiency of memory and communication. Two important open questions are how to convey higher level memory information and how to take advantage of this information in hardware.

We find that a majority of memory accesses follow a small number of simple patterns; we term these streams (e.g. affine, indirect). Streams can often be decoupled from core execution, and patterns persist long enough to express useful behavior. Our approach is therefore to express streams as ISA primitives, which we argue can enable: prefetching stream accesses to hide memory latency, semi-binding decoupled access to remove address computation and optimize the memory interface, and finally inform cache policies.

In this work, we propose ISA-extensions for decoupled-streams, which interact with the core using a FIFO-based interface. We implement optimizations for each of the aforementioned opportunities on an aggressive wide-issue OOO core and evaluate with SPEC CPU 2017 and CortexSuite[1, 2]. Across all workloads, we observe about 1.37× speedup and energy efficiency improvement over hardware stride prefetching.

ACM Reference Format:
Zhengrong Wang and Tony Nowatzki. 2019. Stream-based Memory Access Specialization for General Purpose Processors. In *ISCA '19: 46th International Symposium on Computer Architecture, June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322229

1 INTRODUCTION

Inspired by the slowing of technology scaling and the limits of improving general purpose processors, architects have developed many techniques which specialize aspects of general purpose execution. This includes well-known techniques like vector instructions, custom application specific instructions, and more recently a number of in-core accelerators which specialize for particular computational patterns [3–9]. These approaches add rich semantic information into an ISA and leverage this with a more aggressive or streamlined computation substrate. For example, vector architectures expose instruction parallelism in the ISA; CCA and DySER expose instruction dependence in the ISA; BERET exposes the presence of common control patterns to the ISA, and so on.

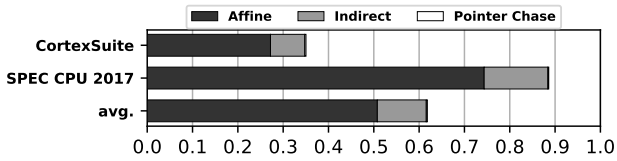


Figure 1: Prevalence of Streams

Though these techniques deliver significant performance and energy advantages, their specialization capabilities end at the boundary to the memory system. This is problematic considering Amdahl's law for energy; one study of general purpose cores showed that the cache hierarchy and network consumed more than 50% of power on chip [10]. Also, specialization paradigms typically make strong assumptions on common code behaviors, limiting their potential scope[1]. Specializing the core computation substrate and relying on only traditional memory abstractions is insufficient.

One unexplored opportunity is to perform the equivalent form of specialization for memory primitives: expose rich semantic information about memory operations at fine grain at the ISA level, and take advantage with microarchitecture mechanisms. To be worth the specialization effort, such an approach should be designed to address many of the sources of inefficiency of memory access, including within the core pipeline, at the interface to cache, and within the cache system itself. More specifically, this includes the overhead in the pipeline for address generation instructions, the overhead of requiring long instruction windows for generating overlapping memory requests, and also reducing the number of requests and transfers in the memory system.

A critical question then is: what is the structure of memory accesses which can be exploited? Our perhaps unsurprising answer is streams – repeated patterns of memory access, occurring due to loops and nested loops. Figure 1 shows the prevalence of streams in CortexSuite [1, 2] and SPEC CPU 2017, with a breakdown to affine, indirect, and pointer-chasing types. On average, more than half of dynamic accesses across both benchmark suites can be considered as streams, with the simplest affine streams being the most common.

Given the premise of specialization for streams, a number of interesting questions arise: How should memory access patterns be communicated to avoid high-overhead? Should data be consumed in a binding way, or only through cache? How to ensure the right decoupling distance? Which patterns should be expressible, and

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322229

[1]For example, CCA [3] and DySER [5] rely on clusters of dependent computations, BERET [4] relies on dynamic control-speculation being unnecessary, etc.

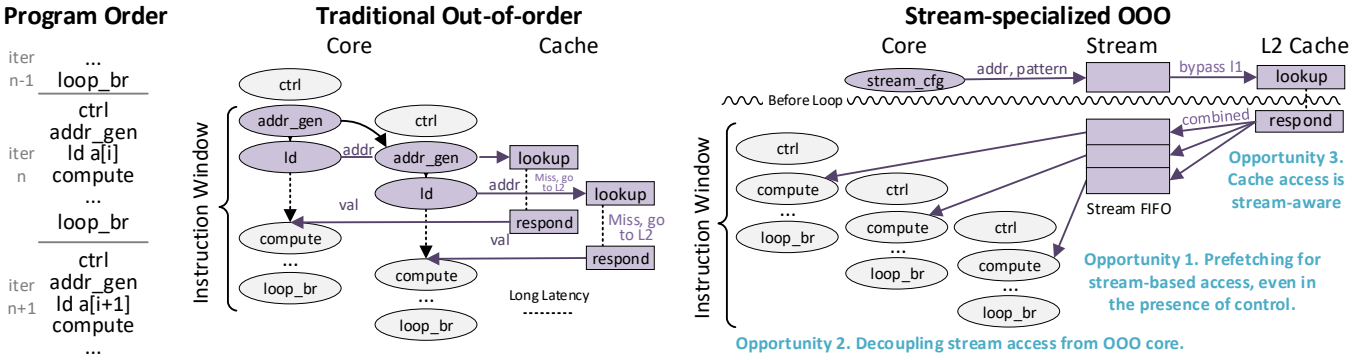


Figure 2: Overview of Decoupled-Stream Paradigm vs Traditional Out-of-Order

can patterns be expressed in the presence of control flow? How should the cache take advantage of this information?

Our goal in this work is to explore some possible answers to these questions. The crux of our approach is twofold: 1. the ISA is extended to express coarse grain streaming patterns, and 2. the core pipeline is responsible to keep track of its relative index into each stream. With such a design, we argue the following opportunities are possible (depicted by Figure 2):

Opportunity 1: Stream-based Prefetching: Having knowledge of access patterns and their relationship to the core's control flow can lead to a very effective stream-based programmable prefetcher. The prefetcher can understand when exactly to make a request based on how far ahead of the core the prefetch is. Similar to other programmable prefetchers, this would enable the scheduling of memory requests far past the limits of a traditional OOO core processor's instruction window.

Opportunity 2: Stream-decoupling: Streams primitives can further be incorporated into the functional semantics of the program to enable what we refer to as *semi-binding* prefetching. Following the principle of decoupled access execute [11], a specialized memory access engine would generate requests corresponding to streams, and ordinary core instructions can accesses stream data through registers that are mapped to this data (we call these pseudo-registers). There are several potential advantages, including the removal of address generation instructions from the general core pipeline, coalescing accesses from related streams into a smaller number of requests to cache, and also reducing the possibility of cache pollution through timely, semi-binding prefetch.

Opportunity 3: Specialized Cache Policies: Streams are precise definitions of an access pattern. Various caches policies could take advantage of advanced knowledge of these patterns: replacement policies, dead-block prediction, cache-bypassing etc. One specific idea is to let the cache bypass streams based on the expected footprint of the stream.

In this work, we propose the design of a stream specialized processor (SSP), which exploits the opportunities of stream prefetching, stream decoupling, and stream-aware caches. We propose a decoupled-stream ISA and associated extensions to an out-of-order (OOO) processor, propose and implement compiler transforms for this ISA, and evaluate in a cycle-level simulator. Our evaluation, consisting of 33 workloads across SPEC CPU 2017 and CorexSuite [1, 2], demonstrates 1.37× speedup and 1.36× energy efficiency improvement over a baseline aggressive OOO core with stride prefetching. Our contributions are:

- **Stream Characterization:** A study on the prevalence of streams with exploitable characteristics, showing streams are common, lengthy, and often interact with program control flow.
- **Stream Specialization Principles/Mechanisms:** The concept of specializing multiple facets of general purpose processor execution (core, memory-interface, and cache) with stream abstractions, along with the development of mechanisms which leverage this information.
- **Stream Specialization ISA/Microarchitecture:** A lightweight set of ISA-extensions for applying stream-specialization to an ISA, along with microarchitecture design for implementation and integration with the core pipeline.

Paper Organization: In the remainder of the paper, we first present an overview (§2), then discuss key related work from several relevant areas (§3). We next characterize the prevalence of streams (§4), and use this insight to propose ISA extensions for decoupling streams (§5). This is followed by a description of how the new information is exploited in an SSP microarchitecture (§6), as well as stream-aware prefetch and cache policies (§7). Finally, we discuss evaluation methodology (§8), results (§9), and conclude (§10).

2 OVERVIEW

In this section, we first make an argument for the requirements of a stream-specialized interface. We then overview the approach, in terms of the ISA and microarchitecture of the stream-specialized processor (SSP), as well as the basics of how we exploit each stream-specialization opportunity.

Stream ISA Requirements: In our stream-characterization study (Section 4), we find that streams are common (>50% of dynamic access), which is promising. However, some streams are shorter (37% less than 100 accesses), streams often have indirect access (about 11%), and streams often coexist with control-flow (>50% of stream accesses). Based on this characterization, we argue that a decoupled-stream ISA interface should have five qualities:

❶ **Integration-simplicity** It should be lightweight and not require excessive core modification, while also efficiently conveying stream patterns to hardware with low overhead for short streams.

❷ **Generality** It should be able to capture both regular and irregular (indirect) memory access patterns.

❸ **Pattern-simplicity** The stream definition should be analyzable by hardware (for stream-aware cache policies).
❹ **Control under streaming** It should enable control dependent access, without interfering with the core speculation.
❺ **Abstract** It should not expose the underlying microarchitecture.

Decoupled-stream ISA Approach: Streams are initialized through a configuration instruction which defines the pattern. To communicate with the core pipeline, each stream is assigned a pseudo-register, which is a register implicitly mapped to stream data. This means that instructions which consume/produce stream data remain unmodified, which keeps the integration simple (req.❶). Streams may specify other streams as dependences, which enables generality across indirect types (req.❷). Streams are simple to analyze (req.❸), because there are only a handful of common patterns.

Streaming under control (req.❹) is possible because of how we update the meaning of each pseudo-register, i.e. the data item a pseudo-register corresponds to within the stream. Specifically, our approach is to add a "step" instruction to the core, which indicates the advancement of the stream from the core's perspective. This implies that data within the stream may be used multiple times, or even ignored if not needed depending on core control flow.

Streams are general and ubiquitous, and therefore useful across subsequent ISA generations (req.❺). The only aspect of the microarchitecture which is exposed is the number of pseudo-registers.

Microarchitecture Approach: We modify the core pipeline's front-end to track the position within each stream based on interpreting step instructions. We add a *stream engine* to generate addresses and interact with the memory system. Finally, we add a load-stream FIFO (and store-stream FIFO), which core instructions may access when loading (and storing) pseudo-registers. Streams may be configured and accessed speculatively; and a simple protocol enables the rollback of stream positions and configurations on mispeculation.

Opportunity 1: Stream-based prefetching: Stream requests are decoupled from the core's instruction window, enabling deep prefetching for regular and irregular memory access. The primary benefits of decoupling is reducing the negative impact of long-latency memory accesses, without requiring a large instruction window. Maintaining the relationship to the control-flow of the core through the "step" instruction enables the prefetcher to keep an accurate distance without running ahead and polluting the cache.

Opportunity 2: Stream-decoupling: The principle of stream-decoupling is to create a direct interface between data which is stream-prefetched, and the core instructions, eliminating the redundant address generation which is typical of programmable prefetchers, and simultaneously reducing instruction pressure on the core pipeline. In addition, the benefits of vectorization of memory are brought to traditionally non-vectorizable code; stream loads fetch data in units of the L1 bandwidth, even though a particular code may have too much control-flow to be otherwise vectorized, and our design requires no vector-shuffling.

Decoupled streams are what we call *semi-binding*. They are binding in that they are obligatory and consume registers. However, they are non-binding in that *not* all data must be consumed, and so the hardware can ignore memory protection faults for non-consumed data. Therefore, stream-decoupling keeps the benefits of binding prefetch, even in the presence of control flow and indirect access. Also, the prefetch distance can be controlled through dynamic throttling, reducing the negative impact of being obligatory.

Opportunity 3: Cache Awareness: The stream engine has access to high-level information regarding streams, through stream configuration instructions. Using this information, and supplemented by the access pattern, the stream engine can make requests to the cache in a way that is aware of stream behaviors.

We specifically explore the idea of exposing the *footprint* of the stream to the cache. A footprint is an under-approximation of the total number of cache lines accessed. Knowing the footprint in advance can lead to an enhanced cache bypassing policy, where requests from a high-footprint stream (that would not fit in e.g. an L2 cache) with low temporal reuse are bypassed to larger caches so that they do not evict useful data.

3 RELATED WORK

While the idea of stream-specialized general purpose processors itself is novel, it derives inspiration from and has an intimate relationship with at least four main areas of architecture research: specialization of address generation, decoupled access-execute, prefetching, and cache policy enhancements.

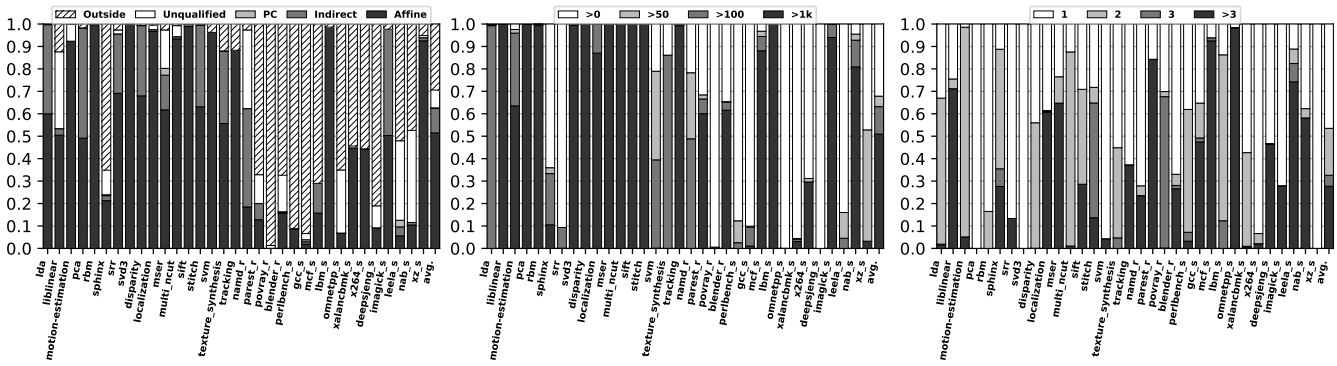
Memory Interface Specialization: The concept of exposing patterns of memory access as "streams" within an ISA perhaps originated with the Imagine Stream Processor [12], designed for media processing. Following in their footsteps, a variety of specialized architectures have employed stream abstractions, like RSVP [13], Q100 [14], Softbrain [15], VEAL [16] and CoRAM++ [17]. None of the above target a traditional general purpose out-of-order core (e.g. no control speculation) or make a general cache stream-aware.

Memory Access Dataflow (MAD) [18] is a reconfigurable front-end/memory-fetch engine for accelerators and SIMD units, but does not use stream abstractions. MAD powers down the OOO core pipeline while it is active, and also does not support exceptions or control speculation. On the other hand, our approach extends the OOO core and does not interfere with its capabilities.

A philosophically similar approach is XMem [19] and the locality descriptor [20], which are cross-layer programming abstractions for conveying memory semantics. The key difference is that our ISA conveys semantics about each access at the instruction level, rather than describing a memory region. This gives our ISA a more fine-grain view of memory patterns.

Decoupled Access Execute (DAE): By encoding and performing streaming memory operations separately from the Von Neumann order of the program, we are implementing a limited form of DAE [11] which is tailored to certain common access patterns. From that perspective, other DAE architectures exploit similar parallelism within programs and can also hide memory latency [21, 22].

One example is Outrider [23], which supports multiple simultaneous decoupled inorder threads; our stream-generator supports multiple concurrent streams. DeSC [24] is a recent example which couples an OOO core with either a second OOO core or an accelerator for the computation. DeSC adds compiler/architecture support to break dependences for certain control-dependent and indirect memory access patterns, which we also address in our work.



(a) Stream Breakdown (PC: Pointer-Chasing) **(b) Average Stream Length** **(c) Number of Control Paths**

Figure 3: Stream Characterization

In the accelerator space, several designs decouple the datapath, like DySER [5], CCA [3], Chainsaw [9] and ASIC accelerators [8, 25, 26]. However, they are fundamentally limited by the instruction window of the general purpose core for latency-hiding. A recent work in this space is Buffets [27], which is a storage idiom for decoupled access-execute accelerators, enabling fine-grain synchronization, flexible data-reuse and composability.

Runahead/Prefetching: Similar to DAE, prefetching also hides memory latency. Stream-specialized processors have an advantage over traditional hardware prefetchers (e.g. stride-based [28] and indirect [29], spatial/temporal memory streaming [30–32], and irregular correlating prefetchers [33]), in that the data they prefetch is guaranteed to be accurate. Also, the stream-FIFOs can be seen as software-exposed stream-buffers [34, 35], eliminating the overhead of dynamic prediction as well as tag-checking in caches.

The type of prefetching performed with SSP is more similar to software/execution-driven prefetching. For example, The stream-generator can be viewed as a highly-specialized helper thread [36–43]. Software prefetching [44] also exposes access patterns through the ISA, and some recent proposals are highly programmable [45] and can be compiler-directed [46].

SSP is different in two key ways: There is no redundant address generation, and there is little potential for cache pollution. These are due to SSP's semi-binding prefetch, which eliminates the problems with traditional binding using regular registers (too much register pressure, cannot prefetch under faults or control flow).

Cache-Policy: Our cache-policy enhancements are inspired primarily by prior works in cache bypassing, like those based on reuse count [47–50]. Using the footprint for modifying the cache replacement policy is inspired by prior cache insertion-policy techniques [51, 52], which are designed to dynamically detect behavior that we have available statically in the stream definition. We also combine static and dynamic information about memory accesses for cache bypassing, as was previously explored in the GPU space [53].

4 STREAM CHARACTERIZATION

A foundational question for a stream specialized processor (SSP) is whether *programs* exhibit enough streaming behavior to take advantage of. We define four key questions:

Q1 - Coverage: Do streams cover program access?
Q2 - Pattern: What are their access patterns?
Q3 - Length: Are streams long enough to be meaningful?
Q4 - Control: Are they entangled with the core's control flow?

This section attempts to answer the above questions through a trace-based analysis of streams. The observations both justify our motivation and provide insights for the ISA and microarchitecture.

Stream Definition: For this analysis, we define a stream to be the dynamic sequence of memory operations associated with a static instruction, where the longest extent is defined as the entry and exit of the outermost containing loop.

Desirable Properties: Extracting access patterns from memory streams and decoupling them from the VonNeumann order of the program is key to achieving high performance and energy efficiency. However, certain memory streams are more amenable than others for specialization. In particular, certain properties are desirable, so we consider streams with these properties to be "qualified":

- **Within an inlinable loop:** This is because saving and restoring streams at function-call boundaries would be more expensive than for a scalar register.
- **Address is Control Independent:** We intend to leave control decisions within the non-stream portion of the program, so that traditional speculative execution may take advantage. We disqualify control-dependent address computation, as supporting this would simultaneously eliminate the benefit of decoupling (close interaction with non-stream instructions), and make analysis by hardware for cache specialization more difficult.
- **Affine Strides:** This restriction keeps the hardware for streams trivial (an integer ALU is sufficient) and also enables simple analysis by cache specialization hardware.

Clarifications: Three clarifications are important. First, these properties only need hold up to some loop nesting level, because they can be considered to start at that level. Second, *data-dependent streams* (indirect and pointer-chasing) are still potentially quite profitable to target, as their address is still control independent.

Third, it can still be profitable to target streams where not all elements of the stream's data are guaranteed to be used – i.e. the memory access *can be* control dependent. Note that the access being control dependent is orthogonal to the address being control dependent, and control-dependent access is not disqualified.

Methodology: We profile SPEC CPU 2017 to capture general application behavior, as well as CortexSuite [1, 2] to reflect the importance of data-processing. To analyze the workloads, we use dynamic instrumentation and trace analysis. We exclude stack spilling accesses as it would inflate the number of affine stream accesses.

Q1 and Q2: Coverage and Type: Figure 3a shows the breakdown of dynamic memory accesses. Memory accesses outside of inlinable loops is labeled "outside". Depending on its access pattern, each qualified stream is further classified as affine, indirect or pointer-chasing (PC). On average, 51.49% dynamic memory accesses belong to affine streams, while 10.90% come from indirect streams and 0.3% from pointer-chasing streams. Although on average indirect streams contributes less than 12%, for some benchmarks more than 40% of stream accesses are indirect, e.g. namd_r. These benchmarks require efficient support for dependence between streams to achieve high performance.

Observation 1: More than 60% of dynamic memory access instructions belongs to a stream with specializable properties.

Observation 2: Affine streams are the most common, while indirect streams are also common for some benchmarks.

Q3: Length: Figure 3b shows the accumulated distribution of average memory stream length, weighted by their dynamic instruction count. 51% of stream accesses belong to a stream of length at least 1000, and 62.1% come from streams with length at least 100. Notice that a stream of length N represents at least N loop iterations (greater if we consider the reuse of stream elements). Combining with other instructions within the loop, even a shorter stream may span across a long instruction window if the loop body is large.

Observation 3: Streams are generally long enough to convey meaningful patterns. However, shorter streams are common enough to require low initialization overhead.

Q4: Interaction with Control: For general purpose workloads, it is common for streams to coexist with the core's control flow. To characterize the degree of this interaction, Figure 3c shows the accumulated distribution of stream accesses, grouped by the number of control paths within the loop containing that static memory access instruction. Loops with 3 or more control paths contribute 27.7% of dynamic stream access.

Observation 4: Because many stream accesses coexist with control flow, it is essential for the ISA to decouple control flow.

5 DECOUPLED-STREAM ISA

Informed by the insights of the previous section, and requirements described in Section 2, we now define a decoupled-stream ISA. We begin by describing the basic concepts, then elaborate with examples. Finally, we discuss compilation support.

5.1 Decoupled-Stream Concepts

The following are the essential components of the ISA extensions:

- **Streams:** Streams are decoupled portions of the program which together generate memory accesses. They are explicitly constructed and deconstructed (`stream_cfg` and `stream_end` instructions), and their data can be accessed by traditional instructions through pseudo-registers.
- **Stream Types and Dependence:** There are two stream types: memory streams describe a memory access pattern; induction streams define a repeating pattern of values. Memory streams are dependent on either 1. induction variable streams (affine access patterns), 2. other memory streams (indirect access patterns), or 3. themselves (pointer-chasing).
- **Pseudo-registers and Stream Stepping:** A pseudo-register is a register which refers to a stream's data. The meaning of the register, the position into the stream, is updated by a `stream_step` instruction to the associated induction variable stream. In other words, a `stream_step` advances the pseudo-register position of all dependent streams.
- **Memory Semantics and Architecture State:** Semantically, a load occurs at the point of the first use of a pseudo-register corresponding to a load stream after stepping or configuring, while a store happens at every write to a pseudo-register of a store stream. Pseudo-registers become part of the architecture state after their first use, and are removed from architecture state after stepping the corresponding induction variable stream.
- **Pseudo-register Width:** Pseudo-registers have a definable width, which determines the amount of data read by each step instruction. Instructions which access narrower portions of the register specify an offset.

5.2 Stream-ISA Extensions

To explain the ISA intuitively, we describe its principles and potential through a series of examples which stress its different aspects. Figure 4 shows five codes, with the decoupled-stream pseudo code, and the stream dependence graph.

Basic Operation – Figure 4(a): The example is a dense vector addition, using three affine streams. There are two load streams (`a[i]`, `b[i]`) and a store stream (`c[i]`), which are both dependent on an induction variable stream (`i`). Each stream is assigned a pseudo-register, which is used by the traditional instructions to interact with streams. Next we explain the use of stream instructions.

`stream_cfg`: A `stream_cfg` instruction is inserted before entering the loop which uses the stream's data. It defines all of the streams within this loop level, including their type (induction/memory), pattern (stride, width, and optional length), dependences, and starting address[2]. This interface conveys stream information at a coarse granularity, using a stable interface.

In practice, after configuration is complete, the hardware may begin fetching data ahead of the core's requests based on the program. Also, note that the stream's data never needs to be consumed, though an unused stream would occupy a pseudo-register.

`stream_step`: As described earlier, the `stream_step` instruction advances the pseudo-register position of the induction variable and dependent streams. In this example, stepping `s_i` will also advance `s_a`, `s_b` and `s_c` by one element. This highlights how the approach of implicitly stepping dependent streams avoids redundant step instructions.

An alternative decoupled ISA could have used a "destructive read" interface, where a read of a pseudo-register implicitly advances the state. This would have worked well in this example, eliminating

[2]This can be implemented with a series of instructions for each stream. While this is shown abstractly in the figure, in our implementation it is an instruction cache load of configuration data, interpreted by the hardware.

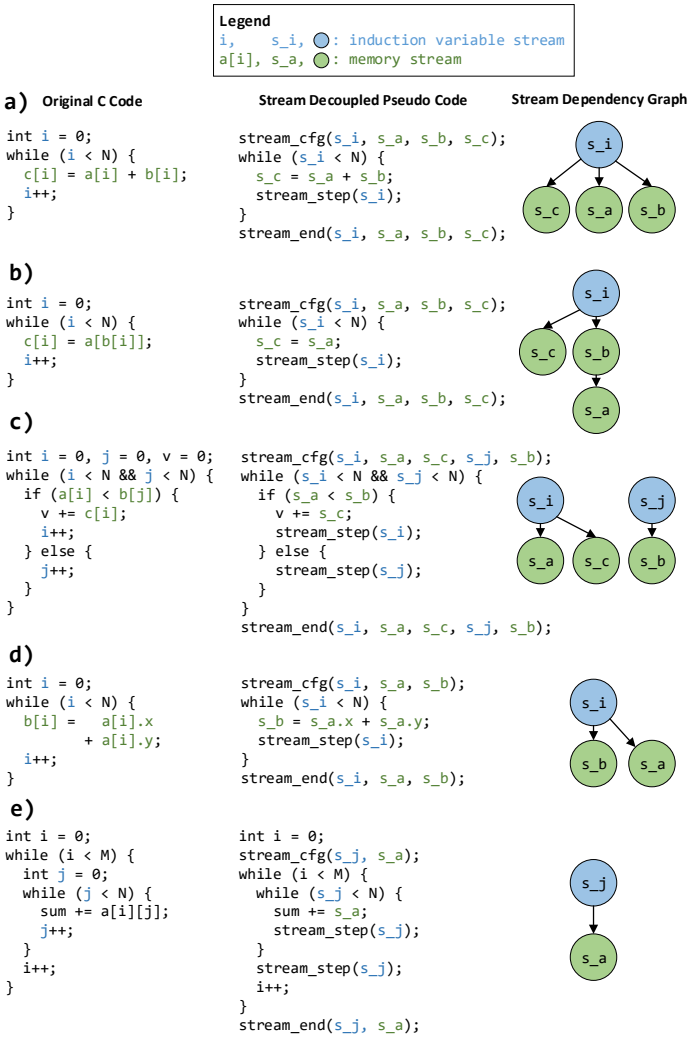


Figure 4: Pseudo Code Examples

the need for the step instruction. However, this would not allow control-dependent access, described shortly.

`stream_end`: The `stream_end` instruction deallocates a set of streams from the corresponding pseudo-registers. Generally this happens after the loop in which the stream use occurred, as it does in the running example. Also note that an explicit `stream_end` enables the termination of a stream to be data dependent.

Indirect Memory Access – Figure 4(b): Indirect memory access is supported by making the address of one memory stream dependent on the value of another. In this example, `s_a` is dependent on `s_b`. We also refer to `s_b` as the base stream of `s_a`. Note that `s_a` is also stepped with the `stream_step` of `s_i`.

Control Flow – Figure 4(c): The `stream_step` interface enables the ISA to specify control-dependent access, meaning that a stream element may be used 0 times, once or many times. This example iterates over the elements of `a[i]` and `b[i]`, but their relative ordering is data dependent. This is implemented by conditionally stepping stream `s_i` and `s_j` depending on the outcome of the comparison. Having a `stream_step` instruction makes it trivial to support such

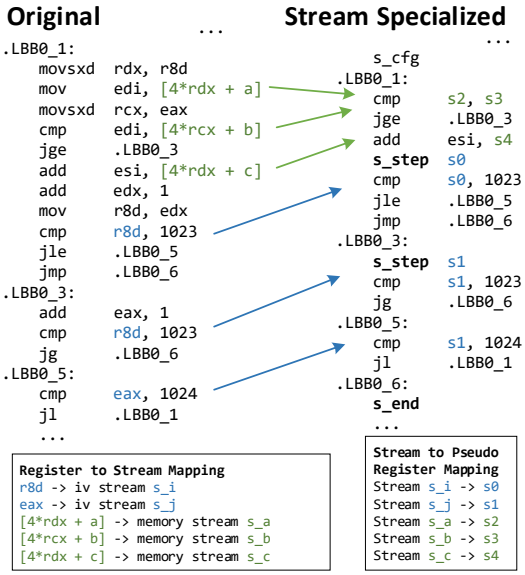


Figure 5: Assembly Example

a scenario, by simply replacing the increment instruction with a corresponding `stream_step`.

Notice that in this example, not every element of `s_c` will eventually be used. In a traditional ISA, such unused elements make it harder for the memory system to figure out the access pattern and prefetch for future elements. With the help of the compiler and the support of explicit control on when to step the stream, we effectively decouple the access pattern from the control flow. This also enables a new opportunity for the hardware, as now it knows the addresses and can speculate whether the stream element will be used and whether it should prefetch.

Coalescing Streams – Figure 4(d): In some situations, memory accesses patterns become more regular when coalescing from two static instructions. A common scenario is iterating through an array of structs, as shown in the example. Here the streams accessing `x` and `y` fields can be coalesced into a single stream, where the pseudo-register width is now 8 bytes. This reduces the total number of streams, and also makes the access pattern contiguous.

To support this, the user of a pseudo-register may add an immediate-offset parameter to specify the offset from the head of the pseudo-register[3]. In this example, `s_a.y` has an offset of 4 bytes.

Nested Loops – Figure 4(e): It is sometimes advantageous to configure a stream at an outer loop level to increase the length. This example iterates over a 2D array, and is transformed into a single memory stream. Because N is known and there is no conditional stepping, the affine access pattern can be determined before entering the outer loop. Note that we need an additional `stream_step` after the inner loop to skip the unused exiting iteration of `j=N`. The induction variable `i` is not specialized as a stream in this example, but implicitly encoded in the configuration of `s_a`.

[3]In theory, this support could be added to the ISA through extending each instruction, or adding a header byte to specify the offset. In our implementation we added this information to the stream configuration.

5.3 Pattern Limitations and Speculation

The address patterns that we support are limited to those which are decouplable, i.e. determined at the point of configuration. There are two relevant caveats: 1. data may be conditionally used, and 2. the outermost dimension of the pattern can have an unknown length. This corresponds to the two forms of speculation that we allow for address patterns: that cache lines in the pattern are likely useful, and that streams are long enough that the overhead of loading a few extra items is acceptable.

This has implications for how many loop levels we can hoist up the configuration of a stream. If at a given outer-level either the trip count of the inner loop becomes unknown, or the induction variable becomes conditionally stepped, then the decoupling invariant can no longer be maintained.

5.4 Compiler Support

We implement compiler support to identify streams and transform the original program to decouple streams. Our implementation uses LLVM. There are three phases: recognizing stream candidates, selecting qualified candidates, and code generation.

Recognizing Stream Candidates: The compiler treats every static memory access instruction in a loop as a candidate for a memory stream, and every ϕ node in the loop entrant basic block as an induction variable stream. ϕ nodes not in the loop entrant basic block represent other control dependent values and are not considered as candidates. Starting from the candidate instruction, the compiler performs a backwards search on its operands, gathering instructions until it encounters a loop-invariant, a constant, or another candidate instruction. It will also record dependences between stream candidates.

Selecting Stream Candidates: After finding the candidates, the compiler identifies all candidates qualified for stream decoupling. First, a candidate can only be qualified if it has a simple enough pattern to match the supported affine, indirect, and pointer chasing patterns. Specifically, it can not contain any ϕ node, which represents control-dependent address generation. Also, it should not contain any unsupported operations, e.g. floating point operations.

Second, the compiler checks the dependencies between streams. A trivial constraint is that if any of its base streams within the same loop level is unqualified, the stream is unqualified. A more sophisticated case is to handle multiple induction variables. To support configuring streams in outer loops, if the address pattern limitations in 5.3 are satisfied, we remove the dependency on any outer loop induction variable so that the memory stream depends on only one inner most induction variable (iteration domain is incorporated into the inner loop variable). If this is not possible, then the stream becomes unqualified.

During this phase, the compiler coalesces affine streams with the same induction variable and small offset between their elements. The compiler will also drop some qualified streams if the total number of streams exceeds the maximum. The compiler prioritizes memory streams with no dependent streams to drop, as they are less likely on the critical path.

Similar to some prior work [18, 54, 55], we take a hardware/software codesign approach to memory aliasing. The compiler records which loads and stores may alias, so that non-aliasing streams can bypass the core's LSQ.

Code Generation: During the code generation phase, the compiler first generates the stream configuration for the selected candidates. The configuration specifies 1. which pseudo-register to represent the stream; 2. the type of the stream (induction, load, store); 3. loop invariant values (stride, width); and 4. stream dependences.

The compiler transforms the loop by 1. inserting `stream_cfg`, `stream_step` and `stream_end` instructions; 2. replacing the operand of a user instruction with the corresponding pseudo register, along with the offset within the element (for a coalesced stream); 3. removing the memory access instruction for a memory stream, and possibly insert a dummy user instruction to ensure the original program order is preserved; and 4. if there are no other users, remove the address computation instructions.

Figure 5 shows both the original and transformed X86 assembly code for example in Figure 4(c). The stream operands are replaced by the corresponding pseudo-registers.

6 MICROARCHITECTURE EXTENSIONS

A traditional processor can be extended with a small number of relatively simple structures to create a stream-specialized processor (SSP), as we depict at a high level in Figure 6. SSP extensions have four basic responsibilities: 1. Maintain the *core's view* of stream position based on configuration and stepping instructions; 2. Maintain the *streams' decoupled view* of their state, and allow streams to issue memory requests; 3. Maintain the data which is decoupled between the core's view and the streams' view, and enable core instructions to access this data; and 4. Keep the above consistent during misspeculation and exceptions. We overview each of the corresponding components:

Core's view – Iteration Map: The frontend of the pipeline maintains the iteration map (Figure 8), which counts iterations of induction-variable streams, as seen by dispatch. A `stream_config` instruction updates the mapping from the stream index to iteration count table. A `stream_step` increments the iteration count table. Stream-consuming instructions access the table to ascertain the current iteration, which is used to index into stream FIFOs.

Streams' View – Stream Engine: The stream engine is the central component of a stream-specialized design, as shown in Figure 7. It contains an induction table to hold iterator parameters, and a load engine and store engine, which generates load and store requests to memory. Multiple streams may be mapped to each engine.

To explain the operation, first, a `stream_config` instruction will load data to the stream engine's configuration unit. This will initialize the designated streams and parameters on the load and store engines. When the unit receives notice of a committed `stream_end`, the associated stream is deallocated from the load or store engine.

The load and store engine maintain three tables describing the state of any stream. The first is the stream's definition, containing the pattern (affine, indirect, linked) and parameters (stride, width). The second is the stream's state, essentially the memory-side view of the induction variables. This is where the current address is stored. Finally are operands, which store any dependences on the data of other streams (for indirect streams). Each stream can have up to two dependences on other streams, and for each dependence,

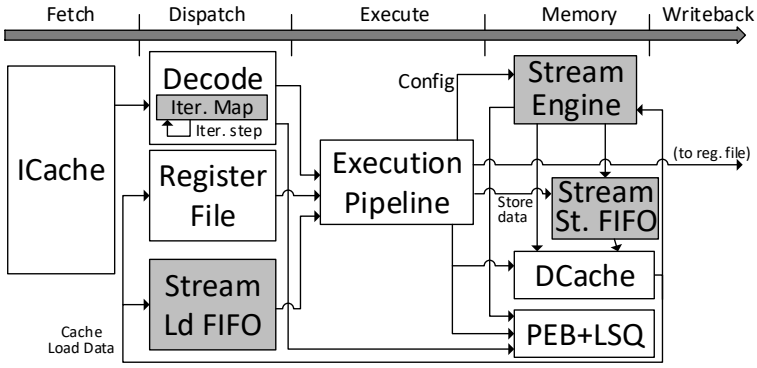


Figure 6: Stream-specialized Pipeline

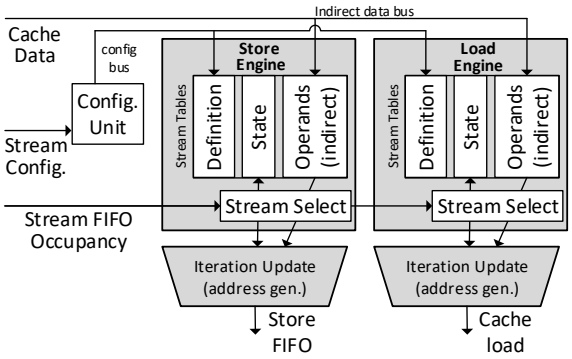


Figure 7: Stream Engine

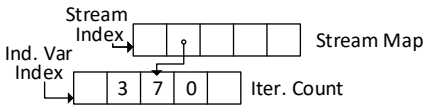


Figure 8: Iteration Map

we keep enough space for four iterations worth of storage for any given dependence to run-ahead.

Each cycle, the stream select unit picks a stream based on the readiness of corresponding operands (if any) and whether remaining FIFO entries are allocated to the stream (see Section 7 on page 9 for allocation policies). Requests are in units of per-port L1 cache bandwidth (64 bytes in our design). In parallel with sending the request, the stream's state is updated for the next iteration.

Decoupled Data – Stream FIFOs: The stream FIFOs are responsible for holding decoupled state either from or to memory (load and store FIFO). We use an implementation similar to the dynamically partitioned queues of Outrider [23], which use a pointer table to virtualize a single wide buffer into multiple FIFO queues (in our case, one for each concurrent stream). For core instructions which consume stream data, they would access the load stream FIFO instead of the register file[4]. For stores to streams, core instructions only produce values, and addresses are produced by the stream engine. These are combined at the store stream FIFO before sending to the memory system.

Control Mispeculation: Stream requests and uses are speculative to avoid pipeline serialization. We discuss implementation in the context of an R10K [56] style merged register file. To maintain the core's view, during mispeculation rollback while the map table is being reverted using register mapping information stored in the core's reorder buffer, the iteration map is also similarly decremented for each mispeculated step instruction. If no `stream_config` instruction is mispeculated, only the core's view of the stream is reverted, because the addresses for streams are control independent. This means we achieve a low-cost form of selective replay [57] by virtue of semi-binding prefetch.

When reverting a `stream_config` instruction, both the core's view and streams' view is reverted. On the core side, the stream map entries are freed, and the corresponding streams are de-configured within the stream engine. Data stored within decoupled FIFOs corresponding to these streams is flushed.

Precise State and Context Switch: Precise state and exceptions can be supported using the same speculation recovery mechanisms as above. Because the stream configuration and pseudo-register values (specifically those which have not been stepped since the last use) are part of the architecture state, they must be saved on context switch. These items amount to less than 1KB for our design.

Interaction with Memory: Before issuing a stream request, the virtual address is translated by the core's MMU. Access to TLB can be delayed to favor core loads, but address translation needs only occur once per-page for affine streams with low stride, reducing TLB access in the common case.

Because stream-loads effectively aggressively reorder loads, may-alias streams require memory disambiguation and recovery. For this, the stream engine relies on the core's LSQ to perform memory disambiguation, along with its memory dependence predictor (similar to MAD [18]). When dispatching a core instruction that semantically triggers the memory access, it is inserted into the core's LSQ as a normal load/store. In order to detect RAW dependence between a store and a prefetch stream element, the stream engine also maintains a prefetch element buffer (PEB). The PEB can be considered a logical extension of the LQ, which contains the prefetched elements by the stream engine. Elements in the PEB are freed when the first use is dispatched, or when the element is released as unused. Traditional memory order checking is performed between SQ and LQ + PEB. Hitting in the PEB indicates a misordered stream access, and the streams' view should be reverted. Overall, may-alias streams can still be aggressively reordered, but do not reduce the LSQ-energy.

To implement a non-relaxed memory model, SSP needs to be integrated with the core's memory-consistency speculation mechanism (e.g. if relevant coherence state changed, flush core pipeline and roll back streams' view).

Finally, because the ISA semantically only performs memory operations if a pseudo-register is accessed, memory faults from prefetching stream requests are delayed until the execution of the corresponding user instruction. Faults are silently ignored if the FIFO entry is unused.

[4] An alternate design could partition the physical register file for use as a stream-FIFO.

7 STREAM-AWARE POLICIES

Here we describe how to leverage stream-information to design effective prefetching and stream-aware cache bypassing policies.

7.1 Stream Prefetch Distance and Throttling

One common problem for prefetching is to determine a suitable prefetch distance. An ideal prefetcher would bring in the data precisely when the user instruction is ready to be issued. Thus, a waiting user instruction is an accurate signal that the prefetcher is falling behind. It is straightforward to leverage this information within a decoupled-stream microarchitecture, as the user instruction checks the readiness of the FIFO entry before issuing. Since the data is prefetched into the FIFO, allocating a different number of FIFO entries to a stream will effectively change its prefetch distance.

A simple policy would be to split the FIFO evenly for all stream pseudo-registers. This reduces the hardware complexity to manage the FIFO. However, this leads to a low utilization, as FIFO entries for unassigned pseudo registers will be wasted. Also, streams with different memory footprints may hit in different cache levels, and require different prefetch distances to hide the memory latency. A better policy is to dynamically allocate FIFO entries on-demand.

Dynamic Throttling: We implement a stream-aware dynamic throttling policy. Each stream is assigned a FIFO occupancy N. Associated with each FIFO entry is a 1-bit `late` flag, which is set by the issue logic when the stream operand is the last operand to be ready. Each stream is assigned a 3-bit `late_counter`. When releasing a FIFO entry, the `late_counter` is incremented if `late` is set, and decremented otherwise. When the late counter reaches a threshold (currently 7), the stream is considered lagging behind the core and its N is increased (by 2) if N is smaller than a maximum threshold T and the sum of all configured streams' N does not exceed the total FIFO size. Having a maximum size T avoids the pathological case when a stream occupies most of the FIFO. N is initialized to a small value when configuring the stream, which helps capture different behaviors of the same stream during different phases.

Possible Extensions: The compiler could provide a suggested initial value for N when generating stream configurations, by leveraging the information of stream memory footprints, profiled latency, etc. Another opportunity is to use the dependencies between streams to prioritize those with dependent streams, as they are more likely on the critical path. These are left to future work.

7.2 Stream-Aware Cache Bypassing

Caching data with low temporal locality unnecessarily wastes the cache capacity and hurts the performance. It is beneficial to identify and bypass such requests.

Our insight is that streams inherently contains useful information for the cache to make such a bypassing decision, e.g. memory footprint, stream length, reuse distance, etc. Ideally, the cache should bypass a stream when the storage required to achieve temporal reuse is beyond its capacity. Bypassing correct streams brings two major benefits: 1. It avoids polluting the cache with data that will not be reused; and 2. Since a bypassed stream is not cached, the cache can speculate that a request from that stream will miss and immediately forward the request to the next level of cache without waiting for the tag lookup or allocating an MSHR. Tag lookup is still necessary to detect misspeculation, but it is removed from the critical path for the common miss case. Not allocating an MSHR increases memory parallelism by allowing more misses to be handled simultaneously.

Field	Description	Field	Description
`sid`	Stream id	`miss`	# cache misses
`footprint`	Est. mem. footprint	`reuse`	# cache reuses
`request`	# stream requests	`bypass`	Whether to bypass

Table 1: Fields of Stream Table

To better understand how stream information can help cache bypassing, consider the following examples:

Example 1: Repeatedly iterating over two affine streams, where the cache can hold only one stream. Without bypassing, the cache tries to keep both streams, and results in a 0% hit rate. With the footprint of the stream, the cache can reason that the total storage required to cache both streams is beyond its capacity, and thus bypass one stream. The other stream now can be fully cached, which improves the hit rate to 50% and reduces the bandwidth pressure to lower cache levels.

Example 2: Iterating over one large affine stream that can not fit in the LLC. In such a case, there are no benefits to caching it. Bypassing it will increasing the memory parallelism and better saturate memory bandwidth. The benefit of stream-awareness is knowing the footprint at the time of stream configuration.

While useful, stream information is not sufficient to handle all situations. For example, it is impossible to accurately estimate the memory footprint of an indirect stream. Also, there may be some temporal reuse from non-stream requests, and bypassing the cache for such a stream hurts the performance.

To mitigate this, a hybrid policy is used to leverage both the stream information and dynamic statistics. In the cache, a stream is identified by the `stream_config`'s PC and pseudo-registers (`sid`). Some lower bits of the PC are used to distinguish streams with the same pseudo-register from different regions. We augment the cache with a stream table. Table 1 gives a basic description of each field of the stream table. An `sid` field is also added to the tag representing which stream brought in this cache line.

Stream Configuration: After configuration, the stream engine will send a request to the cache, which contains all configured streams' `sid`s and their memory footprints. For affine streams with known length at configuration time, their memory footprint can be estimated by the stream engine from the pattern. If not, it sets the memory footprint to 0, and the cache will exclude this information when making bypassing decisions. The cache fills in the corresponding stream table entry when receiving this request. Requests generated by the stream engine contains the stream's `sid`. The cache looks up the stream table to check if it should bypass.

Non-Bypass Stream Requests: If `bypass` is not set, the request is treated as a normal request. The cache updates the stream's dynamic information by: 1. Incrementing the `access` counter. 2. If missed, incrementing the `miss` counter. When the cache line is brought in from the lower level cache, it sets the `sid` field of the tag to the request stream's `sid` so that reuse information can be tracked. 3. If hit, and the `sid` of the tag is valid, incrementing the `reuse` counter of that stream.

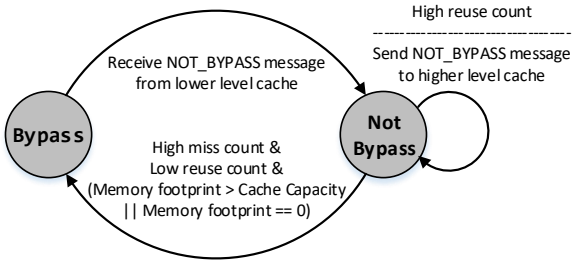


Figure 9: State Transition for Cache Bypassing

Bypass Stream Requests: If the stream is bypassed, i.e. `bypass` is set, the cache will forward the request directly to the lower level cache without waiting for its tag lookup or allocating MSHRs. 1. If missed, the cache forwards the future response from the lower level cache without caching it. 2. If hit, the cache responds normally and drops the future response from the lower level cache.

Bypassing Decision: Figure 9 shows the FSM making bypassing decisions. The cache reconsiders its decision for a stream when its `access` counter hits a threshold. A stream with high `miss` count, a low `reuse` count and a non-zero memory footprint beyond the cache's capacity is marked as bypassed. On the other hand, streams with a high `reuse` count may also have a high reuse rate at the higher level cache. In such a case, the cache will send a `NOT_BYPASS` message to the higher level cache to cancel its bypass decision. The LLC never bypasses any stream. The cache also clears the `access`, `miss` and `reuse` counter after reconsidering the bypassing decision. This is to ensure that the stream table captures the changing dynamic behavior of the stream at run time.

CPU	2.0GHz 8-Way OoO Cores 8-wide fetch/issue/commit 64 IQ, 32 LQ, 32 SQ, 192 ROB 256 Int RF, 256 FP RF speculative scheduling
Function Units	6 Int ALU (1 cycle) 2 Int Mult/Div (3/20 cycles) 4 FP ALU (2 cycles) 2 FP Mult/Div (4/12 cycles) 4 SIMD (1 cycle)
Private L1 ICache	32KB / 8-way 8 MSHRs / 2-cycle latency
Private L1 DCache	32KB / 8-way 8 MSHRs / 2-cycle latency
Private L2 Cache	256KB / 16-way 16 MSHRs / 15-cycle latency
To L3 Bus	16-byte width
Shared L3 Cache	8MB / 8-way 20 MSHRs / 20-cycle latency
DRAM	2 channel / 1600MHz DDR3 12.8 GB/s

Table 2: Simulation Parameters for Baseline

8 METHODOLOGY

Simulation and Compilation: For the simulation, we model an out-of-order processor with a modified version of gem5 [58], extended with support for decoupled-stream ISA extensions and the proposed microarchitecture. As described, we use an LLVM-based compiler to identify streams and transform the program. The simulation is carried out with an approach similar to Aladdin [59, 60] and TDG [61, 62], where compiler transforms are applied to a dynamic dependence graph (DDDG) of LLVM IR operations. We generate wrong-path addresses of streams in the DDDG to ensure fair accounting of unused elements.

Common Parameters: Table 2 summarizes the parameters of the baseline system. We use McPAT [63] for energy estimation, extended to model the stream engine. For the number of pseudo registers, we choose 24, as it is sufficient to cover most of the hot regions in benchmarks we simulated.

Baselines/Configurations: We compare against the following:

Stride Prefetching (Pf-Stride): In this configuration, we add a PC-based stride prefetcher to all three cache levels. The prefetcher takes 1 cycle to generate the prefetch request and it prefetches for 8 requests ahead.

Ideal Helper Thread Prefetcher (Pf-Helper): As discussed earlier, helper-thread approaches [36–43] are a form of aggressive execution-driven prefetching. We evaluate against an ideal SMT-based helper-thread approach, which consumes no core resources (e.g. ROB, RF)[5]. The helper thread is fixed to run k dynamic instructions ahead of the main thread to prefetch the data. We experimentally found $k = 1000$ is sufficient to bring significant speedup for the main thread.

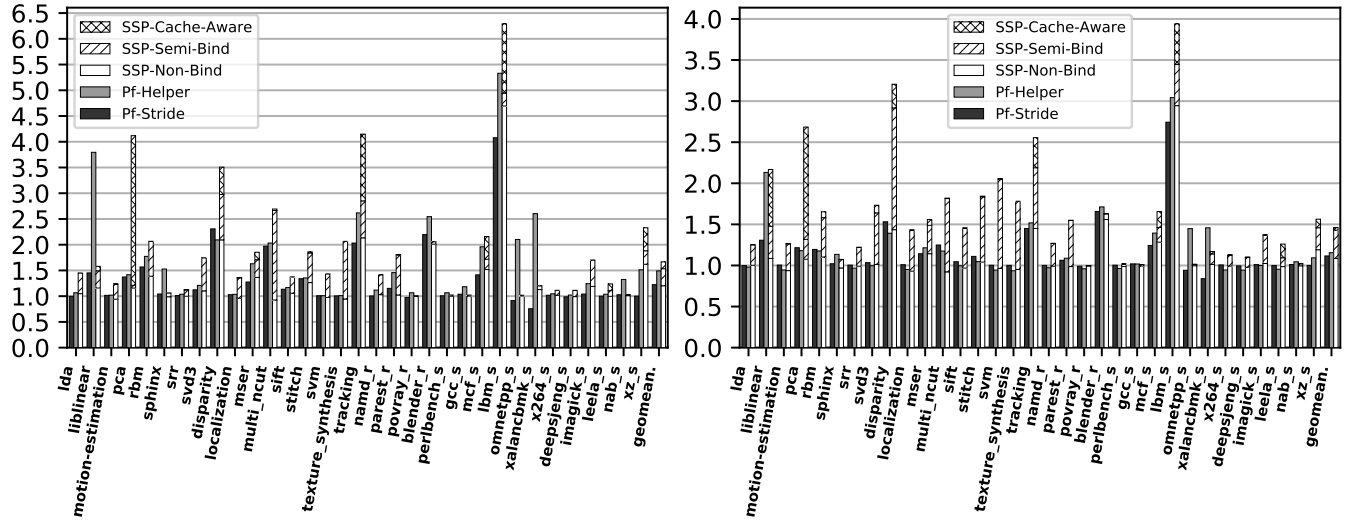
Non-Binding Stream Prefetching (SSP-Non-Bind): This configuration is a limited version of SSP, where the compiler only recognizes the stream and inserts stream instructions, i.e. `stream_cfg`, `stream_step` and `stream_end`. This configuration only uses the stream engine as a prefetcher, and the data fetched is stored in cache. If not specified, we use a 192-entry FIFO for this configuration, with 8 entries per stream. Since most streams will have element size less than or equal to 8 bytes, we set the FIFO entry size to 8 bytes. Note that throttling is not possible in SP as there are no user instructions.

Semi-Binding Stream Prefetching (SSP-Semi-Bind): This configuration is the same as SSP-Non-Bind except that we use the full decoupled-stream ISA, which has the additional benefits of semi-binding streams and address-computation specialization. If not specified, the dynamic throttling policy from Section 7.1 is enabled.

Stream-Aware Cache (SSP-Cache-Aware): This configuration is built upon SSP-Cache-Aware, but with the stream-based cache bypassing policy described in Section 7.2.

We simulate 33 benchmarks from the SPEC CPU 2017 and CortexSuite [1, 2]. We exclude all Fortran benchmarks from SPEC CPU 2017 due to incompatibilities with our current framework. We use the reference input set for SPEC and the largest provided input set for CortexSuite. We use SimPoint [64] to select multiple representative simpoints for simulation from the first 10 billion instructions. Each simpoint contains 10 million dynamic instructions, and on average 10 simpoints are selected for each benchmark. After cache warm-up, we simulate the simpoints and compute the total execution time and energy based on each simpoint's weight.

[5]Properly allocating resources and choosing an instruction slice for a helper thread is the subject of much research, so we abstract here.



(a) Overall Speedup

(b) Energy Efficiency

Figure 10: Overall Speedup and Energy Efficiency

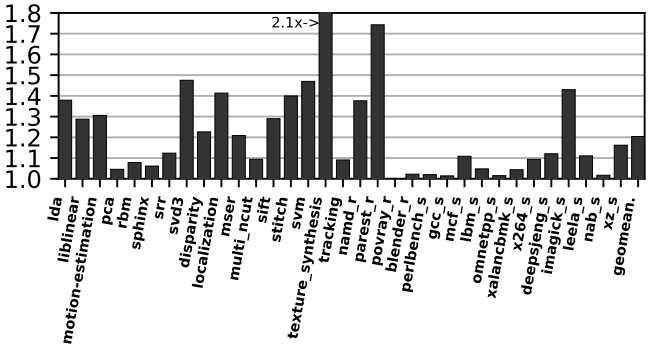


Figure 11: Speedup of SSP-Non-Bind. vs. SSP-Semi-Bind

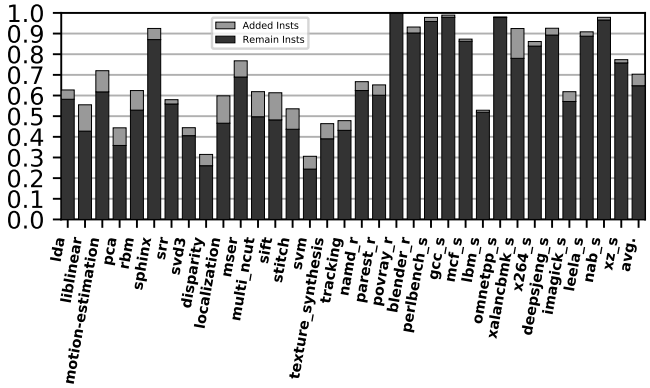


Figure 12: Dynamic Instructions in SSP-Semi-Bind

9 EVALUATION

Our evaluation studies the benefits from the three potential opportunities: stream-prefetching, stream-decoupling, and cache-awareness. We first analyze the overall benefit, then discuss each aspect, and end by discussing the integration with different cores.

Overall Benefit: Figure 10a shows the speedup of all the configurations over the baseline OOO core. Stride prefetching achieves 1.22× speedup, while ideal helper thread yields 1.50× speedup. For SSP, non-binding stream prefetching achieves 1.20× speedup, which is similar to stride prefetching. Semi-binding stream prefetching achieves 1.53× speedup, which outperforms even the ideal helper thread. The main reason is that semi-binding removes the instruction overhead for address computation. It also does not generate duplicate memory requests to L1. Finally, stream-aware cache results in 1.67× speedup over the baseline OOO core.

Figure 10b shows the overall energy efficiency of all the configurations over the baseline OOO core. Stride prefetching improves the energy efficiency by 1.12×, while ideal helper thread achieves 1.16×. Non-binding stream prefetching slightly increases the energy efficiency by 1.09×, while semi-binding stream prefetching gives a significant improvement to 1.47×, as semi-binding removes much instruction overhead. Finally, making the cache stream-aware achieves 1.53× energy efficiency.

Benefits of Semi-Binding Stream Prefetching:

The major benefit of semi-binding stream prefetching versus non-binding stream prefetching comes from a combination of removing address computation and memory access instructions from the pipeline and reducing traffic to the L1 cache. Figure 11 shows the performance of semi-binding stream prefetching, normalized over non-binding stream prefetching. Both configurations use a 192-entry FIFO without throttling. Overall, compared to non-binding prefetching, semi-binding prefetching achieves 1.26× speedup.

Figure 12 shows the number of dynamic instructions committed in semi-binding prefetching, normalized to the original program. On average, semi-binding prefetching removes 35% of the dynamic instructions from the original program, while adding back only 5.6% to control the stream engine. Most of the new instructions added are `stream_step` instructions which advance the stream FIFO – in most cases one per loop iteration. An extreme case is `svm` from CortexSuite. The hot regions of this benchmark involve small matrix multiplication, which has a small memory footprint. The L1 data

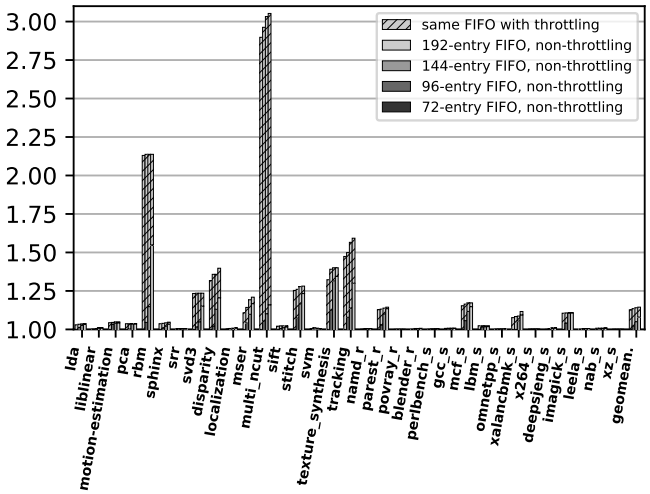


Figure 13: Speedup improvements from dynamic throttling

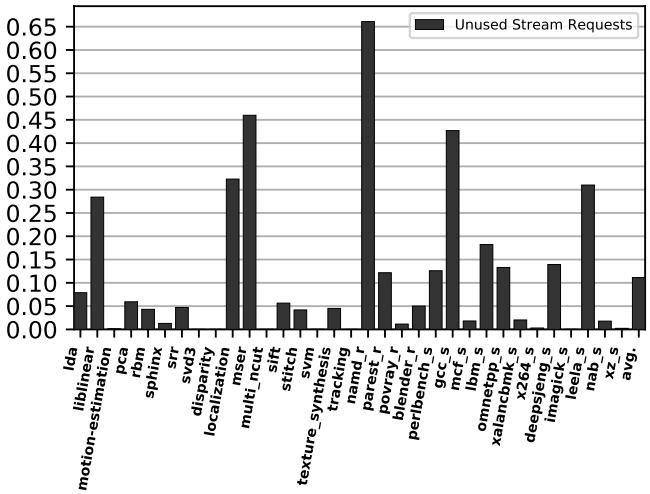


Figure 14: Unused stream requests

cache has less than 1% miss rate, and this explains why neither stride prefetching nor ideal helper threads can improve the performance. When cache is not the bottleneck, semi-binding stream prefetching achieves 1.48× speedup over non-binding stream prefetching for `svm`. A similar analysis also applies to `texture_synthesis`.

Dynamic Throttling: Figure 13 shows the performance of semi-binding stream prefetching with various FIFO sizes and throttling policies, normalized to the configuration with a 72-entry FIFO, non-throttling configuration. Compared with an evenly distributed policy, dynamic throttling improves the performance the most when the FIFO is small, as it achieves a better utilization for the FIFO by allocating more space to streams lagging behind the core. With a 72-entry FIFO, dynamic throttling improves the performance by 13%, while for a larger 192-entry FIFO, it yields a marginal improvement of 5%.

An extreme case is `multi_ncut`, where most of the execution time is spent on a simple loop which iterates through a matrix and generates the sorted indexes. The matrix is too large to be cached in L2, and 68.7% of the memory accesses in this loop goes to L3 cache. Since one stream FIFO entry corresponds to one loop iteration when it is unconditionally stepped, the maximum effective prefetch window measured in dynamic instructions is the dynamic instructions per iteration times the FIFO entries allocated for that stream. As the loop body contains only 9 static instructions, the effective prefetch window achieved by a non-throttling policy is not large enough to fully hide the L3 cache latency.

Unused Stream Requests: Since we are decoupling the stream pattern from the control flow, it is possible that the stream elements are fetched from cache into the stream FIFO but never used by the core. Figure 14 shows the percentage of unused requests issued by the stream engine to the L1 cache. The average unused stream requests is 11.1%. An extreme case is `namd_r`, which has 66% of unused requests. This is partly because some stream elements are unused due to control flow, but also some stream elements are prefetched beyond the termination of the stream (`stream_end`); this is more common for shorter streams. However, these unused stream requests may still be useful as the fetched data may be used by future accesses. It is also possible that the unused stream requests may hit in the L1 cache and do not increase the overall memory traffic. The overhead is mainly extra pressure on the bandwidth between the core and the L1 cache.

Stream-Aware Cache: Figure 15 shows the performance of stream-aware cache, normalized by the performance of binding stream decoupling. Stream-aware cache supports stream-based bypassing (see Section 7.2). Both configurations use a 192-entry FIFO with dynamic throttling.

Stream-aware cache improves the performance from 1.53× to 1.67× (9%), with the highest peak of 3.4× on the `pca` benchmark. For `pca`, the key kernel (based on our simpoints) is computing the correlation matrix, which contains a 3 level nested loop (i, j, k), and the inner most loop accesses two matrix columns $(a[k][i] \times a[k][j])$. The reuse distance is k for the first column and $k \times j$ for the second one. To make things worse, the matrix is accessed in column order, meaning that most data within the cache line goes unused. Without cache awareness, we constantly miss in the L2 cache. Notice that non-binding stream prefetching here can not effectively hide this latency as we are bound by the L1 cache MSHRs, while the stride prefetcher in the L2 cache does not face this constraint. In stream-aware cache, the L2 cache bypasses the second column, which releases enough space to fully cache the first column. This increases the L2 cache hit rate and saves bandwidth on the bus to the L3 cache, and leads to 3.4× speedup over semi-binding stream prefetching and 4.1× over the baseline OOO core.

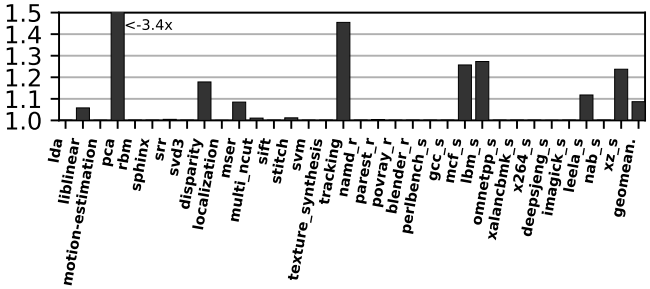


Figure 15: Speedup with Cache Awareness

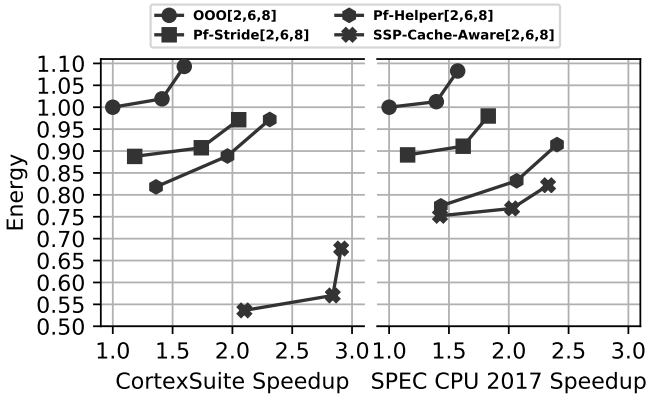


Figure 16: Relative Speedup and Energy Efficiency for Various OOO Processors

Another case is `lbm_s`, whose memory footprint is too large to be cached in L3 and is memory bound. In such a case, the stream-aware cache can forward requests to the L3 cache when all MSHRs of the upper level cache are used. This effectively increases the total number of parallel misses that can be handled by the cache system, and improves the performance of `lbm_s` by 1.3×.

Design Space Interaction: To understand the tradeoffs and interaction with different OOO cores, we simulate several configurations from dual-issue up to 8-issue. Figure 16 shows the relative speedup and energy efficiency of the baseline OOO processor, stride prefetching, ideal helper thread and SSP with stream-aware cache, normalized to a dual-issue OOO core. Compared with traditional prefetching, stream decoupling can greatly improve both the performance and energy efficiency in both SPEC CPU 2017 and CortexSuite. Notably a 6-issue SSP can surpass an 8-issue OOO in both energy-efficiency and performance.

Compared to an ideal helper thread, SSP is much more effective on CortexSuite. This is because most accesses are streams, which can decoupled from the core, and also SSP can intelligently reason about the cache behavior of streams. SSP only sees similar benefits to the ideal helper thread on SPEC CPU, because its advantage of decoupling is offset by its disadvantage in coverage against non-streaming access.

10 CONCLUSION

This work explores the concept of leveraging the inherent structure of streams to specialize the core pipeline, cache interface and cache policies. We find that streams are common and follow simple patterns. Furthermore, they can be decoupled, provided a semi-binding interface that does not require stream data to be consumed. Our proposed decoupled-stream ISA extensions leverage this principle to enable the specification of repeated access patterns in the presence of control flow and indirect memory, opening the door to effective execution-driven prefetching without redundant execution. Our stream-specialized microarchitecture benefits from stream-based prefetching, decoupling of address computation, and stream-awareness in prefetch throttling and cache-bypassing. Broadly, this paradigm of encoding rich memory access semantics could open up new opportunities for specialization of access and communication at even higher levels within the cache and memory hierarchy.

11 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful suggestions and feedback. This work was supported by NSF grants CCF-1751400 and CCF-1823562.

REFERENCES

[1] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 55–64, Oct 2009.

[2] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite.," in *IISWC*, pp. 76–79, 2014.

[3] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *MICRO*, 2004.

[4] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[5] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, pp. 38–51, Sept. 2012.

[6] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 84–95, IEEE Computer Society, 2012.

[7] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 298–310, ACM, 2015.

[8] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm, and A. Shriraman, "Needle: Leveraging program analysis to analyze and extract accelerators from whole programs," pp. 565–576, Feb 2017.

[9] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman, "Chainsaw: Von-neumann accelerators to leverage fused instruction chains," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, Oct 2016.

[10] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, Feb 2014.

[11] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, (Los Alamitos, CA, USA), pp. 112–119, IEEE Computer Society Press, 1982.

[12] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media processing with streams," *IEEE micro*, vol. 21, no. 2, pp. 35–46, 2001.

[13] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (rsvp)," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 141–, IEEE Computer Society, 2003.

[14] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 255–268, ACM, 2014.

[15] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 416–429, ACM, 2017.

[16] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 389–400, IEEE Computer Society, 2008.

[17] G. Weisz and J. C. Hoe, "Coram++: Supporting data-structure-specific memory interfaces for fpga computing," in *25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sept 2015.

[18] C.-H. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 118–130, ACM, 2015.

[19] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 207–220, IEEE, 2018.

[20] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus," ISCA, 2018.

[21] L. Kurian, P. T. Hulina, and L. D. Coraor, "Memory latency effects in decoupled architectures with a single data memory module," in *[1992] Proceedings the 19th*

Annual International Symposium on Computer Architecture, pp. 236–245, May 1992.

[22] L. K. John, V. Reddy, P. T. Hulina, and L. D. Coraor, "Program balance and its impact on high performance risc architectures," in *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pp. 370–379, IEEE, 1995.

[23] N. C. Crago and S. J. Patel, "Outrider: Efficient memory latency tolerance with decoupled strands," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 117–128, ACM, 2011.

[24] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled supply-compute communication management for heterogeneous architectures," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 191–203, Dec 2015.

[25] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *ASPLOS '10*.

[26] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, *et al.*, "The greendroid mobile application processor: An architecture for silicon's dark future," *IEEE Micro*, vol. 31, no. 2, pp. 86–95, 2011.

[27] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 137–151, ACM, 2019.

[28] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 176–186, ACM, 1991.

[29] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 178–190, ACM, 2015.

[30] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 252–263, IEEE Computer Society, 2006.

[31] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," *SIGARCH Comput. Archit. News*, vol. 33, pp. 222–233, May 2005.

[32] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 69–80, ACM, 2009.

[33] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 247–259, ACM, 2013.

[34] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990.

[35] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, (New York, NY, USA), pp. 42–53, ACM, 2000.

[36] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 14–25, IEEE, 2001.

[37] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 37–48, IEEE, 2001.

[38] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 2, pp. 2–13, 2001.

[39] S. Kondguli and M. Huang, "Bootstrapping: Using smt hardware to improve single-thread performance," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 687–700, ACM, 2019.

[40] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings 28th Annual International Symposium on Computer Architecture*, pp. 52–61, June 2001.

[41] J. Lee, C. Jung, D. Lim, and Y. Solihin, "Prefetching with helper threads for loosely coupled multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 9, pp. 1309–1324, 2009.

[42] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and adapting precomputation threads for effcient prefetching," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 85–95, IEEE, 2007.

[43] A. Garg and M. C. Huang, "A performance-correctness explicitly-decoupled architecture," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pp. 306–317, IEEE Computer Society, 2008.

[44] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *ACM SIGARCH Computer Architecture News*, vol. 19, pp. 40–52, ACM, 1991.

[45] N. Kohout, S. Choi, D. Kim, and D. Yeung, "Multi-chain prefetching: effective exploitation of inter-chain memory parallelism for pointer-chasing codes," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 268–279, 2001.

[46] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung, "A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching," *ACM Trans. Comput. Syst.*, vol. 22, pp. 214–280, May 2004.

[47] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, pp. 433–447, April 2008.

[48] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and insertion algorithms for exclusive last-level caches," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 81–92, ACM, 2011.

[49] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive gpu cache bypassing," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, (New York, NY, USA), pp. 25–35, ACM, 2015.

[50] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 293–304, Sep. 2012.

[51] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 381–391, ACM, 2007.

[52] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 222–233, Nov 2008.

[53] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for gpus," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 76–88, Feb 2015.

[54] N. Vedula, A. Shriraman, S. Kumar, and W. N. Sumner, "Nachos: Software-driven hardware-assisted memory disambiguation for accelerators," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 710–723, IEEE, 2018.

[55] R. Huang, A. Garg, and M. Huang, "Software-hardware cooperative memory disambiguation," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pp. 244–253, IEEE, 2006.

[56] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, 1996.

[57] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," in *Software, IEE Proceedings-*, pp. 198–209, IEEE, 2004.

[58] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.

[59] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 97–108, IEEE Press, 2014.

[60] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[61] T. Nowatzki and K. Sankaralingam, "Analyzing behavior specialized acceleration," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 697–711, ACM, 2016.

[62] T. Nowatzki, V. Govindaraju, and K. Sankaralingam, "A graph-based program representation for analyzing hardware specialization approaches," *IEEE Computer Architecture Letters*, vol. 14, pp. 94–98, July 2015.

[63] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO '09*.

[64] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Technique*, pp. 3–14, Sept. 2001.