Cimple: Instruction and Memory Level Parallelism

A DSL for Uncovering ILP and MLP

Vladimir Kiriansky, Haoran Xu, Martin Rinard, Saman Amarasinghe
MIT CSAIL
{vlk,haoranxu510,rinard,saman}@csail.mit.edu

ABSTRACT

Modern out-of-order processors have increased capacity to exploit instruction level parallelism (ILP) and memory level parallelism (MLP), e.g., by using wide superscalar pipelines and vector execution units, as well as deep buffers for in-flight memory requests. These resources, however, often exhibit poor utilization rates on workloads with large working sets, e.g., in-memory databases, key-value stores, and graph analytics, as compilers and hardware struggle to expose ILP and MLP from the instruction stream automatically.

In this paper, we introduce the **IMLP** (Instruction and Memory Level Parallelism) task programming model. IMLP tasks execute as coroutines that yield execution at annotated long-latency operations, e.g., memory accesses, divisions, or unpredictable branches. IMLP tasks are interleaved on a single thread, and integrate well with thread parallelism and vectorization. Our DSL embedded in C++, Cimple, allows exploration of task scheduling and transformations, such as buffering, vectorization, pipelining, and prefetching.

We demonstrate state-of-the-art performance on core algorithms used in in-memory databases that operate on arrays, hash tables, trees, and skip lists. Cimple applications reach 2.5× throughput gains over hardware multithreading on a multi-core, and 6.4× single thread speedup.

CCS CONCEPTS

• **Software and its engineering → Coroutines**;

ACM Reference Format:
Vladimir Kiriansky, Haoran Xu, Martin Rinard, Saman Amarasinghe. 2018. Cimple: Instruction and Memory Level Parallelism : A DSL for Uncovering ILP and MLP. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243176.3243185

PACT '18, November 1–4, 2018, Limassol, Cyprus
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5986-3/18/11.
https://doi.org/10.1145/3243176.3243185

1 INTRODUCTION

Barroso et al. [4] observe that "killer microseconds" prevent efficient use of modern datacenters. The critical gap between millisecond and nanosecond latencies lies outside the traditional roles of software and hardware. Existing software techniques used to hide millisecond latencies, such as threads or asynchronous I/O, have too much overhead to successfully address microsecond latencies and below. On the other hand, out-of-order hardware is capable of hiding at most tens of nanoseconds latencies. Yet, average memory access times now span a much broader range: from ~20 ns for L3 cache hits, to >200 ns for DRAM accesses on a remote NUMA node — making hardware techniques inadequate. We believe an efficient, flexible, and expressive programming model can fill the critical gap and scale the full memory hierarchy from tens to hundreds of nanoseconds.

Processors have grown their capacity to exploit instruction level parallelism (ILP) with wide scalar and vector pipelines, e.g., cores have 4-way superscalar pipelines, and vector units can execute 32 arithmetic operations per cycle. Memory level parallelism (MLP) is also pervasive, with deep buffering between caches and DRAM that allows 10+ in-flight memory requests per core. But long distances between independent operations in existing instruction streams prevent modern CPUs from fully exploiting this source of performance.

Critical infrastructure applications such as in-memory databases, key-value stores, and graph analytics, characterized by large working sets with multi-level address indirection and pointer traversals, push hardware to its limits: large multi-level caches and branch predictors fail to keep processor stalls low. Out-of-order windows of hundreds of instructions are also insufficient to hold all instructions needed in order to maintain a high number of parallel memory requests, which is necessary to hide long latency accesses.

The two main problems are caused by either branch mispredictions that make the effective instruction window too small, or by overflowing the instruction window when there are too many instructions between memory references. Since a pending load prevents all following instructions from retiring in-order, if the instruction window resources cannot hold new instructions, no concurrent loads can be issued. A vicious cycle forms where low ILP causes low MLP when long dependence chains and mispredicted branches do not generate enough parallel memory requests. In turn, low MLP causes low effective ILP when mispredicted branch outcomes depend on long latency memory references.

Context switching using high number of hardware threads to hide DRAM latency was explored in Tera [1]. Today's commercial CPUs have vestigial simultaneous multithreading support, e.g., 2-way SMT on Intel CPUs. OS thread context switching is unusable as it is 50 times more expensive than a DRAM miss. We therefore go back to 1950s coroutines [49] for low latency software context switching in order to hide variable memory latency efficiently.

We introduce a simple Instruction and Memory Level Parallelism (IMLP) programming model based on concurrent tasks executing as coroutines. Coroutines yield execution at annotated long-latency operations, e.g., memory accesses, long dependence chains, or unpredictable branches. Our DSL Cimple (Coroutines for Instruction and Memory Parallel Language Extensions) separates program logic from programmer hints and scheduling optimizations. Cimple allows exploration of task scheduling and techniques such as buffering, vectorization, pipelining, and prefetching, supported in our compiler **Cimple** for C++.

Prior compilers have struggled to uncover many opportunities for parallel memory requests. Critical long latency operations are hidden in deeply nested functions, as modularity is favored by current software engineering practices. Aggressive inlining to expose parallel execution opportunities would largely increase code cache pressure, which would interact poorly with out-of-order cores. Compiler-assisted techniques depend on prefetching [40, 46], e.g., fixed lookahead prefetching in a loop nest. Manual techniques for indirect access prefetching have been found effective for the tight loops of database hash-join operations [10, 32, 45, 55] - a long sequence of index lookups can be handled in batches (*static scheduling*) [10, 45], or refilled dynamically (*dynamic scheduling*) [32, 55]. Since the optimal scheduler style may be data type and data distribution dependent, Cimple allows generation of tasks for both styles, additional code-generation optimizations, as well as better optimized schedulers.

High performance database query engines [15, 34, 45, 48] use Just-In-Time (JIT) compilation to remove virtual function call overheads and take advantage of attendant inlining opportunities. For example, in Impala, an open source variant of Google's F1 [66], query generation uses both dynamically compiled C++ text and LLVM Instruction Representation (IR) [13]. Cimple offers much higher performance with lower complexity than using an LLVM IR builder: Cimple's Abstract Syntax Tree (AST) builder is close to C++ (and allows verbatim C++ statements). Most importantly, low level optimizations work on one item at a time, while Cimple kernels operate on many items in parallel.

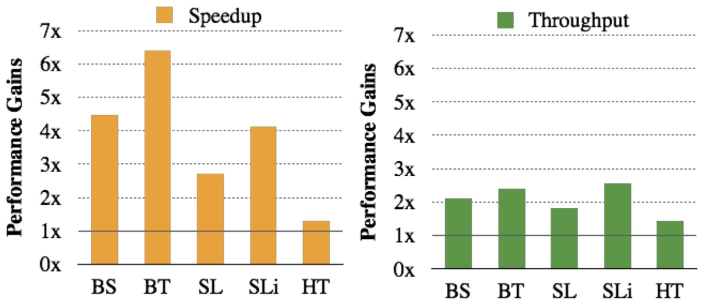


Figure 1: Speedup on a single thread and throughput gains on a full system (24 cores / 48 SMT [25]). Binary Search, Binary Tree, Skip List, Skip List iterator, and Hash Table.

We compare Cimple performance against core in-memory database C++ index implementations: binary search in a sorted array, a binary tree index lookup, a skip list lookup and traversal, and unordered hash table index. As shown on Figure 1, we achieve 2.5× peak throughput on a multi-core system, and on a single-thread — 6.4× higher performance.

The rest of this paper is organized as follows: In Section 2, we walk through an end-to-end use case. In Section 3, we explore peak ILP and MLP capabilities of modern hardware. We introduce the IMLP programming model and Cimple compiler and runtime library design in Section 4, with more details of the Cimple DSL in Section 5, and implementation details of Cimple transformations in Section 6. We demonstrate expressiveness by building a template library of core indexes in Section 7, and performance – in Section 8. Section 9 surveys related work and Section 10 concludes.

2 EXAMPLE

We next present an example that highlights how the Cimple language and runtime system work together to efficiently expose available memory-level parallelism on current hardware. We use a classic iterative binary search tree lookup, which executes a while loop to traverse a binary tree in the direction of `key` until a match is found. It returns the node that contains the corresponding key/value pair, or `nil`.

2.1 Binary Search Tree Lookup in Cimple

Listing 1 presents the Cimple code for the example computation. The code identifies the name of the operation (`BST_find`), the result type (`node*`), and the two arguments (`n`, the current node as the computation walks down the tree, and `key`, the key to lookup in the tree).

In this code, there is one potentially expensive memory operation, specifically the first access to `n->key` in the if condition that checks to see if the key at the current node `n`

```
auto c = Coroutine(BST_find);
c.Result(node*).
Arg(node*, n).
Arg(KeyType, key).
Body().
  While(n).Do(
    Prefetch(n).Yield().
    If( n->key == key ).
    Then( Return(n) ).
    Stmt( n = n->child[n->key < key]; )
  ).
  Return(n);
```

Listing 1: Binary Search Tree Lookup in Cimple.

matches the lookup key. Once the cache line containing this value has been fetched into the L1 data cache, subsequent accesses to n->key and n->child are accessed quickly. The Cimple code issues a prefetch, then yields to other lookup operations on the same thread.

```
struct Coroutine_BST_Find {
  node* n;
  KeyType key;
  node* _result;
  int _state = 0;
  enum {_Finished = 2};

  bool Step() {
    switch(_state) {
    case 0:
      while(n) {
        prefetch(n);
        _state = 1;
        return false;
    case 1:
        if(n->key == key) {
          _result = n;
          _state = _Finished;
          return true;
        }
        n = n->child[n->key < key];
      } // while
      _result = n;
      _state = _Finished;
      return true;
    case _Finished:
      return true;
  }}};
```

Listing 2: Generated Cimple coroutine for BST_find.

Listing 2 presents the coroutine that our Cimple compiler (automatically) generates for the code in Listing 1. Each coroutine is implemented as a C++ struct that stores the required state of the lookup computation and contains the generated code that implements the lookup. The computation state contains the key and current node n as well as automatically generated internal state variables _result and _state. Here after the Cimple compiler has decomposed the lookup computation into individual steps, the computation can be in one of three states:

Before Node: In this state the lookup is ready to check if the current node n contains key. However, the required access to n->key may be expensive. The step therefore issues a prefetch on n and returns back to the scheduler. To expose additional memory level parallelism and hide the latency of the expensive memory lookup, the scheduler will proceed on to multiplex steps from other lookup computations onto the scheduler thread.

At Node: Eventually the scheduler schedules the next step in the computation. In this step, the prefetch has (typically) completed and n is now resident in the L1 cache. The computation checks to see if it has found the node containing the key. If so, the lookup is complete, the coroutine stores the found node in _result, and switches to the Finished state. Otherwise, the coroutine takes another step left or right down the search tree, executes the next iteration of the while loop to issue the prefetch for left or right node, and then returns back to the scheduler.

Finished: Used only by schedulers that execute a batch of coroutines that require different number of steps.

2.2 Request Parallelism

Cimple converts available request level parallelism (RLP) into memory-level parallelism (MLP) by exposing a queue of incoming requests to index routines, instead of queuing or batching in the network stack [44]. Our example workload is inspired by modern Internet servers [2, 8, 60] that process a high volume of aggregated user requests. Even though the majority of requests are for key lookups, support for range queries requires an ordered dictionary, such as a binary search tree or a skip list. Here each worker thread is given a stream of independent key lookup requests.

A coroutine *scheduler* implements a lightweight, single-threaded queue of in-flight partially completed request computations (e.g., BST lookups). The scheduler multiplexes the computations onto its thread at the granularity of steps. The queue stores the state of each partially completed computation and switches between states to multiplex the multiple computations. The Cimple implementation breaks each computation into a sequence of steps. Ideally, each step performs a sequence of local computations, followed by a prefetch or expensive memory access (e.g., an access that is typically satisfied out of the DRAM), then a yield.

Note we never wait for events, since loads are not *informing* [24]. We simply avoid reading values that might stall. This is the fundamental difference between Cimple and heavy-weight event-driven I/O schedulers. We also avoid non-predictable branches when resuming coroutine stages.

We maintain a pipeline of outstanding requests that covers the maximum memory latency. The scheduler queue has a fixed number of entries, e.g., ~50 is large enough to saturate the memory level parallelism available on current hardware platforms. The scheduler executes one step of all of queued computations. A queue refill is requested either when all lookups in a batch complete (*static scheduling* [10]), or as soon as any lookup in a batch has completed (*dynamic scheduling* [32]). The scheduler then returns back to check for and enqueue any newly arrived requests. In this way the scheduler continuously fills the queue to maximize the exploited memory level parallelism.

2.3 Cimple Execution On Modern Computing Platform

For large binary search trees, the aggregated lookup computation is memory bound. Its performance is therefore determined by the sustained rate at which it can generate the memory requests required to fetch the nodes stored in, e.g., DRAM or other remote memory. Our target class of modern microprocessors supports nonblocking cache misses, with up to ten outstanding cache misses in flight per core at any given time. The goal is therefore to maximize the number of outstanding cache misses in flight, in this computation by executing expensive memory accesses from different computations in parallel.

Here is how the example Cimple program works towards this goal. By breaking the tree traversals into steps, and using the Cimple coroutine mechanism to quickly switch between the lookup computation steps, the computation is designed to continuously generate memory requests (by issuing prefetch operations from coroutined lookups). This execution strategy is designed to generate an instruction stream that contains sequences of fast, cache-local instructions (from both the application and the Cimple coroutine scheduler) interspersed with prefetch operations. While this approach has instruction overhead (from the Cimple coroutine scheduler), the instructions execute quickly to expose the available MLP in this example.

2.4 Performance Comparison

We compare the performance of the Cimple binary tree lookup with the performance of a baseline binary tree lookup algorithm. The workload is a partitioned tree search in which each thread is given a stream of lookups to perform. The Cimple implementation interleaves multiple lookups on each thread, while the baseline executes the lookups sequentially. We use a 24 core Intel Haswell machine with 2 hyperthreads per core (see Section 8.1).

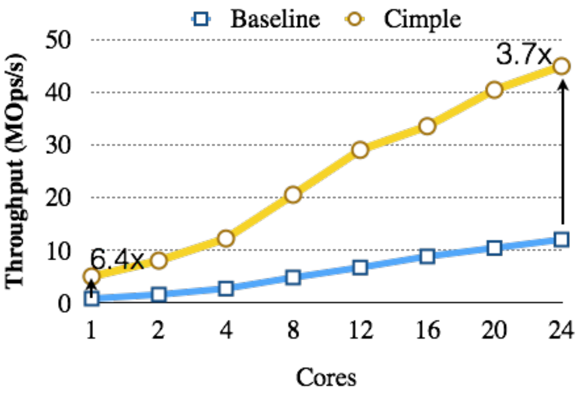


Figure 2: Throughput improvements for lookup in a partitioned binary search tree index (1GB per thread).

Figure 2 presents the results. The X axis is the number of cores executing the computation, with each core executing a single lookup thread. The Y axis presents the number of lookups in millions of operations per second. On one thread, the Cimple computation performs 6.4 times as many lookups per second as the baseline computation. This is true even though 1) due to coroutine scheduling overhead, the Cimple computation executes many more instructions than the baseline computation and 2) in theory, the baseline computation has as much memory parallelism across all requests as the Cimple computation (but the baseline MLP is unavailable to the processor because it is separated within the instruction stream by the long sequential lookups).

The performance of both computations increases up to 24 cores, with the Cimple implementation performing 3.7 times as many lookups per second as the baseline implementation (the difference narrows because the memory and coherence systems become increasingly loaded as the number of cores increases). Our machine supports two hyperthreads per core. Increasing the number of threads from 24 to 48 requires placing two threads on at least some of the cores. With this placement, the Cimple threads start interfering and performance decreases. The performance of the baseline computation increases (slowly) between 24 and 48 threads. Nevertheless, the best Cimple computation (on 24 threads) still performs 2.4 times as many operations per second as the best baseline computation (on 48 threads).

2.5 Three Key Techniques to Improve MLP and ILP

The example on Listing 1 illustrates the three essential techniques for achieving good performance with Cimple on current hardware. The first and most important is to identify independent requests and allow parallelism across them by breaking up execution at `Yield` statements (line 7). The second is to enable longer computation chains between memory requests via explicit software prefetching `Prefetch`. The third is to eliminate unpredictable branches — by replacing a control dependence (if) with an address generation dependence (line 10). Otherwise branch mispredictions would also discard unrelated (correct) subsequent coroutines, since hardware speculative execution is designed to capture the control flow of only one sequential instruction stream.

3 HARDWARE BACKGROUND

We now examine the hardware mechanisms for handling cache misses and memory level parallelism in DRAM and CPUs. The achievable MLP is further limited by the size of the buffers connecting memory hierarchy layers.

3.1 DRAM Parallelism

The two main MLP limiters are the number of DRAM banks and the size of pending request queues.

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*). The typical maximum of simultaneously open banks on a DDR4 server is 16 banks×2 ranks×(4—6) memory channels. The memory controllers track more pending requests in large queues, e.g., 48+ cache lines per memory channel [54].

While DRAM latency has stagnated, higher DRAM bandwidth and more memory controllers have kept up with providing high per-core memory bandwidth. The share of total bandwidth for cores on current Intel Xeon servers is 4–6 GB/s. Although DDR4 latency is ~50 ns, additional interconnect, cache coherence, and queuing delays add to total memory latency of 80 ns–200 ns.

3.2 Cache Miss Handling

The primary MLP limit for single threaded execution is the number of Miss Status Holding Registers (MSHR) [35], which are the hardware structures that track cache lines with outstanding cache misses. Modern processors typically have 6–10 L1 cache MSHRs: since a content-associative search is expensive in area and power, the number of MSHRs is hard to scale [72]. Intel's Haswell microarchitecture uses 10 L1 MSHRs (Line Fill Buffers) for handling outstanding L1 misses [26]. The 16 L2 MSHRs limit overall random and prefetchable sequential traffic.

For current software with low MLP, the MSHRs are hardly a bottleneck. Hardware prefetching and speculative instructions (after branch prediction) are important hardware techniques that put to use the rest of the MSHRs. Hardware prefetching is effective for sequential access – in that case a few MSHRs are sufficient to hide the access latency to the next level in the memory hierarchy. When hardware prefetches are wrong, or when mispredicted branches never need the speculatively-loaded cache lines, these techniques are wasting memory bandwidth and power.

By Little's law, the achievable bandwidth equals the number of MSHR entries divided by the average memory latency. Applications that are stalling on memory requests but do not saturate the memory bandwidth are typically considered "latency bound". More often than not, however, the real bottleneck is in the other term of Little's law - a very low queue occupancy due to low application MLP. The effective MLP of several graph frameworks is estimated in [5].

3.3 Software Prefetching

Using software prefetch instructions allows higher MLP than regular loads. The instruction reorder buffer, or any resource held up by non-retired instructions may become the limiting factor: 192-entry reorder buffer, 168 registers, 72-entry load and 42-entry store buffers on Haswell [26]. These resources are plentiful when running inefficiently one memory request at a time. Dividing the core resources over 10 parallel memory requests, however, means that each regular load can be accompanied by at most 19 μops using at most 16 registers, 7 loads and 4 memory stores.

Prefetch instructions free up the instruction window as they retire once the physical address mapping is known, e.g., either after a TLB hit, or after a page walk on a TLB miss. As soon as the virtual to physical translation has completed, an L2 memory reference using the physical address can be initiated. On current Intel microarchitectures the PREFETCH*h* family of instructions always prefetch into the L1 cache. Software prefetches are primarily limited by the number of L1 MSHR entries. Maintaining a longer queue of in-flight requests (limited by load buffers), however, helps to ensure that TLB translations of the following prefetches are ready as soon as an MSHR entry is available. If hardware performance counters show that dependent loads miss both the L1 cache and MSHRs then prefetches are too early; if loads hit MSHRs instead of L1 then prefetches are too late.

3.4 Branch Misprediction Handling

Highly mispredicted branches are detrimental to speculative execution, especially when a burst of branch mispredictions results in a short effective instruction window. Mispredicted branches that depend on long latency loads also incur a high speculation penalty. Instruction profiling with hardware performance counters can be used to precisely pinpoint such critical branches. The most portable solution for avoiding branch misprediction is to use data or address dependence instead of control dependence. While no further execution of dependent instructions is possible, independent work items can still be serviced.

Most instruction set architectures (ISAs) also support conditional move instructions (`cmov` on x86) (or `csel` on ARM), as simple cases of instruction predication. Automatic predication is also available in simple cases on IBM Power7 [67] where unpredictable branches that jump over a single integer or memory instructions are converted to a predicated conditional selection. The ternary select operator in C (`?:`) is often lowered to conditional move instructions, however, use of assembly routines is unfortunately required to ensure that mispredictable branch instructions are not emitted instead of the desired conditional move instructions.

4 DESIGN OVERVIEW

The Cimple compiler and runtime library are used via an embedded DSL similar to Halide [57], which separates the basic logic from scheduling hints to guide transformations. Similarly we build an Abstract Syntax Tree (AST) directly from succinct C++ code. Unlike Halide's expression pipelines, which have no control flow, Cimple treats expressions as opaque AST blocks and exposes conventional control flow primitives to enable our transformations. Section 5 describes our Cimple syntax in more detail.

Coroutines are simply routines that can be interleaved with other coroutines. Programmers annotate long-latency operations, e.g., memory accesses or unpredictable branches. A `Yield` statement marks the suspension points where another coroutine should run. Dynamic coroutines are emitted as routines that can be resumed at the suspension points, with an automatically generated `struct` tracking all live variables.

Listing 1 presents a traditional Binary Search Tree written in Cimple. A coroutine without any Yield statements is simply a routine, e.g., a plain C++ routine can be emitted to handle small-sized data structures, or if Yield directives are disabled. The bottleneck in Listing 1 is the expensive pointer dereference on line 8. Yet, prefetching is futile unless we context switch to another coroutine. Listing 2 presents a portable [16] unoptimized coroutine for a dynamic coroutine scheduler (Section 6.1.2).

4.1 Target Language Encapsulation

Cimple emits coroutines that can be included directly in the translation unit of the original routines. Our primary DSL target language is C++. All types and expressions are opaque; statements include opaque raw strings in the syntax of the target language, e.g., native C++.

4.2 Cimple Programming Model

A coroutine yields execution to peer coroutines only at **Yield** suspension points.

Expensive memory operations should be tagged with **Load** and **Store** statements (which may yield according to a scheduling policy), or with an explicit **`Prefetch`** directive (see Section 5.7). Loads and stores that hit caches can simply use opaque native expressions.

`If`/**`Switch`** or **`While`**/**`DoWhile`** statements should be used primarily to encapsulate mispredicted branches. Most other control-flow statements can use native selection and iteration statements.

A coroutine is invoked using a coroutine scheduler. Regular routines are called from coroutines as usual in statements and expressions. Coroutines are called from inside a coroutine with a **Call**.

4.3 Scheduling Hints

Scheduling hints guide transformations and can be added as annotations to the corresponding memory access or control statements. The example in Listing 1 showed how a single source file handles four largely orthogonal concerns. First, the program structure is described in Cimple, e.g., `While`. Second, optional inline scheduling directives are specified, e.g., `Yield`. Third, scheduler configuration can be selected via AST node handles in C++, e.g., `auto c`. Finally, all target types and expressions are used unmodified, e.g., `list*`.

4.4 Parallel Execution Model

To maintain a simple programming model, and to enable efficient scheduling (Section 5.8), Cimple coroutines are interleaved only on the creating thread. IMLP composes well with thread and task parallelism [7, 52]. Instead of running to completion just a single task, a fork-join scheduler can execute multiple coroutines concurrently. The embedded DSL approach allows easy integration with loop and task parallelism extensions, e.g., `#pragma` extensions integrated with OpenMP [52] or Cilk [7, 42].

5 CIMPLE SYNTAX AND SEMANTICS

An original C++ program is easily mapped to the conventional control flow primitives in Cimple. Table 1 summarizes our statement syntax and highlights in bold the unconventional directives.

Statement	Section
`Return`, **`Yield`**	Section 5.1
`Arg`, `SharedArg`, `Result`, `Variable`	Section 5.2
`If`, `Switch`, `Break`	Section 5.5
`While`, `DoWhile`, `Continue`	Section 5.6
`Load`, `Store`, `Assign`, **`Prefetch`**	Section 5.7
`Call`	Section 5.8

Table 1: Cimple Statements.

5.1 Coroutine Return

A coroutine may suspend and resume its execution at specified **`Yield`** suspension points, typically waiting on address generation, data, or branch resolution. Programmers must ensure that coroutines are reentrant.

`Return` stores a coroutine’s result, but does not return to the caller. It instead may resume the next runnable coroutine. **`Result`** defines the coroutine result type, or `void`.

5.2 Variable Declarations

The accessible variables at all coroutine suspension points form its context. A target routine’s internal variables need to be declared only when their use-def chains cross a yield suspension point. A **`Variable`** can be declared at any point in a block and is presumed to be live until the block end. **Arg**uments to a coroutine and its **Result** are Variables even when internal uses do not cross suspension points. Shared arguments among coroutines using the same scheduler can be marked **SharedArg** to reduce register pressure.

References in C++ allow variables to be accessed directly inside opaque expressions, e.g.:

```
Arg(int, n).Variable(int, x, {n*2})
```

For C `Variable` accesses must use a macro: `Var(a)`. We do not analyze variable uses in opaque expressions, but judicious block placements can minimize a variable’s scope.

5.3 Block Statement

A block statement encapsulates a group of statements and declarations. Convenience macros wrap the verbose Pascal-like `Begin` and `End` AST nodes, e.g., we always open a block for the **Then**/**Else** cases in `If`, **Do** in `While`, and **Body** for the function body block.

5.4 Opaque Statements and Expressions

Types and expressions used in Cimple statements are strings passed to the target compiler. Opaque statements are created from string literals, though convenient preprocessor macros or C++11 *raw strings* allow clean multi-line strings and unmodified code wrapping in a `.cimple.cpp` source file, e.g.:

```
<< R""( // Murmur3::fmix32
     h ^= h >> 16; h *= 0x85ebca6b;
     h ^= h >> 13; h *= 0xc2b2ae35;
     h ^= h >> 16;
  )""
```

5.5 Selection Statements

`If` and **`Switch`** selection statements can be used for more effective if-conversion to avoid mispredicted branches. For conventional 2-way branch and case selection, If and Switch statements give more control over branch-free if-conversion.

Well-predicted branches do not need to be exposed, and can simply use native `if/switch` in opaque statements. Opaque conditional expressions (`?:`) and standard if-conversion, which converts branches into conditional moves, are effective when only data content is impacted. Traditional predicated execution and conditional moves are less effective when address dependencies need to be hidden, especially for store addresses. Predicated execution also inefficiently duplicates both sides of a branch.

A **`Switch`** must also be used instead of a `switch` when a case has a suspension point, see Section 6.1.2.

5.6 Iteration Statements

`While` and `DoWhile` iteration statements are exposed to Cimple when there are internal suspension points to enable optimizations. Conventional `Continue` and `Break` respectively skip the rest of the body of an iteration statement, or terminate the body of the innermost iteration or `Switch` statement.

5.7 Informed Memory Operations

`Load` and **`Store`** statements mark expensive memory operations that may be processed optimally with one or more internal suspension points. **`Prefetch`** explicitly requires that one or more independent prefetches are issued before yielding. **`Assign`** can mark explicitly other assignments that are expected to be operating on cached data.

5.8 Coroutine Calls

A tail-recursive **`Call`** statement resumes execution to the initial state of a coroutine. Regular function calls can be used in all expressions, and are inlined or called as routines as usual. A Return calling a `void` coroutine is also allowed, as in C++, for explicit tail-recursion.

6 DSL COMPILER AND RUNTIME LIBRARY

The DSL allows exploration of multiple coroutine code generation variants and combinations of data layout, code structure, and runtime schedulers. We use two main code generation strategies for handling a *stage* (the code sequence between two `Yield` statements, or function entry/exit): *static* where a stage becomes a `for` loop body, and *dynamic* where a stage forms a `switch` case body. The `Yield` directive marking the boundary of a coroutine stage can select the schedule explicitly.

We first discuss the context of a single coroutine, and storage formats for tracking active and pending coroutines. Then we discuss how these are used in runtime schedulers that create, execute, and retire coroutines.

6.1 Coroutine Context

A coroutine's closure includes all private arguments and variables of a coroutine. Shared arguments between instances are stored only once per scheduler and reduce register pressure. Additional variables are optionally stored in the context depending on the code generation choices: a Finite State Machine `state` is used for dynamic scheduling on Yield; a `result` value (of user-defined type) holds the final result; a `condition` – when If yields before making decisions on hard to resolve branches; an `address` (or index) – when Load or Store yields before using a hard to resolve address.

```
struct BST::find__Context_AoS {
  node* n;      // Arg
  KeyType key; // Arg
  int    _state;  // for dynamic Yield
  node* _result; // for Return
  bool   _cond;   // for If
  void* _addr;   // for Load/Store
```

Vectorization-friendly Context Layout. The primary distinctive design choice of Cimple is that we need to run multiple coroutines in parallel, e.g., typically tens. For homogeneous coroutines we choose between Struct-of-Array (SoA), Array-of-Struct (AoS), and Array-of-Struct-of-Array (AoSoA) layouts. Variable accesses are insulated from these changes via convenient C++ references.

6.1.1 Static Fused Coroutine. Homogeneous coroutines that are at the same stage of execution can be explicitly unrolled, or simply emitted as a loop. The target compiler has full visibility inside any inlined functions to decide how to spill registers, unroll, unroll-and-jam, or vectorize. An example of SIMD vectorization of a hash function (Listing 7 in Section 7.3) is shown on Listing 3. The hash finalization function called on line 5 has a long dependence chain (shown inlined earlier in Section 5.4). C++ references to variables stored in SoA layout, shown on lines 3–4 and 9–10, allow the opaque statements to access all Variables as usual.

Exposing loop vectorization across strands offers an opportunity for performance gains. Since we commonly interleave multiple instances of the same coroutine, we can fuse replicas of the basic blocks of the same stage working on different contexts, or stitch different stages of the same coroutine, or even different coroutines. These are similar to unroll-and-jam, software pipelining, or function stitching [19]. Stage fusion benefits from exposing more vectorization opportunities, reducing scheduling overhead, and/or improving ILP.

```
1    bool SuperStep() {
2      for(int _i = 0; _i < _Width ; _i++) {
3        KeyType& k = _soa_k[_i];
4        HashType& hash = _soa_hash[_i];
5          hash = Murmur3::fmix(k);
6          hash &= mask;
7      }
8      for(int _i = 0; _i < _Width ; _i++) {
9        KeyType& k = _soa_k[_i];
10       HashType& hash = _soa_hash[_i];
11         prefetch(&ht[hash]);
12     }
```

Listing 3: Stages of a Static Coroutine for Listing 7.

Basic block vectorization, e.g., SLP [38], can be improved by better Variable layout when contexts are stored in array of struct (AoS) format.

6.1.2 Dynamic Coroutine. Coroutines may be resumed multiple times unlike one-shot continuations. Typical data structure traversals may require coroutines to be suspended and resumed between one and tens of times.

Listing 2 presents the basic structure of a `switch` based coroutine that uses "Duff's device" [16] state machine tracking. This method takes advantage of the loose syntax of `switch` statements in ANSI C. Surprisingly to some, `case` labels can be interleaved with other control flow, e.g., `while` loops or `if` statements. Only enclosed `switch` statements can not have a suspension point. Mechanical addition of case labels within the existing control flow is appealing for automatic code generation: we can decorate the original control flow graph with jump labels at coroutine suspension points and add a top level `switch` statement.

This standard C syntax allows good portability across compilers. However, the reliance on `switch` statements and labels precludes several optimization opportunities. Alternatives include relying on computed `goto` (a `gcc` extension), indirect jumps in assembly, or method function pointers as a standard-compliant implementation for C++. The first two are less portable, while the latter results in code duplication when resuming in the middle of a loop.

Short-lived coroutines suffer from branch mispredictions on stage selection. Using a `switch` statement today leaves to compiler optimizations, preferably profile guided, to decide between using a jump table, a branch tree, or a sequence of branches sorted by frequency. Unlike threaded interpreters, which benefit from correlated pairs of bytecodes, [17, 61], the potential correlation benefits from threading coroutines come from burstiness across requests. An additional optimization outside of the traditional single coroutine optimization space is to group across coroutines branches with the same outcome, e.g., executing the same stage.

6.2 Coroutine Schedulers

We discuss the salient parameters of coroutine runtime scheduler flavors, and their storage and execution constraints. We target under 100 active coroutines (*Width*) with under 100B state each to stay L1-cache resident. Below is a typical use of a simple coroutine scheduler (for Listing 6):

```
template<int Width = 48>
void SkipListIterator_Worker(size_t* answers,
                             node** iter, size_t len) {
   using Next = CoroutineState_SkipList_next_limit;
   SimplestScheduler<Width, Next>(len,
      [&](Next* cs, size_t i) {
            *cs = Next(&answers[i], IterateLimit,
                       iter[i]);
   });
}
```

Static Batch Scheduler. Tasks are prepared in batches similar to manual *group prefetching* [10, 45]. Storage is either static AoS, or in SoA format to support vectorization. Scaling to larger batches is less effective if tasks have variable completion time, e.g., on a binary search tree. Idle slots in the scheduler queue result in low effective MLP.

Dynamic Refill Scheduler. Tasks are added one by one, and refilled as soon as a task completes, similar to the manual approach in AMAC [32]. Storage is in static or dynamic-width AoS. Further optimizations are needed to reduce branch mispredictions to improve effective MLP.

Hybrid Vectorized Dynamic Scheduler. Hybrid across stages, where the first stages of a computation can use a static scheduler, but following stages use a dynamic scheduler while accessing the SoA layout.

6.2.1 Common Scheduler Interface. Runtime or user-provided schedulers implement common APIs for initialization, invoking coroutines, and draining results. A homogeneous scheduler runs identical coroutines with the same shared arguments. New tasks can either be pulled via a scheduler callback or pushed when available. A pull task with long latency operations or branch mispredictions, may become itself a bottleneck. Routines with non-`void` results can be drained either in-order or out-of-order. Interfaces are provided to drain either all previous tasks or until a particular task produces its result.

In the appendices of our extended paper [30], we show the simplest scheduler and a typical scheduler use, simple enqueue/dequeue initiated by an outside driver, and more flexible callback functors to `push`/`pull` tasks.

7 APPLICATIONS

We study Cimple's expressiveness and performance on core database data structures and algorithms. Simple near-textbook implementations in Cimple ensure correctness, while scheduling directives are used to fine-tune performance. We compare Cimple C++, against naïve native C++ and optimized baselines from recent research.

We start with a classic binary search, which is often the most efficient solution for a read-only dictionary. For a mutable index, in addition to the binary search tree we have shown in Section 2, here we show a skip list. Since both of these data structures support efficient range queries in addition to lookup, these are the default indices of VoltDB and RocksDB respectively. Finally, we show a hash table as used for database join queries.

7.1 Array Binary Search

```
1    Arg(ResultIndex*, result).
2    Arg(KeyType, k).
3    Arg(Index, l).
4    Arg(Index, r).
5    Body().
6    While( l != r ).Do(
7      Stmts(R""( {
8        int mid = (l+r)/2;
9        bool less = (a[mid] < k);
10       l = less ? (mid+1) : l;
11       r = less ? r : mid;
12        } )"").
13      Prefetch(&a[(l+r)/2]).Yield()
14   ).
15   Stmt( *result = l; );
```

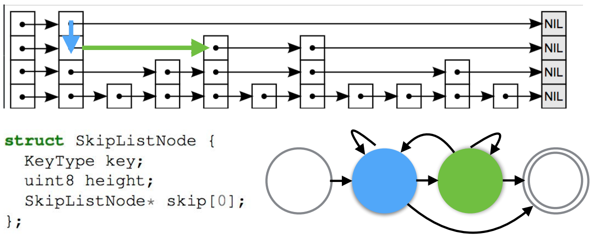
Listing 4: Cimple binary search.

Listing 4 shows our Cimple implementation. Current `clang` compilers use a conditional move for the ternary operators on lines 10–11. However, it is not possible to guarantee that compilers will not revert to using a branch, especially when compiling without Profile Guided Optimization. For finer control, programmers use provided helper functions or write inline assembly with raw statements.

Perversely, a naïve baseline performs better with a mispredicted branch as observed in [29], since speculative execution is correct 50% of the time. When speculative loads have no address dependencies, hardware aggressively prefetches useless cache lines, as we show in Section 8.3.

The Cimple version works on multiple independent binary searches over the same array. All of our prefetches or memory loads are useful.

7.2 Skip List



```
struct SkipListNode {
  KeyType key;
  uint8 height;
  SkipListNode* skip[0];
};
```

Figure 3: SkipList traversal, data layout, and coroutine state machine

```
1   VariableInit(SkipListNode*, n, {}).
2   VariableInit(uint8, ht, {pred->height}).
3   While(true).Do(
4       While(ht > 0).Do( // down
5         Stmt( n = pred->skip[ht - 1]; ).
6         Prefetch(n).Yield().
7         If(!less(k, n->key)).Then(Break()).
8         Stmt( --ht; )
9       ).
10      If (ht == 0).Then( Return( nullptr )).
11      Stmt( --ht; ).
12      While (greater(k, n->key)).Do(
13        Stmt( pred = n; n = n->skip[ht]; ).
14        Prefetch(n).Yield().
15      ).
16      If(!less(k, n->key)).Then(
17         Return( n )));
```

Listing 5: Cimple Skip List lookup.

```
While( limit-- ).Do(
    Prefetch(n).Yield().
    Stmt( n = n->skip[0]; )
).
Prefetch(n).Yield().
Return( n->key );
```

Listing 6: Cimple Skip List Iteration.

Lookup. Our skip list baseline is Facebook's folly template library implementation of ConcurrentSkipList [22]. Figure 3 shows the skip list data structure layout, and the state machine generated for the code in Listing 5; we also illustrate how a lookup follows *down* and then *right*. Note that in the *down* direction (line 5) an array of pointers is explored, therefore speculative execution in the baseline is not blocked by address dependencies; the *right* direction (line 13) cannot be speculated.

Range Query. Range queries are the main reason ordered dictionaries are used as default indices. Skip list iteration requires constant, but still inefficient, pointer chasing (Listing 6). Request level parallelism in range queries is handled similarly to lookup by interleaving multiple independent queries for both finding the first node and for iterating and aggregating over successive nodes.

7.3 Hash tables

```
1   Result(KeyValue*).
2   Arg(KeyType, k).
3   Variable(HashType, hash).
4   Body().
5     Stmt ( hash = Murmur3::fmix(k); ).
6     Stmt ( hash &= this->size_1; ).Yield().
7     Prefetch( &ht[hash] ).Yield()
8     << R""(
9     while (ht[hash].key != k &&
10           ht[hash].key != 0) {
11        hash++;
12        if (hash == size) hash = 0;
13    } )"" <<
14    Return( &ht[hash] );
```

Listing 7: Cimple Hash Table lookup (linear probing).

We compare the performance of an open-address hash table for the special case of database hash-join. An ephemeral hash table optimized for hash-join [3] only needs to support bulk insertion followed by a phase of lookups. The identity hash function can not be used in real workloads, both for performance due to non-uniform skew, and for security due to Denial-of-Service complexity attacks [14].

Listing 7 shows our classic linear probing hash table, similar to the implementation suggested in Menon et al. [45] — linear probing at 50% load factor, and Murmur3's finalization as a hash, masked to the table size. Menon et al. report 1.2× gains from LLVM SIMD vectorization and group prefetching [10], on a well-engineered hash table for state-of-the-art TPC-H performance.

A requested static schedule for all three stages (delineated by Yield on lines 6 and 7 of Listing 7) generates three independent static stages (shown in [30]). Using the SoA layout enables compiler loop vectorization to use AVX2 or AVX512 to calculate multiple hash functions simultaneously.

Variants. Menon et al. [45] analyze the inefficient baseline used in AMAC [32], i.e., identity hash, chaining at 400% load factor, and using a linked list for handling duplicates.

For a chained hash table, which traverses a linked list, we can also produce a hybrid schedule. The first two steps use a static schedule (with SoA storage), while the third stage can use a dynamic scheduler to handle a variable number of cache lines traversed.

8 EVALUATION

We report and analyze our performance gains using Cimple used as a template library generator. Our peak system throughput increases from 1.3× on HashTable to 2.5× on SkipList iteration; Cimple speedups of the time to complete a batch of queries on a single thread range from 1.2× on HashTable to 6.4× on BinaryTree (Figure 1).

8.1 Configuration

System Configuration. We used a dual socket Haswell system [25] with 24 cores at 2.9GHz clock frequency, or 3.3GHz for a single core. Each socket has 4 memory channels populated with dual rank, 16-bank DDR4-2133 memory [62]. The DRAM memory level parallelism on each socket therefore allows 128 open banks for random access. The test applications were compiled with `gcc` 4.8 and executed on Linux 4.4 using huge pages.

Cimple Configuration. We implemented the Cimple DSL in a combination of 2,500 lines of C++14 and 300 lines of C preprocessor macros. Cimple to C++ code was built with Apple clang 9.0. The template library of runtime schedulers adds less than 500 lines of C++11 code. Cimple suspend/resume of the minimum coroutine step (on **SLi** — 21 extra instructions) adds 4ns. We use 48 entry scheduler width — optimal for all DRAM-resident benchmarks.

8.2 Performance Gains

Binary Search (BS). The multithreaded version has all threads searching from the same shared array of 1 billion 64-bit keys. Branch-free execution is important for good performance as discussed in Section 2.5. When a branch is used on lines 10 and 11 of Listing 4, we see only a 3× performance gain. While CMOV in the baseline leads to a 0.7× slowdown, Cimple+CMOV reaches 4.5× over the best baseline.

Binary Tree lookup (BT). Each thread works on a private tree to avoid synchronization, as used in the context of partitioned single-threaded data stores, such as VoltDB or Redis. We use 1GB indexes scaled by the number of threads, i.e., 48GB for the full system. We achieve 2.4× higher peak throughput and 6.4× speedup for a single thread of execution. Our ability to boost a single thread performance much higher above average, will support handling of skewed or bursty workloads, which can otherwise cause significant degradation for partitioned stores [70].

SkipList lookup (SL). Concurrent SkipLists are much easier to scale and implement [22] compared to a binary tree, therefore practical applications use multiple threads looking up items in a shared SkipList.

All items are found after a phase of insertions with no deletions or other sources of memory fragmentation. We achieve 2.7× single thread speedup and 1.8× multithreaded throughput. Note that for SkipList lookup the “down” direction follows an array of pointers, therefore the baseline benefits from speculative execution prefetching nodes.

SkipList Iterator (SLi). We evaluated range queries on the same shared skip list index as above. For 1,000 node limit iterations, similar to long range queries in [2, 69] our total throughput gain is 2.5× and single thread speedup is 4.1×.

Hash Table lookup (HT). We evaluate hash table join performance on a table with 64-bit integer keys and values. We use a 16 GB hash table shared among all threads for an effective load factor of 48%. We replicate similar single thread speedups [45] of 1.2× when either no results or all results are materialized. Since there are few instructions needed to compare and store integer keys and values, hardware is already very effective at keeping a high number of outstanding requests. However, both the hash table load factor and the percentage of successful lookups impact branch predictability, and thus ILP and MLP for the baseline. For 50% materialized results, our speedup is 1.3×. When using 48 threads with 100% hits, we get a 1.3× higher throughput of 650 M operations/s.

We also compared to other traditional but inefficient on modern cores variants, e.g., if division by a prime number is used [10] the corresponding Cimple variant is 2× faster. When there are serializing instructions between lookups our speedup is 4×.

8.3 Performance Analysis

We analyze hardware performance counters to understand where our transformations increase effective ILP and MLP.

8.3.1 ILP Improvements.

Table 2 shows our improvements in ILP and IPC by increasing the useful work per cycle and reducing the total number of cycles. The ILP metric measures the average μinstructions executed when not stalled (max 4). Cimple may have either higher or lower instruction count: e.g., a pointer dereference in **SLi** is a single instruction, while with a dynamic scheduler that instruction is replaced by context switches with attendant register spills and restores. For a static scheduler, vector instructions reduce additional instructions in **HT**. The remaining stall cycles show that there is sufficient headroom for more expensive computations per load.

Benchmark	MLP		ILP		IPC	
	B	C	B	C	B	C
BS	7.5	8.5	1.6	2.3	0.13	1.10
BT	1.2	4.3	1.6	2.3	0.10	0.70
SL	2	5	1.8	2.4	0.07	0.60
SLi	1	5	1.3	2.0	0.01	0.22
HT	4.9	6.4	1.9	2.4	0.37	0.40

Table 2: Memory Level Parallelism (MLP), Instruction Level Parallelism (ILP), and Instructions Per Cycle (IPC). Baseline (B) vs Cimple (C).

8.3.2 MLP Improvements. Improving MLP lowers the stall penalty per miss, since up to 10 outstanding L1 cache misses per core can be overlapped.

In Table 2 we show that measured MLP improved by 1.3–6× with Cimple. Measured as the average outstanding L2 misses, this metric includes speculative and prefetch requests. Therefore the baseline MLP may be inflated due to speculative execution which does not always translate to performance. Cimple avoids most wasteful prefetching and speculation, therefore end-to-end performance gains may be larger than MLP gains.

In BinarySearch the baseline has high measured MLP due to speculation and prefetching, however, most of it is not contributing to effective MLP. For BinaryTree the addresses of the children cannot be predicted, therefore the baseline has low MLP. For SkipList lookup the down direction is an array of pointers therefore speculative execution may prefetch correctly needed values, thus while the measured MLP is 2, the effective MLP is 1.5. SkipList iteration is following pointers and therefore has MLP of 1. For HashTable at low load and 100% hit rate, speculative execution is always correct, thus the baseline has high effective MLP.

There is also sufficient headroom in memory bandwidth and queue depth for sequential input and output streams, e.g., for copying larger payload values.

9 RELATED WORK

We survey related work in hardware multithreading, coroutines and tasks, and software optimizations.

9.1 Hardware Multithreading

Hardware context switching was explored in supercomputers of the lineage of Denelcor HEP [68] and Tera MTA [1], e.g., Tera MTA supported 128 instruction streams that were sufficient to hide the latency of 70 cycles of DRAM latency without using caches. Yet locality is present in real workloads, and caches should be used to capture different tiers of frequently used data. Larrabee [63] threading and vectorization model allowed SIMD rebundling to maintain task efficiency. Current GPUs offer large number of hardware threads, yet relying solely on thread-level parallelism is insufficient [74], and taking advantage of ILP and MLP is critical for GPU assembly-optimized libraries [37, 47].

Out-of-order CPUs can track the concurrent execution of tens of co-running coroutines per core, but provide no efficient notification of operation completion. Informing loads [24] were proposed as a change to the memory abstraction to allow hardware to set a flag on a cache miss and trap to a software cache-miss handler, similar to a TLB-miss handler. Proposals for hardware support for overlapping instructions from different phases of execution with compiler transformations have shown modest performance gains [53, 58, 65, 71, 73].

9.2 Coroutines and Tasks

Coroutines have been a low-level assembly technique since the 1950s, originally used in limited-memory stackless environments [49, 50]. Lowering continuation overhead has been approached by specialized allocation [23] and partial [56] or one-shot [9] continuations.

The C++20 standard is also slated to support coroutines with the keywords `co_yield`, `co_await`, and `co_return`. The original proposals [20, 51] motivate the goal to make asynchronous I/O maintainable. The runtime support for completion tracking is acceptable at millisecond scale for handling network and disk, but is too heavy weight for tolerating tens to hundreds of nanoseconds memory delays targeted by IMLP tasks. The concise syntax and automatic state capture are attractive and the underlying mechanisms in LLVM and Clang can be used to add Cimple as non-standard C++ extensions to delight library writers. Library users can use the generated libraries with mainstream compilers.

Cilk [7, 18] introduced an easy programming model for fork-join task parallelism, divide-and-conquer recursive task creation and work-stealing scheduler. More recently the CilkPlus [42] extensions to C/C++ were added to `icc` and `gcc`. C++20 proposals for `task_block` [21] incorporate task parallelism like Cilk, albeit using a less succinct syntax.

Our rough guide to these overlapping programming models would be to use C++20 tasks for compute bottlenecks, C++20 coroutines for I/O, and Cimple coroutines for memory bottlenecks.

9.3 Software Optimizations

Software prefetching by requesting data at a fixed distance ahead of the current execution is complex even for simple loop nests and reference streams without indirection [46]; and more recently surveyed in [40]. Augmented data structures help deeper pointer chasing [12, 33].

Optimized-index data-structures for in-memory databases [11, 28, 36, 39, 41, 43, 59, 64] try to reduce the depth of indirect memory references and use high fan-out and extra contiguous accesses while performing one-at-a-time requests. Techniques that uncover spatial or temporal locality by reordering billions of memory requests [6, 31] are not applicable to index queries which often touch only a few rows.

Group prefetching (static scheduling) and prefetching with software pipelining techniques were introduced in [10] where a group of hash table lookups are processed as a vector; similarly used for an unordered key value store [44]. AMAC [32] is an extension to group prefetching to immediately refill completed tasks (dynamic scheduling) in order to handle better variable work or variable access time per-item on skewed inputs. In a well-engineered baseline in the state-of-the-art database engine Peloton [45], however, AMAC was deemed ineffective and only group prefetching on hash tables was beneficial and maintainable.

Using C++20 coroutines for easier programmability of AMAC-style dynamic scheduling was evaluated in concurrent work in SAP HANA [55], and more recently by [27]. While easier to use and more maintainable than manual interleaving methods [10, 32], C++20 coroutine backends preclude static or hybrid scheduling. Dependence on I/O-oriented compiler coroutine implementations adds high overhead compared to manual AMAC [32]; C++20 coroutines [27, 55] are even outperformed by static scheduling [10] in some cases in which dynamic scheduling with AMAC is otherwise better than static scheduling. Using a black-box compiler is also not practical for JIT query engines used in modern databases for these critical inner loops. For less critical code-paths implemented in C++, their promising results are a step in the right direction. We expect to be able to offer a similar C++ front-end, once coroutines are mature in Clang, with additional Cimple AST annotations as C++ `[[attributes]]`. Cimple's back-end seamlessly enables static and hybrid scheduling, with efficient dynamic scheduling coroutines optimized for caches and out-of-order processors.

10 CONCLUSION

Cimple is fast, maintainable, and portable. We offer an optimization methodology for experts, and a tool usable by end-users today.

We introduced the IMLP programming model and methodology for uncovering ILP and MLP in pointer-rich and branch-heavy data structures and algorithms. Our Cimple DSL and its AST transformations for C/C++ in Cimple allow quick exploration of high performance execution schedules. Cimple coroutine annotations mark hotspots with deep pointer dereferences or long dependence chains. Cimple achieves up to 6.4× speedup.

Our compiler-independent DSL allows low-level programmers to generate high-performance libraries that can be used by enterprise developers using standard tool-chains. We believe ours and others' promising early results are the first steps towards efficient future Coroutines for Instruction and Memory Parallel Language Extensions to C++.

ACKNOWLEDGMENTS

This material is based upon work supported by DOE under Grant No. DE-SC0014204, and the Toyota Research Institute under Grant No. LP-C000765-SR.

Thanks to Carl Waldspurger for his valuable feedback on early versions of this manuscript. We also thank our anonymous reviewers and Georgios Psaropoulos for helpful questions and suggestions for improving our presentation.

REFERENCES

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera Computer System. In *Proceedings of the 4th International Conference on Supercomputing (ICS '90)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/77726.255132

[2] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1185–1196. https://doi.org/10.1145/2463676.2465296

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1754–1766. https://doi.org/10.1109/TKDE.2014.2313874

[4] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (March 2017), 48–54. https://doi.org/10.1145/3015146

[5] Scott Beamer, Krste Asanović, and David A. Patterson. 2015. 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*. 56–65. https://doi.org/10.1109/IISWC.2015.12

[6] Scott Beamer, Krste Asanović, and David A. Patterson. 2017. Reducing PageRank Communication via Propagation Blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 820–831. https://doi.org/10.1109/IPDPS.2017.112

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.* 30, 8 (Aug. 1995), 207–216. https://doi.org/10.1145/209937.209958

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 49–60. http://dl.acm.org/citation.cfm?id=2535461.2535468

[9] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-shot Continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, New York, NY, USA, 99–107. https://doi.org/10.1145/231379.231395

[10] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2004. Improving Hash Join Performance Through Prefetching. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. IEEE Computer Society, Washington, DC, USA, 116–. http://dl.acm.org/citation.cfm?id=977401.978128

[11] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving Index Performance Through Prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. ACM, New York, NY, USA, 235–246. https://doi.org/10.1145/375663.375688

[12] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. 1999. Cache-conscious Structure Layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/301618.301633

[13] Cloudera. 2013. Inside Cloudera Impala: Runtime Code Generation. http://blog.cloudera.com/blog/2013/02/inside-cloudera-impala-runtime-code-generation/. (2013).

[14] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1251353.1251356

[15] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[16] Tom Duff. 1988. Duff's Device. http://doc.cat-v.org/bell_labs/duffs_device. (1988).

[17] M. Anton Ertl and David Gregg. 2003. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 278–288. https://doi.org/10.1145/781131.781162

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

[19] Vinodh Gopal, Wajdi Feghali, Jim Guilford, Erdinc Ozturk, Gil Wolrich, Martin Dixon, Max Locktyukhin, and Maxim Perminov. 2010. Fast Cryptographic Computation on Intel Architecture Processors Via Function Stitching. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-cryptographic-paper.pdf. (2010).

[20] Niklas Gustafsson, Deon Brewis, and Herb Sutter. 2014. Resumable Functions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3858.pdf. (2014).

[21] Pablo Halpern, Arch Robison, Hong Hong, Artur Laksberg, Gor Nishanov, and Herb Sutter. 2015. Task Block (formerly Task Region) R4. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf. (2015).

[22] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2006. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*.

[23] R. Hieb, R. Kent Dybvig, and Carl Bruggeman. 1990. Representing Control in the Presence of First-class Continuations. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/93542.93554

[24] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. 1996. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*. ACM, New York, NY, USA, 260–270. https://doi.org/10.1145/232973.233000

[25] Intel. 2015. Intel Xeon Processor E5-2680 v3(30M Cache, 2.50 GHz). http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz. (2015).

[26] Intel. 2017. 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. (2017).

[27] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *In Proceedings of the 44th International Conference on Very Large Data Bases (VLDB'18), August 2018, Rio de Janeiro, Brazil, VLDB Endowment* 11, 11 (2018), 1702–1714.

[28] Alfons Kemper, Thomas Neumann, Jan Finis, Florian Funke, Viktor Leis, Henrik Mühe, Tobias Mühlbauer, and Wolf Rödiger. 2013. Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *IEEE Data Eng. Bull.* 36, 2 (2013), 41–47. http://sites.computer.org/debull/A13june/hyper1.pdf

[29] Paul-Virak Khuong and Pat Morin. 2017. Array Layouts for Comparison-Based Searching. *J. Exp. Algorithmics* 22, Article 1.3 (May 2017), 39 pages. https://doi.org/10.1145/3053370

[30] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. 2018. Cimple: Instruction and Memory Level Parallelism. *ArXiv e-prints* (July 2018). arXiv:1807.01624

[31] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. 2016. Optimizing Indirect Memory References with Milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 299–312. https://doi.org/10.1145/2967938.2967948

[32] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 252–263. https://doi.org/10.14778/2856318.2856321

[33] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. 2001. Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*. IEEE Computer Society, Washington, DC, USA, 268–279. http://dl.acm.org/citation.cfm?id=645988.674157

[34] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf

[35] David Kroft. 1981. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA '81)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 81–87. http://dl.acm.org/citation.cfm?id=800052.801868

[36] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-optimized Databases Using Multi-core CPUs. *Proc. VLDB Endow.* 5, 1 (Sept. 2011), 61–72. https://doi.org/10.14778/2047485.2047491

[37] Junjie Lai and Andre Seznec. 2013. Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/CGO.2013.6494986

[38] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[39] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. *PVLDB* 8, 12 (2015), 1740–1751. http://www.vldb.org/pvldb/vol8/p1740-Larson.pdf

[40] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9, 1, Article 2 (March 2012), 29 pages. https://doi.org/10.1145/2133382.2133384

[41] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[42] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *The Journal of Supercomputing* 51, 3 (2010), 244–257. https://doi.org/10.1007/s11227-010-0405-3

[43] Justin Levandoski, David Lomet, Sudipta Sengupta, Adrian Birka, and Cristian Diaconu. 2014. Indexing on Modern Hardware: Hekaton and Beyond. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2014*. ACM. http://research.microsoft.com/apps/pubs/default.aspx?id=213089

[44] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 476–488. https://doi.org/10.1145/2749469.2750416

[45] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.* 11 (September 2017), 1–13. Issue 1. http://www.vldb.org/pvldb/vol11/p1-menon.pdf

[46] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/143365.143488

[47] Nervana. 2017. SGEMM. https://github.com/NervanaSystems/maxas/wiki/SGEMM. (2017).

[48] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[49] A. Newell and H. Simon. 1956. The logic theory machine–A complex information processing system. *IRE Transactions on Information Theory* 2, 3 (September 1956), 61–79. https://doi.org/10.1109/TIT.1956.1056797

[50] A. Newell and F. M. Tonge. 1960. An Introduction to Information Processing Language V. *Commun. ACM* 3, 4 (April 1960), 205–211. https://doi.org/10.1145/367177.367205

[51] Gor Nishanov and Jim Radigan. 2014. Resumable Functions v.2. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4134.pdf. (2014).

[52] OpenMP. 2015. OpenMP Application Program Interface 4.5. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. (2015).

[53] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE Computer Society, Washington, DC, USA, 105–118. https://doi.org/10.1109/MICRO.2005.13

[54] I.E. Papazian, S. Kottapalli, J. Baxter, J. Chamberlain, G. Vedaraman, and B. Morris. 2015. Ivy Bridge Server: A Converged Design. *Micro, IEEE* 35, 2 (Mar 2015), 16–25. https://doi.org/10.1109/MM.2015.33

[55] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with coroutines: a practical approach for robust index joins. *In Proceedings of the 44th International Conference on Very Large Data Bases (VLDB'18), August 2018, Rio de Janeiro, Brazil, VLDB Endowment* 11, 2 (2017), 230–242.

[56] Christian Queinnec and Bernard Serpette. 1991. A Dynamic Extent Control Operator for Partial Continuations. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*. ACM, New York, NY, USA, 174–184. https://doi.org/10.1145/99583.99610

[57] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[58] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage Decoupled Software Pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*. ACM, New York, NY, USA, 114–123. https://doi.org/10.1145/1356058.1356074

[59] Jun Rao and Kenneth A. Ross. 2000. Making B+- Trees Cache Conscious in Main Memory. *SIGMOD Rec.* 29, 2 (May 2000), 475–486. https://doi.org/10.1145/335191.335449

[60] RocksDB. 2017. RocksDB. http://rocksdb.org/. (2017).

[61] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. 2015. Branch Prediction and the Performance of Interpreters: Don't Trust Folklore. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 103–114. http://dl.acm.org/citation.cfm?id=2738600.2738614

[62] Samsung. 2015. 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).

[63] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH '08)*. ACM, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/1399504.1360617

[64] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB* 4, 11 (2011), 795–806. http://www.vldb.org/pvldb/vol4/p795-sewall.pdf

[65] Rami Sheikh, James Tuck, and Eric Rotenberg. 2012. Control-Flow Decoupling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 329–340. https://doi.org/10.1109/MICRO.2012.38

[66] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle LittleïňĄeld, and Phoenix Tong. 2012. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. In *SIGMOD*. Talk given at SIGMOD 2012.

[67] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. 2011. IBM POWER7 Multicore Server Processor. *IBM J. Res. Dev.* 55, 3 (May 2011), 191–219. https://doi.org/10.1147/JRD.2011.2127330

[68] B J Smith. 1986. Advanced Computer Architecture. IEEE Computer Society Press, Los Alamitos, CA, USA, Chapter A Pipelined, Shared Resource MIMD Computer, 39–41. http://dl.acm.org/citation.cfm?id=17956.17961

[69] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. 2016. Cache-sensitive skip list: Efficient range queries on modern CPUs. In *International Workshop on In-Memory Data Management and Analytics*. Springer, 1–17.

[70] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[71] K. A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean. 2017. Clairvoyance: Look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 171–184. https://doi.org/10.1109/CGO.2017.7863738

[72] James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 409–422. https://doi.org/10.1109/MICRO.2006.44

[73] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative Decoupled Software Pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE Computer Society, Washington, DC, USA, 49–59. https://doi.org/10.1109/PACT.2007.66

[74] Vasily Volkov. 2010. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC'10*. http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf