Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching

Dan Zhang
The University of Texas at Austin
dan.zhang@utexas.edu

Xiaoyu Ma
Google
xiaoyuma@google.com

Michael Thomson
Centaur Technology
mthomson@centtech.com

Derek Chiou
Microsoft & The University of Texas at Austin
derek@utexas.edu

Abstract

The importance of irregular applications such as graph analytics is rapidly growing with the rise of Big Data. However, parallel graph workloads tend to perform poorly on general-purpose chip multiprocessors (CMPs) due to poor cache locality, low compute intensity, frequent synchronization, uneven task sizes, and dynamic task generation. At high thread counts, execution time is dominated by worklist synchronization overhead and cache misses. Hardware worklist accelerators lower scheduling costs, but have inflexible scheduling policies and do not address high cache miss rates.

We address this with *Minnow*, a technique that augments each CMP core with a *Minnow engine*, a programmable accelerator that offloads worklist scheduling and performs *worklist-directed prefetching*, exploiting knowledge of upcoming tasks to issue accurate and timely prefetch operations. On a simulated 64-core CMP running a parallel graph benchmark suite, Minnow improves scalability and reduces L2 cache misses from 29 to 1.2 MPKI on average, resulting in 6.01x average speedup over an optimized software baseline for only 1% area overhead.

Keywords Accelerators; Prefetching; Graph Analytics

ACM Reference Format:
Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173197

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173197

1 Introduction

The ubiquity of multicore processors has shifted the burden of improving software performance from chip manufacturers to software developers. While *regular* programs targeting dense matrices with predictable access patterns are well-understood, *irregular* programs are difficult to parallelize due to unpredictable access patterns, poor locality, low arithmetic intensity, and data-dependent control flow [57].

In this work, we explore these issues in the context of graph analytics. Graph workloads are notoriously difficult to execute efficiently on modern general-purpose processors [33] and have become an increasingly important problem over a wide range of domains, including web search [42], social networks [16] [28], and recommendation systems [10]. With growing dataset sizes and graph query complexity, the demand for fast performance has pressured researchers to develop more efficient parallel algorithms [58] [35] [17] [30].

The difficulty of creating customized parallel graph applications has led to the development of graph processing frameworks using common design patterns [54] [13] [51] [43] [34]. Graph algorithms can be decomposed into parallel tasks where each task processes a single node [43]. *Active nodes* that require processing are stored as *work items* in a global queue called a *worklist*, from which they are dynamically scheduled to worker threads.

Many graph algorithms are *unordered*: tasks can be applied to nodes in any order until convergence. Some unordered algorithms can improve their convergence rate by enforcing some algorithmic priority order. Improving convergence rate improves *work efficiency*, the total amount of work executed by the parallel algorithm relative to the amount of work executed by an efficient serial algorithm. Worklists implement partial ordering to gain most of the efficiency benefit while uncovering more parallelism and reducing overhead relative to a strict priority worklist [31]. However, previous work has either downplayed or ignored the effects of partial priority ordering on overall performance [54] [47] [24]. We show in Section 3.1 that exploiting partial prioritization can result in over 100x speedup compared to a state-of-the-art unordered graph framework [54] due to improved work efficiency.

Despite considerable research from both software and hardware directions, general-purpose CMPs are still inefficient at executing parallel graph workloads [5]. The problem is two-fold: firstly, worklists only provide good performance when tasks are large compared to worklist overhead. However, tasks in graph analytics tend to be small, on the order of hundreds of instructions [27], leading to high scheduling overhead and thus reduced scalability. The problem is further exacerbated when attempting to balance overheads with work efficiency. Hardware worklist accelerators [27] [29] [46] reduce worklist overhead by hardening specific scheduling policies using hardware queues with message passing interfaces, but are inflexible and cannot support priority ordering. Furthermore, these designs have unscalable centralized structures [27], limited buffering capacity [27], and/or require custom on-chip networks [29] [46].

Secondly, graph algorithms generally have hard-to-predict memory access patterns and low computational intensity, resulting in workloads struggling to fully utilize both compute resources and memory bandwidth [5]. Prior work suggests that this poor performance can be fixed with larger reorder buffers (ROBs) [5], but we show in Section 3.3 that increasing ROB size alone does not significantly improve performance. Techniques such as helper threading [6][11][8] can help, but generally only target single-threaded workloads due to their dependency on otherwise-idle SMT threads or cores. State-of-the-art hardware prefetchers such as IMP [59] target specific patterns, but cannot adapt if the data structure does not match their memory layout assumptions.

In this paper we propose *Minnow*, a technique that attacks both memory and scheduling bottlenecks in parallel graph workloads by augmenting each CMP core with a lightweight multithreaded *Minnow engine* connected through an accelerator interface. The engines offload worklist operations, removing scheduling from the critical path to improve scalability. Minnow engines are programmable to enable custom priority scheduling, and have access to the CMP memory subsystem through its core's L2 cache, resulting in a decentralized design without on-chip network modifications or large dedicated buffers. Minnow engines use their knowledge of upcoming tasks to perform *worklist-directed prefetching*, a novel technique that launches prefetch threads in response to worklist scheduling decisions, enabling accurate and timely prefetches well ahead of the execution stream.

We modify Galois, a parallel task framework, to support Minnow and simulate 64-core CMP performance on several graph workloads. By addressing key bottlenecks, Minnow reduces scheduling overhead, improves scalability, and virtually eliminates L2 cache misses, resulting in 6.01x average speedup at only 1% area overhead. Our contributions include:

- We characterize the bottlenecks in parallel data-driven graph workloads, contradicting conclusions from previous work and motivating the design of Minnow.

```
Graph graph = { /* init */ };
WorkList<GraphNode> workQ();
workQ.enq(0, srcNode);

foreach(GraphNode node : workQ) {
  for(Edge edge : node.edges) {
    Node destNode = edge.destNode;
    int newDist = node.distance + edge.weight;
    if(newDist < destNode.distance) {
      destNode.distance = newDist;
      int priority = newDist >> lgBucketInt;
      workQ.enq(priority, destNode);
    }
  }
}
```

Figure 1. Galois pseudocode for SSSP Delta-Stepping

- We propose the use of programmable engines with access to the core's memory subsystem to offload worklist operations without needing a separate interconnect network or hardening a specific scheduling policy.
- We propose worklist-directed prefetching, a software prefetching technique that exploits application knowledge of upcoming tasks to issue prefetch requests with nearly perfect accuracy and timeliness.
- We propose Minnow, combining worklist offload and worklist-directed prefetching to accelerate parallel task-based irregular applications and demonstrate its effectiveness on parallel graph workloads.

2 Background

Minnow can be used to accelerate most software task-based frameworks. We chose Galois since its task-based programming model is well-suited to exploit priority ordering.

2.1 The Galois Framework

Galois is an asynchronous task-based framework for parallel data-driven algorithms that can exploit partial priority ordering [43]. Galois provides a **foreach** loop construct, where loop bodies, representing tasks, can be scheduled for parallel execution by a worklist following user-specified priorities. The *ordered by integer metric* (OBIM) worklist [31] used in Galois relaxes strict prioritization to reduce scheduling overhead. OBIM discretizes priorities into *buckets* as follows:

$$bucket\_number = priority \gg lg\_bucket\_interval$$

The worklist attempts to process buckets in order, but work within each bucket is unordered and can be scheduled in parallel. The contents of each bucket are held in a concurrent FIFO, and a concurrent ordered map stores pointers to each bucket. Increasing the bucket interval increases parallelism but lowers work efficiency; the optimal bucket interval depends on the algorithm, graph input, and host machine.

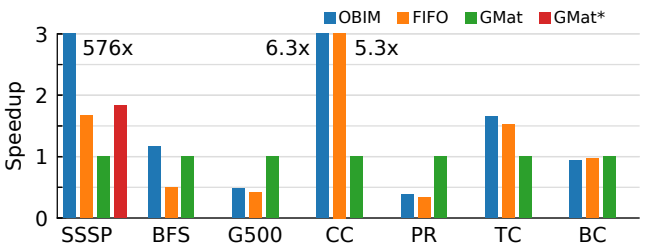

Figure 2. Galois and GraphMat speedup normalized to 1-thread GraphMat

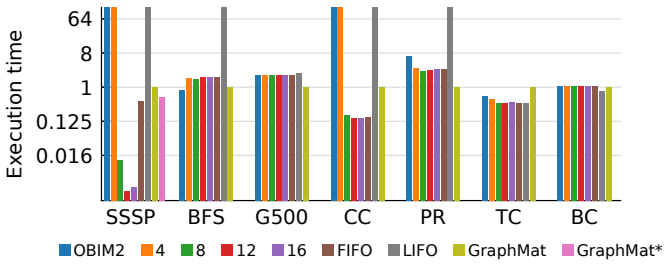

Figure 3. Runtime for various Galois schedulers normalized to GraphMat (lower is better). High bars represent the benchmark timing out. Intel Carbon uses the LIFO policy.

Galois is designed to parallelize irregular algorithms, and is well-suited for graph analytics. An example is SSSP, shown in Figure 1. The worklist is initialized with the source node, and each task processes a single node. Tasks explore node neighbors, conditionally updating nodes and generating more tasks. The code shown can represent Dijkstra's algorithm [25], Bellman-Ford [12], or Delta-stepping [35] depending on the instantiated Galois worklist. Dijkstra processes nodes shortest-distance-first in strict priority order. While this is efficient with complexity $O(v \log v + e)$, it exposes no parallelism since each task may produce the next-highest priority task. Bellman-Ford trades off work efficiency for parallelism by processing tasks in any order, increasing complexity to $O(ve)$. Delta-stepping uses partial ordering to discretize priorities into buckets with Dijkstra's-like efficiency while maintaining high levels of parallelism, and maps to OBIM.

3 Motivation

3.1 The Benefits of Priority Ordering

Many graph frameworks overlook the benefits of partial priority ordering. To characterize the benefits of frameworks that specialize in supporting partial priority ordering, we compare Galois against GraphMat, a state-of-the-art graph framework built on a highly tuned sparse matrix library with an unordered bulk synchronous execution model and up to 7x faster than competing frameworks [54]. Each iteration, GraphMat processes all active nodes in parallel, generates the next set of active nodes, and repeats until convergence.

Figure 2 shows Galois and GraphMat performance at 10 threads on a 10-core Intel Xeon E5-2680v2 normalized to GraphMat on the benchmarks listed in Section 6.1. We test Galois with partial priority OBIM and unordered FIFO worklists. GraphMat performs much better on G500 and PR due to its heavy optimizations and simpler bulk synchronous execution model. However, SSSP is especially sensitive to priority ordering, with Galois achieving 576x speedup compared to GraphMat. Since ordering improves Big-O, running SSSP with larger graphs will result in even more disparate results. For example, Galois achieves a 927x speedup over GraphMat running SSSP on USA-road-d.USA, an graph with 23M nodes and 58M edges. We contacted the GraphMat authors, who wrote a new SSSP Delta Stepping kernel taking advantage of ordering, shown as *GMat** in Figure 2.

However, GraphMat kernels can only perform unordered scheduling. To implement bucketed priorities, they execute a full GraphMat kernel for each bucket, resulting in high overheads and thus much larger optimal bucket interval than Galois with OBIM. Their Delta Stepping implementation results in only 2x improvement over their unordered implementation at 10 threads, still 285x slower than Galois with OBIM. BFS, G500, CC, and PR are less sensitive but still benefit from priority ordering compared to the Galois worklist. Although many workloads do not require priority ordering for good performance, failure to consider it can result in especially poor performance scenarios like SSSP.

The optimal scheduler depends on the application and graph input. To measure the effects, we ran our graph workloads with various Galois scheduling policies. Figure 3 shows execution time normalized to the same workloads run in GraphMat. For workloads sensitive to ordering, choosing an improper scheduling policy may result in workloads sensitive to ordering never converging and timing out. Several OBIM configurations time out as well, since OBIM assumes changing buckets is rare. OBIM's optimal configuration requires tuning for each workload and input, but choosing a conservative bucket interval does not severely impact performance. Priority sensitivity is a function of the algorithm and input. Graph inputs with high diameters and low degrees will be more sensitive to priority ordering. Prior work has generally focused on social network and web graphs with low diameters and high degrees less sensitive to priority ordering, or used suboptimal baselines such as Bellman-Ford.

Prior work on hardware worklist acceleration has not considered the effects of priority ordering, and generally only support a fixed scheduling policy. Carbon [27] only supports LIFO scheduling, which can improve cache locality. This results in faster performance on BC, but times out on SSSP, BFS, CC, and PR, making it impractical for general use. IsoNet [29], Rigel [20], and HWWL [24] use FIFO-mostly schemes which don't capture any benefits from prioritization, which may result in significant slowdown compared to using a software priority worklist on workloads like SSSP.

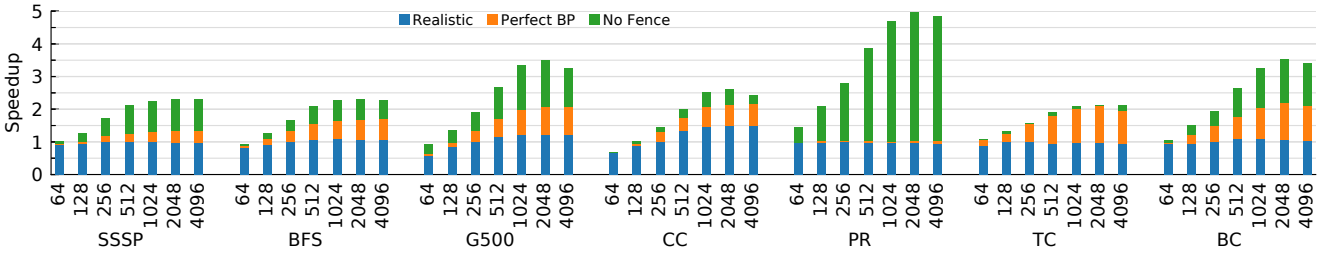

Figure 4. Benchmark sensitivity to ROB size, speedup normalized to 256-entry ROB with 128-entry reservation station, 64-entry load queue, and 64-entry store queue. "Realistic" has realistic branch prediction and atomics with memory fences.

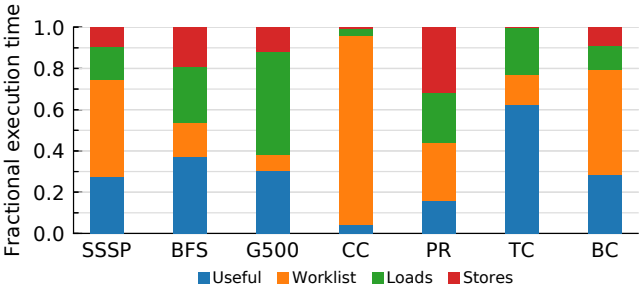

Figure 5. Galois overhead breakdown

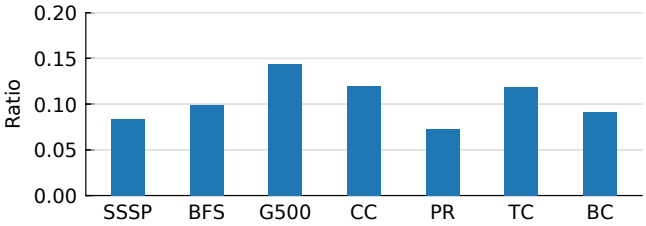

Figure 6. Delinquent load density, i.e. the ratio of frequently-missing loads to all loads

3.2 Software Inefficiencies in Graph Analytics

Previous work [14] has already demonstrated that general purpose processors are inefficient at graph analytics. To characterize this inefficiency, we simulated our workloads at 64 threads as described in Section 6. Figure 5 shows the cycle breakdown spent on useful work, worklist operations, and data cache misses. On average, only 28% of execution time is spent performing useful work. The remaining cycles are spent on synchronization and cache misses. CC is significantly worklist-bottlenecked past 16 threads, resulting in worklist operations comprising 92% of all cycles. Aside from CC, Galois worklist operations comprise 8% to 51% of all cycles and cache misses comprise the rest, from 12% on BC to 50% on G500. PR has a significant 32% store cycle bottleneck since atomic operations are classified as stores. These bottlenecks present an opportunity to massively improve performance from multiple fronts.

3.3 Exploiting Memory Level Parallelism

Previous work claims that ROB size is the main limiting factor for memory throughput [5]. However, there are two serializing events that can limit MLP: firstly, branch mispredictions are notably costly when the branch depends on a long-latency load [53] as is common in graph workloads. Secondly, the memory consistency model in x86 states that atomics are not reordered with loads and stores [1], thus requiring all previous loads and stores to be completed, i.e. a memory fence, prior to each atomic [49]. This is costly for benchmarks like PageRank, where the residual is unconditionally pushed to all outgoing edges. To evaluate the effect of these serializing events, we calculate speedup while sweeping ROB size in simulation, normalized to a 256-entry ROB configuration with 128-entry RS, 64-entry LQ, and 64-entry SQ. Each configuration keeps the same buffer sizing ratio. We tested a realistic baseline, and ideal versions with perfect prediction and no fencing.

Our results, shown in Figure 4, demonstrate that ROB size is not the main limiting factor: realistic speedup past 256 ROB entries is minimal. Once branch prediction and fencing limiting factors are removed, ROB size becomes the limiting factor and scales well with performance. PR executes many more atomics than other benchmarks, resulting in up to 5x speedup once fences are removed. Unfortunately, these branches and atomics are essential for correctness and cannot be removed when running on realistic hardware, although x86 chip designers may wish to consider adding special atomic instructions that relax the consistency model and enable the elimination of fences. These results suggest that helper threads, which can speculatively prefetch ahead while avoiding serializing events, may be a promising approach.

3.4 Maximizing MLP with Delinquent Load Density

Helper threads can improve MLP not just by reducing serializing events, but also by increasing the density of *delinquent loads*, loads with frequent cache misses, in the load queue. Cores have large OOO windows to find and issue delinquent loads in parallel, but these loads are sparsely populated in the instruction stream. Increasing the number of delinquent loads in the load queue improves MLP and thus performance.

Figure 6 shows the ratio of delinquent loads relative to all other loads in our benchmark suite, where delinquent loads are first accesses to graph nodes and edges in a task. Despite the low compute density, only about 10% of all loads

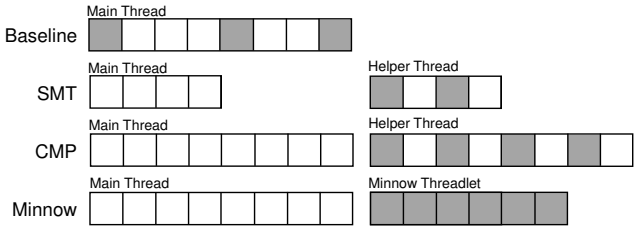

Figure 7. Increasing in-flight delinquent loads in the load queue with helper threads. Boxes represent load queue entries, and shaded boxes represent delinquent loads.

are delinquent, with remaining loads comprising secondary accesses to a node or edge, stack reads, and register spills/fills. This is especially an issue on x86 due to its two-operand instructions and limited number of general-purpose registers. On modern cores such as Intel Skylake with 72-entry load queues, only about 7 entries will be delinquent, not enough to tolerate frequent accesses to memory.

As shown in Figure 7, helper threads can improve delinquent load density by executing a subset of the program. However, each delinquent load now requires two load queue entries. Loads in the main thread are not marked delinquent since the helper thread accesses them first. When running helper threads in SMT, the load queue is shared, which may only slightly improve density. Running helper threads on separate cores doubles load queue resources but limits potential improvement due to only prefetching into shared LLC, and also doubles area and power. In Minnow, Minnow engines executing helper threads reserve its load queue for only delinquent loads, exposing greater MLP at lower cost.

4 Minnow Overview

We propose *Minnow*, a technique that addresses the bottlenecks presented in Section 3 by coupling each core in a general-purpose CMP with a *Minnow engine*, a highly multithreaded programmable core augmented with hardened logic to accelerate worklist scheduling and prefetching operations, as shown in Figure 8. Minnow engines are fully programmable and have access to the CMP memory subsystem through its paired core's L2 cache and L2 TLB. We assume that the L2 TLB is inclusive with the L1 TLB and choose not to connect to the core's L1 cache and L1 TLB due to routing, timing, and capacity concerns. Cores may share a single Minnow engine to reduce resources, but this work focuses on cores with dedicated Minnow engines.

Minnow engines are designed to perform two tasks: firstly, software worklist operations are offloaded to Minnow engines through accelerator calls. Unlike previous work, the full programmability of the Minnow engine enables any software worklist algorithm to be offloaded, including priority worklists such as OBIM. Although Minnow engines also have small hardware queues to store pending tasks, Minnow does not require a dedicated global task unit to handle spilled tasks

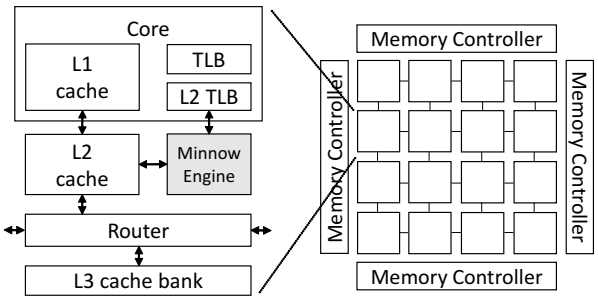

Figure 8. Example microarchitecture implementing Minnow

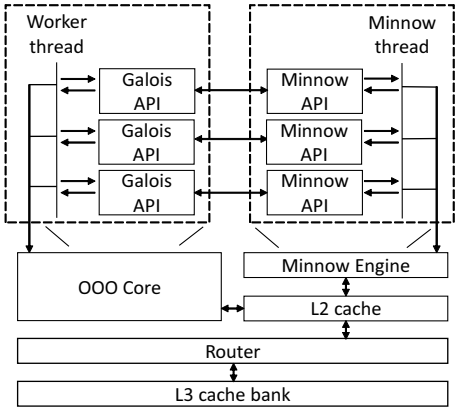

Figure 9. Galois-Minnow interface overview

[27] or a dedicated task routing network [29] [46]. Instead, Minnow leverages the existing cache hierarchy by implementing the global priority worklist in software running on the Minnow engine. Minnow engines are therefore able to store spilled tasks in the memory subsystem, only requiring processor intervention on TLB misses. Secondly, once a task has been scheduled to a core by the worklist, Minnow engines perform worklist-directed prefetching, spawning a helper thread to prefetch the task's input data.

Rather than executing helper threads in a different SMT context on the same core as the worker thread, creating contention for execution resources, helper threads are offloaded to Minnow engines that support a large number of threads in flight and context switch after each memory request. Overall, Minnow provides three potential benefits: improved convergence rates through priority scheduling, improved scalability through worklist offloading, and improved IPC through worklist-directed prefetching.

4.1 Minnow API

To support Minnow, framework developers simply need to modify their framework to perform the appropriate Minnow API calls, thus hiding Minnow complexity from the end user. For this work, we added Minnow support to Galois [43]. As shown in Figure 9, Galois worker threads are not aware of Minnow, instead calling worklist enqueue and dequeue operations through the Galois API. These Galois API calls

are translated into Minnow accelerator calls, whereafter the Minnow engine executes the offloaded worklist operations.

A Minnow task consists of two 64-bit values: an integer priority, and a pointer to the task data. Cores primarily communicate with Minnow engines to enqueue and dequeue tasks, but additional instructions are needed to initialize, context switch, and detect when all worker threads are done. To keep Minnow engines simple, it cannot handle TLB misses. Instead, the Minnow instruction that caused the TLB miss throws an exception, leveraging the host processor to properly handle the miss. We extended the ISA with the following instructions:

- **minnow_init**: initializes Minnow engines across all threads.
- **minnow_enqueue**: enqueues a task to the core's Minnow engine. Takes priority and pointer to task as input. May cause TLB miss exception.
- **minnow_dequeue**: dequeues a task from the core's Minnow engine. Stalls until a task is available, and returns a pointer to the task. If worklist is empty, returns a null pointer. May cause TLB miss exception.
- **minnow_flush**: prepares Minnow engine for a core context switch. Flushes local queue tasks to the global worklist, and returns a pointer to the worklist.
- **minnow_done**: determines if all Minnow engines are idle and global worklist is empty.

5 Minnow Engines

Minnow engines are programmable engines that interface with their paired core through an accelerator interface to offload worklist and prefetching operations. These operations are heavily memory-bound with low arithmetic intensity and frequent synchronization. In such scenarios, rather than focusing on IPC, a popular approach is to expose memory-level parallelism explicitly through multiple threads [4][41][7][9]. We use a similar approach with Minnow engines.

The Minnow engine contains a lightweight in-order programmable microcontroller core that executes *threadlets*, short lightweight threads spawned within the Minnow engine without OS overhead. Threadlets are spawned in response to accelerator calls or by other threadlets to perform worklist spill/fill operations and prefetching. In addition, each Minnow engine contains a private instruction memory for storing threadlet instructions and a private data memory storing threadlet context data and temporary variables.

Since accessing the memory subsystem through the core's L2 cache is slow, Minnow engine microcontrollers access private memory for most operations and only explicitly access L2 when accessing shared data structures, i.e. the priority worklist and input graph. Minnow engines expose memory-level parallelism by supporting many threadlets in flight and context switching on every L2 access. Pending loads are stored in a CAM-based load buffer and threadlets are dynamically woken when their pending load completes. Unlike in a standard OOO core, since temporary values are stored in the Minnow engine private data memory, the load buffer only contains delinquent load operations. Therefore, even small load buffers can support more delinquent loads in flight than a much larger traditional OOO window.

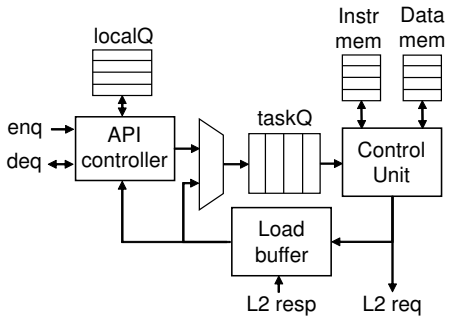

Figure 10. Minnow engine microarchitecture

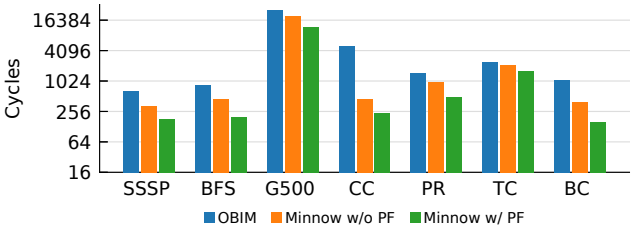

Figure 11. Average cycles per worklist enq/deq operation

5.1 Minnow Engine Microarchitecture

Figure 10 depicts a high-level overview of our proposed Minnow engine microarchitecture. The design is split into two parts: a hardened low-latency front-end interfacing with the core with dedicated buffering for storing high-priority tasks, and a throughput-oriented fully programmable back-end handling accesses to the core's memory subsystem. There are several key considerations for designing a high-performance, low-cost Minnow engine. Since the core cannot make forward progress while waiting for a task, the Minnow engine front-end is optimized to reduce **minnow_dequeue** common-case latency. In addition, the Minnow engine must be optimized to sustain many long-latency loads in flight to effectively perform worklist-directed prefetching. As shown in Figure 11, since the Minnow engine is only accessed once every few hundreds of cycles for worklist enqueue/dequeue operations, an aggressive front-end design is not required to meet performance goals. Instead, the base design is kept simple and hardware is added only where necessary.

The front-end presents an accelerator interface to the core and contains dedicated buffering for accelerating worklist enqueue and dequeue operations. It can be performed using hardened logic such as a simple finite state machine (FSM). For less common operations that require accessing system

```
void minnowEnq(Task* task) {
  int bucket = task->priority >> lgBucketInt;

  // Put in localQ if not full and high priority
  if(!localQ.full() && (bucket <= localQ.bucket)) {
    localQ.enq(task);
    localQ.bucket = bucket;
  }
  // Else, spill to global worklist
  else {
    globalQ.enq(task);
  }
}

Task* minnowDeq() {
  if(!localQ.empty()) {
    return localQ.deq();
  }
  // Else, fetch from global worklist (slow)
  // If globalQ is empty, returns NULL
  else {
    return globalQ.deq();
  }
}
```

Figure 12. Minnow worklist operation pseudocode

memory such as worklist spills and fills, the front-end spawns a threadlet by placing it into the back-end threadlet queue.

The back-end is responsible for all operations that interface with the memory hierarchy, including worklist spills, fills, and helper thread prefetching. It consists of a threadlet queue, in-order instruction pipeline (Control Unit), instruction and data memories, and OOO load buffer. The Control Unit instruction pipeline processes threadlets from the threadlet queue in order. To handle multiple loads in flight, the pipeline context switches after every L2 cache load request, allowing it to sustain one load in flight per threadlet. Since threadlets only consist of a few instructions, threadlet contexts are small, requiring only about 64B per context stored in data memory.

When the load request returns, the load address is matched with the proper load buffer entry, and the threadlet is re-added to the threadlet queue for further processing. Unlike a standard load queue, since the load buffer is optimized for throughput rather than latency, performing the CAM address search can take several cycles without affecting performance. In our experiments, we assume a 32-entry load buffer, and a 4-cycle CAM search latency.

5.2 Worklist Offloading

The Minnow engine interfaces with the core to offload worklist operations. Figure 12 shows pseudocode for Minnow enqueue and dequeue operations. Task priorities are discretized into bucket priorities. Minnow engines assign their

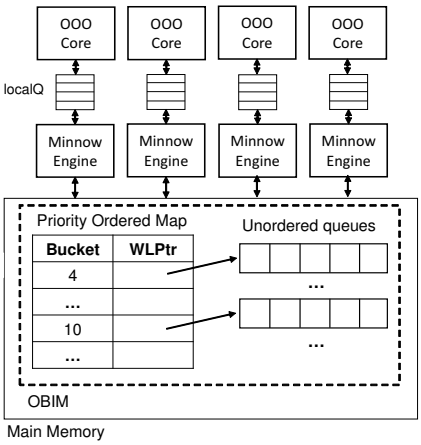

Figure 13. Minnow worklist hierarchy

local queue a bucket priority number and attempt to only store tasks associated with the highest-priority bucket in the local queue. Within each local queue, all tasks are treated as the same priority and are processed in FIFO order. Once a task is stored in the local queue, it is guaranteed to be scheduled in FIFO order barring a **minnow_flush** operation. On a **minnow_enqueue** operation, if the enqeued task is lower priority than the local bucket priority or if the local queue is full, then the task is sent to the global priority worklist by spawning a threadlet. If the task has a higher bucket priority than the local queue, the local bucket priority is updated but local queue contents remain unchanged. On a **minnow_dequeue** operation, the operation stalls on an empty local queue until it is populated with tasks from the global worklist. If the global worklist is also empty, the operation fails and returns NULL. The requesting core can then perform termination detection to see if all threads are idle or if there simply aren't enough tasks at the moment.

Since **minnow_dequeue** operations block until the next task is fetched, accessing the global worklist hurts performance and should be avoided. To minimize stalling, when the number of task in the local queue is below a programmable threshold, the Minnow engine proactively attempts to dequeue tasks in the highest-priority bucket from the global priority worklist by spawning a threadlet. If tasks at the head of the global worklist are of equal or higher priority than the local queue, then they are streamed into the local queue and the local bucket priority is updated. However, if the local queue is empty, tasks are unconditionally accepted.

Since the Minnow engine is programmable, the global priority worklist can be implemented using any concurrent priority scheduler software algorithm. A simplified version of OBIM [31] was implemented. Figure 13 shows an overview of the worklist hierarchy. Pending tasks are stored in the local queue, with all other tasks stored in memory as part of the global priority worklist. The global priority worklist is constructed with a concurrent **ordered_map**, mapping

bucket numbers to pointers to concurrent unordered worklists. Since worklist operations are now off the critical path, higher overhead algorithms producing better priority schedules can potentially be selected. Minnow instructions may throw an exception to perform the required memory allocation or deallocation tasks. To lower overheads, several memory allocation and deallocation tasks may be grouped together. During a context switch, Minnow engines flush all tasks from the local queue into the global worklist.

5.3 Worklist-Directed Prefetching

Worklist offload provides a unique opportunity for prefetching. Since the Minnow engine schedules all upcoming tasks to its worker thread, the Minnow engine knows exactly what the worker thread will execute in the future and thus can prefetch each task prior to execution. We call this technique *worklist-directed prefetching*, in which task-parallel applications use scheduler knowledge to launch helper threads for prefetching when tasks are assigned to a worker. With multiple pending tasks per worker and thousands of cycles per task, prefetches can be issued well before worker thread access, enabling accurate and timely prefetches. Unlike traditional helper threads, worklist-directed prefetching does not require compiler support or delinquent load profiling, and can be supported by any parallel task framework. A challenge in helper threading is determining the best time to launch a helper thread. In worklist-directed prefetching, prefetches are simply launched when tasks are scheduled.

Although worklist-directed prefetching can be used on any parallel architecture, it is especially well-suited for Minnow due to its low area and support for high MLP. Helper threads are generally executed on SMT threads [8] or separate OOO cores [21], wasting hardware resources that could potentially be used to execute more worker threads. Using smaller OOO cores [56] limits potential speedup since helper thread performance depends heavily on the core's ability to extract MLP, preventing the helper thread from running sufficiently far ahead of the worker thread.

To perform worklist-directed prefetching, the Minnow engine is initialized with the task prefetching function shown in Figure 14. Many graph algorithms have the same per-node access pattern, enabling graph framework developers to write helper thread code once and hide the complexity from users. For many graph algorithms, for each task, the operator accesses a node in the graph, the node's outgoing edges, and the destination nodes for all edges. These tasks, nodes, and edges should all be prefetched. Prefetch requests are treated as normal load operations and cannot be dropped. If users require a different graph access pattern, they can write a custom prefetch function to fit their usage model. All our workloads except for TC use this prefetching function. For TC, we wrote a custom TC prefetching function.

Whenever a Minnow engine enqueues a task into its local queue, the task is now guaranteed to be consumed by its core barring a core context switch, thereby triggering a task prefetch which enqueues a helper threadlet to the threadlet queue. When the threadlet is scheduled, it then executes **prefetchTask()** shown in Figure 14. Helper threads call **load_L2()** to read from the core's L2 cache into the Minnow engine's private data memory, consume a prefetch credit, and context switch to a different threadlet.

```
// Main prefetch helper function
void prefetchTask(Address taskAddr) {
  // Prefetch task
  Task task = load_L2(taskAddr, sizeof(Task));
  // Prefetch source node
  Address srcAddr = nodeBasePtr + task.nodeID;
  Node src = load_L2(srcAddr, sizeof(Node));
  // Index into edge array
  Address edgeAddr = edgeBasePtr + src.edgePtr;

  // Prefetch edges and dest nodes in parallel
  for(int i = 0; i < src.numEdges; i++) {
    // Spawn edge prefetch threadlet
    threadletQ.enq(PREFETCH_EDGE, edgeAddr + i);
  }
}

// Spawned from prefetchTask()
void prefetchEdge(Address edgeAddr) {
  // Prefetch edge
  Edge edge = load_L2(edgeAddr, sizeof(Edge));
  // Prefetch destination node
  Node dest = load_L2(edge.dest, sizeof(Node));
}
```

Figure 14. Minnow task prefetch pseudocode

To achieve high prefetch throughput, multiple edges and destination nodes must be prefetched in parallel per task. Since threadlets are unordered, they can explicitly represent the dataflow graph, with **prefetchTask()** spawning a **prefetchEdge()** threadlet for each edge in the source node. Each **prefetchEdge()** threadlet prefetches a single edge and destination node, but many **prefetchEdge()** threadlets can execute in parallel.

5.3.1 Prefetch Rate Control

While prefetched data is guaranteed to be accessed in the future, timeliness is a major concern. Tasks generally process nodes, and nodes may have millions of outgoing edges. The memory footprint of a task may therefore exceed cache capacity, requiring a smarter prefetching solution than simply fully prefetching a task before dispatching it to the worker thread. Instead, tasks must be dispatched to worker threads and concurrently prefetched within the Minnow engine. Prefetches issued too late are not useful, and prefetches issued too early may be evicted from the cache prior to use. Ideally, the prefetch thread should run ahead of the worker

thread by some fixed distance such that the prefetched data is ready in the cache by the time it is accessed by the worker thread, yet not too far ahead such that the prefetched data is evicted from the cache prior to access.

We propose a credit-based prefetch throttling mechanism in which each Minnow engine is initialized with a fixed number of credits representing the maximum number of L2 cachelines reserved by the prefetcher. Credits are decremented when the Minnow engine issues a prefetch. When credits run out, prefetching pauses until credits become available. Each L2 cacheline is augmented with one bit of prefetch metadata, and cachelines prefetched by Minnow engines are marked. When a cacheline is accessed or evicted, if the prefetch bit is set, the prefetch bit is cleared and a credit is returned.

In contrast to helper thread techniques such as periodic synchronization [56], Minnow's credit-based scheme operates on cachelines rather than loop iterations, allowing the throttling mechanism to be completely abstracted from the end-user. Loop iteration distance throttling can work well, but every helper function has a different optimal loop iteration distance since each loop may access an arbitrary amount of data per iteration. In contrast, the number of initial credits is simply the number of reserved cachelines for prefetching and can be generalized to throttle any other prefetcher in the system. Our results show that Minnow can be initialized with a fixed number of credits which produces near-optimal speedup across all tested workloads.

5.3.2 Avoiding Deadlock

Since Minnow engine prefetch requests may stall due to lack of credits and can spawn an unbounded number of new threadlets, proper care must be taken to avoid deadlock. Deadlock and starvation can be avoided by context switching whenever a threadlet stalls, using virtual queues for each type of threadlet, and reserving entries in each structure. For example, when a threadlet tries to spawn a new threadlet, there may not be enough space in the threadlet queue for the new task. When this occurs, the Minnow engine should context switch so that other threadlets can complete, thus freeing up space. However, there must be space reserved for the context switch. Each threadlet must reserve an entry in the threadlet queue, context buffer, and load buffer for itself prior to being created, and entries can only be deallocated after the threadlet completes. Deadlock may also occur if the threadlet queue is full and all threadlets must spawn new threadlets to continue. This can be avoided if the max threadlet spawn depth is less than the threadlet queue size and if the initial threadlet reserves queue and buffer entries equal to the max spawn depth guaranteed by the programmer prior to execution. For example, in Figure 14, **prefetchTask()** must reserve at least two entries, one for itself and one for spawned **prefetchEdge()** tasks since **prefetchEdge()** does not spawn any additional tasks. These reserved entries are freed when **prefetchTask()** completes.

Name	Nodes	Edges	Est. Diam.	Largest Node	Size
USA-road-d.W	6.2M	15.1M	4,420	9	420 MB
r4-2e23	8.4M	33.6M	17	16	759 MB
rmat16-2e22	4.2M	67.1M	4	18,436,730	1152 MB
wikipedia-20051105	1.6M	19.8M	18	4,970	351 MB
wiki-Talk	2.4M	5.0M	9	100,022	150 MB
com-dblp-sym	426K	2.1M	21	343	58 MB
amazon-ratings	3.4M	11.5M	16	12,180	383 MB

Table 1. Evaluated graph inputs

Workload	Algorithm	Input	Cycles
SSSP	Single-Source Shortest Path	USA-road-d.W	4.6B
BFS	Breadth-First Search	r4-2e23	8.3B
G500	Breadth-First Search	rmat16-2e22	4.1B
CC	Connected Components	wikipedia-20051105	2.4B
PR	PageRank	wiki-Talk	10.7B
TC	Triangle Counting	com-dblp-sym	1.7B
BC	Bipartite Coloring	amazon-ratings	2.1B

Table 2. Benchmark configuration

5.4 Area Estimation

As Minnow leverages the existing cache subsystem for task storage and no additional storage for prefetched data, area cost is limited to the Minnow engine and the credit system. SRAM area numbers were generated for the data structures using an in-house memory compiler on a popular 28nm process technology. The additional 1-bit per L2 cacheline metadata can be stored in separate SRAM arrays. We assume Minnow will be augmented on a CMP with cores similar to Intel Skylake. Assuming 256KB L2 caches with 64B lines, 64-entry local queue, 128-entry threadlet queue, 2KB instruction and data memories, and 32-entry load buffer, the total area on 28nm is ~0.03mm^2, which scales to 0.008mm^2 on 14nm.

The area of the Minnow engine control unit is estimated using the area of the P54C-based Intel Quark, a simple x86 embedded processor with an in-order pipeline. Based on die photo analysis, we estimate a Quark core is 0.5 mm^2 on 32nm, which scales to 0.1 mm^2 on 14nm. We examined a Skylake-K die photo to estimate area overhead for adding Minnow. Each Skylake processor-router-L3 slice consumes 12.1 mm^2. Since each Skylake slice would be augmented with a Minnow engine, we estimate the total area overhead for Minnow to be less than 1% per slice.

6 Experimental Results

6.1 Evaluation Benchmarks

To evaluate Minnow, we chose a variety of parallel graph algorithms implemented in Galois. Not all algorithms tested benefit from priority ordering or have significant worklist bottlenecks, which help determine the minimum performance benefits of Minnow.

Single-Source Shortest Path (SSSP) calculates the shortest distance between a source node to all other nodes in a weighted graph. We implement non-blocking Delta-Stepping [35]. Work is prioritized by ascending distance.

Cores	64 Skylake-like cores 2.5GHz, x86-64 ISA
Branch Predictor	64Kb 5-table TAGE[50]
Reservation Station	97 entries, unified
Load-Store Queue	72 load entries, 56 store entries
Reorder Buffer	224 entries
L1 Instruction Cache	32KB per-core 4-way assoc, 3 cycle latency
L1 Data Cache	32KB per-core 8-way assoc, 4 cycle latency
L2 Cache	256KB per-core 8-way assoc, 7 cycle latency
L3 Cache	64MB, 2MB bank/core 16-way assoc, 27 cycle latency
NoC	8x8 mesh, 512-bits/cycle/link X-Y routing, 3 cycles/hop
Main mem	12-channel DDR4-2400 CL17
Minnow Engine	64-entry localQ 10 cycle localQ access latency 32-entry loadQ 4-cycle loadQ wakeup latency

Table 3. Configuration of baseline microarchitecture

Breadth-First Search (BFS, G500) traverses all nodes in an unweighted graph from a source node in breadth-first order. We implement non-blocking push-based BFS, and prioritize work by ascending hop distance. We test BFS running on a random mesh network (*BFS*), and a Kronecker graph generated as specified by the Graph500 [37] benchmark (*G500*).

Connected Components (CC) finds all connected components in an undirected graph using a standard non-blocking minimum-label propagation algorithm [40]. Work is prioritized by ascending component ID.

PageRank (PR) estimates the relative importance of nodes in a directed graph [42]. We implement a non-blocking, data-driven, push-based algorithm presented in [58], with work prioritized by descending node residual.

Triangle Counting (TC) counts the number of triangles, sets of three mutually adjacent nodes, in a graph. We implement a *node-iterator-hashed* algorithm [48], which uses binary search to determine if two nodes share a common edge. TC does not benefit from priority ordering or dynamically generate new work, and its tasks do not require atomics.

Bipartite Coloring (BC) determines if nodes can be colored with one of two colors such that no two adjacent nodes share the same color. A non-blocking algorithm was implemented in which each node propagates colors to its neighbors. BC does not benefit from priority ordering.

6.2 Experimental Setup

Minnow was implemented within a modified version of ZSim, a trace-driven, Pin-based microarchitectural simulator correlated against Intel Westmere [45]. For our baseline shown in Table 3, we updated ZSim's OOO core model to represent the Intel Skylake microarchitecture, and improved overall model accuracy. We added details such as cache port contention and network contention, and addressed atomic instruction model inaccuracy. The simulated system was scaled to 64 cores on a mesh network with similar parameters as Intel Knights Landing, keeping a core to memory bandwidth ratio similar to modern server designs.

We evaluated our design on the benchmarks and inputs shown in Tables 1 and 2 implemented in Galois 2.2.1 [43], selecting relatively realistic inputs sized such that each each single-threaded baseline runs for billions of cycles. Input graphs are stored in memory in standard CSR format, with 32B nodes (64B for TC) and 16B edges. Input sizes range from 150MB to 1GB except for **com-dblp-sym** used in TC. A small input had to be selected for TC due to execution time, with the unfortunate side effect of fitting within LLC. We extended Galois with a new worklist and graph class that implements Minnow support. We also addressed several Galois bottlenecks, which improved overall scalability in both our modified Galois and our Galois software baseline.

6.2.1 Galois Framework Optimizations

The latest version of Galois (2.2.1) scales poorly at high thread counts. While Galois framework performance is irrelevant to the design of Minnow and worklist-directed prefetching, improving Galois was required to get good scaling for our results. We introduced the following optimizations to Galois to address two key bottlenecks.

Firstly, OBIM was tuned for the authors' specific host platform which had a small number of cores per socket, resulting in their per-socket scheduler scaling poorly at higher thread counts. We overrode the Galois hardware topology detection mechanism into treating our simulated 64-core platform as an 8-socket, 8-core platform, resulting in significantly improved scalability when using OBIM.

Secondly, the Galois framework cannot properly load balance when some tasks are abnormally large since it cannot break up tasks into smaller subtasks to be processed in parallel. This is common in power-law graphs, where some nodes have vastly more edges than average. For example, **rmat16-2e22** contains a node with 18.4M edges, 27% of all edges in the graph. Therefore, under Amdahl's Law, the maximum speedup cannot exceed 3.65x.

We address this with *task splitting*, a technique to break up large tasks over a defined threshold into smaller subtasks that can be processed in parallel. Task splitting is applicable to graph algorithms where edges can be processed in any order, as long as edge updates are atomic. Tasks are augmented with an edge start and end index representing the subset of edges to process. Our implementation splits tasks when adding tasks to the worklist when the number of outgoing edges exceeds 10K. Task splitting overhead is negligible since only the few largest nodes need to be split.

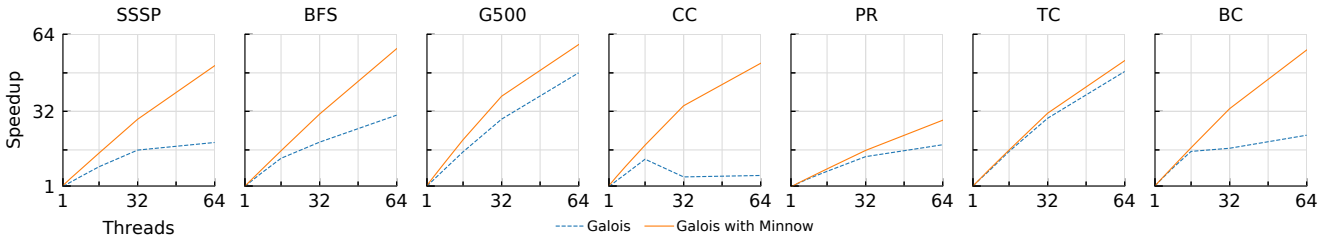

Figure 15. Galois scalability with and without Minnow, relative to optimized serial baseline

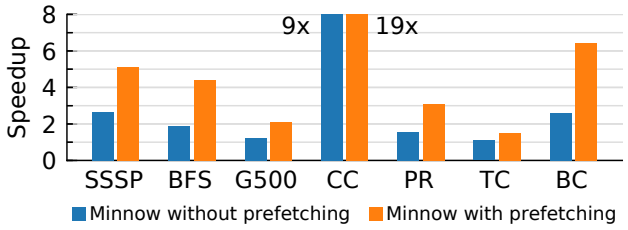

Figure 16. Overall Minnow speedup vs software baseline

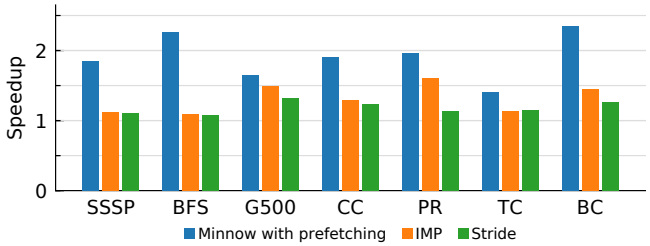

Figure 17. 16-thread Minnow prefetching speedup vs. IMP normalized to Minnow without prefetching

6.3 Experimental Results

Figure 16 shows Minnow overall speedup compared to the optimized Galois software baseline at 64-threads. Overall, Minnow achieves major performance gains, with average speedups of 2.96x for Minnow without prefetching and 6.01x with both Minnow and worklist-directed prefetching enabled. TC shows the least speedup since it is the least-bottlenecked by the worklist and memory accesses. TC neither dynamically produces new work nor benefits from priority ordering, resulting in minimal worklist overhead. As a result, TC does not show much improvement when using Minnow without prefetching. The graph input selected for TC fits within the simulated CMP last-level cache, reducing but not eliminating the memory latency bottleneck due to the long L3 latency. With worklist-directed prefetching enabled, TC shows an overall 1.53x speedup. Speedup gains for Minnow with prefetching disabled do not perfectly correlate with the worklist bottleneck shown in Figure 5 due to how worklist overhead was measured. The measured worklist overhead does not include portions of Galois worklist enqueue overhead that are no longer called when using Minnow.

6.3.1 Minnow Scalability

Figure 15 shows Galois speedup from 1 to 64 threads with and without Minnow relative to an optimized serial software baseline. To isolate the impact of worklist offloading, Minnow was evaluated with worklist-directed prefetching disabled. We found that Galois worklists were faster than C++ STL queues and priority queues even for serial operation, so our serial baseline still uses Galois but has atomics removed. After the optimizations applied in Section 6.2, the Galois without Minnow scales well up to 32 threads on many workloads with the exception of CC, but runs into significant bottlenecks at higher thread counts. CC is severely worklist-bottlenecked, resulting in performance slowdowns with more than 16 threads. Minnow improves scalability for all workloads and enables CC to scale past 16 threads. TC is the least worklist-bottlenecked and thus sees the least improvement from worklist offload.

6.3.2 Worklist-Directed Prefetching Analysis

Figure 18 shows the impact of worklist-directed prefetching on L2 misses per kilo-instruction (MPKI) as the number of prefetch credits is swept from 1 to 256. With prefetching disabled, all benchmarks except for TC have MPKI values greater than 20, resulting in a significant performance bottleneck. MPKI rates decrease as the number of prefetch credits increases since the prefetcher is allowed to run further ahead of the execution stream. Prefetching too aggressively results in L2 cache thrashing in several benchmarks, as shown by an increase in MPKI. L2 miss rate is minimized between 32 and 128 credits, resulting in less than 1 MPKI for all benchmarks except for SSSP. In SSSP, the prefetcher cannot run far enough ahead of the execution stream to hide all L2 misses.

Figure 19 shows prefetching speedup at 64 threads relative to a design using Minnow with prefetching disabled. All applications and inputs show speedup with prefetching, ranging from 1.39x for TC to 2.47x for BC. Prefetching speedup is closely correlated with MPKI reduction. There are diminishing gains around 32 to 64 credits, and G500 shows significant performance degradation past its optimal credit threshold. **r16-2e22** is a scale-free graph, with some nodes having millions of edges. At high credit counts, G500 runs too far ahead, completely overflowing the cache hierarchy, thus resulting in lower performance.

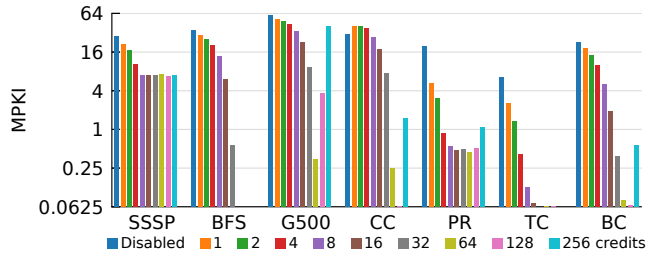

Figure 18. Minnow prefetching effect on L2 misses per kilo-instruction

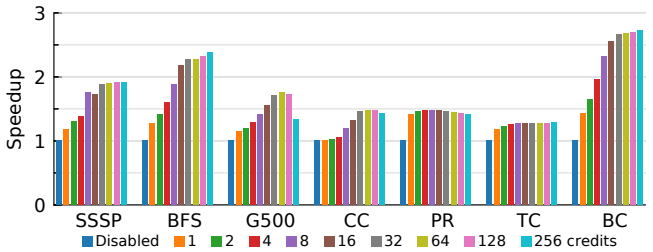

Figure 19. Minnow prefetching speedup vs. credits

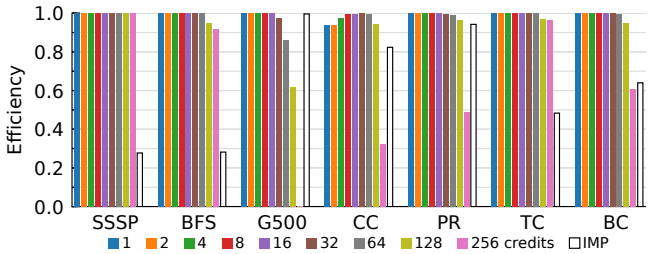

Figure 20. Minnow prefetch efficiency

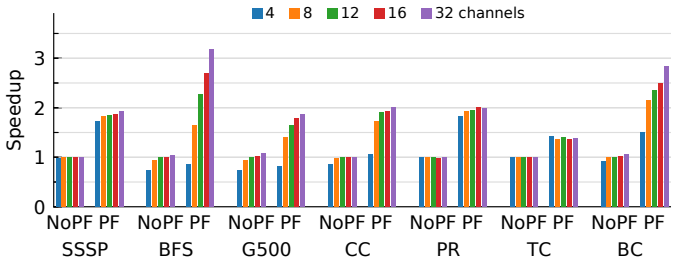

Figure 21. Minnow 64-thread speedup vs. memory channels

6.3.3 IMP Comparison

Minnow worklist-directed prefetching is compared to a standard stride prefetcher and IMP [59], a state-of-the-art table-based graph pointer prefetcher, in Figure 17. IMP extends a classic stride based table to support indirect data structures by assuming some arrays in memory contain indices to other tables, prefetching array-of-pointers with the form *A[B[i]]*. The IMP configuration and table sizes as evaluated in [59] were suboptimal for our workloads and microarchitectural baseline. IMP was re-tuned to maximize performance, quadrupling buffer sizes to eliminate prefetcher table capacity misses and selecting the best prefetch distance (4) for our workloads. While IMP performs well on several benchmarks, it has no feedback mechanism like our credit system to throttle prefetches and can only prefetch addresses once the processor is already accessing the indirect index table, resulting in similar performance as basic stride prefetching with the exception of G500, PR, and TC.

IMP uses prefetch distance similar to stride prefetchers, in which the prefetcher reacts to specific load instruction addresses by prefetching the next data element by some offset. This offset results in the first few edges in a graph never being prefetched. In the worst case, if the prefetched graph node has equal to or fewer edges than the prefetch distance, then every issued prefetch request will be incorrect since the distance is larger than the size of the array. Mesh-type graph inputs such as **USA-road-d.W** used in SSSP and **r4-2e23** used in BFS have low edge counts per node, resulting in IMP being unable to prefetch these graphs. Worklist-directed prefetching is proactive, comprehensively prefetching tasks well before use rather than reacting on first use.

The effect of prefetch aggressiveness on L2 caching behavior is explored in Figure 20. Prefetch efficiency is defined as ratio of the number of prefetched lines used before eviction vs. the total number of prefetch fill requests. At low prefetch credits, there are no cache contention issues and almost every prefetched line brought into the L2 is used. As prefetch credits increase, for G500, CC, PR, and BC, cache contention increases and performance degrades. Setting the number of initial prefetch credits to 32 results in near-maximum performance and more than 99% prefetch efficiency for all benchmarks and inputs evaluated. In contrast, while IMP is extremely efficient for a traditional prefetcher, its efficiency is significantly lower than worklist-directed prefetching using the proposed credit-based scheme.

6.3.4 Scaling CMP Bandwidth

Figure 21 shows speedup relative to a 12 memory channel design with and without worklist-directed prefetching. When prefetching is disabled, all benchmarks are significantly memory latency-bound, only resulting in significant performance penalties after the number of memory channels is reduced to 4 for 64 cores. Since the graph input used for TC fits within LLC, TC is insensitive to the number of memory channels regardless of prefetching. With prefetching enabled, Minnow can utilize the available memory bandwidth by sustaining a large number of memory operations in flight, converting several benchmarks (BFS, G500, BC) from being memory latency-bound to bandwidth-bound. Minnow can potentially enable general-purpose CMPs to close the gap with throughput architectures on memory-bound workloads while still providing good single-threaded performance.

7 Related Work

Researchers have proposed hardware accelerated worklists for both CPU [27] [46] [29] and GPU [24], and even hardened graph processors [14], but they do not exploit priority ordering. Minnow removes the centralized global structures found in Carbon by moving them to software and leveraging the existing cache hierarchy. Hardware worklists [27] [29] [24] require considerable area for buffering work items and have hard limits for storage. IsoNet [29] does not discuss this issue and simply uses large-enough buffering for their test cases. Carbon [27] throws exceptions to trigger handlers to explicitly spill/fill from its task queues. HWWL [24] adds a virtualization unit that spills/fills from a dedicated region in GPU memory allocated during initialization. Minnow is *software-mostly* like ADM, but removes the custom message-passing protocol and executes manager threads on Minnow engines instead of cores. By accessing its core's cache hierarchy and TLB, Minnow engines allow the most flexibility in worklist spills/fills. These modifications reduce complexity, allow for partial priority scheduling, and improve generality, while potentially enabling Minnow to accelerate shared structures beyond worklists. Prior hardware worklists also cannot support worklist-directed prefetching.

State-of-the-art streaming prefetchers [39] [18] maintain large buffers for correlation prefetching. These schemes work poorly for graph workloads [59]. Recent hardware prefetch schemes provide large speedups if workloads match targeted access patterns. Widx [26] accelerates hash and pointer traverse operations. Tesseract [2] is a processor-in-memory for graph workloads that augments its message passing protocol with a prefetch target address and prefetch buffer. IMP [59] prefetches array-of-pointers with the form *A[B[i]]*. [3] describes an L1 prefetcher for graphs and unordered worklists with the form *visited[edge[node[worklist[i]]]]*. The assumption of a worklist in linear virtual space and the visited bit make a parallel partial priority worklist impractical. In contrast, Minnow has full control of the worklist, and therefore can launch prefetches without monitoring address ranges. Minnow is programmable, enabling full data structure and access pattern flexibility for prefetching, and supports parallel priority worklists. In addition, our credit-based prefetch timeliness scheme results in consistently high prefetch utilization at 98% with 32 credits for all workloads.

There exists a large body of work exploiting multiple threads or cores to improve single-threaded performance. Helper threads [6] [11] target single-threaded programs running on CMPs, utilizing otherwise-idle SMT threads [8] or cores [21] [22] to run subroutines to prefetch to shared cache. These may be generated by the user [8], compiler [32], or dynamically at runtime [36]. However, previous work only targeted single-threaded applications since helper threads require idle hardware contexts, and has a high area and power cost. Researchers have proposed using lower power cores through DVFS [23] [19] and smaller OOO cores [56]. Decoupled access-execute [52] splits a single-threaded program into compute and memory instruction streams which communicate through shared FIFOs. Assisted execution [11] spawns nanothreads on cache misses to perform stride prefetching. SSMT [6] spawns microthreads for prefetching and branch prediction. DDMT [44] forks program subsets into separate threads and integrates results back to the main thread. Slipstream processors [55] are paired cores where a core executing on a program subset runs ahead feeding results to a core executing a redundant stream. Runahead execution [38] [15] unblocks the pipeline to prefetch further ahead of the execution stream during cache misses. In contrast, worklist-directed prefetching exploits scheduling information and user knowledge of graph access patterns. Rather than executing on general purpose cores and competing with the main thread for resources, Minnow's prefetching threads are run on specialized hardware optimized for memory throughput. Minnow exploits the task-based programming model to spawn prefetches much farther ahead of the main thread, resulting in significant speedups.

8 Conclusion

We presented Minnow, which accelerates task-based parallel graph workloads by augmenting each core in a CMP with a memory throughput-oriented Minnow engine. Minnow engines reduce the synchronization bottleneck by offloading the software priority worklist, and reduce the memory latency bottleneck with worklist-directed prefetching, launching helper threads in response to worklist scheduling decisions. Minnow engines enable high memory throughput at low area by spawning and executing many threadlets in parallel, which explicitly encode memory dependency information. By spawning helper threads at the earliest possible time with custom hardware designed for helper thread throughput, our technique can prefetch far ahead, requiring throttling. Minnow's credit-based feedback system dynamically returns credits as prefetched data is consumed. Our results show that Minnow can greatly improve scalability and eliminate almost all cache misses, resulting in an average speedup of 6.01x over an optimized software baseline for only 1% area overhead. We plan on extending Minnow to accelerate other classes of irregular workloads in the future.

Acknowledgments

The authors would like to thank the anonymous reviewers, Joseph Greathouse, Christopher Rossbach, Stephen Pruett, and Steven Reinhardt for their valuable feedback. This material is based on work supported by the National Science Foundation under Grant No. 1205721 and CNS-1111766. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the views of the sponsors or authors' employers.

References

[1] 2011. Intel® 64 and IA-32 Architectures Software DeveloperâĂŹs Manual. *Volume 3: System Programming Guide* 3B (2011).

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2749469.2750386

[3] Sam Ainsworth and Timothy M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2925426.2926254

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera computer system. In *ACM SIGARCH Computer Architecture News*, Vol. 18. ACM, 1–6.

[5] S. Beamer, K. Asanovic, and D. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *2015 IEEE International Symposium on Workload Characterization*. 56–65. https://doi.org/10.1109/IISWC.2015.12

[6] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. 1999. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture*. 186–195. https://doi.org/10.1109/ISCA.1999.765950

[7] Derek Chiou, Boon S Ang, Robert Greiner, James C Hoe, Michael J Beckerle, James E Hicks, Andy Boughton, et al. 1995. StarT-NG: Delivering seamless parallel computing. In *EURO-PAR'95 Parallel Processing*. Springer, 101–116.

[8] J. D. Collins, Hong Wang, D. M. Tullsen, C. Hughes, Yong-Fong Lee, D. Lavery, and J. P. Shen. 2001. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings 28th Annual International Symposium on Computer Architecture*. 14–25. https://doi.org/10.1109/ISCA.2001.937427

[9] William J Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J Knight, et al. 2003. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. ACM, 35.

[10] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. 2010. The YouTube Video Recommendation System. In *Proceedings of the Fourth ACM Conference on Recommender Systems (RecSys '10)*. ACM, New York, NY, USA, 293–296. https://doi.org/10.1145/1864708.1864770

[11] Michel Dubois and Y Song. 1998. Assisted execution. *University of Southern California CENG Technical Report* 98, 25 (1998).

[12] LR Ford. 1956. Network Flow Theory. (1956).

[13] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

[14] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[15] Milad Hashemi and Yale N. Patt. 2015. Filtered Runahead Execution with a Runahead Buffer. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 358–369. https://doi.org/10.1145/2830772.2830812

[16] Kirsten Hildrum and Philip S Yu. 2005. Focused community discovery. In *Data Mining, Fifth IEEE International Conference on*. IEEE, 4–pp.

[17] B. Hong and Z. He. 2011. An Asynchronous Multithreaded Algorithm for the Maximum Network Flow Problem with Nonblocking Global Relabeling Heuristic. *IEEE Transactions on Parallel and Distributed Systems* 22, 6 (June 2011), 1025–1033. https://doi.org/10.1109/TPDS.2010.156

[18] Akanksha Jain and Calvin Lin. 2013. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 247–259. https://doi.org/10.1145/2540708.2540730

[19] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. 2014. Fix the Code. Don'T Tweak the Hardware: A New Compiler Approach to Voltage-Frequency Scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 262, 11 pages. https://doi.org/10.1145/2544137.2544161

[20] Daniel R Johnson, Matthew R Johnson, John H Kelm, William Tuohy, Steven S Lumetta, and Sanjay J Patel. 2011. Rigel: A 1,024-core single-chip accelerator architecture. *IEEE Micro* 31, 4 (2011), 30–41.

[21] Changhee Jung, Daeseob Lim, Jaejin Lee, and Y. Solihin. 2006. Helper thread prefetching for loosely-coupled multiprocessor systems. In *Proceedings of the 20th International Parallel Distributed Processing Symposium*. 10–. https://doi.org/10.1109/IPDPS.2006.1639375

[22] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2011. Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 393–404. https://doi.org/10.1145/1950365.1950411

[23] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. 2012. Underclocked Software Prefetching: More Cores, Less Energy. *IEEE Micro* 32, 4 (July 2012), 32–41. https://doi.org/10.1109/MM.2012.54

[24] Ji Yun Kim and C. Batten. 2014. Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists. In *Proceedings of the 47th Annual International Symposium on Microarchitecture*. 75–87. https://doi.org/10.1109/MICRO.2014.24

[25] Donald E Knuth. 1977. A generalization of Dijkstra's algorithm. *Inform. Process. Lett.* 6, 1 (1977), 1–5.

[26] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 468–479. https://doi.org/10.1145/2540708.2540748

[27] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 162–173. https://doi.org/10.1145/1250662.1250683

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[29] Junghee Lee, Chrysostomos Nicopoulos, Hyung Gyu Lee, Shreepad Panth, Sung Kyu Lim, and Jongman Kim. 2013. Isonet: Hardware-based job queue management for many-core architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 21, 6 (2013), 1080–1093.

[30] Charles E. Leiserson and Tao B. Schardl. 2010. A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 303–314. https://doi.org/10.1145/1810479.1810534

[31] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2011. Priority queues are not good concurrent priority schedulers. *The University of Texas at Austin, Department of Computer Sciences, Tech. Rep. TR-11-39*

(2011).

[32] Steve S.W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoflehner, Daniel Lavery, and John P. Shen. 2002. Post-pass Binary Adaptation for Software-based Speculative Precomputation. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/512529.512544

[33] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in Parallel Graph Processing. *Parallel Processing Letters* 17, 01 (2007), 5–20. https://doi.org/10.1142/S0129626407002843 arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0129626407002843

[34] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[35] U. Meyer and P. Sanders. 2003. Delta-stepping: A Parallelizable Shortest Path Algorithm. *J. Algorithms* 49, 1 (2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[36] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. 2001. Slice-processors: An Implementation of Operation-based Prediction. In *Proceedings of the 15th International Conference on Supercomputing (ICS '01)*. ACM, New York, NY, USA, 321–334. https://doi.org/10.1145/377792.377856

[37] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray UserâĂŹs Group (CUG)* (2010).

[38] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, Washington, DC, USA, 129–. http://dl.acm.org/citation.cfm?id=822080.822823

[39] Kyle J. Nesbit and James E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA '04)*. IEEE Computer Society, Washington, DC, USA, 96–. https://doi.org/10.1109/HPCA.2004.10030

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP '13)*. 456–471. https://doi.org/10.1145/2517349.2522739

[41] Michael D Noakes, Deborah A Wallach, and William J Dally. 1993. The J-machine multicomputer: an architectural evaluation. *ACM SIGARCH Computer Architecture News* 21, 2 (1993), 224–235.

[42] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the Web. (1999).

[43] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 12–25. https://doi.org/10.1145/1993498.1993501

[44] A. Roth and G. S. Sohi. 2001. Speculative data-driven multithreading. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. 37–48. https://doi.org/10.1109/HPCA.2001.903250

[45] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[46] Daniel Sanchez, Richard M Yoo, and Christos Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. In *ACM Sigplan Notices*, Vol. 45. ACM, 311–322.

[47] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 979–990. https://doi.org/10.1145/2588555.2610518

[48] Thomas Schank. 2007. Algorithmic aspects of triangle-based network analysis. (2007).

[49] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the Cost of Atomic Operations on Modern Architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 445–456. https://doi.org/10.1109/PACT.2015.24

[50] André Seznec and Pierre Michaud. 2006. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction Level Parallelism* 8 (2006), 1–23.

[51] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[52] James E Smith. 1982. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 10. IEEE Computer Society Press, 112–119.

[53] Srikanth T. Srinivasan and Alvin R. Lebeck. 1998. Load Latency Tolerance in Dynamically Scheduled Processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 31)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 148–159. http://dl.acm.org/citation.cfm?id=290940.290973

[54] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[55] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. 2000. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices* 35, 11 (2000), 257–268.

[56] Bharath Narasimha Swamy, Alain Ketterlin, and André Seznec. 2014. Hardware/Software Helper Thread Prefetching on Heterogeneous Many Cores. In *Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '14)*. IEEE Computer Society, Washington, DC, USA, 214–221. https://doi.org/10.1109/SBAC-PAD.2014.39

[57] A. Tumeo and J. Feo. 2015. Irregular Applications: From Architectures to Algorithms [Guest editors' introduction]. *Computer* 48, 8 (Aug 2015), 14–16. https://doi.org/10.1109/MC.2015.233

[58] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S Dhillon, and Keshav Pingali. 2015. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. In *Euro-Par 2015: Parallel Processing*. Springer, 438–450.

[59] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect Memory Prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 178–190. https://doi.org/10.1145/2830772.2830807