2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

# Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism

Quan M. Nguyen
*MIT CSAIL*
qmn@csail.mit.edu

Daniel Sanchez
*MIT CSAIL*
sanchez@csail.mit.edu

***Abstract*—Applications with irregular memory accesses and control flow, such as graph algorithms and sparse linear algebra, use high-performance cores very poorly and suffer from dismal IPC. Instruction latencies are so large that even SMT cores running multiple data-parallel threads suffer poor utilization.**

**We find that irregular applications have abundant *pipeline parallelism* that can be used to boost utilization: these applications can be structured as a pipeline of stages decoupled by queues. Queues hide latency very effectively when they allow producer stages to run far ahead of consumers. Prior work has proposed *decoupled architectures*, such as DAE and streaming multicores, that implement queues in hardware to exploit pipeline parallelism. Unfortunately, prior decoupled architectures are ill-suited to irregular applications, as they lack the control mechanisms needed to achieve decoupling, and target decoupling *across cores* but suffer from poor utilization within each core due to load imbalance across stages.**

**We present *Pipette*, a technique that enables cheap pipeline parallelism *within each core*. Pipette decouples threads within the core using architecturally visible queues. Pipette's ISA features control mechanisms that allow effective decoupling under irregular control flow. By time-multiplexing stages on the same core, Pipette avoids load imbalance and achieves high core IPC. Pipette's novel implementation uses the physical register file to implement queues at very low cost, putting otherwise-idle registers to use. Pipette also adds cheap hardware to accelerate common access patterns, enabling fine-grain composition of accelerated accesses and general-purpose computation. As a result, Pipette outperforms data-parallel implementations of several challenging irregular applications by gmean 1.9× (and up to 3.9×).**

## I. Introduction

Irregular workloads such as graph analytics and sparse linear algebra use high-performance cores poorly: these workloads suffer from frequent long-latency memory accesses and hard-to-predict branches that limit instruction-level parallelism and render out-of-order execution mechanisms ineffective. In this paper we focus on non-invasive modifications to existing out-of-order cores to make these challenging workloads run efficiently.

Leveraging multithreaded cores is a common way to improve core utilization. But structuring irregular applications into multiple *data-parallel* threads suffers from three key problems: *(i)* latencies are larger than what can be effectively hidden by a moderately large number of threads per core (e.g., four); *(ii)* operating on disjoint parts of the input increases pressure on the memory hierarchy, limiting performance [14]; and *(iii)* data-parallel implementations suffer from overheads because they need to synchronize through shared memory.

In this work we explore a different and more effective approach to improve utilization in simultaneous multithreading (SMT) cores: exploiting *pipeline parallelism*. A pipeline-parallel program is structured as a series of feed-forward pipeline stages, with each stage executing on a separate thread. Decoupling stages with queues hides latency by allowing producer stages to run far ahead of consumer stages.

Abundant prior work has proposed *decoupled architectures* to exploit pipeline parallelism (Sec. II): decoupled access-execute (DAE) architectures [11, 15, 37, 38, 41, 44], streaming multicores [5, 8, 42], and spatial architectures [31, 32, 34, 50] use queues as latency-insensitive interfaces between cores, threads, or specialized processing elements. Unfortunately, these architectures are ineffective for our use case because they *(i)* suffer from load imbalance, as they decouple stages across separate cores or processing elements, *(ii)* lack control-flow mechanisms, preventing decoupling of irregular applications, and/or *(iii)* fail to target threads within SMT cores, so their implementations miss opportunities to reuse already-existing resources to implement decoupling cheaply.

To address these limitations, we present *Pipette*. Pipette introduces architectural support for pipeline parallelism within the threads of a multithreaded core. Pipette's novel ISA (Sec. III) allows threads to define inter-thread queues. By exploiting pipeline parallelism within a multithreaded core, Pipette hides latencies more effectively than the same number of data-parallel threads. Pipeline parallelism's naturally smaller memory footprint alleviates cache pressure and reduces the need to synchronize through shared memory.

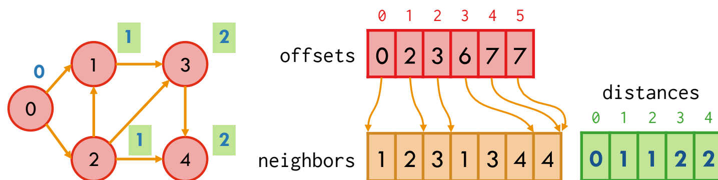
By using SMT to time-multiplex stages in the same core, Pipette avoids load imbalance issues that arise when decoupling stages across separate cores or processing elements. Nevertheless, Pipette allows queues to span multiple cores, avoiding limitations on the number of stages (and thus opportunities for decoupling). Pipette also adds out-of-band control flow to keep producer and consumer loops running despite complex control flow. With these features, Pipette effectively decouples irregular applications, unlike prior work.

In addition to using SMT for load balancing, Pipette's microarchitecture (Sec. IV) features two more novel aspects. First, the implementation reuses core structures: it uses the physical register file to implement queues cheaply, avoiding the storage costs of prior techniques. Second, the implementation exposes a decoupled interface that cleanly accommodates *reference accelerators*, simple hardware units that further accelerate common memory access patterns like indirections.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00056

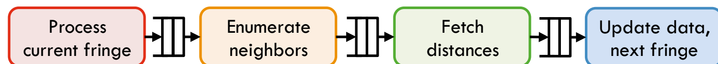
```
define bfs(src):
  distances[src] = 0
  cur_fringe = [src]
  cur_dist = 1
  while not cur_fringe.empty():
    for v in cur_fringe:
      start, end = offsets[v], offsets[v+1]
      for e in range(start, end):
        ngh = neighbors[e]
        dist = distances[ngh]
        if dist is unset:
          distances[ngh] = cur_dist
          next_fringe.push(ngh)
    cur_fringe = []
    swap(cur_fringe, next_fringe)
    cur_dist += 1
```

(a) Pseudocode for serial BFS.




(b) An example graph $G$.

(c) $G$'s CSR representation and, at right, the output (distances) produced by BFS.




(d) A pipeline-parallel implementation of BFS.

Fig. 1: An implementation of breadth-first search (BFS).

Whereas prior work proposed coarse-grain specialized units to access complex data structures [17, 19, 26], Pipette enables composable, fine-grain interleaving of accelerated accesses and general-purpose computation.

We evaluate Pipette on applications from graph analytics, sparse linear algebra, and databases (Sec. VI). Pipette substantially outperforms prior work, by gmean 1.9× and up to 3.9× over SMT with data-parallel threads. Moreover, Pipette is more efficient because it achieves high core utilization.

In summary, we make the following contributions:

- We identify the architectural support needed to efficiently express many irregular applications as a pipeline of decoupled stages.
- We present a novel ISA and control-flow primitives that enable effective decoupling in these applications.
- We present a novel implementation of this ISA that reuses existing core machinery to achieve decoupling and load-balanced execution cheaply, and adds simple, composable specialized units to accelerate common access patterns.
- We demonstrate the effectiveness of this approach on a wide range of applications.

## II. Background and Motivation

In this section we show that pipeline parallelism is common in irregular workloads, and yet prior techniques cannot use it effectively. We show this concretely through a very simple algorithm: breadth-first search (BFS).

**BFS:** Fig. 1(a) shows the pseudocode of serial BFS. Given an input graph, BFS visits all the vertices reachable from a given source vertex `src`, and tags them with the shortest distance to the source, in number of edges.

This algorithm iteratively tags all vertices at a given distance from the source, `cur_dist`, before moving onto the next distance. A *fringe* tracks the set of all vertices at the previous distance (`cur_dist-1`). As BFS visits the neighbors of each vertex in the fringe, it checks whether the neighbor's distance has been set. If not, BFS sets its distance, and adds it to the next iteration's fringe (`next_fringe`). At the end of the current iteration, BFS processes the vertices of the next fringe, until an iteration results in an empty fringe—indicating that all vertices reachable from the source have been visited.

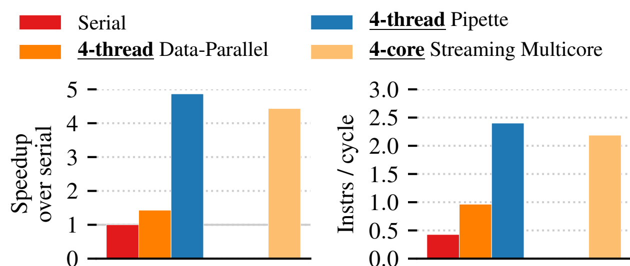



Fig. 2: BFS performance and instructions per cycle on serial, data-parallel, and Pipette versions on a 4-way multithreaded out-of-order core, as well as on a 4-*core* streaming multicore.

The BFS implementation in Fig. 1(a) uses a graph in compressed sparse row (CSR) format, the most commonly used representation [29, 36, 39]. Fig. 1(b) shows an example graph and Fig. 1(c) shows its CSR representation. CSR stores the graph using two arrays, `offsets` and `neighbors`. For each vertex id, the `offsets` array stores where its neighbors begin in the `neighbors` array. Thus, vertex $v$ has edges to `neighbors`[$i$] for i from `offsets`[$v$] to `offsets`[$v$+1]. The `neighbors` array stores the vertex id of each neighbor.

**BFS performance:** Fig. 2 shows the performance of various BFS implementations. The serial and data-parallel implementations are from highly optimized PBFS [21], and we derive the Pipette implementation from serial PBFS. The simulated system uses high-performance cores modeled after Skylake. Each core is 4-thread SMT and features 6-wide out-of-order (OOO) execution with a deep ROB and large load-store queues (see Sec. V for methodology). Fig. 2 shows the speedup of each BFS variant over the serial version, as well as their IPCs.

BFS operates on a large road network graph, so the irregular accesses to graph data miss on the caches and cause poor IPC: just 0.43 instructions/cycle for the serial version, despite using the state-of-the-art 6-wide OOO execution.

SMT alone still achieves poor core utilization: data-parallel BFS running on four SMT threads is only 30% faster than serial, and IPC for the whole core is just 0.96. This happens because four threads are insufficient to hide memory access

latencies. Moreover, threads compete for cache capacity and incur synchronization overheads, further capping speedup.

By contrast, the Pipette BFS version uses pipeline parallelism to hide long-latency accesses effectively. Pipette BFS is 4.9× faster than serial, and its IPC is a healthy 2.4. We now discuss how Pipette achieves this. (Fig. 2 also shows results for a streaming multicore, which we discuss in Sec. II-A.)

**Pipeline parallelism in BFS:** Fig. 1(d) shows how Pipette effectively decouples BFS by partitioning the code into four stages. Stage colors correspond to colored code regions in Fig. 1(a). BFS has three levels of indirection per fringe vertex, to fetch *(i)* the vertex's offsets (line 7), *(ii)* its corresponding neighbors (line 9), and *(iii)* each neighbor's distance (line 10). (Accesses to the fringes may also miss frequently, but they are sequential and trivially handled by a stream prefetcher.) Pipette achieves high performance by *splitting the program across each long-latency indirection*, resulting in four stages.

Achieving this effective decoupling requires a combination of features that prior decoupled architectures do not provide:

- *More than two stages.* Merging any two stages in BFS would add a costly indirection and make it the rate-limiting stage. High performance is achieved only when all frequent long-latency events are decoupled.
- *Independent control flow for each stage.* For example, the update data stage uses conditional execution, and the enumerate neighbors stage has a variable-length loop.
- Related to the above, *accommodating large, fast variations in work across stages.* For example, the amount of work of the last two stages depends on the degree of each vertex, and the work in the update data stage depends on how many neighbors have been visited.

Pipette achieves the first two objectives through its ISA, and achieves the last objective by load-balancing stages within a single core. We now review how prior work in this area lacks some or all of these features.

### A. Prior work on decoupled architectures

We now review the key types of decoupled architectures and discuss why their approach is insufficient for irregular workloads like BFS. In general, prior decoupled architectures suffer from two limitations: *(i)* their queue-based communication and control mechanisms target applications with regular control flow, and impose restrictions on the number of stages or the types of activities within each stage, so they are *insufficient to decouple stages* in irregular programs; and *(ii)* most of this prior work places each stage on a different core, which causes *high load imbalance* in irregular programs. Table I summarizes the differences between Pipette and prior work.

**Decoupled Access-Execute (DAE)** architectures [37] feature two specialized units: an access core that performs memory operations and an execute core that performs compute operations. Both cores are decoupled by queues, allowing the access core to run ahead. Unfortunately, DAE-based architectures [11, 15, 37, 38, 41, 44] suffer from loss of decoupling because they allow only two stages, access and execute, and each stage has a limited set of operations, which causes tight

TABLE I
FEATURE COMPARISON OF RELATED WORK AND PIPETTE.

| Name | Flexible number of stages > 2 | Independent control flow | Dynamic load balancing | Reuses core structures |
|---|---|---|---|---|
| DAE [37], DeSC [15] | ✘ | ✘ | ✘ | ✘ |
| MT-DCAE [41] | ✘ | ✘ | ✔ | ✘ |
| Raw [42], MPPA [8] | ✔ | ✔ | ✘ | ✘ |
| Triggered instructions [32] | ✔ | ✔ | ✘ | ✘ |
| DSWP [35] | ✔ | ✘ | ✘ | ✘ |
| Outrider [5] | ✔ | ✘ | ✔ | ✘ |
| Pipette (this work) | ✔ | ✔ | ✔ | ✔ |

two-way dependences between the access and execute cores. For example, DAE is unable to decouple BFS as described.

**Streaming multicores** like Raw [42] and Kalray's MPPA [8] introduce hardware support for decoupled communication between cores, which can stream values over the network [6, 7, 18, 43]. Unlike DAE, streaming multicores allow more than two pipeline stages and let each core execute arbitrary instructions. However, this cross-core decoupling is inefficient for irregular workloads due to *load imbalance*: since the work per stage varies quickly, cores incur many idle cycles. In fact, these streaming multicores were only used for *regular* pipeline-parallel applications, coded in languages like StreamIt [12] or StreamC [7]. These systems relied on precise knowledge of the execution time and communication requirements of all stages, gathered through static analysis or annotations, to statically map stages to cores [18, 20, 33].

To see the effect of load imbalance, Fig. 2 shows BFS performance on a 4-core streaming multicore. This 4-core pipeline-parallel BFS has similar performance (in fact, 10% slower) than *single-core, 4-thread* Pipette: cores stay idle most of the time, and per-core IPC is a measly 0.55, 4.4× worse than single-core Pipette. Thus, load imbalance is not a second-order issue, but a major roadblock to use streaming multicores well.

Though not the focus of this paper, spatial and specialized architectures such as coarse-grained reconfigurable arrays (CGRAs) [13, 31, 34, 50] and Triggered Instructions [32] also use queues as latency-insensitive channels between processing elements. These systems also cannot load-balance stages across processing elements dynamically.

**Decoupled multithreaded cores** introduce support for queue-based communication among cores. Outrider [5] is the closest work to Pipette. Outrider introduces hardware queues to decouple the threads of a single multithreaded core. In principle, this makes load-balancing across stages easy. However, Outrider was designed for applications with regular control flow, and *lacks the control-flow mechanisms needed to accelerate irregular applications*. Specifically, Outrider uses a global queue for control decisions and requires that all control instructions reside within the first thread to achieve any decoupling. For example, Outrider would not work for BFS, as three out of the four stages have control flow. In addition to this lack of flexibility, Outrider targets simple cores. This makes its implementation quite different from Pipette's: Pipette leverages existing OOO core structures to implement queues, whereas Outrider adds separate queue storage.

In addition to Outrider, Decoupled software pipelining (DSWP) [35] uses a synchronization array to facilitate communication between cores or the threads of a multithreaded core. But DSWP focuses on pipelining a single loop across different threads, which is too limiting for irregular applications. For example, BFS uses a 3-level nested loop, with stages across several loop levels (and because inner loops are short, decoupling within the inner loop is insufficient).

**Indirect prefetchers** seek to hide the latency of accesses common in irregular workloads. IMP [51] prefetches accesses of the form `A[B[i]]`, which is insufficient work for BFS, as it has several indirections. Ainsworth and Jones propose a prefetcher tailored to BFS [1] and a more general event-driven prefetcher [2] that can handle multiple levels of indirection. However, these prefetchers are complex, taking significant energy to infer dependent accesses from memory traffic; they cannot handle all accesses accurately (like fetching the right set of offsets and neighbors in BFS, which requires iteration [2]); they can handle a limited set of access patterns; and they duplicate much of the work done in the cores, hurting efficiency.

**Data structure fetchers** are similar to prefetchers, but feed fetched data to cores to avoid duplicating work. HATS [26] performs graph traversals, Widx [17] accelerates hash indexing, and SQRL [19] handles vector, hash table, and tree traversals. Fetchers avoid the inefficiencies of prefetchers, but are limited to specific data structures and to operations where data structure traversal and computation are not interleaved.

In summary, prior work lacks the ingredients needed to achieve high core utilization on irregular workloads. We now describe how Pipette's implementation reuses existing core structures to implement decoupling cheaply.

## III. Pipette ISA

**Design goals:** Pipette's design is driven by three main goals:

1) Providing inter-thread queues at extremely low overheads, so that threads can communicate very frequently, potentially on almost every instruction. This enables a fine-grain slicing of the program into stages, which is crucial, as we saw in Sec. II. For example, some stages of BFS have as little as one dereference per enqueue and dequeue.
2) Providing control flow primitives that avoid instruction overheads when a serial thread is split into multiple stages. For example, in BFS, stages must synchronize on distance changes. If each stage had to check on every dequeue whether a distance increase was needed, control overheads would negate the benefits of splitting work into stages.
3) Achieving an efficient implementation that reuses existing core structures and accelerates common access patterns.

Pipette's ISA is designed to achieve all these goals. We first discuss how the Pipette ISA achieves extremely low-overhead queues (Sec. III-A), enabling the first design goal. We then present Pipette ISA's control primitives for efficient inter-stage coordination (Sec. III-B), enabling the second goal. Sec. IV presents Pipette's microarchitecture, which efficiently implements the Pipette ISA to achieve all design goals. Table II details the Pipette instruction set.

TABLE II
Pipette instruction set additions. `$q` is a queue id; `%rd`, `%rs`, `%rq` are general-purpose registers used as a destination, source, or queue.

| Mnemonic | Function |
|---|---|
| `map_enq $q, %rq` | Map writes to architectural register `%rq` as enqueues to queue `$q`. |
| `map_deq $q, %rq` | Map reads from architectural register `%rq` as dequeues from queue `$q`. |
| `unmap %rq` | Revert `%rq` to a non-queue register. |
| `peek %rd, %rq` | Peek top element from queue `%rq`, writing `%rd` without dequeueing `%rq`. |
| `enq_ctrl %rq, %rs` | Enqueue a control value (§ III-B). |
| `skip_to_ctrl %rd, %rq` | Skip to the next occurrence of a control value (§ III-B). |

### A. Enqueue and dequeue operations

Pipette provides a fixed number of FIFO queues per core (e.g., 16 in our implementation). To minimize overheads, Pipette does *not* have explicit enqueue or dequeue instructions. Instead, each thread can map the input or output of a queue to a general-purpose register. Each write to a queue input register implicitly enqueues the written value, and each read of a queue output register implicitly performs a dequeue. As we will see in Sec. IV, this register-mapped communication is cheap to implement through register renaming.

It is sometimes useful to read the value at the head of the queue without dequeueing it. To accomplish this, Pipette provides a `peek` instruction, as shown in Table II.

Pipette queues have a maximum size (e.g., 32 values). To avoid full/empty checks, queues have blocking semantics: dequeue or peek operations to an empty queue block until a value is enqueued, and enqueues to a full queue block until free space is available. We later describe how producers and consumers can use *control values* to coordinate without adding instruction overheads in the common case.

Fig. 3 shows why register-mapped, implicit enqueues and dequeues are crucial for performance in the enumerate neighbors stage of BFS. Fig. 3(b) shows assembly code corresponding to the excerpt of C code in Fig. 3(a). If a pipeline-parallel implementation used an `enq` instruction to enqueue values to queues, as done in Fig. 3(c), it would expand the inner loop by one instruction, a 33% increase for this short loop. This instruction would add pressure to the core frontend (to fetch and decode it) and backend (to execute and commit a micro-op that merely copies a value). Instead, Pipette uses implicit communication, implemented through register renaming, to avoid all these overheads: the Pipette code in Fig. 3(d) maps register `t1` so that the load instruction directly enqueues `q1`.

### B. Efficient control flow

Producers often need to communicate control flow changes or exceptional conditions to consumers. Doing this through normal enqueues and dequeues would be inefficient. Instead, Pipette provides *control values* (CVs). Control values are similar to other values passed through queues except that they convey changes to control flow instead of application data. To differentiate them from application data, each CV is enqueued

```
int start = offsets[v];
int end = offsets[v+1];
for (int e = start;
     e < end;
     e++) {
  int ngh = neighbors[e];
  // fetch distances[ngh]
  // if unset:
  //   update distance
  //   add to next fringe
}
```

(a) C code enumerating neighbors of a vertex.

```
  ; vertex v's first neighbor
  a2 = &(neighbors[offsets[v]])
  ; vertex v+1's first neighbor
  a3 = &(neighbors[offsets[v+1]])
  ...
loop:
  ; ngh = neighbors[offsets[v]]
  lw    t1, 0(a2)
  ; fetch distance
  ; set if unset
  addi  a2, a2, 4 ; next neigh. addr
  blt   a2, a3, loop ; more neighs?
```

(b) Serial assembly code.

```
loop:
  lw    t1, 0(a2)
  enq   q1, t1 ; overhead
  addi  a2, a2, 4
  blt   a2, a3, loop
```

(c) Pipeline-parallel assembly code using an enq instruction, which does not exist in Pipette, to manipulate a queue.

```
  ; writes to t1
  ; enqueue q1
  map_enq q1, t1
  ...
loop:
  ; q1 enq ngh
  lw    t1, 0(a2)
  addi  a2, a2, 4
  blt   a2, a3, loop
```

(d) Pipette code tightens the inner loop by making writes to t1 enqueue q1.

Fig. 3: Example showing the importance of tight inner loops: adapting the code from Fig. 3(b) to Fig. 3(c) using explicit enqueue instructions adds an instruction to a tight inner loop. Pipette addresses this in Fig. 3(d) with its implicit queue semantics.

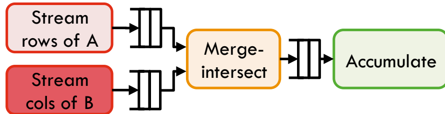



Fig. 4: Sparse matrix-matrix multiplication (SpMM), with one stage receiving two inputs.

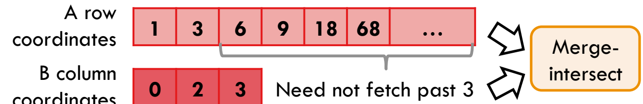



Fig. 5: In SpMM, an inner product in which the input row and column differ greatly in length.

with the `enq_ctrl` instruction, which sets the *control bit* for its queue entry.

CVs let programs avoid checks for infrequent conditions by using semantics similar to those of exceptions. Before starting execution, each thread registers a *dequeue control handler*, similar to an exception handler. A thread dequeueing from or peeking at a queue with a control value at its head instead jumps to the dequeue control handler (this jump happens entirely in user level and does not involve the OS). The dequeue control handler receives the control value and the id of the queue that triggered it. The handler processes the control value, then jumps back to mainline Pipette code to continue computation.

Going back to the BFS example from Fig. 3(d), it's easy to see why control values make execution efficient: the inner loop in Fig. 3(d) does not check for termination or level switches in the inner loop. Instead, stages handle these conditions through control values and dequeue control handlers, leaving the inner loops to deal with data values only.

For a more sophisticated use of control values, consider the inner-product sparse matrix-matrix multiply (SpMM) kernel, shown in Fig. 4. SpMM computes the dot product of a row of `A` and a column of `B` at a time. Both matrices are sparse, so the leftmost stages (stream rows/cols) stream the non-zero *coordinates* of a row and a column at a time. Then, the merge-intersect stage finds the matching non-zeros, which the accumulate stage fetches and accumulates.

Control values make SpMM efficient by letting the stream rows/cols stages *delineate each row and column*. For example, the stream rows stage enqueues all non-zeros for a row of A, followed by a control value denoting the index of the next row, and then the non-zeros of the next row. The merge-intersect code need not check for row or column termination, and the stream rows stage can fetch multiple rows ahead, which is useful as rows often have few non-zeros.

**Consumer-producer coordination:** So far, we have seen how producers can delimit data values with control values to communicate with consumers. The dual case is also desirable: consumers may need to communicate with producers. For example, a consumer may discover that the work a producer is enqueueing is no longer useful, and should alter the producer's control flow to reduce unnecessary work.

To achieve this, the `skip_to_ctrl` instruction finds and dequeues the next control value in a queue, discarding all earlier data values. If the queue does not have a control value, `skip_to_ctrl` blocks waiting for one, and the next time the producer attempts an enqueue, it jumps to an *enqueue control handler* instead. This lets the producer redirect control flow and enqueue a control value that unblocks the consumer.

SpMM shows why `skip_to_ctrl` and enqueue control handlers are useful. Fig. 5 shows an inner product where $A$'s row is much longer than $B$'s column, and the last coordinate in $B$'s column is seen very early in $A$'s row, so no more matched coordinates are possible. It would be wasteful for stream rows to stream the full row of $A$, but only merge-intersect can detect this condition. To address this, when merge-intersect sees the end of $B$'s column (its dequeue control handler fires), it performs `skip_to_ctrl` on the rows queue to skip to the next row. If stream rows is still working on the same row, the queue has no control value, so on the next enqueue, the enqueue control handler of stream rows fires and moves to the next row. If stream rows is already on a later row, then `skip_to_ctrl` lets merge-intersect discard the current row without undue interruptions to stream rows.

In summary, control values and enqueue/dequeue control handlers enable producers and consumers to coordinate out-of-band, in a way similar to *user-level* exceptions. This avoids frequent checks on inner loops that would add significant overheads to the pipelined version. Beyond the two instructions required to enqueue and dequeue control values, this mechanism requires two control registers per thread to store the PCs of enqueue and dequeue control handlers.

### C. Integrating Pipette into the system

**Code transformations to use Pipette:** We use a simple, systematic procedure to split applications into Pipette stages: we split programs along every long-latency indirect load, starting

at the innermost loop and moving outwards. BFS in Sec. II demonstrated this procedure.

We currently transform applications manually, similar to prior work in this area [5, 32]. We transform source code rather than assembly, by using a simple C/C++ API that encapsulates Pipette functionality (e.g., abstracting the mapping and use of queue registers). While some prior work targets compiler transformations [15, 35], we find that irregular applications are harder to transform automatically due to complex indirections that may be impacted by aliasing and races.

For example, one such race condition arises in the last stage of BFS, update data. To decouple this stage from the previous one, the Pipette BFS implementation fetches distances in advance. However, this distance may be stale, as the neighbor may have been recently reached from another fringe vertex. It would be incorrect to use the distance as-is. Our manually transformed code uses this distance for an initial check, but if unset, it re-fetches the distance to ensure it was not set in the interim (this second access is cheap, as it hits in the L1). It would be hard for a compiler to infer when such races are possible, since aliasing information is conservative and whether races are possible depends on application semantics.

Though general-purpose compiler transformations may be unattainable, we nonetheless believe programmers need not directly write Pipette programs. Instead, the compilers of domain-specific languages such as TACO [16] (sparse linear algebra) or GraphIt [52] (graph analytics) could easily generate Pipette code, as they have full information about data structures, aliasing, and high-level semantics.

Pipette is orthogonal to the memory consistency model, and programs behave like normal multithreaded programs.

Finally, if a Pipette application is incorrectly synchronized, it may deadlock. Deadlocks leave user-level threads blocked, but the OS can use interrupts to break those deadlocks (like e.g., a blocked `monitor/mwait` instruction).

**Architectural state and context switches:** Pipette queues are architectural state, and must be drained and saved across context switches. As is done for FPU state, OSs need not save and restore this state on every system call or interrupt, only when the process is descheduled. As these context switches occur infrequently, saving queue contents represents a negligible fraction of the time spent in OS code.

Draining and refilling queues can be done with normal Pipette instructions. In addition to the OS, debuggers could inspect queues by draining and refilling them.

**Privileged code and virtualization:** Since threads are an OS abstraction, and threads from multiple processes may share the same core, some of Pipette's operations must be privileged. Specifically, the map and unmap operations and the registration of control handlers must happen though system calls. Similar to virtual memory, the OS can provide each process with a set of virtual queues, which it can then map to physical queue ids within each core. This allows descheduling and rescheduling individual threads in any order. Since each queue is shared between a producer and a consumer thread, the last of the two threads to be descheduled saves the queue's state. Threads can migrate across cores (using cross-core queues, Sec. IV-C). A producer/consumer thread can enqueue/dequeue to a queue while the other thread is descheduled; however, in practice it will quickly block on a full/empty queue. Thus, the OS should co-schedule the threads of a Pipette program, e.g., using gang scheduling [54].

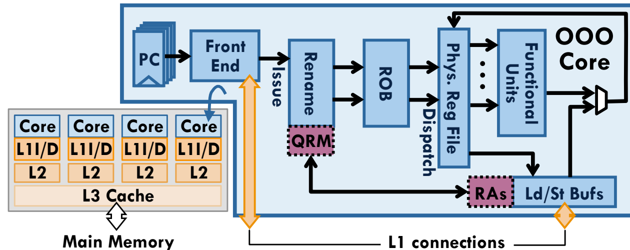


Fig. 6: Pipette implementation overview and modified out-of-order pipeline. Modifications (QRM and RAs) are shown in purple with dotted borders.

OS-mediated queue mappings prevent accessing queues from other processes. Side channels are possible just like in normal SMT cores; to avoid them, Pipette threads should not be co-scheduled with other processes on the same core.

## IV. Pipette Microarchitecture

Fig. 6 gives an overview of Pipette's implementation, focusing on its two distinguishing features. First, our Pipette implementation *uses the physical register file to implement queues* (Sec. IV-A). We observe that physical registers are underutilized in irregular applications, where deep out-of-order execution is not efficient. This enables a cheap implementation that leverages existing OOO structures: physical registers and register renaming. Second, we introduce *reference accelerators* to speed up common access patterns (Sec. IV-B). We also introduce *connectors* to enable cross-core queues (Sec. IV-C), and evaluate Pipette's implementation costs (Sec. IV-D).

### A. Register-based inter-thread queues

Pipette maintains queues within the *physical register file*, and adds minor changes to register renaming to implement FIFO queue semantics. Pipette prevents queues from starving threads of physical registers by sizing each queue and limiting the space all queues may collectively occupy. Since queues are embedded within speculatively managed structures, we first explain the basic Pipette bookkeeping structure, then how it interacts with speculative execution.

**Basic operation:** Fig. 7 shows the Queue Register Map (QRM), the structure that tracks the state of all queues. The QRM has as many entries as the maximum capacity of all queues. Each queue takes a contiguous chunk of entries (shown in different colors in the figure), and manages it as a circular buffer. The chunk associated with each queue determines its capacity. This mapping is configurable by the OS, but cannot change while queues are active.

Fig. 7 also shows how each queue is managed. Each queue has both *speculative* and *committed* pointers for *head* and *tail*. Enqueues happen to the tail of the queue, and dequeues happen from the head. We restrict each queue to be point-to-point, so there is a single enqueuer and dequeuer thread.

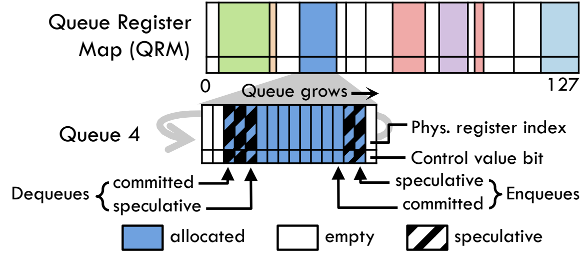



Fig. 7: The Queue Register Map (QRM) tracks the physical registers of each Pipette queue. Each queue is managed as a circular buffer.

Each entry between the head and tail pointers tracks the physical register index that holds the enqueued value. Moreover, each entry has a *control value bit* that denotes whether the entry holds a control value (enqueued with `enq_ctrl`).

The QRM is designed to require simple changes to register renaming. Enqueue operations are nearly identical to normal register writes. On issue, the rename stage acquires a free register index from the freelist, and uses it to store the enqueued value. As usual, the reorder buffer (ROB) stores the *previous* physical register index. The only difference is that, on commit, the ROB *does not free* the previous physical register index. Instead, the QRM manages it, as it is part of a queue.

Dequeue operations are also very similar to reads. For each dequeue-mapped queue, the thread's register map simply holds the index for the *head* of the queue. A dequeue simply uses this value, and additionally modifies the register map to point to the next register in the queue, supplied by the QRM. On commit, the QRM returns the register to the freelist. Finally, peek operations are exactly like normal reads.

**Speculative value management:** Because registers are written and read speculatively, there are multiple value management options. We choose the simplest one: *enqueued values cannot be dequeued until they are non-speculative.*

This leads to a simple implementation: QRM's *speculative* head and tail pointers are the only eagerly managed values. Each enqueue advances the speculative tail pointer on issue, and the committed tail pointer on commit; similarly, each dequeue advances the speculative head pointer on issue, and the committed head pointer on commit. The queue is considered *empty* if the speculative head is about to catch up with the committed tail, and *full* if the speculative tail is about to catch up with the committed head. The issue stage stalls the thread on enqueues to full queues or dequeues from empty queues.

Recovery from misspeculation simply requires rolling back the speculative head and tail pointers, as well as releasing the registers from rolled-back enqueues to the freelist.

A key benefit of consuming only committed values is that *misspeculation in a producer thread does not propagate to the consumer thread*. This allows us to implement Pipette with simple changes to the issue and rename stages.

We also tried a more complex variant of Pipette that allowed dequeues to consume still-speculative enqueued values. This version barely improved performance (by about 1% on average), and occasionally caused minor performance degradations. Intuitively, this result makes sense because the point of Pipette is to keep threads decoupled, so while allowing dequeues to dip into the speculative region of the queue may get some out-of-order benefits, in well-decoupled programs producers should already run far ahead of consumers.

**Issue logic modifications:** Pipette requires minor changes to the issue logic. First, the per-thread issue logic stalls on a dequeue from an empty queue or an enqueue from a full queue. Second, every dequeue of a control value triggers a jump to the dequeue control handler. For simplicity, we reuse the exception logic to implement this redirection.

Our current Pipette implementation does not change the thread prioritization logic. We use the standard ICOUNT policy [46] to avoid issue queue clog. Further gains might be achieved by controlling thread priorities to increase decoupling, e.g., by prioritizing producers over consumers. However, we find Pipette works well with ICOUNT, and leave exploration of more advanced issue policies to future work.

### *B. Accelerating common access patterns*

By exposing a queue-based interface, Pipette makes it easy to add specialized units to accelerate long-latency memory accesses. Pipette achieves this with *reference accelerators* (RAs), simple configurable units that perform indirect loads and communicate with threads through queues.

**Benefits:** BFS (in Fig. 1(d)) showcases the two key benefits of RAs. First, some stages, like fetch distances in BFS, are very simple, performing indirect or strided accesses. Using a thread for such simple work is overkill: a simple RA can perform them much more efficiently. Second, to get decoupling, Pipette divides the code across each long-latency indirection. While this lets producers run ahead of consumers, *producers still suffer from long-latency loads*. For example, the process current fringe stage in BFS issues loads to the `offsets` array. These loads take a long time to commit and stress the ROB, limiting MLP. RAs allow offloading these *producer* long-latency accesses. This results in producers with short, tight loops that do not stress OOO resources, improving performance. For example, in our RA-enhanced BFS, the process current fringe stage passes `v` to an RA, which fetches `offsets[v]` and `offsets[v+1]` autonomously and non-speculatively, producing `start` and `end`.

**Interface:** Each RA is a configurable unit with a single input and output queue. The RA takes in a stream of input elements, uses them to perform indirect accesses, and places the resulting data in its output queue. RA accesses are independent from those of general-purpose threads. Consistency-wise, programs simply see each RA as a separate thread.

RAs are configured once, by specifying which queues to use, a starting address `A`, an element size `S`, and the access *mode* with can be *indirect* or *scan*. The RA interprets `A` as an array with elements of size `S`. In *indirect* mode, the RA takes a stream of indices at its input, and for each index `i`, it fetches `A[i]`. In *scan* mode, the RA takes a stream of starting and ending indices at its input, and for each pair of input values {`start, end`}, it fetches elements `A[start:end-1]`.

We find that these simple modes cover most indirection patterns and benefit all our applications. For instance, in BFS,

the indirect mode covers the first and third stages, and the scan mode covers the second stage.

**Implementation:** RAs use existing core machinery. RAs opportunistically use spare rename and register bandwidth, and manipulate the QRM like ordinary threads on enqueues and dequeues. When performing memory accesses, RAs use the load/store unit and use virtual addresses. Each RA has a small completion buffer to track outstanding loads (Sec. IV-D presents implementation costs). On a virtual memory exception, the core interrupts the producer thread associated with the RA.

### C. Extending Pipette to cross-core queues

We have so far described how queues work within a core, but allowing queues to span threads in multiple cores is desirable for three reasons. First, although Pipette's main goal is to improve core utilization, achieving effective decoupling may require more stages than a core has threads. Second, as we will see later (Sec. VI-F), Pipette can scale and balance work across cores in new ways, by using inter-core queues to improve locality and avoid shared-memory synchronization costs. Third, as Sec. III-C discussed, it is desirable to let the OS schedule threads individually, in separate cores if needed.

We achieve this through *connectors*, simple hardware structures that stream a queue from a producer to a consumer core. Producer and consumer threads are both given intra-core queues. The connector is a simple FSM that sits on the producer's core. It has a similar but simpler implementation than RAs: rather than interacting with the load/store unit, it just sends values from the producer to the consumer core, using credit-based flow control to avoid saturating the on-chip network and strictly limits the receiver queue's state to its capacity. When descheduling a consumer thread, the OS must wait for its connectors to quiesce; this requires a simple teardown protocol on top of credit-based flow control.

### D. Implementation costs

Pipette's storage and logic additions impose minimal overheads.

TABLE III
PIPETTE STORAGE REQUIREMENTS.

| Component | Size (bits) |
|---|---|
| 148 9-bit QRM entries | 1,332 |
| 64 8-bit QRM pointers | 512 |
| 8 64-bit handler PCs | 512 |
| **Pipette total** | **2,356** |
| 212-entry Int. Phy. Reg. File (for comparison) | 13,568 |

Table III summarizes Pipette's storage requirements. In our configuration, Pipette can map up to 148 physical registers, and takes 1844 bits (231 bytes). This is only 14% of the physical register file, showing the benefits of leveraging physical registers to implement queue storage. Beyond the QRM, 512 bits (64 bytes) are required for the per-thread enqueue and dequeue control handler PCs. Overall, only 2356 bits (295 bytes) of additional storage are needed to implement Pipette, a small overhead for a modern core.

RAs, the other hardware addition, are small. We write complete RTL for RAs, including configuration registers, address generation, and a 32-entry completion buffer. We synthesize RAs using yosys [49] and the 45 nm FreePDK45 library [28]. Four RAs take 0.0014 $mm^2$ at 45 nm, adding an estimated 0.007 % to core area, a tiny overhead.

TABLE IV
CONFIGURATION PARAMETERS OF THE EVALUATED SYSTEM.

| | |
|---|---|
| **Cores** | 1 or 4 cores, 3.5 GHz, x86-64 ISA, Skylake-like: 6-wide out-of-order issue, 224-entry ROB, 97-entry issue window, 72-entry load buffer, 56-entry store buffer, 212 integer physical registers, 168 vector physical registers; 4-thread SMT with ICOUNT issue policy and dynamically shared ROB, issue window, PRF, and LSQs |
| **Pipette** | QRM with 148 physical register entries, 16 queues max; 4 RAs; 4 connectors; queues are sized 24 elements deep by default |
| **L1 cache** | 32 KB/core, 8-way set-associative, 4 cycle latency |
| **L2 cache** | 256 KB/core, 8-way set-associative, 12 cycle latency |
| **L3 cache** | 2 MB/core, 16-way set-associative, 40 cycle latency |
| **Main mem** | 120-cycle minimum latency, 2 controllers, 25 GB/s each |

## V. EXPERIMENTAL METHODOLOGY

### A. Simulated System

We implement Pipette on a detailed event-driven, cycle-level simulator based on Pin [23]. Table IV lists the parameters of our simulated system, whose cores are modeled after Intel's Skylake [9], scaled to 4 SMT threads from the usual two. Core structures are sized as in Skylake; we grow the physical register file (PRF) from 180 to 212 entries to accommodate the 32 architectural registers of the two extra threads. Thus, the PRF entries left for renaming are the same as in Skylake. We extend cores to faithfully simulate Pipette additions, with the configuration shown in Table IV. We use McPAT [22] to model core and uncore energy at 22 nm, and Micron DDR3L datasheets [25] to model main memory energy.

We evaluate 1- and 4-core systems. Since Pipette's main goal is to improve core utilization, we first compare Pipette and data-parallel programs on a single 4-thread SMT core. Then, we compare 1-core, 4-thread Pipette with a baseline decoupled architecture: a 4-core streaming multicore. Finally, we show that Pipette also scales across cores.

### B. Benchmarks

We evaluate Pipette on six applications from graph analytics, sparse linear algebra, and databases. For each application, we start from a state-of-the-art implementation that includes serial and data-parallel versions. We derive the Pipette version of each benchmark from the serial version. The wide variety of Pipette applications highlights its generality and the abundance of pipeline parallelism.

**Breadth-first search (BFS)**, first described in Sec. II, determines the distance of graph vertices to a source vertex. We base our implementation on PBFS [21].

**Connected components (CC)**, **PageRank-Delta (PRD)**, and **Radii estimation (Radii)** are graph algorithms from the Ligra framework [36]. CC uses multiple invocations of BFS to discover graph connectivity. PRD is a PageRank variant that only visits vertices whose PageRank value changes by more than a certain amount. Radii launches several breadth-first searches from random points in the graph to estimate the radii of its vertices. These algorithms process only a subset of graph vertices in each iteration, so their memory access patterns are

TABLE V
INPUT GRAPHS, SORTED BY NUMBER OF EDGES.

| Domain | Graph | Vertices | Edges |
|---|---|---|---|
| Human collaboration | coAuthorsDBLP-symmetric | 299K | 1.9M |
| Dynamic simulation | hugetrace-00000 | 4.6M | 14M |
| Circuit simulation | Freescale1 | 3.4M | 19M |
| Internet graph | as-Skitter | 1.7M | 22M |
| Road network | USA-road-d-USA | 24M | 58M |

TABLE VI
INPUT MATRICES, SORTED BY AVERAGE NON-ZERO ELEMENTS PER ROW.

| Domain | Matrix | Size ($n \times n$) | Avg. nnz/row |
|---|---|---|---|
| Graph as matrix | amazon0312 | 400,727 | 8.0 |
| Collaboration | ca-CondMat | 23,133 | 8.1 |
| Gel electrophoresis | cage12 | 130,228 | 15.6 |
| Electromagnetics | 2cubes_sphere | 101,492 | 16.2 |
| Fluid dynamics | rma10 | 46,835 | 49.7 |
| Structural | pwtk | 217,918 | 52.9 |

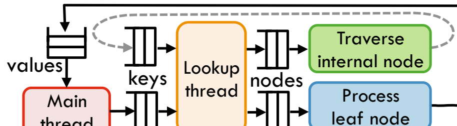


Fig. 8: Silo, with bounded feedback loops.

very irregular. The pipelines for these algorithms resemble the pipeline for BFS in Fig. 1(d).

**Sparse matrix-matrix multiplication (SpMM)**, introduced in Sec. III, is a key component of sparse linear algebra, and its merge-intersection parallels similar operations on databases.

**Silo** [45] is an in-memory database. Silo is dominated by lookups to B+tree indexes. Our Pipette implementation, shown in Fig. 8, pipelines multiple tree traversals. The lookup thread performs tree lookups level by level, requeueing the key and lookup status in its input queue if the lookup needs to go to the next level by traversing internal nodes. This recursive process manifests as a cycle in the pipeline diagram (dashed gray line).

Silo shows that Pipette programs can feature cycles in their application graphs. As in dataflow systems, we avoid application-level deadlock as long as cycles are bounded—in this case, each lookup thread re-enqueues *at most* one element for each element it processes.

**Input sets:** Graph applications use five large, real-world graphs that include road networks, Web connectivity graphs, and academic collaboration graphs, listed in Table V. SpMM uses six diverse sparse matrices, listed in Table VI. Silo uses the YCSB-C workload [4] on a 52 GB dataset.

On some of PRD, Radii, and SpMM's largest inputs, we use *iteration sampling* to keep simulation times reasonable: we simulate only a subset of iterations, uniformly distributed. Even with sampling, simulated periods are long (*e.g.*, ~3 billion cycles per phase of PRD), so no warmup is needed. For all other benchmarks, we simulate the full algorithm.

**Reference accelerators:** We build Pipette benchmark variants with and without RAs. We apply RAs systematically, offloading every producer load to an RA as described in Sec. IV-B. All workloads benefit from RAs. We report results with RAs on by default; Sec. VI-E studies the impact of RAs.

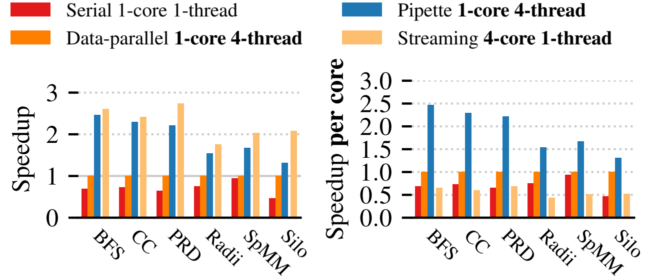


Fig. 9: Performance of Pipette implementations on a single 4-thread core and a *4-core* streaming multicore, as well as *performance per core*.

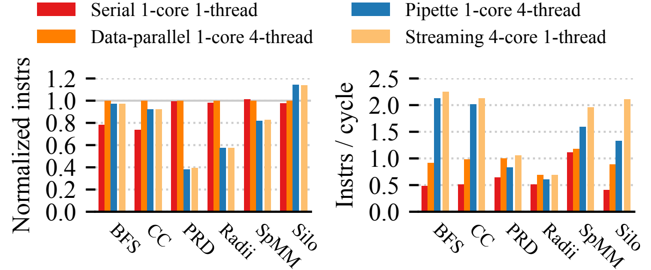


Fig. 10: Instructions executed (lower is better) and average IPC (higher is better), averaged across program inputs, for 1-core data-parallel and Pipette as well as 4-core Streaming.

## VI. EVALUATION

We first analyze Pipette's single-core performance and compare Pipette and data-parallel implementations. Then, we compare Pipette to a 4-core streaming multicore to show the utilization and efficiency benefits of time-multiplexing stages among threads. We then report microarchitectural efficiency metrics, study sensitivity to configuration parameters, and conclude by studying Pipette on multiple multithreaded cores.

### A. Pipette vs. data-parallel implementations

Fig. 9 (left) summarizes the performance of the serial, data-parallel, and Pipette versions of all benchmarks. These versions use a single 4-thread SMT core. Performance is reported as speedup *over the data-parallel version* (not serial), averaged (gmean) across inputs. Fig. 9 shows that Pipette substantially outperforms the data-parallel versions, by 1.9× gmean across applications; by up to 2.5× for an application (BFS); and by up to 3.9× on particular inputs (Fig. 13).

### B. Pipette vs. cross-core decoupling

Beyond data-parallel implementations, we compare Pipette against a baseline decoupled architecture, a streaming multicore. Pipette's key benefit over prior decoupled architectures is its ability to time-multiplex stages across the same core to achieve load balance and high utilization. To focus on evaluating this effect, we model the streaming multicore simply as multi-core, single-thread Pipette: the streaming multicore uses 4 single-threaded cores, and benefits from Pipette's ISA and features. *This includes reference accelerators*, even though they are our contribution.

Fig. 9 compares the performance of Pipette on a *single multithreaded core* to the *single-threaded 4-core* streaming

multicore. To measure how effectively Pipette uses its core resources, we also show the performance *per core*.

Fig. 9 shows that, while the streaming multicore outperforms Pipette, it does so by relatively small margins given that it uses *four times the cores*: BFS, CC, Radii, and SpMM perform similarly, and Streaming is 24% faster on PRD and 59% faster on Silo. Overall, Streaming is only 22% faster than Pipette.

This happens because *load imbalance hampers utilization of decoupled cores in irregular applications*: the right plot in Fig. 9, which normalizes by the number of cores, shows that each core in the Streaming baseline contributes similar performance as Serial. This is because, unlike in regular applications where stages proceed at matched rates, irregular applications have highly variable utilization across stages.

### C. Pipette is resource-efficient

To further understand these results, Fig. 10 compares the instructions executed by each benchmark version, relative to those of the data-parallel implementation (left graph, lower is better) as well as instructions per cycle (IPC, right graph, higher is better). Each group of bars shows results for a single benchmark, averaged across all inputs.

These figures reveal that Pipette consistently uses cores efficiently, but the reasons are application-dependent:

- In BFS and CC, Pipette's improvement mainly comes from its dramatic gain in IPC. Pipette executes nearly the same instructions as the sequential code, whereas the data-parallel versions incur some synchronization overheads.
- In PRD and Radii, Pipette's benefits mainly come from reducing the number of instructions. Instruction overheads in these benchmarks stem from synchronization overheads. (These benchmarks come from Ligra, and unfortunately, the serial Ligra version also carries these overheads.) Pipette avoids this and reduces instruction count by up to 3.2×. Thus, while Pipette's IPCs are slightly lower, each instruction does more work.
- In SpMM, Pipette's benefits stem from both increasing IPC and reducing the number of executed instructions.
- Pipette improves Silo by increasing IPC, which is slightly attenuated by a modest increase in executed instructions.

Fig. 11 gives more insight into the factors contributing to IPC by showing a breakdown of cycles spent by cores, derived using the CPI stack methodology [10]. Each group of bars reports breakdowns of each variant across benchmarks (averaged across inputs), relative to the data-parallel baseline. Each bar within a group reports cycles for one technique, broken down in cycles spent *(i)* issuing micro-ops, and waiting on *(ii)* backend stalls (including memory latency), *(iii)* full or empty queues (for Pipette and Streaming), or *(iv)* other stalls (e.g., frontend).

Fig. 11 shows that the serial and data-parallel versions are limited by backend stalls, which are caused by long memory accesses. Meanwhile, the streaming multicore is limited by queue stalls, i.e., load imbalance. By contrast, Pipette incurs few stalls: proper decoupling dramatically reduces backend stalls, and time-multiplexing stages in the same core keeps queue stalls low.

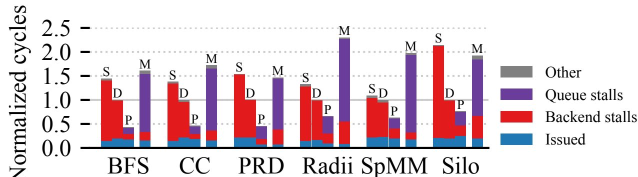


Fig. 11: Breakdown of cycles spent executing each application, normalized to data-parallel and averaged across inputs. (S: Serial, D: Data-parallel, P: Pipette, M: streaming Multicore)

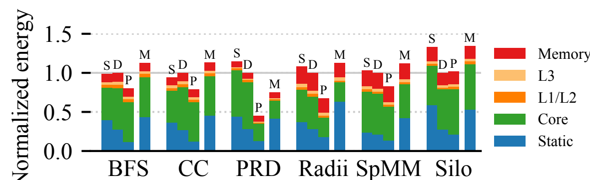


Fig. 12: Breakdown of energy consumed by each application, normalized to data-parallel and averaged across inputs. (S: Serial, D: Data-parallel, P: Pipette, M: streaming Multicore).

Finally, Fig. 12 shows the breakdown of energy consumption of each variant across benchmarks (averaged across inputs), relative to the data-parallel baseline. Pipette is the most efficient variant for BFS, CC, PRD, Radii, and SpMM, reducing energy by up to 2.2× (PRD). Pipette's savings mainly come from reducing dynamic core energy (fewer instructions) and reducing static energy (fewer cycles, as performance is higher). Fig. 12 shows that the streaming multicore is not efficient overall, as it suffers from high static energy due to poor core utilization.

### D. Per-input results

Fig. 13 reports the performance of all variants across each input. Per-input results reveal some interesting behaviors.

**BFS:** Pipette widely outperforms the serial and data-parallel BFS versions, as shown in Fig. 13(a). Pipette outperforms the data-parallel BFS by gmean 2.5× and by up to 3.9×.

Speedups mainly depend on two factors: graph size and average degree. Pipette yields more benefits in larger graphs, where misses are more frequent; and Pipette is more efficient at enumerating small sets of edges than conventional code, where hard-to-predict control flow is inefficient. Thus, Pipette achieves the best speedups in low-degree graphs (`Dy` and `Rd`).

**CC, PRD, and Radii** (Fig. 13(b-d)) show similar trends to BFS: Pipette consistently outperforms the data-parallel versions, of gmean speedups of 2.3×, 2.2×, and 1.5×.

Unlike BFS, these algorithms operate on a fraction of the graph that changes slowly over iterations, so they get better reuse. However, synchronization is more complex, so the data-parallel versions suffer from costly overheads that add substantial extra work. Thus, Pipette's low instruction counts contribute substantially to speedups, as explained above.

**SpMM** (Fig. 13(e)) shows more mixed performance results: Pipette outperforms data-parallel SpMM by up to 2.1×, but it is slightly slower than data-parallel SpMM on one input.

The slight slowdown on the `Co` input results from a combination of two factors. First, `Co` has only 8 non-zeros per row (Table VI), so control values are common and the merge-intersect stage spends a significant fraction of time in the dequeue control handler. Second, `Co` is a small matrix that

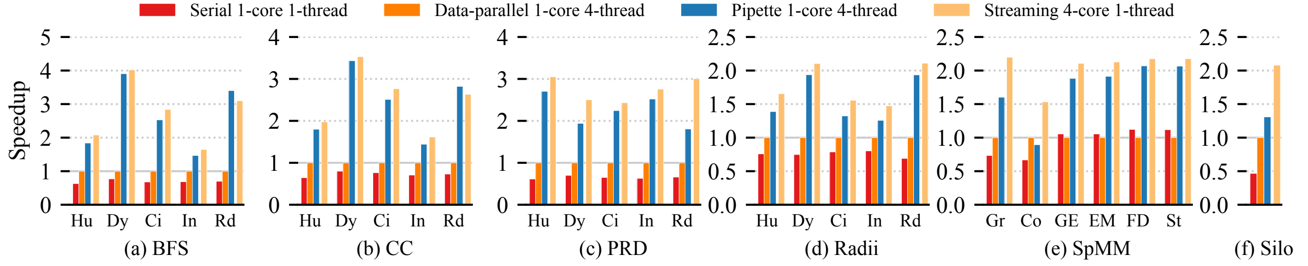


Fig. 13: Per-input performance of all evaluated applications.

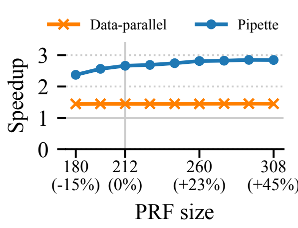


Fig. 14: Performance of 4-thread Pipette and data-parallel over PRF sizes. Gmean speedups are over Serial with default parameters (0%).

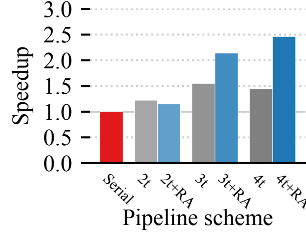


Fig. 15: Sensitivity studies measuring performance of BFS while varying number of stages and use of RAs. Higher is better; speedups are geomean over all inputs.

fits on-chip, so decoupling yields limited benefits. This result shows that, while Pipette's control flow mechanisms work well even under frequent control flow, in some cases data parallelism (i.e., processing several row-column pairs in parallel) is slightly better. An adaptive implementation could detect this and switch between Pipette and data-parallel versions.

Finally, **Silo** (Fig. 13(f)) yields a modest 24% gain for Pipette. Silo's high-radix B+tree is cache-friendly, and the data-parallel version hides occasional misses reasonably well, but Pipette achieves further decoupling and hence performs better.

### *E. Sensitivity studies*

**Sensitivity to physical register count:** Fig. 14 compares the performance of Pipette and data-parallel variants as the physical register file (PRF) changes. The graph shows the gmean speedup over all benchmarks, relative to the serial version with the default 212-entry PRF. We scale the PRF from 180 to 308 entries. We scale Pipette's queues proportionally with PRF size, so larger PRFs result in larger queues and thus more decoupling.

Fig. 14 shows that implementing queues using physical registers is a good choice. Pipette maintains a substantial performance advantage over the full range of PRF sizes. Moreover, while data-parallel benchmarks are insensitive to PRF capacity (as they are bound by backend stalls, the issue queue and ROB limit them), Pipette can modestly benefit from larger PRFs, which improve decoupling.

**Pipette vs. software techniques:** Pipette programs feature fine-grain stages that communicate extremely frequently: as many as one in six register file reads/writes are enqueues/dequeues (BFS, SpMM) or as few as one in 27 (Silo). This shows the need for hardware support: state-of-the-art software queues take tens of cycles per enqueue/dequeue [53], so using them instead of Pipette would add very high overheads.

**Effect of the number of stages on decoupling:** Fig. 15 examines the performance of 2-, 3-, and 4-stage versions of BFS. This shows that proper decoupling requires more than two stages: the best-performing implementation is the 4-stage one. We also measure the effect of RAs these pipelines. Without RAs, performance peaks in the 3-stage implementation, as the 4-stage version has higher ROB pressure. The 2-stage implementation decouples the distance updates, but leaves the fringe accesses and neighbor enumeration needlessly tightly coupled. RAs and decoupling go hand-in-hand; the 2t+RA point demonstrates the pitfalls of adding RAs without first properly decoupling the application. A distance fetched by an RA can become stale if that distance is updated later. This race condition requires an extra check in the second stage, whose latency cannot be overcome by the limited decoupling. When RAs offload *all* long-latency loads, they reduce backend pressure and enable peak performance with 4 stages—a 1.7× speedup over the conventional 4-stage pipeline.

**Effect of RAs:** Fig. 16 shows Pipette's per-application performance without and with RAs. BFS, CC, and SpMM benefit substantially from RAs, whereas PRD, Radii and Silo see modest gains. Overall, RAs improve performance by gmean 38%, by reducing instruction count and core backend pressure.

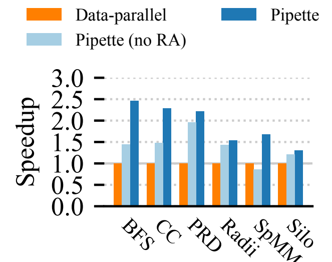


Fig. 16: Performance of Pipette without and with reference accelerators (RAs).

### *F. Multicore Pipette*

Finally, we compare the performance of a different Pipette BFS variant on a 4-core system, showing that data and pipeline parallelism are complementary, and that Pipette's techniques scale outside of the core.

Fig. 17 (right) compares the performance of four different BFS implementations: Serial (1 core, 1 thread), data-parallel (with 4 cores and 4 threads/core), streaming single-threaded (with each BFS stage running on a separate core), and the Pipette multicore BFS shown in Fig. 17 (left). In this version, all stages are replicated across cores to achieve load balance, so each core processes a fraction of the fringe. Furthermore, instead of using shared-memory synchronization, neighbors

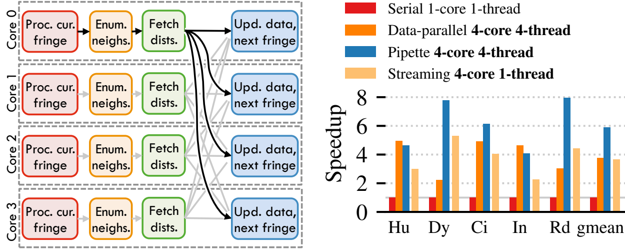



Fig. 17: Multicore+multithreaded BFS Pipette implementation, and its performance vs. serial, data-parallel, and pipeline-parallel versions.

are partitioned across cores and sent to be processed at their corresponding core, as shown by the cross-core communication in Fig. 17 (left) among the last two stages.

Fig. 17 shows that, like in the single-core case, the data-parallel version leaves performance on the table, with a gmean speedup of 3.8× vs. serial despite using 16 threads on 4 cores. The streaming version sometimes outperforms the data-parallel version, though it is limited by load imbalance as each core runs a single stage. Finally, the Pipette multicore version performs best, achieving a gmean speedup of 5.9×.

To analyze scalability, we also evaluate a 16-core system for a total of 64 threads. At 16 cores, the data-parallel version is 1.5× faster than the 4-core system, while Pipette is 1.8× faster (and thus 1.9× faster than 16-core data-parallel). While Pipette scales better, at 64 threads BFS suffers synchronization overheads that limit its scalability.

In summary, this result shows that Pipette continues to be attractive in multicore systems. Multicore Pipette achieves high core utilization and avoids the synchronization overheads of the data-parallel implementation by using connectors to join queues across cores.

## VII. Additional Related Work

We now discuss prior work not introduced in Sec. II.

**Decoupled Access-Execute Architectures:** PIPE [11], ZS-1 [38], ACRI-1 [44], and DeSC [15] are all variants of decoupled access-execute architectures, which have specialized access and execute units separated by queues. These systems extend DAE with facilities for changing control flow or to reduce the impact of loss-of-decoupling events. Software techniques for DAE [55] leverage the same insights, but without the need for specialized hardware. Pipette uses multiple general-purpose threads for improved decoupling and addresses load imbalance by time-multiplexing threads.

DeSC [15] also observes that long-latency producer loads limit performance in OOO cores, and proposes to partially commit them out of order. Pipette's RAs achieve the same effect but with simpler machinery: RAs offload these loads and run them non-speculatively instead. Supporting multiple queues and stages per core is key to this simplification: DeSC has a single supply queue, making RAs harder to adopt.

**Streaming Architectures:** Stream dataflow [30] provides an interface to express streaming semantics and uses coarse-grained reconfigurable arrays (CGRAs) to reduce instruction count. Another approach, employed on general-purpose graphics processing units (GPGPUs), communicates values between threads without going through memory [47].

Similar work on stream specialized processors [48] explicitly informs the processor of application-level memory patterns, including multiple levels of indirection. While similar to Pipette in that it adds a decoupled interface to the microarchitecture, such techniques only source operands from memory. Pipette specifically targets pipelined communication between the threads of multithreaded cores. Moreover, prior streaming abstractions do not effectively handle the unpredictable control flow in irregular applications.

**Helper Threads:** Architectures with helper threads [24], including runahead execution [27], slipstream processing [40], and "flea-flicker" two-pass pipelining [3], perform redundant computation so that the main thread benefits from improved branch prediction and prefetched operands. Pipette instead creates pipelines of threads whose work is never discarded.

## VIII. Conclusion

Applications with irregular access patterns and control flow have latent pipeline parallelism that can be exploited to improve core utilization. However, prior decoupled architectures are insufficient for these irregular applications. We have presented Pipette, which achieves high utilization by exploiting fine-grain pipeline parallelism within the threads of a multithreaded core. This new regime not only allows fast and inexpensive local communication, but also sidesteps the load balancing issues that affect prior decoupled architectures and enables a cheap implementation that reuses otherwise-idle registers and accelerates common access patterns. As a result, Pipette achieves significant speedups on several applications over a wide variety of inputs. Pipette thus offers a high-performance, practical substrate for pipeline-parallel programs.

## Acknowledgments

We thank Maleen Abeydeera, Joel Emer, Axel Feldmann, Mark Jeffrey, Hyun Ryong (Ryan) Lee, Anurag Mukkara, Po-An Tsai, Yifan Yang, Victor Ying, Guowei Zhang, and the anonymous reviewers for their feedback. This work was supported in part by DARPA SDH under contract HR0011-18-3-0007. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

## References

[1] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proc. ICS'16*, 2016.

[2] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proc. ASPLOS-XXIII*, 2018.

[3] R. D. Barnes, J. W. Sias, E. M. Nystrom, S. J. Patel, N. Navarro, and W. W. Hwu, "Beating in-order stalls with "flea-flicker" two-pass pipelining," *IEEE Trans. Computers*, vol. 55, no. 1, 2006.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. SoCC'10*, 2010.

[5] N. C. Crago and S. J. Patel, "OUTRIDER: Efficient memory latency tolerance with decoupled strands," in *Proc. ISCA-38*, 2011.
[6] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi, "Merrimac: Supercomputing with streams," in *Proc. SC03*, 2003.
[7] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *Proc. PACT-15*, 2006.
[8] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, "A clustered manycore processor architecture for embedded and accelerated applications," in *Proc. HPEC*, 2013.
[9] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation Intel Core: New microarchitecture code-named Skylake," *IEEE Micro*, vol. 37, no. 2, 2017.
[10] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *Proc. ASPLOS-XII*, 2006.
[11] J. R. Goodman, J. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: A VLSI decoupled architecture," in *Proc. ISCA-12*, 1985.
[12] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proc. ASPLOS-XII*, 2006.
[13] V. Govindaraju, C. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proc. HPCA-17*, 2011.
[14] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," *IEEE CAL*, vol. 8, no. 1, 2009.
[15] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled supply-compute communication management for heterogeneous architectures," in *Proc. MICRO-48*, 2015.
[16] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," in *Proc. OOPSLA*, 2017.
[17] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proc. MICRO-46*, 2013.
[18] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proc. PLDI*, 2008.
[19] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips, "SQRL: Hardware accelerator for collecting software data structures," in *Proc. PACT-23*, 2014.
[20] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE TC*, vol. 100, no. 1, 1987.
[21] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proc. SPAA*, 2010.
[22] 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 *Proc. MICRO-42*, 2009.
[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.
[24] P. Marcuello, A. Gonzalez, and J. Tubella, "Speculative multithreaded processors," in *Proc. ICS'98*, 1998.
[25] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.
[26] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proc. MICRO-51*, 2018.
[27] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proc. HPCA-9*, 2003.
[28] Nangate Inc, "The NanGate 45nm open cell library," http://www.nangate.com/?page_id=2325, 2008.
[29] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. SOSP-24*, 2013.
[31] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *Proc. HPCA-22*, 2016.
[30] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. ISCA-44*, 2017.
[32] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. C. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. L. Allmon, R. Rayess, S. Maresh, and J. S. Emer, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Proc. ISCA-40*, 2013.
[33] J. Park and W. J. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," in *Proc. SPAA-22*, 2010.
[34] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proc. ISCA-40*, 2013.
[35] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proc. PACT-13*, 2004.
[36] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. PPoPP*, 2013.
[37] J. E. Smith, "Decoupled access/execute computer architectures," in *Proc. ISCA-9*, 1982.
[38] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. Laudon, "The ZS-1 central processor," in *Proc. ASPLOS-II*, 1987.
[39] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proc. VLDB*, 2015.
[40] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," in *Proc. ASPLOS-IX*, 2000.
[41] M. Sung, R. Krashinsky, and K. Asanović, "Multithreading decoupled architectures for complexity-effective general purpose computing," *SIGARCH Comput. Archit. News*, 2001.
[42] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: A computational fabric for software circuits and general-purpose programs," in *Proc. MICRO-35*, 2002.
[43] M. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal, "Scalar operand networks," *IEEE TPDS*, vol. 16, no. 2, 2005.
[44] N. P. Topham and K. McDougall, "Performance of the decoupled ACRI-1 architecture: The perfect club," in *Proc. HPCN*, 1995.
[45] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. SOSP-24*, 2013.
[46] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proc. ISCA-23*, 1996.
[47] D. Voitsechov, O. Port, and Y. Etsion, "Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays," in *Proc. MICRO-51*, 2018.
[48] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *Proc. ISCA-46*, 2019.
[49] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/, 2014.
[50] 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 *Proc. ASPLOS-XIX*, 2014.
[51] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proc. MICRO-48*, 2015.
[52] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A high-performance graph DSL," in *Proc. OOPSLA*, 2018.
[53] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proc. EuroPar*, 2012.
[54] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, 1992.
[55] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. Don't tweak the hardware: A new compiler approach to Voltage-Frequency scaling," in *Proc. CGO*, 2014.