Optimizing Indirect Memory References with `milk`

Vladimir Kiriansky
MIT CSAIL
vlk@csail.mit.edu

Yunming Zhang
MIT CSAIL
yunming@csail.mit.edu

Saman Amarasinghe
MIT CSAIL
saman@csail.mit.edu

ABSTRACT

Modern applications such as graph and data analytics, when operating on real world data, have working sets much larger than cache capacity and are bottlenecked by DRAM. To make matters worse, DRAM bandwidth is increasing much slower than per CPU core count, while DRAM latency has been virtually stagnant. Parallel applications that are bound by memory bandwidth fail to scale, while applications bound by memory latency draw a small fraction of much-needed bandwidth. While expert programmers may be able to tune important applications by hand through heroic effort, traditional compiler cache optimizations have not been sufficiently aggressive to overcome the growing DRAM gap.

In this paper, we introduce **`milk`** — a C/C++ language extension that allows programmers to annotate memory-bound loops concisely. Using optimized intermediate data structures, random indirect memory references are transformed into batches of efficient sequential DRAM accesses. A simple semantic model enhances programmer productivity for efficient parallelization with OpenMP.

We evaluate the Milk compiler on parallel implementations of traditional graph applications, demonstrating performance gains of up to 3×.

1. INTRODUCTION

Memory bottlenecks limit the performance of many modern applications such as in-memory databases, key-value stores, graph analytics, and machine learning. These applications process large volumes of in-memory data since DRAM capacity is keeping pace with Moore's Law. Indirect memory references in large working sets, however, cannot be captured by hardware caches. Out-of-order execution CPUs also cannot hide the high latency of random DRAM accesses, and most cycles are wasted on stalls. Compiler cache optimizations also cannot statically capture locality in irregular memory accesses even when loop-level locality exists.

Any execution order is often acceptable, yet performance dramatically depends on the access pattern. The following read-modify-write loop is typical in many domains, e.g., graph analysis, image processing, and query planning:

```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]++;
```

On contiguous DRAM-resident inputs this loop achieves 18% of peak DRAM bandwidth of Intel Haswell CPUs; if inputs are always sorted we can optimize to remove atomics and reach 54% of peak bandwidth. On uniform random inputs, however, this loop's effective bandwidth is only 2%.

For this loop on random inputs, the Milk compiler achieves 4× end-to-end speedup. Our compiler transformations dynamically partition the index values in `d` to capture all available locality in accesses to `count`. Each partition accesses a subset of memory that fits in cache. In addition, synchronization is eliminated since threads process disjoint partitions. Just adding **`milk`** clauses to the OpenMP directives enables these optimizations.

To achieve high performance and low overhead, Milk's *DRAM-conscious Clustering* transforms loops into three logical phases: Collection, Distribution, and Delivery. In Collection, dynamically generated indices of indirect accesses (e.g., simply `d[i]` above) are first collected in cache local buffers. In Distribution, indices are written to DRAM resident partitions using efficient sequential DRAM access. In Delivery, each partition's deferred updates are read from DRAM and processed, along with dependent statements.

These three logical phases are similar to inspector-executor style optimizations [18, 32, 40]; in Milk, however, to eliminate materialization of partitions and conserve DRAM bandwidth, the phases are fused and run as coroutines. Prior research either focused on expensive preprocessing that resulted in net performance gain only when amortized over many loop executions, or explored simple inspection for correspondingly modest gains. In contrast, our transformations are well optimized to pay off within a single execution — as required in real applications that use iterative algorithms with dynamically changing working sets or values.

The Milk execution model is a relaxed extension of the bulk-synchronous parallel (BSP) model [50], in which communicating processors work on local memory during a *superstep* before exchanging data at a global barrier. Milk virtual processors access randomly only one DRAM-resident cache line per superstep, in addition to sequential memory or cache-resident data. All indirect references are deferred to the next superstep. A Milk superstep may encompass loop nests that include billions of deferred references. Ref-

PACT '16 September 11–15, 2016, Haifa, Israel

ACM ISBN 978-1-4503-4121-9/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2967938.2967948

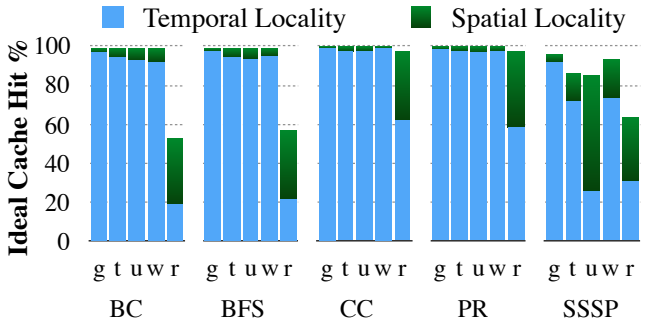

Figure 1: Temporal and spatial locality on an infinite cache with 64-byte lines of indirect memory references per superstep of **B**etweenness **C**entrality, **B**readth-**F**irst **S**earch, **C**onnected **C**omponents, **P**age**R**ank, and **S**ingle-**S**ource **S**hortest **P**aths. Synthetic graphs with V=128M vertices and avg. degree d=16: power law Graph500 (**g**) and uniform (**u**); and real-world graphs: US roads V=24M, d=2.4 (**r**); Twitter V=62M, d=24 (**t**); weblinks V=51M, d=39 (**w**).

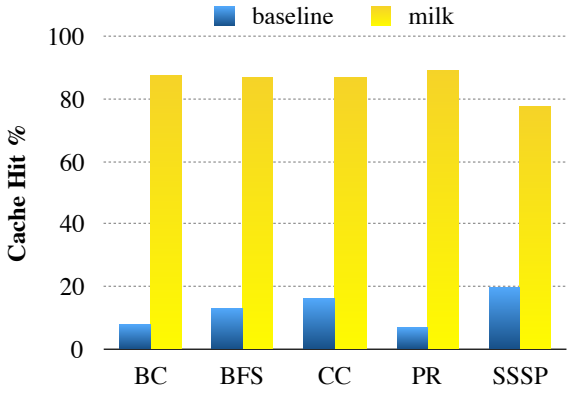

Figure 2: Cache hit rates of demand reads on 256 KB L2, 8 MB shared L3 [2], for V=32M, d=16(uniform).

erences targeting the same cache line are grouped to maximize locality, and are processed by a single virtual processor to eliminate synchronization. This abstract programming model is similar to MapReduce [16] with discrete phases of *Map* and *Reduce* for processing references to distinct memory locations. Milk programs, however, allow programmers to continue thinking in terms of individual references and to maintain traditional serial or OpenMP syntax, usually without semantic change.

We investigate the total locality available in applications, compare to locality captured by hardware, and show Milk's effectiveness. To find an upper bound on locality within a Milk superstep, we use an ideal cache model. An ideal cache would not incur cache capacity or cache conflict misses, or true or false sharing coherence misses on a multiprocessor system. The only memory demands should be compulsory cache misses and output write-backs. (We assume all data is written to memory between supersteps, therefore reuses across supersteps are counted as compulsory misses.)

Since compulsory cache misses, temporal locality, and spatial locality are program- and data-dependent, we instrumented the indirect memory references of commonly used [11, 36,43,47] graph applications with state-of-the-art implementations in the Graph Algorithm Platform Benchmark Suite (GAPBS) [10]. The GAPBS datasets include public samples of real graphs (*Twitter*, *USA road*, and *web crawl*), and synthetic graphs (*uniform* and *power law* [13, 21]) to compare sensitivity to graph degree distributions and diameters. We describe the applications in more detail in Section 6, where we also show their handwritten in OpenMP critical loops.

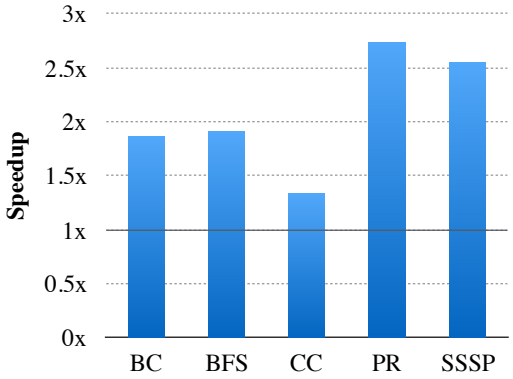

Figure 3: Milk overall speedup. 8MB L3, V=32M, d=16(u).

Figure 1 shows that, on an ideal cache, the total locality within a superstep is close to 100% for most graphs and applications, and above 50% for all. Temporal locality on an infinite cache (i.e., reuse of the same address) captures more than 90% of accesses for high-degree low-diameter graphs. However, for the road graph and for SSSP, harnessing spatial locality (i.e., capturing reuse of 64-byte cache lines) is important. Unlike hit rates on the ideal infinite cache, the cache hit rates observed on real hardware are much lower, and most cycles are wasted on memory stalls.

Figure 2 shows the low cache hit rates of real hardware are improved by 4× with Milk. On a synthetic graph with 32 M vertices, most applications have greater than 128 MB total working sets, and little locality is captured by the 8 MB shared L3 cache and 256 KB L2 per core on a Haswell CPU [2]. To isolate indirect memory reads during full program execution, we use hardware performance counters that track non-prefetch demand reads that hit L2 or L3 caches. In the baseline 80–90% of all L1 misses require a DRAM access, while with Milk they are served by L2 and L3 caches. Figure 3 shows that these improved hit rates translate to end-to-end speedups, ranging from 1.3× to 2.7×.

The rest of this paper is organized as follows: In Section 2, we explore memory system complexities and opportunities in more detail. We introduce the Milk programming model and Milk compiler and runtime library design in Section 3, `milk` syntax and semantic extensions in Section 4, and implementation details of Milk optimizations in Section 5. We demonstrate Milk expressiveness on parallel graph applications in Section 6, and we evaluate performance in Section 7. Section 8 surveys related work and Section 9 concludes.

2. BACKGROUND

We now examine the organization of DRAM to show why random memory references are often 10–20× slower than sequential memory accesses. We also explain why hardware approaches for hiding long latency penalties are ineffective.

2.1 DRAM Organization

The memory hierarchy is optimized for sequential accesses, often at the expense of random accesses, at all levels: DRAM chip, DRAM channel, memory controllers, and caches.

Sequential DRAM reads benefit from spatial locality in DRAM chip *rows*. Before a DRAM read or write can occur, a DRAM row – typically 8–16KB of data – must be destructively loaded (*activated*) into an internal buffer for its contents to be accessed. Consecutive read or write requests to the same row are limited only by DRAM channel bandwidth, thus sequential accesses take advantage of spatial locality in row accesses. In contrast, random accesses must find independent request streams to hide the high latency of a row cycle of different banks (*DRAM page misses*), or worse the additional latency of accessing rows of the same bank (*DRAM page conflicts*).

Hardware memory controllers reorder requests to minimize page misses, and other high-latency request sequences to minimize bus turnarounds and read-to-write transitions. But requests can be rescheduled only within a short window of visibility, e.g., 48 cache lines per memory channel [38] shared among cores. Requests are equally urgent from hardware perspective, and cannot be reordered outside of this window.

DRAM interfaces are also optimized for sequential accesses. Modern CPUs transfer blocks of at least 64 bytes from DRAM. Double Data Rate (DDR) DRAM interfaces transfer data on both rising and falling edges of a clock signal: e.g., a DDR3-1600 [24] chip operating at 800 MHz delivers a theoretical bandwidth of 12.8 GB/s. Current generation CPUs use DDR3/DDR4 interfaces with burst transfers of 4 cycles (mandatory for DDR4), leading to a 64-byte minimum transfer for a 64-bit DRAM. Even though larger transfers would improve DRAM power, reliability, and peak performance, this size coincides with current cache line sizes.

Sequential memory accesses benefit significantly from large cache lines, as they have good spatial locality and can use all of the data within each line. By contrast, random memory accesses use only a small portion of the memory bandwidth consumed. In many applications with irregular memory accesses, each DRAM access uses only 4–8 bytes out of cache lines transferred – 64 bytes transferred for reads, and 128 bytes for writes (a cache-line read and write-back) – netting 3–6% effective bandwidth.

Applications that are stalling on memory but do not saturate the memory bandwidth are bound by memory latency. Random DRAM device latency has been practically stagnant at ∼50 ns for the last decade [41, 42]. A DRAM request at peak bandwidth has 100× worse latency than an L1 cache lookup, and in the time it takes to wait for memory, a SIMD superscalar core can execute ∼5,000 64-bit operations.

2.2 CPU Request Reordering

CPU out-of-order execution allows multiple long delay memory accesses to be handled but only up to the limits of small hardware structures. By Little's Law, memory throughput is the ratio of outstanding memory requests over DRAM latency. The visible effect of either low memory level parallelism or high DRAM latency is underutilized memory bandwidth. Such "latency bound" programs perform well in-cache, and may have no explicit data dependencies, but become serialized on large inputs.

Hardware prefetchers for sequential accesses increase memory level parallelism with more outstanding DRAM requests. Therefore, sequential reads and writes achieve higher DRAM bandwidth than random accesses. Current hardware has no prefetchers for random requests.

Current CPUs can handle 10 demand requests per core (Line Fill Buffers between L1 and L2) [1, 11] as long as all requests fit within the out-of-order execution window. The effective Memory Level Parallelism (MLP) is most constrained by the capacity limits of resources released in FIFO order, e.g., reorder buffer, or load and store buffers. A loop body with a large number of instructions per memory load may reduce the effective parallelism, e.g., a 192 *micro-op* entry reorder buffer (ROB) must fit all micro-ops since the first outstanding non-retired instruction. Branch misprediction, especially of hard to predict branches that depend on indirect memory accesses further reduce the effective ROB size. Finally, atomic operations drain store buffers [44] and reduce Instruction Level Parallelism (ILP). While hardware mechanisms, including memory controllers and out-of-order schedulers, can reorder only a limited window of tens of operations, Milk efficiently orchestrates billions of accesses.

3. DESIGN

Given the inefficiency of random DRAM references and the limitations of hardware latency-hiding mechanisms, in order to harvest locality beyond hardware capabilities, the Milk compiler uses a software based approach to plan all memory accesses. To use the Milk compiler, programs must fit the Milk execution model and have **`milk`**[1] annotations.

Milk achieves significant performance improvement by reordering the memory accesses for improved cache locality. The reordered memory references maximize temporal locality by partitioning all indirect references to the same memory location within the cache capacity. The planned accesses also improve spatial locality by grouping indirect references to neighboring memory locations. Furthermore, Milk avoids true sharing, false sharing, and expensive synchronization overheads by ensuring only one core writes to each cache line.

However, naively reorganizing indirect references can add non-trivial overhead. Milk keeps all additional bandwidth low by using only efficient sequential DRAM references. Although data transformations require an investment of additional CPU cycles and sequential DRAM bandwidth, we show how to minimize these overheads with *DRAM-conscious Clustering*.

In order for the Milk compiler to perform the optimization automatically, users must annotate indirect accesses in parallel OpenMP loops with a **`milk`** clause, which is sufficient for simple loops like Figure 4a. Explicit **`milk`** directives can select indirect references that should be deferred, along with their context (see line 12 in Figure 4b). Optional *combiner* functions allow programmers to summarize the combined effects of updates targeting the same address. Section 4 describes **`milk`**'s syntax in more detail.

The Milk execution model is similar to MapReduce [16]. We do not offer a *Map* interface, however, since efficient iteration over in-memory data structures can use domain-specific knowledge (e.g., incoming vs. outgoing neighbor traversal in graph processing, 2-D loop nests for image pro-

[1] Milk's **`milk`** is an homage to Cilk's **`cilk`** [19].

```
01 void f(int a[], int ind[], int x) {
02 #pragma omp parallel for milk
03 for(int i=0; i<n; i++)
04     a[ind[i]] = x;
05 }
```

(a) Implicit Indirect References with **milk**.

```
06 void g(float a[], int ind[],
07        float b[]) {
08 #pragma omp parallel for milk
09   for(int i=0; i<n; i++) {
10      float value = b[i];
11      int index = ind[i];
12 #pragma milk pack(value:+) tag(index)
13 #pragma omp atomic if(!milk)
14      a[index] += value;
15   }
16 }
```

(b) Explicit Indirect References with **milk**.

```
17 .milk.outlined.f(int _tag,
18                    Context *c) {
19   c->a[_tag] = c->x;
20 }
21 .milk.outlined.g(int index,
22                    float value,
23                    Context *c) {
24     c->a[index] += value;
25 }
```

(c) Outlined functions for *Delivery*.

```
26 void f(int a[], int ind[], int x) {
27   _Milk_containers c;
28   #pragma omp parallel
29   {
30     #pragma omp for nowait
31     for(int i=0; i<n; i++)
32       _Milk_collect(ind[i], &c);
33     _Milk_distribute(&c);
34     _Milk_deliver(&c, Context(a, x),
35                   .milk.outlined.f);
36   }
37 }
38 void g(float a[], int ind[],
39        float b[]) {
40   _Milk_containers c;
41   #pragma omp parallel
42   {
43     #pragma omp for nowait
44     for(int i=0; i<n; i++)
45       _Milk_collect(&c, ind[i], b[i]);
46     _Milk_distribute(&c, SumCombine);
47     _Milk_deliver(&c, Context(a),
48                   .milk.outlined.g,
49                   SumCombine);
50   }
51 }
```

(d) Logical transformation overview.

Figure 4: DRAM-conscious Clustering transformation.

cessing, 3-D loop nests in physical simulations, etc.). Leaving the outer loop structure to programmers allows further algorithm-specific optimizations, such as use of bit vectors, Bloom filters, priority queues, etc. Further knowledge about power-law data distributions can be used to maximize load balance in parallel loop nests, or to identify accesses that are likely to be cached. Domain-specific frameworks that apply user-supplied *Map* functions over the input can be built on top of our **milk** primitives.

3.1 Collection, Distribution, and Delivery

In the *Collection* and *Distribution* phases, Milk plans memory accesses by grouping references into cache-sized partitions. In the *Delivery* phase, the reordered memory references and deferred dependencies are applied. These phases are a generalization of cache partitioning [45] used in heavily optimized database hash-join [5, 8, 26, 30, 45, 49, 51], with additional DRAM optimizations shown in Section 5.2.

Figure 4d shows a logical overview of the DRAM-conscious Clustering transformation to Collection, Distribution, and Delivery phases. The transformation for line 4 in `f()` shows how a tag index is collected, while the more complex case of line 14 in `g()` also includes a payload value, and a combiner function.

During Collection, indirect reference expressions are evaluated as *tags*, e.g., `ind[i]` on lines 4 and 11. If additional per-reference state is needed, it is also collected with *pack* as a payload; an optional *combiner* enables coalescing of payloads targeting the same tag (e.g., line 12). Tag and pack variables can be initialized with any dynamically evaluated expression and can have side effects (e.g., queue iterators can consume items), since they are written to temporary buffers.

During Distribution, tags are partitioned into clusters such that maximum cache line and DRAM page locality can be achieved for accesses during Delivery. Depending on the range of tag values, if too many partitions would be needed, multiple Distribution passes are required. Each additional Distribution pass uses only sequential transfers to DRAM.

During Delivery, an outlined program slice from the original code is executed using a dynamically constructed context valid for each Delivery thread. Milk captured statements allow arbitrary C++ expressions and function calls, e.g., arbitrary containers can be accessed during Delivery. Figure 4c shows the outlined functions created to capture the deferred execution of lines 4 and 14, respectively, on lines 19 and 24.

Distribution phases are always fused with the previous or next processing stage: the first Distribution pass is fused with Collection, and the final Distribution pass is fused with Delivery. The three distinct phases shown as functions are coroutines, i.e., Collection receives one element at a time to pass to Distribution, and early Delivery can be triggered at any time if memory for temporary buffers is exhausted. Unlike inspector/executor-style [40] preprocessing, the three phases are distinct only logically, and different threads and different tags may be in different phases at the same time.

3.2 Performance Model

A simple model predicts Milk DRAM bandwidth savings as a high order estimate of performance on bandwidth-bound applications. Let us take a workload that makes m random read-modify-writes to n distinct location, e.g., processing a graph with n vertices and m edges. Assume n is

	Original	MILK		
		Collection	Distribution	Delivery
L2		$8m$		$128m$
DRAM	$128m$		$8m$	$8m{+}8n$

Table 1: MILK reduction of DRAM bandwidth in bytes, for m random references to n locations.

much larger than the cache size, and that updated values, indices, and payloads are 4 bytes. If $n \ll m$, the MILK version uses 8× less bandwidth than the original.

Table 1 summarizes the bandwidth required by the transformed workload and overheads of Milk phases. Original references read and write back a random cache-line on each update. The transformed updates are within L2 caches and use DRAM bandwidth only for compulsory cache-fill and write-back of accessed cache lines. When spatial locality is high, as seen earlier in Figure 1, these use $2n \cdot 4B$. Collection arranges tag and pack records in L1/L2 caches, and Distribution and Delivery write and read DRAM sequentially.

3.3 Milk Correctness Conditions

All BSP-model [50] applications are correct under Milk, as are some non-BSP programs allowed under the more relaxed Milk execution model. Correctness proofs of BSP algorithms hold for Milk execution. The only difference is the number of virtual processors, i.e., tens of threads vs. millions of cache lines. Loops in which read and write sets do not overlap will not change behavior when split in Milk supersteps. For non-BSP programs, a loop *may* read results produced within the same step. Programs that produce externally deterministic [12] output but vary intermediate results (e.g., Connected Components in Figure 9a) may incur more per-iteration work or a lower convergence rate with Milk but will be correct.

Programs with read/write dependencies, which produce non-deterministic outputs under OpenMP, have to be validated or modified for Milk. Domain knowledge may allow divergence from sequential semantics, e.g., in traditional bank check processing all checking account debits are processed or bounced first, and only then any account credits are applied. Splitting a loop into separate parallel loops to break dependence chains allows the alternative sequential execution, OpenMP, and Milk to all agree on the output.

4. MILK SYNTAX AND SEMANTICS

The MILK compiler allows programmers to use the Milk execution model for C/C++ programs with OpenMP extensions and loop annotations. Adding a **`milk`** clause to a loop directive is often sufficient to enable optimizations. To achieve further optimization control, programmers can select which references in a loop should be deferred by adding a **`milk`** directive and its clauses. A `tag` clause selects the destination, while `pack` clauses specify the values necessary for deferred dependent statements. The `if` clause allows programmers to express dynamic knowledge about data distribution by deferring only references that are unlikely to be cached. Additional language and library features support safe avoidance of synchronization, deterministic performance, and deterministic outputs with minimal overhead. Table 2 summarizes the clauses of the `pragma` **`milk`** directive and extended OpenMP directives.

Directive	Clause	Section
`milk`	`if`	4.1.1
`milk`	`pack(`*v*`[:all])`	4.1.3
`milk`	`pack(`*v*`:+|*|min|max|any)`	4.1.4
`milk`	`pack(`*v*`:first|last)`	4.1.4, 4.4
`milk`	`tag(`*i*`)`	4.1.2
`atomic`	`if`	4.3
`for`	**`milk`**	4.1
`ordered`	**`milk`**	4.4

Table 2: **`milk`** OpenMP extensions.

```
01 void BFS_Depth(Graph &g, int current,
02                int depths[]) {
03 #pragma omp parallel for milk
04   for (Node u=0; u<g.num_nodes(); u++)
05     if (depths[u] == current)
06       for (Node v : g.out_neigh(u))
07 #pragma milk
08           if(depths[v] == -1)
09               depths[v] = current + 1;
10 }
```

Figure 5: Breadth-First Search (BFS) with **`milk`**.

4.1 `milk` Directive and Clause

`milk` directives within a **`milk`** loop afford fine-level control when Collecting indirect accesses. This use is similar to the OpenMP `ordered` loop clause and the identically named directive used in the loop body; we borrow the same nesting and binding rules [3]. Figure 5 illustrates this annotation syntax for a Breadth-First-Search traversal (BFS), which produces a vector of depths from the source node. Sequential access on line 5 can be processed directly, but random accesses on lines 8–9 are deferred with **`milk`**.

4.1.1 `if` *Filters*

MILK optimizations are effective only when locality exists within a superstep, but is beyond the reach of hardware caches. When programmers can cheaply determine effectiveness, they can provide such a hint as a superstep property, or as a dynamic property of each indirect reference. A loop can be marked for execution without deferral if programmers can determine that the optimization will be ineffective because the range of accesses is small, there are too few requests, or if little locality is expected. For example, a `parallel for` can be marked with OpenMP 4.5 `if:` syntax, e.g.: `#pragma omp` **`milk`** `for if(milk: numV<cacheSize).`

Individual **`milk`** directives can be tagged with an `if` clause when domain-specific knowledge can cheaply determine that a specific address is likely in cache. For example, in graph algorithm implementations where graph nodes are ordered by degree, the high-degree nodes that are in cache can be skipped, e.g.: `#pragma` **`milk`** `if(v>hot).`

4.1.2 `tag` *Clause*

The `tag` clause can make explicit the definition of the index variable used in the following statement or block, as seen in Figure 4b. The clause takes as an argument the identifier of a variable that programmers should define to use the smallest possible type for supported inputs. An explicit tag is also required when using user-defined collections that do not use the subscript operator `[]`.

4.1.3 `pack` *Clause*

The `pack` clause explicitly specifies data needed for correct continuation execution. It takes a list of variables that must be carried as a different payload for every tag. Programmers should minimize the number of variables whenever it is possible to evaluate expressions early (e.g., lines 47 and 98 on Figures 8b and 9c).

Reducing the size of `pack` variables also directly reduces overhead. The performance of traditional indirect reference-bound kernels is a function of the number of cache lines touched regardless of the amount of data. The costs are identical whether 1 bit or 2 doubles are written: a random DRAM read, and a random DRAM writeback upon cache eviction. Reducing precision is ineffective traditionally if the reduced working set is still larger than cache. MILK transformations, however, are bandwidth bound. Thus, the smallest type for the expected range should be used for `pack` variables. Floating point variables can be cast to lower precision types, e.g., users can explore the non-standard `bfloat16` communication type from TensorFlow [4], which is simply a 16-bit bitcast of `float`.

4.1.4 `pack` *Combiners*

For each packed variable, a combiner function can be specified, e.g., `pack(u:min)`, `pack(value:+)`, etc., which may be used to combine `pack` payloads before Delivery. When a combiner is used, programmers must ensure the program is correct, whether all, one, or a subset of all items is delivered. Valid combiners can be defined for all applications we study in Section 6.

Combiners are optional, as their performance advantage depends on the data distribution and access density for each partition. For uniform distributions, for example, early evaluation of combiners will not discover temporal locality to match tags. Combiners evaluated immediately before executing the continuation allow faster processing of all updates targeting a given tag. While not saving bandwidth, late combiner execution enables deterministic outputs, e.g., with `min`/`max` combiners.

The default binary operations supported include the standard OpenMP reductions, e.g., `+`, `*`, `min`. When OpenMP 4.0 user defined reductions [3] are supported by `clang`, we will also allow similar syntax for user defined combiner expressions. While OpenMP `reduction` clauses for scalars (or short dense arrays in OpenMP 4.5) use unit value initializers (e.g., 0, 1, `INT_MAX`), the explicit initialization incurs significant overhead on large and sparsely accessed data. Therefore, Milk combiners are assigned or copy/move constructed only on first access.

Additional pre-defined pack combiner identifiers include `all`, `any`, `first`, and `last`. Specifying `all` explicitly documents that each value must be delivered without a combiner. When any value can be used in any order, `any` marks that duplicate pack values can be dropped. Alternatively, `first` and `last` select one pack value in the respective order it would be collected in serial execution.

4.2 Elision Version Semantics

Same-source programs can be compiled in several versions – traditional serial, OpenMP parallel, and Milk (serial or parallel) – dispatched depending on input size. Eliding Milk annotations produces valid serial programs. Milk programs without atomics are succinct and easier to reason about than OpenMP parallel programs. Parallel non-Milk programs, however, may be desired for efficient execution on small cache-resident inputs. When correctness of parallel non-Milk versions requires additional synchronization, integration with OpenMP allows correct and efficient execution.

```
01 CountDegrees(vector<Edge>& edges) {
02     vector<Node> degrees(n, 0);
03 #pragma omp parallel for milk
04     for (Edge e : edges) {
05 #pragma omp atomic if(!milk)
06         degrees[e.u] += 1;
07         if (!directed)
08 #pragma omp atomic if(!milk)
09             degrees[e.v] += 1;
10     }
11 }
```

Figure 6: `CountDegrees` with multiple atomic updates.

4.3 Atomic Accesses

When unnecessary for **`milk`** loops, synchronization can be eliminated by marking `omp atomic if(!milk)`. This OpenMP syntax extension allows an `if` clause in `atomic`.

All threads updating data concurrently must be in the same **`milk`** loop. As a safe default, we preserve the behaviour of unmodified `omp atomic` directives to use atomics, in case they were intended for synchronization with other thread teams, or with external threads via shared memory. During Delivery, atomic operations operate on cache resident data and are executed by a single thread. Such atomics are faster than DRAM resident or contended cache data with expensive cache-to-cache transfers. However, atomics still drain store buffers and destroy instruction and memory parallelism, incurring 3× overhead. When possible, unnecessary atomics should be eliminated.

Milk safely allows either only reads or only writes to the same array. Mixing reads and writes, or `atomic capture` clauses may change non-BSP program semantics. Currently, loops with internal producer-consumer dependencies should be transformed manually to unidirectional accesses per array per loop to ensure program semantics does not change. Deferring an `atomic update` can be treated as a write and requires no semantic change.

Read-modify-write `update` operations that do not peek into results can issue in parallel. For example, the CountDegrees kernel – used for building a graph in GAPBS – shown in Figure 6 can access two edges with an `atomic update`.

4.4 Deterministic and Ordered Extensions

Adding an `ordered` clause ensures deterministic outputs by restricting **`milk`** processing order. This improves programmer productivity and enables new application use cases, e.g., database replication and fault tolerance. Milk processing can give deterministic execution guarantees very efficiently: there are very few thread synchronization points during Distribution, while the cost of enforcing deterministic order is amortized across thousands of individual memory references. The main concern in deterministic execution is that it exacerbates load imbalance during a Collection work-sharing loop, therefore programmers have to explicitly mark these after addressing load imbalance.

5. COMPILER AND RUNTIME LIBRARY

The MILK compiler and runtime library are implemented within the LLVM [27] framework, as Abstract Syntax Tree (AST) transformations in Clang and code generation of LLVM IR. Collection and Delivery phases are implemented as AST lowering phases emitting specializations of the Distribution container library. Table 3 summarizes additional programmer-facing intrinsics.

5.1 Dynamically Constructed Closure

Milk Delivery's indirect memory references are logically separate continuations, but we dynamically construct the full context using four classes of data: a static context shared among all continuations for each deferred block, a tag to construct address references to one or more arrays, an optional payload carried with each tag, and optional thread-local variables.

While similar to C++ lambda expressions, continuations need to allow access to out-of-scope variables across different threads. Compared to `delegate` closures in distributed systems [34], we minimize per-item overhead by optimizing for space and capturing only unique values across iterations. Explicit uses of `tag` and `pack` define the only variables that are unique per tag and packaged as payloads. All variables other than globals used in a continuation must be captured in a shared context.

5.1.1 Shared Context

To minimize space overhead, we use a single closure to capture the original block and all variable references necessary for execution during Delivery of continuations.

Variables that are independent of the iteration variable are simply captured by reference only once for all iterations, e.g., the array base and value in `{m[ind[i]] += val;}` can capture `m` and `val` as long as they are not aliased. If `m` may change, however, programmers should either add `pack(m)`, or recompute `m` in the continuation.

Loop-private variables cannot be passed by reference for Delivery; if programmers can hoist them out of the loop, they are automatically stored in the shared context. Otherwise, variables need to be either explicitly passed by value with `pack`, or recomputed.

5.1.2 Tags

Each block must have a single implicit or explicit `tag`, but multiple references based on the same tag can access different arrays, vectors, or C++ objects with overloaded `operator[]`, e.g., `if (a[v]) b[v]++`. Therefore, tags stored in Collection are indices, not pointers.

5.1.3 Pack payloads

Initialization of variables in `pack` clauses is evaluated during Collection. Variables are not added automatically to `pack` clauses; any non-packed variable reference that is not in the shared context results in a compile-time error. Currently, programmers have to decide whether an expression should be evaluated during Collection, Delivery, or in both.

Evaluation during Collection is best for expressions that make sequential accesses, or can be `pack`ed as a compact payload, e.g.: `bool b = (a[i]>1); Pack(b){...}`. (The `Pack` macro expands to `_Pragma("milk pack")`).

Evaluation during Delivery is favorable when indices are smaller than values and refer to cache-resident data, or expressions have larger size, e.g.:
`Pack(i) {double ai=a[i], r = 1.0/i; ...}`.

Evaluation on both sides trades memory bandwidth for arithmetic unit cycles; programmers must explicitly recompute any shadowed variables, e.g.:
`double d = f*f; Pack(f) {double d = f*f; ...}`.

Intrinsic	Section and Use
`milk_collect_thread_num`	5.1.4 Collecting thread ID
`milk_deliver_thread_num`	5.1.4 Delivering thread ID
`milk_set_max_memory`	5.2.2 Maximum memory
`milk_set_strict_bsp`	5.2.2 Superstep splitting
`milk_touch`	5.2.2 Working set estimate

Table 3: MILK API summary.[2]

5.1.4 Thread-Local Variables

Thread-local variable expressions may be expected to be exclusive to their owner, but are now accessed in both Collection and possibly different Delivery threads. Thread-local variables here refers both to OpenMP `threadprivate` global variables and to indirection via CPUID, thread ID (TID), or `omp_get_thread_num`.

TID references in left-hand expressions in continuations are usually intended to use variables of Delivery threads. A common use for thread-local variables is to implement reductions manually, e.g., lines 32 and 103 in Section 6. The alias `milk_deliver_thread_num` explicitly documents the intent that `omp_get_thread_num` should use Delivery thread local variables. When user-defined reductions [3] are explicitly declared, thread-local references use Delivery threads.

TID references in right-hand expressions may have to be evaluated in the Collect threads. For `const` references, the intrinsic `milk_collect_thread_num` should replace TID references. This helps reduce payload size as many requests share the same source TID value. Thread-local variables cannot be passed by non-`const` reference to continuations, as multiple Delivery threads may update the same item. To guarantee correct execution without atomics, expressions can be evaluated and packed during Collection, e.g., to generate unique ID's in a pre-allocated per thread range: `{m[ind[i]] = state[TID].id++;}`.

5.2 Milk Containers and Distribution

Milk Distribution fuses a nested multi-pass radix partitioning optimized for all levels of the memory hierarchy from registers to DRAM. Distribution is a generalization of cache partitioned join [45], and TLB-aware radix join [30], adjusted for the capacity and access costs of modern hardware memory hierarchy with additional optimizations for maximizing DRAM bandwidth. Naively appending to a per-partition buffer in DRAM would require $2 \cdot 64$B random updates, and negate any cache partitioning benefits. The internal Distribution coalesces updates in full cache lines using a container library specialized for transportation of different sized tags and payloads. As illustrated in Figure 7, for each partition there is a *jug* and/or a *pail*, and a list of *tubs*. These append-only two-dimensional structures are logically simple, but they require careful choice of widths (i.e., number of elements), heights (i.e., fan-out), address placement and access patterns.

A *jug* is an L1-cache-resident container, optimized for fast writes with low register pressure. Branch misprediction im-

[2] MILK's API reference is at milk-lang.org/api.html

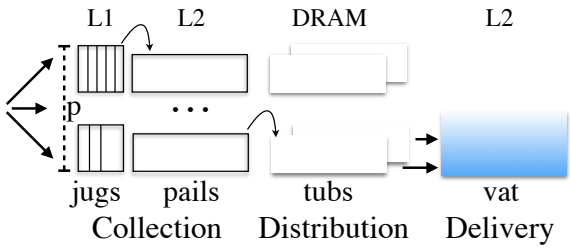

Figure 7: Containers for DRAM-conscious clustering.

pact on jug overflow can be reduced with wider jugs, but that increases total L1 cache occupancy, thus typical widths are 32–512 bytes. A full jug is emptied into one or two *pails*.

A *pail* is an L2-cache-resident container primarily designed to build full cache lines with streaming stores to the next stage. Pails are optimized for cache capacity, and further optimized to avoid cache associativity conflicts. Additionally, when appending to a pail unused bits can be trimmed, e.g., when partitioning into 256 pails with a bit mask, only the remaining 24-bits need to be stored. When compiled for Haswell, streaming and partition indexing can take advantage respectively of 256-bit Advanced Vector Extensions (AVX2) instructions and Bit Manipulation Instructions (BMI).

A *tub* is a large DRAM-resident container used for multi-pass processing. When full or finalized, tubs are linked into thread-local lists and eventually added to global lists. Tubs are tagged with the creating thread-ID if access to thread-local variables is needed in a captured continuation. TLB reach and fan-out constrain the maximum size of a single tub. All active tubs for each thread are TLB-reachable, achieved at the cost of additional memory fragmentation.

A *vat* is a random access cache-resident container for mixing *tubs* targeting a fixed output subset. Depending on the expected spatial or temporal locality different variants are selected. The simplest vat handles the case of expected high spatial locality with dense range updates, e.g., when every cache line is used and the full content of each cache line is overwritten. Other specializations handle low density or low coverage cases, e.g., sparse cache-line writes when few cache lines within a range are modified, or when partial cache line content is either preserved or can be overwritten.

Milk containers are by default emptied in Last-In First-Out (LIFO) order to maximize cache locality. LIFO order is acceptable, when the sequential consistency of a `parallel for` is acceptable. Even if deterministic, this order does not match serial program semantics when that may be desirable, e.g., for `a[ind[i]] = i` to observe the last write. If `ordered` processing is requested, containers are emptied in First-In First-Out (FIFO) order, and a Milk program can match the output of a serial program.

5.2.1 Container Sizing to Hardware

For a desired cache level and a given cache budget, the total number of partitions (i.e., height) has priority over pail width. The height and the range of valid indices determine the number of partitioning passes, e.g., we can partition using 9-bit partitioning at every pass with height 512, while height 256 uses 8-bit partitioning that may require an additional pass to partition the full input.

Wider pails improve DRAM row hits on writes to tubs, as these are otherwise random cache-line writes. Pails width, however, is limited by cache capacity, including the need to fit cache resident additional application data and code during Collection.

Tubs are limited by memory capacity. Memory management is a simple linked list of chunks. Tub layout is guided by low-level optimization opportunities: the TLB entries are a per-core resource, thus the threads on a core fit all their active tubs within TLB reach; the DRAM banks are a global resource, thus the active tubs are interleaved across all threads on a socket to maximize DRAM open page hits.

5.2.2 Container Sizing for Applications

The maximum memory that can be allocated for temporary space during Distribution is the most important attribute, set with `milk_set_max_memory`. If memory limits are reached this will force compaction and/or BSP superstep splitting. Applications targeting Milk-only execution can eschew double-buffering and disallow superstep splitting with `milk_set_strict_bsp`.

Tuning hints that limit the container sizes are expressed from the perspective of the user program, i.e., L1, L2, L3 cache sizes and TLB entries that should be reserved for cache resident data and code that are not managed by Milk. Container implementation details do not need to be exposed to users. When optimal cache-resident size cannot be inferred, tuning hints (i.e., bytes touched per `tag`) control the number of partitions and fan-out. External library footprint can be summarized with the `milk_touch` intrinsic, e.g., a call to a function that touches 3 floats on average for the expected data distribution can be accompanied by a call to `milk_touch(3*sizeof(float))`.

6. APPLICATIONS

To demonstrate the applicability and performance improvements provided by MILK, we use the five fundamental graph applications from the GAP Benchmark Suite [10, 11] building on the high performance OpenMP reference implementations.

In Figure 8 and Figure 9, we show the concise **`milk`** implementations of the inner loops of the kernels. These are also valid serial programs without **`milk`**, but parallel variants need additional compare-and-swap (CAS) logic as noted on lines 16, 31, and 102, e.g., line 102 replaces ten lines of a compare-and-swap loop in the reference implementation. Thread-local data structures are accessed using the delivery thread's ID, therefore the TID macro on lines 17, 32, and 103 is simply `omp_get_thread_num`.

Betweenness Centrality (BC). BC is an important graph algorithm that analyzes the relative importance of nodes based on the number of shortest paths going through each, e.g., to identify critical nodes in power grids, terrorist communication networks, or HIV/AIDS propagation [15,28]. BC is also an integral building block for many state-of-the-art community detection algorithms [35].

The forward- and backward- step kernels of BC are shown in Figure 8b. The continuation block line 28 defers random access to the neighbors of u, and `pack`s the number of paths to be added. An optional sum combiner can accumulate the contributions along all incoming edges of v. For backward propagation line 50, we use a similar floating point combiner for each contribution.

```
Graph<Node> g;
Queue<Node> queue;   // global
Queue<Node>* lqueue; // per-thread

vector<Node> parent;
long outedges;
void BFS_TopDown(int depth) {
#pragma omp parallel for milk \
            reduction(+ : outedges)
  for (Node u : queue)
    for (Node v : g.out_neigh(u))
#pragma milk pack(u : min) tag(v)
    {
        Node curr = parent[v];
        if (curr < 0) {
          parent[v] = u;  // CAS
          lqueue[TID].push_back(v);
          outedges += -curr;
        }
    }
}
```

(a) Breadth-First Search with **milk**

```
vector<Node> paths, depths;
void Brandes_Forward(int depth) {
#pragma omp parallel for milk
  for (Node u : queue) {
    Node pathsU = paths[u];
    for (Node v : g.out_neigh(u))
#pragma milk pack(pathsU : +) tag(v)
      {
        if (depths[v] == -1) {
          depths[v] = depth;  // CAS
          lqueue[TID].push_back(v);
        }
        if (depths[v] == depth)
#pragma omp atomic if(!milk)
          paths[v] += pathsU;
      }
  }
}
vector<Queue<Node>::iterator> wave;
vector<float> deltas;
void Brandes_Reverse(int d) {
#pragma omp parallel for milk
  for (auto it = wave[d-1];
            it < wave[d]; it++) {
    Node v = *it;
    float contribV = (1 + deltas[v]) /
                     paths[v];
    for (Node u : g.in_neigh(v)) {
#pragma milk pack(contribV : +) tag(u)
      if (depths[u] == d - 1)
#pragma omp atomic if(!milk)
        deltas[u] += paths[u] * contribV;
    }
  }
}
```

(b) Betweenness Centrality with **milk**

Figure 8: Modified BFS-traversal kernels from GAPBS [10]

```
vector<Node> comp;
bool change;

void ShiloachVishkin_Step1() {
#pragma omp parallel for milk
  for (Node u = 0; u < g.num_nodes(); u++) {
    Node compU = comp[u];
    for (Node v : g.out_neigh(u))
#pragma milk pack(compU : min) tag(v)
      if ((compU < comp[v]) &&
          (comp[v] == comp[comp[v]])) {
        comp[comp[v]] = compU;
        change = true;
      }
  }
}
```

(a) Connected Components with **milk**

```
vector<float> contrib, new_rank;

void PageRank_Push() {
#pragma omp parallel for milk
  for (Node u=0; u < g.num_nodes(); u++) {
    float contribU = contrib[u];
    for (Node v : g.out_neigh(u))
#pragma milk pack(contribU : +) tag(v)
#pragma omp atomic if(!milk)
      new_rank[v] += contribU;
  }
}
```

(b) Page Rank with **milk**

```
WeightedGraph<Node> wg;
vector<Weight> dist;

Weight delta;
vector<Node> frontier;
vector<vector<Node>>* lbins; // per-thread

void DeltaStep_Relax(int bin) {
#pragma omp parallel for milk
  for (Node u : frontier) {
    if (dist[u] < delta * bin)
      continue;
    for (WEdge e : wg.out_neigh(u)) {
      Weight newDist = dist[u] + e.weight;
      Node v = e.dest;
#pragma milk pack(newDist : min) tag(v)
      if (dist[v] > newDist) {
        dist[v] = newDist; // CAS
        lbins[TID][newDist/delta].push_back(v);
      }
    }
  }
```

(c) Single Source Shortest Path with **milk**

Figure 9: Modified kernels from GAPBS [10]

Breadth-First Search (BFS). BFS traversal is an important component of larger graph kernels, such as BC. BFS is also used in path analytics on unweighted graphs. Figure 8a shows a **milk** version of TopDown, which tracks the parent of each vertex. Unlike the traversal in BC above, or the variant in Figure 5 which tracks only depths of each vertex, this variant tracks parents for validation. Therefore, source vertex id is `pack`ed along each visited edge, while a `min` combiner ensures deterministic results. Adaptive optimizations in state of the art implementations [9] also depend on a sum reduction of the outdegrees of all vertices added to the BFS frontier.

Connected Components (CC). CC is critical for understanding structural properties of a graph. It finds connected components in a large graph by propagating the same label to all vertices in the same connected component.

Our baseline implements Shiloach-Vishkin's algorithm [46] with optimizations [7]. The **milk** implementation of CC is shown in Figure 9a. While the output of CC is deterministic, traditional implementations have non-deterministic performance, i.e., the number of iterations is dependent on race resolution of line 68, and results in 20% run-to-run variation. While the BSP model does not allow a synchronous step to observe updates within the same step, Milk allows updates during Delivery, e.g., `comp[v]` references can still observe simultaneous updates by other neighbors. A `min` combiner allows deterministic performance.

PageRank (PR). PageRank is an iterative algorithm for determining influence within a graph, initially used to sort web search results [37]. The PageRank-Delta variant [29] more efficiently propagates updates from vertices that receive large deltas between iterations. The commonly benchmarked in prior work, however, traditional algorithm propagates till convergence contributions from all vertices in every iteration as shown in Figure 9b.

Single-Source Shortest Paths (SSSP). For weighted graphs, SSSP computes the distance from a start vertex to all reachable vertices, e.g., travel distance or travel time on a road network. Δ-stepping [31] is among the few parallel SSSP variants that outperform a well-optimized serial Dijkstra algorithm. Figure 9c shows Δ-stepping with **milk**.

According to the benchmark specifications all non-pointer data types are 32-bit, therefore all `tag` keys are 32-bit integers, and `pack` values are 32-bit integer or floating point numbers. We do not modify the precision of floating point numbers in `pack` types or combiner accumulators.

7. EVALUATION

We compare the performance of Milk on top of `clang` 3.7 to an OpenMP baseline, on the graph applications from GAPBS [10, 11] v0.8 using the modified kernels shown in Section 6 with **milk** extensions. We report end-to-end performance gains while exploring the effects of varying numbers of vertices and edges, as well as sensitivity to available temporary memory.

7.1 System Characterization

Table 4 summarizes the characteristics of the evaluated platform which has high memory bandwidth per core, as well as low latency. We measured achievable read and streaming write bandwidths using all cores to be >90% of 6.5 GB/s per-core DRAM theoretical bandwidth. Idle and loaded latencies are measured using MLC [33]. We cross-checked

Cores	Cache		TLB	
	L2	L3 Shared	Entries	Reach
4	256 KB	8 MB	1024	2 GB

Core	DRAM Bandwidth			Latency
Frequency	Peak	*Read*	*Write*	*Read*
4.2 GHz	26 GB/s	*92 %*	*98 %*	*58–130* ns

Table 4: System specifications [2] and *measurements*.

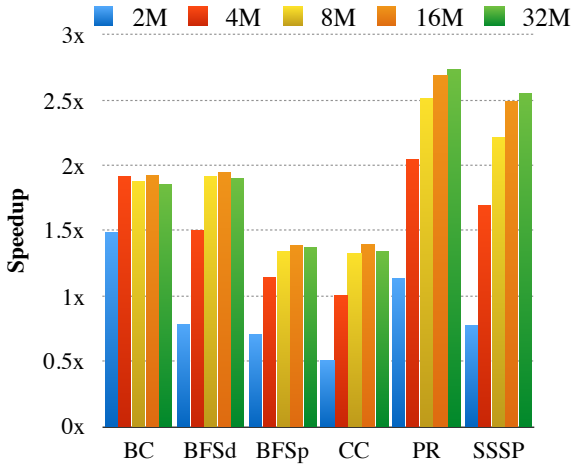

Figure 10: Overall speedup with Milk on random graphs with 2—32 million vertices, d=16(uniform), 8 MB L3.

with `perf` that these measurements match the average L1 miss latency of 452 cycles on DRAM-bound references, e.g., PageRank on uniform graphs, which sustains Memory Level Parallelism (MLP) of 8 outstanding L1 misses.

The Haswell microarchitecture has much improved TLB reach compared to previous generations: each core can reach up to a 2 GB working set with low TLB overhead. For larger working sets, 2 MB page table entries are cache resident, and a TLB miss does not access DRAM.

7.2 Speedup

We show end-to-end speedups, ranging from 1.4× on CC and 2× for BFS, to 2.7× on PR, and analyze Milk's performance on synthesized random graphs with variable working set size and temporal and spatial locality.

Figure 10 shows speedups when vertices range from V=2^{21} to V=2^{25}, with an average degree of 16 in uniform degree distribution (i.e., Uniform⟨21..25⟩). We show two BFS variants – BFSd from Graph500 [21] (in Figure 5), as well as BFSp from GAPBS (Figure 8a). The two variants help compare effects of payload size: the BFSd variant has no payload as it only tracks the constant per superstep depth of each vertex, while the BFSp variant uses more bandwidth as it needs to `pack` the parent ID. On CC, Milk per-iteration speedup is stable, however, CC is sensitive to vertex labeling and race resolution. These effects result in non-deterministic iteration count and per-iteration time, the former may vary by up to 20% in the baseline. All other applications have under 2% noise between runs.

Figure 10 also shows the characteristic transition from L3 to DRAM-bound references with a corresponding increase in Milk speedups. The randomly accessed data per vertex varies for the different benchmarks, e.g., either 1 or 3 random

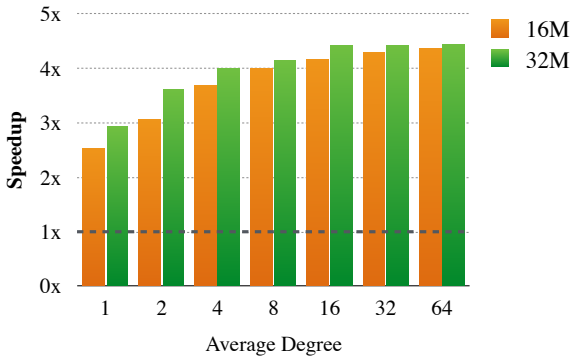

Figure 11: MILK Speedup on Histogram, counting degree of 16M or 32M vertices with 16M–2B edges.

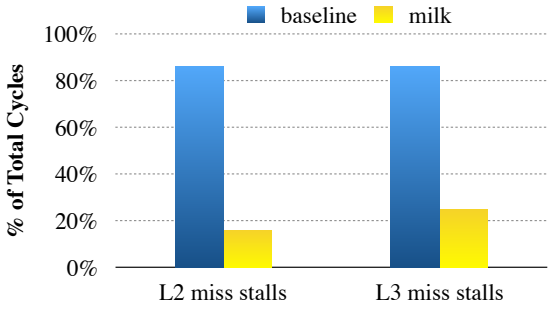

Figure 12: Stall cycle reduction with MILK on PR, V=32M, 256 KB L2, 8 MB L3.

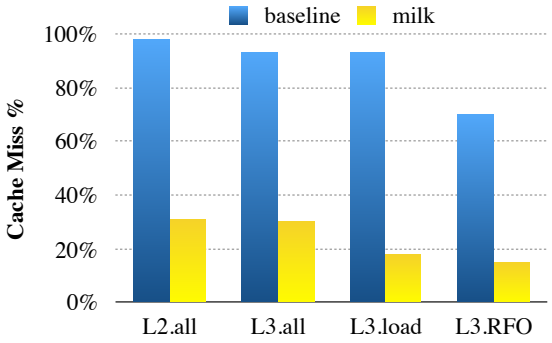

Figure 13: Cache miss rate reduction with MILK on PR, V=32M, 256 KB L2, 8 MB L3. L3 miss rates show all references (including prefetches), non-prefetch loads, and non-prefetch store Requests For Ownership (RFOs).

references to 32-bit values in BC, vs. a single 32-bit value in the other benchmarks. At $V = 2M$ the total working set is already larger than cache size for BC, while for the other applications requests are mostly served by the L3 cache. Once the working set size more than doubles the cache capacity, very little locality is captured by hardware caches and the baseline random accesses run at DRAM latency, while MILK's access time per indirect reference remains constant.

In Figure 11, we compare performance while varying vertex degrees to show sufficient spatial locality can be exploited even at a low average degree. We use the CountDegree (Histogram) kernel of Graph500, shown in Figure 6, which is used to construct a graph data structure from a list of edges in graph applications, and is critical in other applications [25]. Figure 11 shows speedups on Histogram in uniform distributions with average degrees from 1 to 64. Low degree graphs have lower temporal and spatial locality, and only 2.5× speedup. When the average degree is larger than 16, however, all vertices have an incident edge, and spatial locality reaches 100%, i.e., all words accessed per cache line are useful. At degree 16 and higher, we gain 4.2×–4.4×.

Most real-world web and social graphs have much higher degrees than 16, even within subgraphs of graphs partitioned across distributed systems. Characterization of actual social network graphs [6, 14] shows that Twitter's ∼300 million users have ∼200 followers on average, and Facebook's ∼1.5 billion users have 290 friends on average.

Real-world graphs also exhibit better cache hit rates than synthetic uniformly random graphs, due to power law structure, and to a lesser extent due to graph community structure. While current hardware caches are able to capture some locality in power law graphs [11], the actual social and web graphs are orders of magnitude larger than public datasets. We tested on 8 MB L3 cache size in order to simulate the effects of larger graphs while preserving the graph structure. On Twitter(V=62M, d=24) we observe 20–50% L3 hits, and only 5–20% L2 hits.

We evaluated performance while varying the size of synthetic power law graphs – R-MAT [13] used for Graph500 benchmarks as a proxy for real-world graph structure. On RMAT25 (V=32M,d=16) CountDegree achieves 3.2× speedup. Spatial locality is lower here since half of the generated vertices have degree 0, while the hottest vertices are cache-resident, e.g., the top 3% of vertices have 80% of the edges. Although the majority of edges are incident on hot vertices, processing hot vertex edges is faster than handling indirect references that miss caches. Speedups on RMAT25 are comparable to speedups on smaller size uniform graphs; e.g., BC goes from 1.9× speedup on Uniform25 to 1.3× on RMAT25, and SSSP goes from 2.5× to 2.2×.

Filtering accesses to hot references is necessary for effective use of both hardware caches and Milk. In graphs, hot references to high-degree vertices are easy to identify as described in Section 4.1.1.

7.3 Performance Analysis

Cache miss rates and cycles stalled metrics based on hardware performance counters indicate that MILK effectively converts random DRAM references into cache hits.

Figure 12 shows that effective blocking at L2-cache sized partitions dramatically reduces cycles stalled on L2 and L3 misses. While a Haswell core can execute 4 µops per cycle at ideal ILP, these measurements show cycles with complete execution stalls while there is a pending cache miss, i.e., waiting on L3 or DRAM.

Figure 13 further breaks down L3 misses by access type – indirect memory references appear as demand loads and stores, while most sequential accesses are prefetched. With MILK, demand accesses are no longer a major contributor to cache misses, and most misses are on sequential requests.

7.4 Space Overhead

MILK's maximum space overhead is proportional to the number of references per superstep, but even within a fraction of the maximum space, applications with ample spatial

Kernel	Overhead % Resident	pack Type	Tag + Pack Bytes
BC	104 %	int/float	8
BFS_Depth	58 %	—	4
BFS_Parent	113 %	int	8
CC	176 %	int	8
PR	184 %	float	8
SSSP	4 %	int	8

Table 5: Maximum memory overhead of MILK. Measured as maximum resident size over baseline for V=32M. `tag` clauses, and where needed `pack`, use 32-bit variables.

locality attain good performance. For example, at only 12% memory overhead PageRank maintains $> 2.7\times$ speedup. For graph applications, the number of indirect references is a function of the number of active edges accessed during each iteration. Most algorithms make a different number of indirect references during each iteration: few edges are processed in BC/BFS, very few in SSSP, and only CC and PR always touch all edges. If less memory than the maximum required is available, a superstep will have to be forcefully divided, and may not be able to capture all available locality. Superstep splitting has minimal performance impact when temporal locality is high, e.g., as shown in Figure 11 where splitting is required on graphs with degree 64.

Table 5 shows the maximum additional memory required on the evaluated kernels without memory limits, compression, filters, or combiners. Baseline memory consumption varies based on the graph representation and other algorithm overheads, e.g., unweighted graph plus 1 bit per edge for BC vs. a weighted graph plus additional 4 bytes per edge for SSSP. MILK memory consumption is primarily a function of the number of indirect references per superstep, multiplied by the size of the `pack` and `tag` variables. For example, BC forward traversal carries an `int`, and backward traversal – a `float`. Small additional overhead is added by bookkeeping and internal fragmentation in Milk containers.

8. RELATED WORK

Milk's distribution phases are most similar to the heavily optimized table join operations in relational databases. The two best join methods are to sort both database relations, or to build a hash table with tuples of one relation and then probe the hash table for each tuple of the other relation. Increasing data sizes and perennial hardware changes have fueled continuous innovation: Shatdal et al. [45] demonstrated hash join with cache partitioning, further improved to consider TLB impact [30], and to optimize for SIMD, NUMA, non-temporal write optimizations, software buffering, etc [5, 8, 26, 49, 51]. Our runtime library data structures and compiler generated code are designed for modern microarchitectures (e.g., on Haswell TLB overhead is less critical), and focus on avoiding increasingly important DRAM contention and turnaround delays. Milk's compiler support for indirect expressions is more general than database join, and we are the first to use similar techniques to accelerate graph applications.

Graph processing is an important application that we have shown significantly benefits from Milk optimizations. Graph applications expose irregular access patterns that render traditional compiler optimizations ineffective [39]. Creating efficient shared memory graph frameworks [36, 47] has received considerable research attention, but overheads and abstraction mismatch lead to significant slowdown compared to hand-optimized code [43]. Milk allows optimized by hand iterators and algorithm-specific data structures to use familiar and concise indirect array accesses.

The inspector/executor [40] family of two-phase optimizations inspect indirect references before an alternative execution. The original optimization [40] rearranged the base array A in `A[ind[i]]` not the index, as it targeted better scheduling for message passing machines. Follow-up research explored two main directions: either very cheap per-step index array analysis (like determining bounding box (min/max) for MPI and NUMA boundary exchanges [48]), or very expensive data reorganizations to be applied as preprocessing steps. Coalescing references targeting different cores to reduce MPI messaging overhead is explored for the HPCC RandomAccess benchmark [20]. The workload is similar to Histogram of degree 4 (Figure 11) but disallows BSP semantics. Since the benchmark allows at most tens of requests per core to be reordered, short coalescing buffers are needed, yet cache resident buffers can discover very little locality in DRAM-sized working sets. In shared-memory settings, sharding memory and coalescing requests to owning cores eliminate atomics at the cost of message-passing overhead. Inexpensive methods, however, do not improve the performance of DRAM indirect references.

Expensive locality reorganization methods have not been able to amortize costs within a single iteration, and they are limited to applications that repeatedly process the same references, e.g., rearranging the index array [17, 32] and remapping all arrays in a loop, or graph partitioning [22, 23] or cheaper reorderings with lower benefits (e.g., space filling curves). Recent inspector/executor work [18] traded lower cache hit rate for improvement of DRAM row buffer hits for 14% net gains. Milk achieves up to $4\times$ gains on static reference loops, and pays off in one iteration to also allow dynamic references.

9. CONCLUSION

We introduced the **`milk`** C/C++ language extension in order to capture programmer's intention when making indirect memory references, while the compiler and runtime system choose the most efficient access order. As demonstrated, the annotations can easily be added across a suite of popular graph applications and when compiled with MILK enable performance gains of up to $3\times$.

We believe the right level of abstraction to enable high-performance applications for graph analytics, machine learning, or in-memory databases is not necessarily a domain specific framework, nor an optimized collection of algorithms — i.e., hash vs. sort join, nor a data structure — i.e., bit vector vs. hash table. Instead, best performance and flexibility are enabled by the most broadly relied upon low-level primitive — the indirect memory reference — and just adding **`milk`**.

Acknowledgments

We thank our anonymous reviewers and our shepherd Bronis de Supinski for their helpful probing questions and their specific suggestions for improving our presentation. This research is based upon work supported by NSF grant CCF-1533753, and DOE awards DE-SC008923 and DE-SC014204.

10. REFERENCES

[1] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[2] Intel Core i7-4790K Processor (8M Cache, up to 4.40 GHz). http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz.

[3] OpenMP Application Program Interface 4.0. http://openmp.org/wp/openmp-specifications/, 2013.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Available at tensorflow.org.

[5] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, June 2012.

[6] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, pages 33–42, New York, NY, USA, 2012. ACM.

[7] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 547–556, June 2005.

[8] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.*, 27(7):1754–1766, 2015.

[9] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[10] S. Beamer, K. Asanović, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[11] S. Beamer, K. Asanović, and D. A. Patterson. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. In *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*, pages 56–65, 2015.

[12] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 181–192, 2012.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 442–446, 2004.

[14] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[15] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, Mar. 2004.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[17] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *ACM SIGPLAN Notices*, volume 34, pages 229–241. ACM, 1999.

[18] W. Ding, M. Kandemir, D. Guttman, A. Jog, C. R. Das, and P. Yedlapalli. Trading cache hit rate for memory performance. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 357–368, New York, NY, USA, 2014. ACM.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[20] R. Garg and Y. Sabharwal. Optimizing the HPCC RandomAccess benchmark on Blue Gene/L supercomputer. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 369–370, New York, NY, USA, 2006. ACM.

[21] Graph500. Graph 500 benchmark. http://www.graph500.org/specifications.

[22] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *Languages and Compilers for Parallel Computing*, pages 181–196. Springer, 1998.

[23] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, July 2006.

[24] JEDEC. DDR3 SDRAM Standard. http://www.jedec.org/standards-documents/docs/jesd-79-3d.

[25] W. Jung, J. Park, and J. Lee. Versatile and scalable parallel histogram construction. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 127–138, New York, NY, USA, 2014. ACM.

[26] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389, Aug. 2009.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. CGO '04, San Jose, CA, USA, Mar 2004.

[28] F. Liljeros, C. R. Edling, L. A. N. Amaral, H. E. Stanley, and Y. Åberg. The web of human sexual contacts. *Nature*, 411(6840):907–908, 2001.

[29] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[30] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. on Knowl. and Data Eng.*, 14(4):709–730, July 2002.

[31] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, Oct. 2003.

[32] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 192–202. IEEE, 1999.

[33] MLC. Intel Memory Latency Checker v3.0. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[34] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[35] M. E. J. Newman. Detecting community structure in networks. *The European Physical Journal B - Condensed Matter and Complex Systems*, 38(2):321–330, 2004.

[36] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[37] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[38] I. Papazian, S. Kottapalli, J. Baxter, J. Chamberlain, G. Vedaraman, and B. Morris. Ivy Bridge server: A converged design. *Micro, IEEE*, 35(2):16–25, Mar 2015.

[39] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[40] J. Saltz, K. Crowley, R. Michandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, 1990.

[41] Samsung. DDR3L SDRAM 240pin Registered DIMM M393B2G70BH0 datasheet:. http://www.samsung.com/global/business/semiconductor/file/product/ds_ddr3_4gb_b-die_based_1_35v_rdimm_rev16-5.pdf, 2012.

[42] Samsung. DDR4 SDRAM 288pin Registered DIMM M393A2G40DB1 datasheet. http://www.samsung.com/semiconductor/global/file/product/DS_8GB_DDR4_4Gb_D_die_RegisteredDIMM_Rev15.pdf, 2015.

[43] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. 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, pages 979–990, New York, NY, USA, 2014. ACM.

[44] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. San Francisco, CA, 2015. Parallel Architectures and Compilation Techniques (PACT'15).

[45] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[46] Y. Shiloach and U. Vishkin. An O(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1), 1982.

[47] J. Shun and G. E. Blelloch. 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, pages 135–146, New York, NY, USA, 2013. ACM.

[48] J. Su and K. Yelick. Automatic support for irregular computations in a high-level language. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, Washington, DC, USA, 2005. IEEE Computer Society.

[49] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, ICDE '13, pages 362–373, Washington, DC, USA, 2013. IEEE Computer Society.

[50] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[51] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 160–169, Berlin, Heidelberg, 2011. Springer-Verlag.