ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Spring 2024

Implementing an OpenMP runtime for MemPool in LLVM

Semester Project

Diego de los Santos Gausí
dgausi@student.ethz.ch

1st of July 2024

Advisors: Samuel Riedel, sriedel@iis.ee.ethz.ch
Sergio Mazzola, smazzola@iis.ee.ethz.ch

Professor: Prof. Dr. L. Benini, lbenini@iis.ee.ethz.ch

Todo list

Abstract

In order to effectively harness MemPool's parallelism, an initial GCC-compatible OpenMP runtime was developed for this architecture, supporting the most commonly used set of features. However, due to increasing reliance on the LLVM compiler infrastructure by the MemPool team, it is becoming crucial to have an OpenMP runtime compatible for it. This project aims to implement an OpenMP runtime that works with LLVM, supporting at least the same set of features as the previous runtime, while maintaining comparable performance.

Acknowledgments

I'd like to sincerely thank Samuel Riedel and Sergio Mazzola for supervising this project. Their continuous feedback and encouragement has been very helpful, and it enabled me to stay on track and make steady progress.

I'd also like to thank Frank Gürkaynak for getting me in touch with the group, which led to me learning about this project.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

☐ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☐ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

☒ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Implementing an OpenMP runtime for MemPool in LLVM

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
de los Santos Gasi	Diego

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 30.06.24	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard

Contents

Chapter 1

Introduction

As demand for processing of highly parallel workloads grows over time, computing architectures with increasing core/processing element counts become more common. These can range all the way from custom accelerators with domain-specific architectures and programming models, to general-purpose multi-core processors. There is generally a trade-off between ease of programming and performance/efficiency.

One can achieve an interesting middle ground by grouping many relatively simple general purpose cores into a so called *compute cluster*. In practice, however, this approach does not scale well beyond a few tens of cores, which lead to the development of *multi-cluster* architectures comprised of many of such clusters. While this allows for greater core counts, it also introduces additional complexity and memory access overhead between each of them.

MemPool [1] tries to address this issue by scaling a single cluster to the order of hundreds of cores. It is an open-source manycore architecture developed at ETH, which is comprised of up to 1024 [2] 32-bit RISC-V Snitch [3] cores that share a common L1 cache. Unlike on GPUs, each core is individually programmable, therefore enabling the familiar shared memory programming model.

Although MemPool's parallelism can be harnessed by manually managing it using the primitives provided by MemPool's C runtime library, this can become a tedious and error prone task since the programmer has to be knowledgeable about the underlying architecture. Luckily, higher level abstraction layers such as OpenMP [4] allow the programmer to express this parallelism in a clear and concise way using the fork-join paradigm and without needing advanced architectural knowledge. This is what makes OpenMP portable cross architectures and therefore widely used.

OpenMP is comprised of a set of compiler directives for both C/C++ and Fortran that enable this higher level abstraction. In order to use OpenMP, one has to use a compatible compiler that is able to roughly do the following:

1. The compiler must be able to parse the directives and compile the surrounding code according to their semantics.
2. The output of the compilation process must match the target platform in order to make use of its parallelism capabilities and primitives.

Both the GNU Compiler Collection (GCC) and LLVM provide an OpenMP runtime library interface that bridges the gap between both points: instead of directly generating a binary from OpenMP annotated code, the compiler generates calls to the OpenMP runtime library. This library has to be aware of how to parallelize the code given the platform, which makes it generally non-portable, unless it can use parallelism primitives provided by an Operating System (OS) or a Hardware Abstraction Layer (HAL). Naturally, popular compilers ship with their own OpenMP runtime library implementation, however, they target a host running an operating system which is able to provide the necessary parallelism primitives. Since MemPool is designed to run bare-metal applications, this means that a custom OpenMP runtime library has to be developed.

Previously, the authors of MemPool developed a GCC compatible OpenMP runtime library with support for the most commonly used OpenMP constructs, such as *work sharing*, *critical sections* and *atomics*. This implementation will be discussed in more detail in Section 3.1. However, due to the growing popularity of the LLVM compiler infrastructure in research because of its modularity and ease of use, it is essential to have a compatible OpenMP runtime library for LLVM as well.

The goal of this project is to implement an OpenMP runtime for LLVM that works on MemPool, supports at least the same set of features as the previous GCC runtime and achieves comparable performance.

In this work, we present the following contributions:

- A C++ implementation of an OpenMP runtime library compatible with LLVM that achieves feature parity with the previous GCC runtime and adds support for `teams` constructs (Chapter 4).
- An evaluation of the new runtime in terms of performance and correctness against the GCC runtime (Chapter 5).
- A small testing framework for running and evaluating runtime tests automatically (Section 5.1.1).
- C++ support for MemPool (Section 4.4.1).

Chapter 2

Background

2.1. MemPool

2.1.1. Architecture Overview

At the heart of MemPool lies the *Snitch* core, which is a simple, in-order, 32-bit RISC-V core. This core was adapted for this architecture by adding the ability to retire out-of-order load instructions, which, in addition to its previously existing support for issuing multiple outstanding loads and stores through scoreboarding, allows MemPool to completely hide the L1 access latency.

MemPool's architecture is hierarchical and comprised of the following building blocks:

Tile A tile is made up of a number of cores, some shared L1 banks, an L1 instruction cache shared amongst the tile's cores, as well as a set of remote ports to send/receive requests to/from other tiles trying to access the global L1 cache. Although any region of the L1 cache is accessible by any tile, obtaining data from remote tiles is more costly than reading/writing it directly from/to the local banks. Finally, it also has an Advanced eXtensible Interface (AXI) port connected to the system bus, which allows the tile to access the global L2 cache or system memory.

Group A group is made up of multiple tiles. Each tile's remote ports are connected with each other both within the same group as well as with other groups through a series of interconnects. Each tile's AXI port is connected to the rest of the system, also supporting Direct Memory Access (DMA).

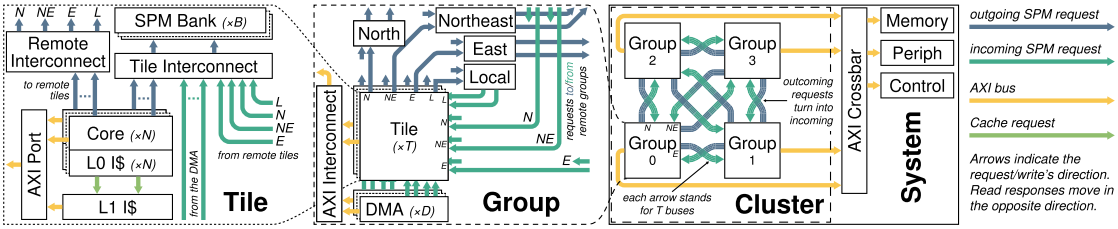

Figure 2.1.: MemPool's Architecture Hierarchy.

Cluster Finally, a cluster is made up of multiple groups. Here, an AXI interface is used to connect MemPool to the rest of the system (e.g., L2 cache, system memory, etc.).

Another interesting aspect of MemPool's architecture is the *hybrid memory addressing scheme*. MemPool's shared L1 memory is interleaved at the word level across all banks. This means that accessing any given sequential region of memory would result in many strided accesses to remote tiles. This is not an issue for data shared across all cores, however, it is needlessly costly for local data (e.g., the stack). For this reason, MemPool is able to dedicate *sequential regions* of memory that are interleaved only within the same tile by re-interpreting the memory address, which is entirely done in hardware.

2.1.2. Programming Model

MemPool aims to provide a conceptually simple and familiar programming model to ease the development of parallel applications. It is based on the following principles:

1. Shared memory: MemPool provides a shared memory programming model, which means that all cores have access to the full shared memory space.
2. Independently programmable cores: Each core is individually programmable, which means that each core can execute different code at the same time unlike on GPUs or SIMD architectures.

2.2. OpenMP

In her report [5], Qiu provides a comprehensive overview of OpenMP and its most commonly used constructs and clauses. In this work, we will implement all of the ones mentioned in her work, as well as the *teams* construct, described in the following:

The *teams* construct uses the following syntax:

Listing 2.1: teams construct

```
1 #pragma omp teams [clause[ [,] clause] ... ] new-line
2   structured-block
```

with the following clauses:

Listing 2.2: teams clauses

```
num_teams(scalar-integer-expression)

thread_limit(scalar-integer-expression)

default(shared | firstprivate | private | none)

private(list)

firstprivate(list)

shared(list)

reduction([default ,] reduction-identifier : list)

allocate([allocator :] list)
```

The *teams* construct is used to create a league of teams, where each team is a group of threads. The number of teams is less then or equal to the value defined by the *num_teams* clause, and the maximum number of threads in each team is less than or equal to the value defined by the *thread_limit* clause.

The *structured_block* is able to contain other OpenMP constructs such as *parallel*, meaning that this construct essentially allows the creation of independent groups of threads, each of which could work on different tasks. This is particularly useful in the context of MemPool, as it allows for the creation of multiple teams, each of which could be assigned to a different tile or group (or even to different clusters in a multi-cluster system).

Chapter 3

Related Work

3.1. GCC Implementation

As previously mentioned, MemPool already had a working OpenMP runtime library for GCC. In the following, we will briefly discuss its architecture and implementation.

3.1.1. Event Loop

Because the entry point of the application is the same for all cores, it is necessary to distinguish between master and worker cores at runtime. Therefore, every application intended to use the GCC OpenMP runtime has to include the code shown in Listing 3.1 in its main file. Essentially, it distinguishes between two cases:

1. If the current core has ID 0 (i.e., it is the master core), it will continue executing standard OpenMP code (i.e., containing `#pragma omp parallel` directives, etc.).
2. Otherwise, it will enter an infinite loop, where the core first goes to sleep and waits for an interrupt, and then participates in the parallel execution of the program by running its assigned task. Waking up other cores and assigning tasks is done by the master core when it encounters OpenMP constructs that require it.

Listing 3.1: Main Event Loop

```
1 uint32_t core_id = mempool_get_core_id();
2
3 if (core_id == 0) {
4   /* OpenMP Code */
5 } else {
6   while (1) {
```

```
7       mempool_wfi();
8       run_task(core_id);
9     }
10  }
```

Listing 3.2: run_task Implementation

```
1   typedef struct {
2     void (*fn)(void *);
3     void *data;
4     uint32_t nthreads;
5     uint32_t barrier;
6     uint8_t thread_pool[NUM_CORES];
7   } event_t;
8
9   void run_task(uint32_t core_id) {
10    if (event.thread_pool[core_id]) {
11      event.fn(event.data);__
12      atomic_add_fetch(&event.barrier, -1, __ATOMIC_SEQ_CST);
13    }
14  }
```

Listing 3.2 shows the implementation of `run_task` as well as `event_t`. Since this implementation assumes at most a single team (i.e., group of threads) working on a single task, it uses a global variable `event` of type `event_t` to store the task information.

The thread first checks if it is supposed to participate in the parallel execution by checking whether `event.thread_pool[core_id]` is set. If it is, it executes the function pointed to by `event.fn` and decrements the barrier, which makes sure that all threads have finished their work before starting the next task.

Listing 3.3: set_event Implementation

```
1   void set_event(void (*fn)(void *), void *data, uint32_t nthreads) {
2     uint32_t num_cores = mempool_get_core_count();
3     event.fn = fn;
4     event.data = data;
5     if (nthreads == 0) {
6       event.nthreads = num_cores;
7       event.barrier = num_cores;
8     } else {
9       event.nthreads = nthreads;
10      event.barrier = nthreads;
11    }
12
13    for (uint32_t i = 0; i < num_cores; i++) {
```

```
14      event.thread_pool[i] = (i < event.nthreads) ? 1 : 0;
15    }
16  }
```

A task is set when the master thread calls `set_event` (Listing 3.3) after encountering an OpenMP construct that requires it. The function pointer and its arguments are passed through `fn` and `data`, respectively. The number of threads that should participate in the parallel execution is passed through `nthreads`. If `nthreads` is 0, all cores will execute it. This is controlled by the `event.thread_pool` array.

3.1.2. Parallel Regions

OpenMP parallel constructs will be transformed into calls to `GOMP_parallel`, which is responsible for setting the current task, waking up all cores, and waiting for them to finish. Note that this function is only called by the master thread.

Listing 3.4: GOMP_parallel Implementation

```
1  void GOMP_parallel_start(void (*fn)(void *), void *data,
2                           unsigned int num_threads) {
3    set_event(fn, data, num_threads);
4    wake_up_all();
5    mempool_wfi();
6  }
7
8  void GOMP_parallel_end(void) {
9    uint32_t num_cores = mempool_get_core_count();
10   while (event.barrier > 0) {
11     mempool_wait(4 * num_cores);
12   }
13 }
14
15 void GOMP_parallel(void (*fn)(void *), void *data,
16                    unsigned int num_threads,
17                    unsigned int flags) {
18   uint32_t core_id = mempool_get_core_id();
19   gomp_new_work_share();
20   GOMP_parallel_start(fn, data, num_threads);
21   run_task(core_id);
22   GOMP_parallel_end();
23 }
```

3.1.3. Work Sharing Constructs

Work sharing constructs (e.g., `for`, `sections`, etc.) allow for threads in the current team to distribute the work to be done in a given parallel region. A global variable of type `work_t` is used for the bookkeeping related to these constructs and is initialized by calling `gomp_new_work_share` (Listing 3.4, Line 19).

Listing 3.5: work_t Struct

```
1  typedef struct {
2    int end;
3    int next;
4    int chunk_size;
5    int incr;
6
7    omp_lock_t lock;
8
9    // for single construct
10   uint32_t checkfirst;
11   uint32_t completed;
12   void *copyprivate;
13
14   // for critical construct
15   omp_lock_t critical_lock;
16
17   // for atomic construct
18   omp_lock_t atomic_lock;
19 } work_t;
```

As an example for a work sharing construct, Listing 3.6 shows the implementation of the functions required for a dynamically scheduled for loop construct (i.e., `#pragma omp parallel for schedule(dynamic)`).

`GOMP_parallel_loop_dynamic` (Line 60) is very similar to `GOMP_parallel`, but it also initializes the loop bounds and chunk size with `gomp_loop_init` (Line 66).

Each thread calls `GOMP_loop_dynamic_start` (Line 1) once and `GOMP_loop_dynamic_next` (Line 36) continuously afterwards until the loop is finished. The former checks if another thread already initialized the loop bounds and chunk size, and initializes them if that is not the case. Then, it atomically increments the start of the next chunk for the next thread to use and sets the bounds of the current chunk for the current thread. This is done again with every call to `GOMP_loop_dynamic_next`.

Listing 3.6: GOMP Dynamic For Loop Implementation

```
1  int GOMP_loop_dynamic_start(int start, int end, int incr,
2                              int chunk_size, int *istart,
3                              int *iend) {
4    int chunk, left;
5    int ret = 1;
6
7    if (gomp_work_share_start()) { // work returns locked
8      gomp_loop_init(start, end, incr, chunk_size);
9    }
10   gomp_hal_unlock(&works.lock);
11
12   chunk = chunk_size * incr;
13
14   start = __atomic_fetch_add(&works.next, chunk, __ATOMIC_SEQ_CST);
15
16   if (start >= works.end) {
17     ret = 0;
18   }
19
20   if (ret) {
21     left = works.end - start;
22
23     if (chunk > left) {
24       end = works.end;
25     } else {
26       end = start + chunk;
27     }
28   }
29
30   *istart = start;
31   *iend = end;
32
33   return ret;
34 }
35
36 int GOMP_loop_dynamic_next(int *istart, int *iend) {
37   int start, end, chunk, left;
38
39   chunk = works.chunk_size * works.incr;
40   start = __atomic_fetch_add(&works.next, chunk, __ATOMIC_SEQ_CST);
41
42   if (start >= works.end) {
```

```
43      return 0;
44    }
45
46    left = works.end - start;
47
48    if (chunk > left) {
49      end = works.end;
50    } else {
51      end = start + chunk;
52    }
53
54    *istart = start;
55    *iend = end;
56
57    return 1;
58  }
59
60  void GOMP_parallel_loop_dynamic(void (*fn)(void *), void *data,
61                                  unsigned num_threads, long start,
62                                  long end, long incr, long chunk_size) {
63    uint32_t core_id = mempool_get_core_id();
64
65    gomp_new_work_share();
66    gomp_loop_init(start, end, incr, chunk_size);
67
68    GOMP_parallel_start(fn, data, num_threads);
69    run_task(core_id);
70    GOMP_parallel_end();
71  }
72
73  void GOMP_loop_end() {
74    uint32_t core_id = mempool_get_core_id();
75    mempool_barrier_gomp(core_id, event.nthreads);
76  }
```

3.1.4. Barriers

To implement barriers, the runtime makes calls to `mempool_barrier` as implemented in MemPool's runtime library. It makes use of MemPool's interrupt capabilities to wake up all cores when the last core reaches the barrier. Note that cores that are not participating in the parallel execution will also be woken up, but they will immediately go back to sleep because of the check on Line 10 in Listing 3.2.

Listing 3.7: mempool_barrier Implementation

```
1  void mempool_barrier(uint32_t num_cores) {
2    // Increment the barrier counter
3    if ((num_cores - 1) == __atomic_fetch_add(&barrier, 1,
4                                              __ATOMIC_RELAXED)) {
5      __atomic_store_n(&barrier, 0, __ATOMIC_RELAXED);
6      __sync_synchronize(); // Full memory barrier
7      wake_up_all();
8    }
9    // Some threads have not reached the barrier --> Let's wait
10   // Clear the wake-up trigger for the last core reaching the barrier
11   // as well
12   mempool_wfi();
13 }
```

3.2. HERO LLVM Implementation

In her work [5], Qiu also implements an OpenMP runtime library compatible with LLVM, however for the HERO [6] platform. The main difference between the GCC and LLVM implementations is that they use a different Application Programming Interface (API) (i.e., the compiler directives get translated into different function calls). The LLVM team provides a reference document [7] describing the internal runtime interface.

Chapter 4

Implementation

4.1. General Architecture

The OpenMP runtime library presented in this report is implemented in C++, as opposed to the previous version, which was implemented in C. The main reason for this is that C++ provides a higher level of abstraction compared to C, allowing us to use features such as *classes*, *templates*, and some parts of the standard library that don't require OS support, such as *lock guards*. If used properly, C++ should have no meaningful performance overhead compared to C, but we will look into this in more detail in chapter 5. Another reason is that, until now, C++ was only used in MemPool in the context of Halide [8], so we use this as an opportunity to explore how to enable C++ support in MemPool.

The runtime is roughly structured into the following three main components:

- KMP entrypoints.
- C++ classes for each relevant concept (e.g., threads, teams, barriers, etc.).
- Supporting code.

Each of them will be described in more detail in the following sections.

4.2. KMP Entrypoints

The KMP entrypoints are the functions that are called at runtime during the execution of an OpenMP program. During the compilation process, the compiler converts OpenMP directives into calls to these functions as defined by the internal API between the LLVM compiler and the runtime library. An LLVM OpenMP runtime reference document is

available at [7], however it is quite out of date and not very helpful in documenting the expected behavior of the runtime library. For this reason, using the source code of the default runtime library[1], as well as smaller third-party implementations[2] can be quite helpful.

In order to maintain some separation of concerns, the functionality associated with each entrypoint is not directly implemented in the function, but rather in a one of the different classes described in the next chapter, which contains the relevant methods associated with the entrypoint. Those are then called by the entrypoint function.

Each implemented entrypoint is documented in Appendix A.1.

4.3. C++ Classes

The runtime library is structured around a set of C++ classes, each representing a different high level concept, which allows us to compartmentalize the implementation.

The main classes are described in the following sections.

4.3.1. Task

The `Task` class is essentially a wrapper around a KMP `microtask`, which represents a section of code to be run by a thread (e.g., the body of a `parallel` section) and is passed in as an argument to, e.g., `__kmpc_fork_call` (Appendix A.1.1). Additionally, this class also stores the arguments that must be passed to the `microtask`. These are pointers used by the compiler to access data outside of the `microtask`'s scope.

Documentation for this class can be found in Appendix A.2.1.

4.3.2. Barrier

This class is used to implement a thread barrier. It provides two implementations: one of them is more performant but only works when there is only a single team running, whereas the other one works with multiple teams.

Documentation for this class can be found in Appendix A.2.2.

[1] https://github.com/llvm/llvm-project/tree/main/openmp
[2] https://github.com/parallel-runtimes/lomp

4.3.3. Thread

This class represents a single OpenMP thread running on a core and contains attributes such as the thread ID and the team it belongs to, as well as methods such as the main event loop equivalent to Listing 3.1 and the logic for executing a fork.

Documentation for this class can be found in Appendix A.2.3.

4.3.4. Team

This class represents a team of threads. It contains attributes such as the team ID and the number of threads participating in it, as well as the task to be executed by all threads in the team, and the barrier associated with it. Additionally, it contains methods for assigning work to the threads and waking them up, as well as for scheduling static and dynamic work shares.

Documentation for this class can be found in Appendix A.2.4.

4.4. Supporting Code

The supporting code consists of various helper functions and global variables in the `runtime` namespace (Appendix A.3.1) used throughout the runtime, as well as as everything surrounding C++ support, and the main function wrapper. In the following sections, we will describe the latter two.

4.4.1. C++ Support

Heap Allocation

In order to support heap allocation in C++, we implement the global `operator new` and `operator delete` functions since they are used by the default allocator[3]. They are essentially wrappers around the `simple_malloc` and `simple_free` functions provided by the MemPool runtime library. Additionally, they use a globally defined mutex `allocLock` to make sure that only one thread uses the allocator at any given time, since neither `simple_malloc` nor `simple_free` are thread-safe.

[3] https://en.cppreference.com/w/cpp/memory/allocator

Listing 4.1: Heap Allocation Functions

```
1  kmp::Mutex allocLock __attribute__((section(".l1")));
2
3  void *operator new(size_t size) {
4    std::lock_guard<kmp::Mutex> lock(allocLock);
5    void *ptr = simple_malloc(size);
6    return ptr;
7  }
8
9  void operator delete(void *ptr) noexcept {
10   std::lock_guard<kmp::Mutex> lock(allocLock);
11   return simple_free(ptr);
12 }
13
14 void *operator new[](size_t size) {
15   return operator new(size);
16 }
17
18 void operator delete[](void *ptr) noexcept {
19   return operator delete(ptr);
20 }
```

Compiler/Linker Flags

The C++ code is compiled with the following compiler flags:

- `-std=c++17` to enable C++17 features.
- `-fno-exceptions` to disable exception handling.
- `-fno-threadsafe-statics` to disable thread-safe initialization of static variables, since they are not used.

And linked with the following linker flags:

- `-nostdlib` to not link against the standard C library because it does not support the architecture.

Global Variables

Global C++ objects need to be initialized at runtime. Usually, this would be done by the startup files that the program is liked to by default, however, they are disabled when using the `-nostdlib` flag. This is why we need to implement this initialization

ourselves. The compiler generates an array of initialization functions [9] that are placed in the `.init_array` section of the Executable and Linkable Format (ELF) file, each of which must be called before the main function. This is done by the `initGlobals` function (Listing 4.2), in conjunction with the additions to the linker script shown in Listing 4.3.

Listing 4.2: initGlobals

```
1  typedef void (*init_func)(void);
2  extern init_func __init_array_start[];
3  extern init_func __init_array_end[];
4
5  static inline void initGlobals() {
6    int32_t len = __init_array_end - __init_array_start;
7    for (int32_t i = 0; i < len; i++) {__
8
9      init_array_start[i]();
10   }
11 }
```

Listing 4.3: Init Array Linker Script

```
1 /* Init array on L2 */
2 .init_array : {
3     HIDDEN (__init_array_start = .);
4     KEEP (*(SORT_BY_INIT_PRIORITY(.init_array.*)))
5     KEEP (*(.init_array))
6     HIDDEN (__init_array_end = .);
7 } > l2
```

4.4.2. Main Function Wrapper

In order to not have to write the code from Listing 3.1 manually in every OpenMP program, we implement a wrapper around the `main` function by compiling them with the `-wrap main` linker flag, which renames the `main` function to `__real_main` and `__wrap_main` to `main`. This means that the branch depending on the core ID is done transparently to the user. Additionally, we can also perform initialization tasks before calling the real `main` function, such as initializing the MemPool heap allocators and initializing global variables.

Listing 4.4: Main Function Wrapper

```
1 extern "C" int __real_main();
2
3 bool initLock = true;
4
```

```
5  extern "C" int __wrap_main() {
6    const mempool_id_t core_id = mempool_get_core_id();
7    if (core_id == 0) {
8      // Init heap allocators
9      mempool_init(0);
10
11     // Call C++ global constructors
12     initGlobals();
13
14     initLock = false;
15
16     // Run the program__
17     real_main();
18
19     printf("Program done\n");
20   } else {
21     while (initLock) {
22       // Wait for initialization to finish
23     }
24
25     kmp::runtime::runThread(static_cast<kmp_int32>(core_id));
26   }
27
28   return 0;
29 }
```

Chapter 5

Results

5.1. Evaluation setup

For our evaluation, we simulate a MemPool System on Chip (SoC) in its *MinPool* (16-core) configuration using the Verilator [10] simulator. Additionally, we increase the stack size for each core to 1024 bytes to accommodate the higher stack usage of the new runtime, and disable the `Xpulp` instructions, since they are not fully supported when compiling with LLVM.

In order to compare the performance of the new runtime against the previous one, we run a part of the existing benchmarks available in the MemPool repository[1] using both of them.

For correctness, we run the set of tests also provided in the MemPool repository, however, we adapt them to use the new testing framework described in the following section. Additionally, we add new tests for the `teams` and `sections` constructs, which we describe in more detail in Section 5.2.

5.1.1. Testing Framework

The testing framework consists of a small set of C preprocessor macros that allow us to make assertions, define test cases and run them. It is inspired by Jera's minimal testing framework [11].

It is comprised of the following macros:

[1] https://github.com/pulp-platform/mempool

- `ASSERT_TRUE(condition, error_message)`: Checks that `condition` evaluates to `true`. If it does not, it prints `error_message` and fails the test.
- `ASSERT_EQ(left, right)`: Checks that `left` is equal to `right`. If it is not, it prints an error message and fails the test.
- `ASSERT_NEQ(left, right)`: Checks that `left` is not equal to `right`. If it is, it prints an error message and fails the test.
- `EXPECT_TRUE(condition, error_message)`: Similar to `ASSERT_TRUE`, but does not fail the test if the condition is not met, it only prints the error message.
- `TEST(testname)`: Defines a new test with the name `testname` and adds it to the list of tests to be run.
- `RUN_TEST(testname)`: Runs the test `testname`.
- `RUN_ALL_TESTS()`: Runs all tests defined in the test suite.
- `PRINT_TEST_RESULTS()`: Prints the results of the test suite (i.e., how many tests were run and how many failed).

When printing, the framework includes easy to parse markers to the output such as, `[FAIL]` or `[SUCCESS]`, which can be used by, e.g., a python script to run multiple tests in succession and automatically show the results to the user.

5.2. Correctness

Our implementation passes all of the correctness tests provided in the MemPool repository, as well as the new tests described in the following sections.

5.2.1. `teams` Construct

The test shown in Listing 5.1 checks that the `teams distribute` construct correctly distributes the iterations of the loop across the teams. It does so by checking that the team number is correct for each iteration. It also checks that the number of teams matches the number of requested teams by comparing the value returned by `omp_get_num_teams()` to the expected value (4).

Listing 5.1: test_teams_distribute

```
1 TEST(test_teams_distribute) {
2   for (int rep = 0; rep < REPETITIONS; rep++) {
3     int num_teams = 0;
4     int team_num[12];
5
```

```
6  #pragma omp teams distribute num_teams(4)
7      for (int i = 0; i < 12; i++) {
8        team_num[i] = omp_get_team_num();
9
10       if (omp_get_team_num() == 0) {
11         num_teams = omp_get_num_teams();
12       }
13     }
14
15     for (int i = 0; i < 12; i++) {
16       ASSERT_EQ(team_num[i], i / 3);
17     }
18
19     ASSERT_EQ(num_teams, 4);
20   }
21 }
```

The test shown in Listing 5.2 checks that each team is able to work independently by performing a reduction onto a team-local variable. The result of each reduction is then compared against the expected value (32).

Listing 5.2: tests_teams_reduce

```
1  TEST(test_teams_reduce) {
2    for (int rep = 0; rep < REPETITIONS; rep++) {
3      int sum[4];
4
5  #pragma omp teams num_teams(4)
6      {
7        int local_sum = 0;
8
9  #pragma omp parallel for reduction(+ : local_sum)
10       for (int i = 0; i < 32; i++) {
11         local_sum += 1;
12       }
13
14       sum[omp_get_team_num()] = local_sum;
15     }
16
17     for (int i = 0; i < 4; i++) {
18       ASSERT_EQ(sum[i], 32);
19     }
20   }
21 }
```

Finally, the test shown in Listing 5.3 checks that barriers work correctly within teams and don't affect other teams. It does so by performing an independent barrier test in each team, that checks that the barrier is functional.

Listing 5.3: test_teams_barrier

```
1  TEST(test_teams_barrier) {
2    for (int rep = 0; rep < REPETITIONS; rep++) {
3      int results[2];
4  #pragma omp teams num_teams(2)
5      {
6        uint32_t team_num = omp_get_team_num();
7        int result = 0;
8
9  #pragma omp parallel
10       {
11         uint32_t rank = omp_get_thread_num();
12         if (rank == 1) {
13           mempool_wait(1000);
14           result = 3;
15         }
16 #pragma omp barrier
17
18         if (rank == 2) {
19           results[team_num] = result;
20         }
21       }
22     }
23
24     ASSERT_EQ(results[0], 3);
25     ASSERT_EQ(results[1], 3);
26   }
27 }
```

5.2.2. `sections` Construct

The test shown in Listing 5.4 checks that the `sections` construct correctly distributes the sections across the threads. It does so by checking that each section is executed by a different thread.

Listing 5.4: test_omp_parallel_sections

```
1  TEST(test_omp_parallel_sections) {
2    for (int rep = 0; rep < REPETITIONS; rep++) {
```

```
3     uint32_t section_1 = 0;
4     uint32_t section_2 = 0;
5
6  #pragma omp parallel sections
7     {
8
9  #pragma omp section
10       { section_1 = omp_get_thread_num(); }
11
12 #pragma omp section
13       { section_2 = omp_get_thread_num(); }
14    }
15
16    ASSERT_NEQ(section_1, section_2);
17  }
18 }
```

5.3. Performance

To account for performance differences that arise purely from using different compilers, we compare the performance of the new runtime against the previous one in two different ways:

First, we obtain the speedup of running a given workload in parallel versus sequentially with the same compiler, and compare this across runtimes. This way, the effect of a more performant compilation output will be mitigated, since the speedup is calculated relative to the sequential baseline. The results of this approach will be discussed in Section 5.3.1.

Then, we look at the cost of specific runtime operations (e.g., barriers, startup time, etc.) and compare them across runtimes as well. This way we can isolate individual components and see if any of them could benefit from further optimizations. The results of this approach will be discussed in Section 5.3.2.

5.3.1. Speedup

Sparse Matrix-Vector Multiplication

In this benchmark, we perform Sparse Matrix-Vector Multiplication (SPVM), where the matrix is stored in Compresed Sparse Row (CSR) format. It has 512 columns and we vary the number of rows from 1 to 512.

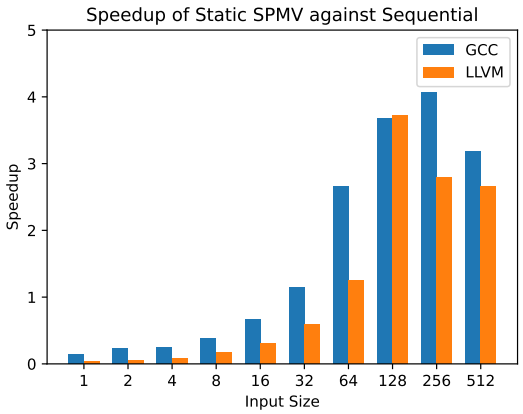

Figure 5.1.: Static SPVM Speedup.

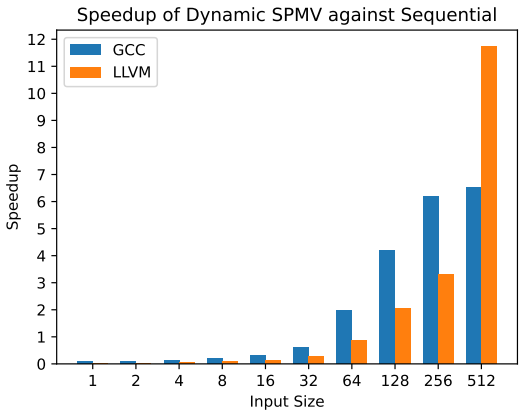

Figure 5.2.: Dynamic SPVM Speedup.

Matrix-Matrix Multiplication

In this benchmark, we perform Matrix-Matrix Multiplication (MMM) of square matrices, where we vary the size of both matrices from 1 to 32.

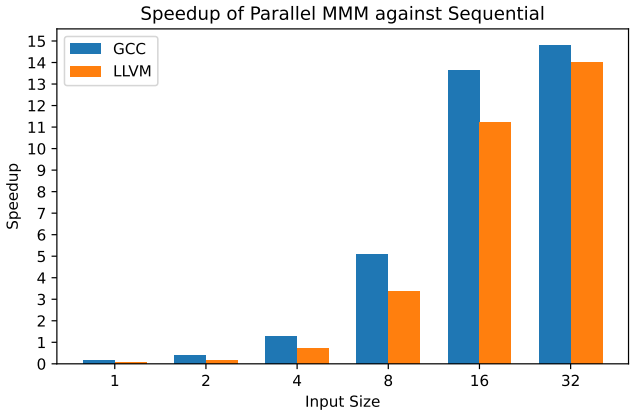

Figure 5.3.: Matrix-Matrix Multiplication Speedup.

Dot Product

In this benchmark, we perform the dot product of two vectors of size 1 to 2048.

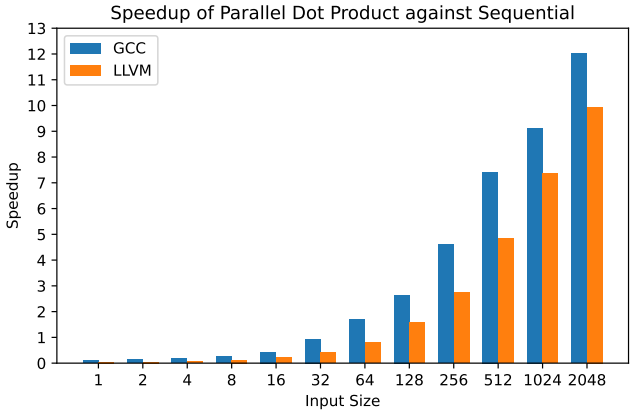

Figure 5.4.: Dot Product Speedup.

5.3.2. Runtime Overhead

Startup Time

We measure the runtime startup time using the benchmark shown in Listing 5.5. A timer is started before executing the parallel region and stopped by the master thread inside a master construct. We do this because the master thread is the one that performs the fork and we can therefore know that when the master thread stops the timer, all of the other threads have at least been initialized. The duration is averaged across 100 repetitions.

Listing 5.5: Startup Time Benchmark

```
1  void startup_time() {
2
3    uint32_t duration = 0;
4
5    for (int i = 0; i < REPETITIONS; i++) {
6
7      mempool_timer_t cycles = mempool_get_timer();
8      mempool_start_benchmark();
9
10     #pragma omp parallel
11     {
12
13       #pragma omp master
14       {
15         mempool_stop_benchmark();
16         cycles = mempool_get_timer() - cycles;
17         duration += cycles;
```

```
18        }
19      }
20    }
21
22    printf("Startup time duration: %d\n", duration / REPETITIONS);
23  }
```

The results of the benchmarks are shown in Table 5.1, where we can see that the new runtime has a startup time that is more than twice as long as the previous one.

Runtime	Startup Time (cycles)	Speedup (vs. GCC)
GCC	154	1
LLVM	355	0.43

Table 5.1.: Startup Time Benchmark Results.

Barrier

In this benchmark, we measure the number of cycles it takes to execute an increasing number of barriers, as shown in Listing 5.6.

Listing 5.6: Barrier Benchmark

```
1  int main() {
2  #pragma omp parallel
3    {
4      unsigned int counter = 0;
5      unsigned int cycles = 0;
6      unsigned int start_time = 0;
7
8      for (int i = 1; i < MAX_BARRIERS + 1; i++) {
9
10 #pragma omp barrier
11
12       start_time = mempool_get_timer();
13       mempool_start_benchmark();
14
15       for (int j = 0; j < i; j++) {
16 #pragma omp barrier
17         counter++;
18       }
19
20       mempool_stop_benchmark();
```

```
21        cycles = mempool_get_timer() - start_time;
22
23  #pragma omp single
24        printf("%d barriers: %d cycles\n", i, cycles);
25      }
26    }
27
28    return 0;
29  }
```

As expected, the number of cycles increases linearly with the number of barriers, as shown in Fig. 5.5. The execution time of a barrier using the new runtime is about 18% faster than with old one, as shown in Fig. 5.6.

Explain why for one barrier it is faster

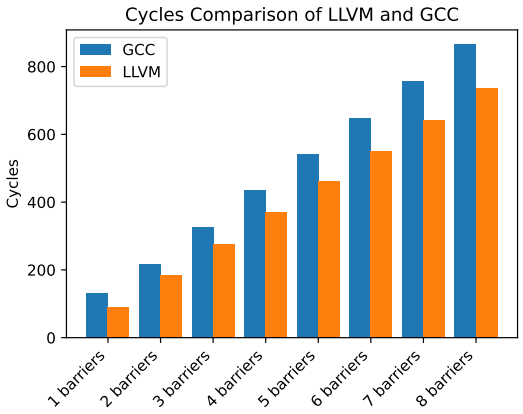

Figure 5.5.: Barrier Benchmark Cycles.

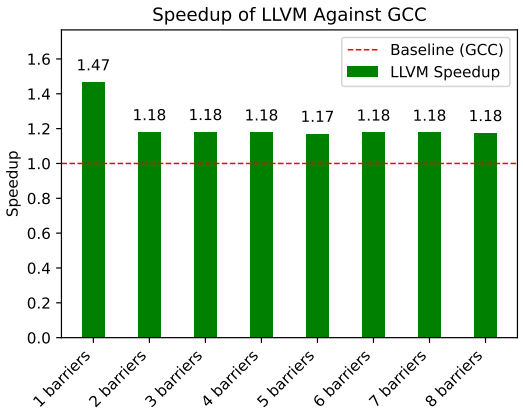

Figure 5.6.: Barrier Benchmark Speedup.

Critical Section

In this benchmark, we measure the number of cycles it takes to execute a `critical` section, as shown in Listing 5.7. The timer is started before the critical section and stopped right after it is done executing. The master thread then aggregates the duration of each run to finally print the average over 100 repetitions.

Listing 5.7: Critical Benchmark

```
1  void omp_parallel_critical() {
2    uint32_t num_cores = mempool_get_core_count();
3    uint32_t duration = 0;
4
5    for (int i = 0; i < 100; i++) {
```

```
 6
 7  #pragma omp parallel num_threads(num_cores)
 8      {
 9        mempool_timer_t cycles = mempool_get_timer();
10        mempool_start_benchmark();
11
12  #pragma omp critical
13        { result += 100; }
14
15        mempool_stop_benchmark();
16        cycles = mempool_get_timer() - cycles;
17
18  #pragma omp master
19        { duration += cycles; }
20      }
21    }
22
23    printf("OMP Critical Duration: %d\n", duration / 100);
24  }
```

As shown in Table 5.2, the new runtime achieves comparable performance in this benchmark, needing less than 10 extra cycles to execute a critical section.

Runtime	Critical Section (cycles)	Speedup (vs. GCC)
GCC	74	1
LLVM	82	0.9

Table 5.2.: Critical Section Benchmark Results.

Single Construct

This benchmark follows the same structure as the previous one, except that the `critical` section is replaced by a `single` construct.

Listing 5.8: Single Benchmark

```
1  void omp_parallel_single() {
2    uint32_t num_cores = mempool_get_core_count();
3    uint32_t duration = 0;
4
5    for (int i = 0; i < 100; i++) {
6
7  #pragma omp parallel num_threads(num_cores)
```

```
 8      {
 9        mempool_timer_t cycles = mempool_get_timer();
10        mempool_start_benchmark();
11
12  #pragma omp single
13        { result += 100; }
14
15        mempool_stop_benchmark();
16        cycles = mempool_get_timer() - cycles;
17
18  #pragma omp master
19        { duration += cycles; }
20      }
21    }
22
23    printf("OMP Single Duration: %d\n", duration / 100);
24  }
```

As shown in Table 5.3, the new runtime achieves a speedup of around 5.73 in this benchmark compared to the previous one. This is because the `single` construct is implemented in a completely different way: The GOMP runtime uses a global `work_t` struct (Listing 3.5) to signal to every thread whether the `single` construct has already been executed. This requires locking, since all threads are trying to access it concurrently. Contrary to that, the new runtime implements `single` constructs the same way as `master` constructs, which just requires checking the thread's ID.

Runtime	Single Construct (cycles)	Speedup (vs. GCC)
GCC	837	1
LLVM	146	5.73

Table 5.3.: Single Construct Benchmark Results.

Master Construct

This benchmark follows the same structure as the previous one, except that the `single` construct is replaced by a `master` construct.

Listing 5.9: Master Benchmark

```
1  void omp_parallel_master() {
2    uint32_t num_cores = mempool_get_core_count();
3    uint32_t duration = 0;
4
```

```
5     for (int i = 0; i < 100; i++) {
6
7   #pragma omp parallel num_threads(num_cores)
8       {
9         mempool_timer_t cycles = mempool_get_timer();
10        mempool_start_benchmark();
11
12  #pragma omp master
13        { result += 100; }
14
15        mempool_stop_benchmark();
16        cycles = mempool_get_timer() - cycles;
17
18  #pragma omp master
19        { duration += cycles; }
20      }
21    }
22
23    printf("OMP Master Duration: %d\n", duration / 100);
24  }
```

A shown in Table 5.4, the new runtime achieves only about half of the performance of the previous runtime.

Runtime	Master Construct (cycles)	Speedup (vs. GCC)
GCC	38	1
LLVM	69	0.55

Table 5.4.: Master Construct Benchmark Results.

Try to explain difference between single and master for LLVM since they are implemented the same way

Chapter 6

Conclusion and Future Work

- Interesting missing features (like nested parallelism and explicit tasks, as well as more clever ways to create teams) and how performance may be improved further.
- This can also be split into two chapters.

Draw your conclusions from the results and summarize your contributions. Point out aspects that need to be investigated further.

The conclusion can be structured inversely to the introduction: Summarize how the *evaluation* backs the *solution*, which solves the *problem*. Describe how your contributions improve the *situation* and what other (potentially newly discovered) problems have to be solved in the future.

Be concise: the conclusion normally fits on a single page and is rarely longer than two pages.

Appendix A

Implementation Details

A.1. KMP Entrypoints

A.1.1. Parallel Construct

`__kmpc_fork_call`

Description This function is called by the master thread of the current team (the default team contains all cores and the core with ID 0 runs the master thread). It is responsible for assigning the appropriate number of threads to the team and waking them up, as well as setting up the task that they will run.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `argc`: Number of arguments passed to the microtask function.
- `microtask`: Function pointer to the task to be run by each thread in the team.
- `...`: Variable number of arguments to be passed to the microtask function.

Implementation First, it creates a `kmp::Task` (Appendix A.2.1) object with the microtask, arguments casted to a void pointer array, and the number of such arguments. Then, it obtains the object representing the current thread using `kmp::runtime::getCurrentThread()` (Appendix A.3.1) and calls the `forkCall` method on it with the task as an argument.

Listing A.1: __kmpc_fork_call

```
1 void __kmpc_fork_call(ident_t *loc, kmp_int32 argc,
2                        kmpc_micro microtask, ...) {
3   va_list args;
4   va_start(args, microtask);
5   kmp::Task kmpMicrotask(microtask, reinterpret_cast<void **>(args),
6                          argc);
7   kmp::runtime::getCurrentThread().forkCall(kmpMicrotask);
8   va_end(args);
9 };
```

`__kmpc_push_num_threads`

Description This function is called when using the `num_threads` clause in a parallel construct. It is used to request a specific number of threads to be used in the parallel section.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `global_tid`: Global ID of the calling thread.
- `num_threads`: Requested number of threads.

Implementation This function essentially just calls the `requestNumThreads` method on the object representing the current thread (Appendix A.2.3), which sets the `requestedNumThreads` field, obtained by calling `kmp::runtime::getThread` (Appendix A.3.1) with `global_tid`.

Listing A.2: __kmpc_push_num_threads

```
1 void __kmpc_push_num_threads(ident_t *loc, kmp_int32 global_tid,
2                               kmp_int32 num_threads) {
3   kmp::runtime::getThread(global_tid).requestNumThreads(num_threads);
4 };
```

A.1.2. Work Sharing Constructs

The following entrypoints are used when running work sharing constructs such as static `for` loops or `sections`. Contrary to GOMP, LLVM uses the same entrypoints for both of them.

`__kmpc_for_static_init_4`

Description This function is called by every thread at the beginning of a static work sharing construct. It is responsible for setting the values of `plastiter`, `plower`, `pupper`, and `pstride` in order to assign a range of iterations to each thread. Because this assignment is static, the function only needs to be called once per thread. There is a variant of this function for handling the case where the loop iteration variable is unsigned. The semantics and implementation are the same except for the type of `plower`, `pupper` and `plastiter`, which becomes unsigned.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `gtid`: Global ID of the calling thread.
- `schedtype`: Type of scheduling to be used. [1]
- `plastiter`: Pointer to the *last iteration* flag. This is supposed to be set to 1 if the calling thread is the one to execute the last iteration of the loop.
- `plower`: Pointer to the lower bound of the iteration range for the current thread. This is initially set to the global lower bound for the loop.
- `pupper`: Pointer to the upper bound of the iteration range for the current thread. This is initially set to the global lower bound for the loop.
- `pstride`: Pointer to the stride of the iteration range. This is the distance between two work chunks assigned to the same thread.
- `incr`: Increment amount of the loop iteration variable.
- `chunk`: Chunk size.

[1] https://github.com/llvm/llvm-project/blob/f28c006a5895fc0e329fe15fead81e37457cb1d1/openmp/runtime/src/kmp.h#L357

Implementation This function just calls the `forStaticInit` method (Appendix A.2.4) on the object representing the current team.

Listing A.3: __kmpc_for_static_init_4

```
1 void __kmpc_for_static_init_4(ident_t *loc, kmp_int32 gtid,
2                               kmp_int32 schedtype,
3                               kmp_int32 *plastiter, kmp_int32 *plower,
4                               kmp_int32 *pupper, kmp_int32 *pstride,
5                               kmp_int32 incr, kmp_int32 chunk) {
6   kmp::runtime::getThread(gtid).getCurrentTeam()->forStaticInit(
7       loc, gtid, static_cast<kmp_sched_type>(schedtype), plastiter,
8       plower, pupper, pstride, incr, chunk);
9 };
```

`__kmpc_for_static_fini`

Description This function is called when the static work sharing construct is finished.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `global_tid`: Global ID of the calling thread.

Implementation This function is not required for the correct implementation of the our runtime so it does nothing.[2]

Listing A.4: __kmpc_for_static_fini

```
1 void __kmpc_for_static_fini(ident_t *loc, kmp_int32 global_tid){};
```

A.1.3. Dynamic Loops

`__kmpc_dispatch_init_4`

Description This function is called once by each thread before running a dynamic `for` loop. It is responsible for storing the loop iteration variables and scheduling type so that they can be used by future calls to `__kmpc_dispatch_next_4` (Appendix A.1.3). Just

[2] Because the code generated from the OpenMP directives still calls this function, it is necessary to implement it even if it is left empty.

like for the static loop, there is also a variant of this function that handles unsigned loop iteration variables.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `gtid`: Global ID of the calling thread.
- `schedtype`: Type of scheduling to be used. [3]
- `lower`: Lower bound of the iteration range for the current thread.
- `upper`: Upper bound of the iteration range for the current thread.
- `incr`: Increment amount of the loop iteration variable.
- `chunk`: Chunk size.

Implementation This function just calls the `dispatchInit` method (Appendix A.2.4) on the object representing the current team after removing any scheduling modifiers from the schedule variable using the `SCHEDULE_WITHOUT_MODIFIERS` macro since they are not supported by our runtime.

Listing A.5: __kmpc_dispatch_init_4

```
1 void __kmpc_dispatch_init_4(ident_t *loc, kmp_int32 gtid,
2                              kmp_int32 schedtype, kmp_int32 lower,
3                              kmp_int32 upper, kmp_int32 incr,
4                              kmp_int32 chunk) {
5   kmp::runtime::getThread(gtid).getCurrentTeam()->dispatchInit(
6     loc, gtid,
7     static_cast<kmp_sched_type>(SCHEDULE_WITHOUT_MODIFIERS(schedtype)),
8     lower, upper, incr, chunk);
9 }
```

[3] https://github.com/llvm/llvm-project/blob/f28c006a5895fc0e329fe15fead81e37457cb1d1/openmp/runtime/src/kmp.h#L357

`__kmpc_dispatch_next_4`

Description This function is called by each thread after calling `dispatchInit` and after completing the previously assigned chunk of iterations. It is responsible for setting the iteration variables for the current chunk of the loop that the calling thread should execute, as well as updating the loop information shared across the team so that future calling threads can know what work is left to do. There is also a variant of this function that handles unsigned loop iteration variables.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `gtid`: Global ID of the calling thread.
- `plastiter`: Pointer to the *last iteration* flag. This is supposed to be set to 1 if the calling thread is the one to execute the last iteration of the loop.
- `plower`: Pointer to the lower bound of the iteration range for the current thread.
- `pupper`: Pointer to the upper bound of the iteration range for the current thread.
- `pstride`: Pointer to the stride of the iteration range.

Return Value 1 if there is still work to do for the calling thread, 0 otherwise.

Implementation This function just calls the `dispatchNext` method (Appendix A.2.4) on the object representing the current team.

Listing A.6: __kmpc_dispatch_next_4

```
1 int __kmpc_dispatch_next_4(ident_t *loc, kmp_int32 gtid,
2                            kmp_int32 *plastiter, kmp_int32 *plower,
3                            kmp_int32 *pupper, kmp_int32 *pstride) {
4   return static_cast<int>(
5       kmp::runtime::getThread(gtid).getCurrentTeam()->dispatchNext(
6           loc, gtid, plastiter, plower, pupper, pstride));
7 }
```

A.1.4. Critical sections

`__kmpc_critical and __kmpc_end_critical`

Description These functions are used to implement `critical` sections. `__kmpc_critical` is called before entering a `critical` section and `__kmpc_end_critical` is called before leaving it.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `gtid`: Global ID of the calling thread.
- `crit`: Pointer to a region of memory associated with the `critical` section. 8×32 bits are made available by the compiler at the location pointed to by this argument.

Implementation Both of these functions reinterpret the memory location pointed to by `crit` as a `kmp::Mutex` object and use it to lock and unlock access to the `critical` section. We use a static assertion to make sure that the size of the `kmp::Mutex` object is smaller than the size of the memory region.

Listing A.7: __kmpc_critical and __kmpc_end_critical

```
1  typedef kmp_int32 kmp_critical_name[8];
2
3  void __kmpc_critical(ident_t *loc, kmp_int32 gtid,
4                       kmp_critical_name *crit) {
5    static_assert(sizeof(kmp::Mutex) <= sizeof(kmp_critical_name));
6
7    kmp::Mutex *mutex = reinterpret_cast<kmp::Mutex *>(*crit);
8    mutex->lock();
9  };
10
11 void __kmpc_end_critical(ident_t *loc, kmp_int32 gtid,
12                          kmp_critical_name *crit) {
13   Mutex *mutex = reinterpret_cast<kmp::Mutex *>(*crit);
14   mutex->unlock();
15 };
```

A.1.5. Master and Single Constructs

`__kmpc_master and __kmpc_single`

Description These functions are used to implement the `master` and `single` constructs respectively. They are called by every thread before entering a `master` or `single` section. `__kmpc_master` needs to make sure that only the master thread of the team executes the section, while `__kmpc_single` needs to make sure that only a single thread executes it, however, it does not need to be the master thread.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `gtid`: Global ID of the calling thread.

Return Value 1 if the calling thread should execute the section, 0 otherwise.

Implementation Both `__kmpc_master` and `__kmpc_single` are implemented identically for simplicity, since only letting the master thread execute `single` sections will always be correct, as there is only a single master thread per team. It essentially obtains the thread ID of the calling thread by calling `getTid()` on the thread object and compares it against 0, returning 1 if they are equal and 0 otherwise.

Listing A.8: __kmpc_master and __kmpc_single

```
1  kmp_int32 __kmpc_master(ident_t *loc, kmp_int32 gtid) {
2    return static_cast<kmp_int32>(
3      kmp::runtime::getThread(gtid).getTid() == 0
4    );
5  };
6
7  kmp_int32 __kmpc_single(ident_t *loc, kmp_int32 gtid) {
8    return static_cast<kmp_int32>(
9      kmp::runtime::getThread(gtid).getTid() == 0
10   );
11 };
```

`__kmpc_end_master and __kmpc_end_single`

Description These functions are called after the execution of a `master` or `single` section respectively by the thread that executed it.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `gtid`: Global ID of the calling thread.

Implementation These functions are not required for the correct implementation of our runtime so they do nothing.[4]

Listing A.9: __kmpc_end_master and __kmpc_end_single

```
1 void __kmpc_end_master(ident_t *loc, kmp_int32 gtid){};
2
3 void __kmpc_end_single(ident_t *loc, kmp_int32 gtid){};
```

A.1.6. Copyprivate Clause

`__kmpc_copyprivate`

Description This function is used to implement the `copyprivate` clause associated with a `single` section. It is called by every thread that participates in the `parallel` section after the `single` section is executed and it is responsible for broadcasting the private data of the thread that executed it to the other threads.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `gtid`: Global ID of the calling thread.
- `cpy_size`: Size of the private data to be copied.
- `cpy_data`: Pointer to the private data to be copied.
- `cpy_func`: Pointer to a function that copies the private data.

[4] Because the code generated from the OpenMP directives still calls these functions, it is necessary to implement them even if they are left empty.

- `didit`: 1 if the calling thread executed the single section, 0 otherwise.

Implementation This function just calls the `copyPrivate` method (Appendix A.2.4) on the object representing the current team that the calling thread belongs to.

Listing A.10: __kmpc_copyprivate

```
1 void __kmpc_copyprivate(ident_t *loc, kmp_int32 gtid, size_t cpy_size,
2                         void *cpy_data,
3                         void (*cpy_func)(void *, void *),
4                         kmp_int32 didit) {
5   kmp::runtime::getThread(gtid).getCurrentTeam()->copyPrivate(
6       loc, gtid, cpy_size, cpy_data, cpy_func, didit);
7 };
```

A.1.7. Reduction Clause

`__kmpc_reduce and __kmpc_reduce_nowait`

Description These functions are used to implement the reduction clause. The `nowait` version is called when the `nowait` clause is present in the reduction directive and indicates that no barrier should be executed after the reduction.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `global_tid`: Global ID of the calling thread.
- `num_vars`: Number of variables to reduce.
- `reduce_size`: Size of the data to be reduced (in bytes).
- `reduce_data`: Pointer to the data to be reduced.
- `reduce_func`: Pointer to the function that performs the reduction on a pair of elements. The result is made available at the location pointed to by the first argument after calling it.
- `lck`: Pointer to a region of memory associated with the `critical` section used for the reduction. 8×32 bits are made available by the compiler at the location pointed to by this argument.

Return Value 1 for the master thread, 0 for others. 2 if the reduction should be performed using build in atomic operations.

Implementation Both functions are implemented identically since they only differ in the execution of a barrier, which is differentiated when calling `__kmpc_end_reduce_nowait` and `__kmpc_end_reduce` (Appendix A.1.7). They always return 2 in order to use built-in atomic operations for the reduction.

Listing A.11: __kmpc_reduce and __kmpc_reduce_nowait

```
kmp_int32 __kmpc_reduce_nowait(ident_t *loc, kmp_int32 global_tid,
                               kmp_int32 num_vars, size_t reduce_size,
                               void *reduce_data,
                               void (* reduce_func)(void *lhs_data,
                                                    void *rhs_data),
                               kmp_critical_name *lck) {
  return 2; // Atomic reduction
}

kmp_int32 __kmpc_reduce(ident_t *loc, kmp_int32 global_tid,
                        kmp_int32 num_vars, size_t reduce_size,
                        void *reduce_data,
                        void (*reduce_func)(void *lhs_data,
                                            void *rhs_data),
                        kmp_critical_name *lck) {
  return __kmpc_reduce_nowait(loc, global_tid, num_vars, reduce_size,
                              reduce_data, reduce_func, lck);
}
```

`__kmpc_end_reduce` and `__kmpc_end_reduce_nowait`

Description These functions are called after the reduction is done. The first one performs a barrier, while the second one does not.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `global_tid`: Global ID of the calling thread.
- `lck`: Pointer to a region of memory associated with the `critical` section used for the reduction. 8×32 bits are made available by the compiler at the location pointed to by this argument.

Implementation `__kmpc_end_reduce_nowait` does nothing[5], while `__kmpc_end_reduce` performs a barrier using `__kmpc_barrier` (Appendix A.1.9).

Listing A.12: __kmpc_end_reduce and __kmpc_end_reduce_nowait

```
1 void __kmpc_end_reduce_nowait(ident_t *loc, kmp_int32 global_tid,
2                               kmp_critical_name *lck) {}
3
4 void __kmpc_end_reduce(ident_t *loc, kmp_int32 global_tid,
5                        kmp_critical_name * lck) {
6   return __kmpc_barrier(loc, global_tid);
7 }
```

A.1.8. Teams Construct

`__kmpc_fork_teams`

Description This function is used to implement the `teams` construct. It is responsible for creating a league of teams as described in Section 2.2.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `argc`: Number of arguments passed to the microtask function.
- `microtask`: Function pointer to the task to be run by each thread in the team.
- `...`: Variable number of arguments to be passed to the microtask function.

Implementation This function works very similarly to `__kmpc_fork_call` (Appendix A.1.1) except that it calls the `forkTeams` method (Appendix A.2.3) and that the microtask will only be executed by the master thread of each team.

Listing A.13: __kmpc_fork_teams

```
1 void __kmpc_fork_teams(ident_t *loc, kmp_int32 argc,
2                        kmpc_micro microtask, ...) {
3   va_list args;
4   va_start(args, microtask);
5   kmp::Task kmpMicrotask(microtask, reinterpret_cast<void **>(args),
```

[5] Because the code generated from the OpenMP directives still calls this function, it is necessary to implement it even if it is left empty.

```
6                           argc);
7    kmp::runtime::getCurrentThread().forkTeams(kmpMicrotask);
8    va_end(args);
9  }
```

`__kmpc_push_num_teams`

Description This function is used to implement both the `num_teams` and `thread_limit` clauses of the teams construct. `num_teams` requests the number of teams to create and `thread_limit` limits the number of threads in each team.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `global_tid`: Global ID of the calling thread.
- `num_teams`: Requested number of teams.
- `num_threads`: Requested maximum number of threads per team.

Implementation This function first checks if either `num_teams` or `num_threads` are greater than 0 and, if so, it sets the corresponding global variables (Appendix A.3.1) accordingly. The maximum number of teams is limited to half the number of cores.

Listing A.14: __kmpc_push_num_teams

```
1  void __kmpc_push_num_teams(ident_t *loc, kmp_int32 global_tid,
2                             kmp_int32 num_teams,
3                             kmp_int32 num_threads) {
4    if (num_teams > 0) {
5      kmp::runtime::requestedNumTeams = std::min(num_teams, NUM_CORES / \
           2);
6    }
7
8    if (num_threads > 0) {
9      kmp::runtime::requestedThreadLimit = num_threads;
10   }
11 }
```

A.1.9. Barrier

`__kmpc_barrier`

Description This function is called to execute a barrier.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.
- `global_tid`: Global ID of the calling thread.

Implementation This function obtains the barrier object associated with the current team by calling the `getBarrier` method and executes it by calling `wait`.

Listing A.15: __kmpc_barrier

```
1 void __kmpc_barrier(ident_t *loc, kmp_int32 global_tid) {
2   kmp::runtime::getThread(global_tid).getCurrentTeam()
3                                      ->getBarrier().wait();
4 };
```

A.1.10. Miscellaneous

`__kmpc_global_thread_num`

Description This function is used to obtain the global ID of the calling thread.

Arguments

- `loc`: Pointer to a struct containing information about the source code location of the call.

Return Value Global ID of the calling thread.

Implementation This function just returns the value returned by `mempool_get_core_id`, which is provided by the MemPool runtime library. It is used to obtain the current core ID, which is the same as the global thread ID since each thread runs on a single core with the same ID.

Listing A.16: __kmpc_global_thread_num

```
1 kmp_int32 __kmpc_global_thread_num(ident_t *loc) {
2   return static_cast<kmp_int32>(mempool_get_core_id());
3 };
```

A.2. C++ Classes

A.2.1. Task

Attributes

- `kmpc_micro func`: Pointer to the `microtask` function.
- `kmp_int32 argc`: Number of arguments passed to the `microtask` function.
- `void **args`: Array containing the `microtask` arguments.

Relevant Methods

- `void run(kmp_int32 gtid, kmp_int32 tid)`: Calls the `microtask` function with the arguments stored in `args` in addition to `gtid` and `tid`, which are always expected to be passed as the first and second argument respectively.

A.2.2. Barrier

Attributes

- `std::atomic<kmp_int32> barrier`: Atomic counter used to keep track of the number of threads that have reached the barrier.
- `std::atomic<kmp_int32> generation`: Atomic counter used to distinguish between generations when using the *generation barrier*. This is not used in the *WFI barrier*.
- `int32_t numThreads`: Number of threads participating in the barrier.

Relevant Methods

- `void wait()`: This function implements the actual barrier. It distinguishes between two cases: if there is currently only a single team running, then a *Wait For Interrupt (WFI) barrier* is used, otherwise a spinning *generation barrier* is used.

  The first kind is essentially the same as in Section 3.1.4, with the only difference being that the `barrier` variable is not global, but a member of the `Barrier` class. This is important for the second kind of barrier, but it can also be used in this case for simplicity.

  If there are multiple teams, then a *generation barrier* is used. This is because the *WFI barrier* does not work when multiple teams are running concurrently, since the call to `wake_up_all` would wake up all threads in all teams, which would interfere with other barriers. This barrier works similarly, but instead of waiting for an interrupt, the threads spin on the `generation` variable, which is incremented by the last arriving thread.

Listing A.17: Barrier::wait

```
1  inline void wait() {
2    if (runtime::numTeams == 1) {
3      // WFI barrier
4
5      // Increment the barrier counter
6      if ((numThreads - 1) == barrier.fetch_add(1,
7                                      std::memory_order_relaxed)) {
8        barrier.store(0, std::memory_order_relaxed);
9        std::atomic_thread_fence(std::memory_order_seq_cst);
10       wake_up_all();
11     }
12
13     // Some threads have not reached the barrier --> Let's wait
14     // Clear the wake-up trigger for the last core reaching
15     // the barrier as well
16     mempool_wfi();
17
18   } else {
19     // Spin generation barrier
20     kmp_int32 gen = generation;
21
22     // Increment the barrier counter
23     if ((numThreads - 1) == barrier.fetch_add(1,
24                                      std::memory_order_relaxed)) {
25       barrier.store(0, std::memory_order_relaxed);
26       generation.fetch_add(1, std::memory_order_relaxed);
```

```
27        std::atomic_thread_fence(std::memory_order_seq_cst);
28      }
29
30      while (gen == generation.load(std::memory_order_relaxed)) {
31        // Spin
32      }
33    }
34  };
```

A.2.3. Thread

Attributes

- `kmp_int32 gtid`: Global ID of the thread.
- `kmp_int32 tid`: Team-local ID of the thread.
- `std::optional<Task> teamsRegion`: Optional task representing the body of the `teams` construct. This is only present in the master thread of the team.
- `Team *currentTeam`: Pointer to the team the thread is part of.
- `Mutex running`: Mutex indicating whether the thread is running. It is locked if it is, unlocked otherwise. [6]
- `std::optional<kmp_int32> requestedNumThreads`: Optional number of threads requested by a previous call to `__kmpc_push_num_threads` (Appendix A.1.1).

Relevant Methods

- `void Thread::run()`: This function implements the infinite loop that each thread except the thread with global ID 0 runs to wait for and execute work assigned to it.

  First, the thread waits for an interrupt. After it is woken up, it acquires the `running` mutex and checks that it is part of a team by ensuring that `currentTeam` is not `null`, and whether it is the master thread of a team by checking if `teamsRegion` has a value.

  If `currentTeam` is not `null` and `teamsRegion` is empty, meaning that the calling thread is a worker thread, the thread executes the implicit task of the team (Appendix A.2.4), resets the `currentTeam` pointer, sets its `tid` to match its `gtid`, and executes the team's barrier associated with the end of the `parallel` region. The `tid`

[6] This is only necessary to support the Banshee [12] simulator, which does not currently support the accumulation of interrupts and can cause threads to miss their wake-up signal.

is reset in this way to speed up the forking process when launching only one team later, since then every thread's `tid` matches its `gtid`. The team pointer has to be reset before executing the barrier to avoid the case where a new team is assigned to the thread after the barrier but before resetting the pointer, which would cause the thread to miss the assignment to the new team.

If `teamsRegion` has a value, that means that the current thread is the master thread of a team and therefore has to execute the body of the `teams` construct. After doing that, it resets the `teamsRegion` pointer, deletes the team object, and executes the runtime's barrier associated with the end of the `teams` construct. (Section 2.2).

Listing A.18: Thread::run

```
1  void Thread::run() {
2    while (true) {
3      mempool_wfi();
4      std::lock_guard<Mutex> lock(running);
5
6      if (currentTeam != nullptr && !teamsRegion.has_value()) {
7
8        (*currentTeam).getImplicitTask().run(gtid, tid);
9
10       Team *prevTeam = currentTeam;
11       currentTeam = nullptr;
12       tid = gtid;
13
14       (*prevTeam).getBarrier().wait();
15
16     } else if (teamsRegion.has_value()) {
17       teamsRegion->run(gtid, tid);
18
19       teamsRegion.reset();
20
21       delete currentTeam;
22       currentTeam = nullptr;
23       tid = gtid;
24
25       runtime::teamsBarrier.wait();
26     }
27   }
28 };
```

- `void Thread::forkCall(Task parallelRegion)`: This function is called by the master thread of a team when it encouters a `parallel` construct.

It first checks if a specific number of threads has been requested and uses the total number of cores as the default if not. Then, it sets the number of threads for the team as well as the implicit task, which is then run on all threads in the team including the calling thread. Finally, it executes the team's barrier.

Listing A.19: Thread::forkCall

```
1  void Thread::forkCall(Task parallelRegion) {
2    kmp_int32 numThreads = this->requestedNumThreads
3                                  .value_or(NUM_CORES);
4    this->requestedNumThreads.reset();
5
6    Team *team = currentTeam;
7
8    // Setup
9    team->setNumThreads(numThreads);
10   team->setImplicitTask(parallelRegion);
11
12   // Run on all threads
13   team->run();
14   parallelRegion.run(gtid, tid);
15
16   team->getBarrier().wait();
17 };
```

- `void Thread::forkTeams(Task teamsRegion)`: This function is called by the thread with global ID 0 when it encounters a `teams` construct.

  First, it checks if a specific number of teams was requested and uses the number of groups (Section 2.1.1) as the default if not. This hints at the possibility of having different MemPool groups work on different parts of a given problem or on different problems altogether in the future. Then it sets both the global variable `runtime::numTeams` (Appendix A.3.1) and the number of threads participating in the global `runtime::teamsBarrier` (Appendix A.3.1) accordingly, and resets `runtime::requestedNumTeams` for the next call.

  Next, it identifies the master threads of each team by dividing the total number of cores into `runtime::numTeams` equally sized chunks. Then, for each thread except for the calling thread[7], a new team is created with master thread `coreId` and team ID `i`. After setting the `teams` region and the requested thread limit, if any, the thread is woken up.

[7]The thread with global ID 0 is always part of the default team (Appendix A.3.1), therefore it is not necessary to create a new team for it.

The calling thread also sets its `teams` region and explicitly runs it after setting the requested thread limit if applicable. After executing the region, it is reset and the global `teams` barrier is executed, finally resetting the number of teams to 1.

Listing A.20: void Thread::forkTeams

```
1  void Thread::forkTeams(Task teamsRegion) {
2    runtime::numTeams = runtime::requestedNumTeams
3                                  .value_or(NUM_GROUPS);
4    runtime::teamsBarrier.setNumThreads(runtime::numTeams);
5    runtime::requestedNumTeams.reset();
6
7    kmp_int32 coresPerTeam = NUM_CORES / runtime::numTeams;
8
9    for (kmp_int32 i = 1; i < runtime::numTeams; i++) {
10     kmp_int32 coreId = i * coresPerTeam;
11
12     Thread &thread = runtime::getThread(coreId);
13
14     thread.setCurrentTeam(new Team(coreId, i));
15     thread.setTeamsRegion(teamsRegion);
16
17     if (runtime::requestedThreadLimit) {
18       thread.requestNumThreads(
19               runtime::requestedThreadLimit.value());
20     }
21
22     thread.wakeUp();
23   }
24
25   this->setTeamsRegion(teamsRegion);
26   if (runtime::requestedThreadLimit) {
27     this->requestNumThreads(
28             runtime::requestedThreadLimit.value());
29   }
30
31   teamsRegion.run(gtid, tid);
32   this->teamsRegion.reset();
33
34   runtime::teamsBarrier.wait();
35
36   runtime::numTeams = 1;
37 };
```

A.2.4. Team

Attributes

- `kmp_int32 masterGtid`: Global ID of the master thread of the team.
- `kmp_int32 teamId`: ID of the team.
- `kmp_int32 numThreads`: Number of threads in the team.
- `Barrier barrier`: Barrier associated with the team.
- `DynamicSchedule dynamicSchedule`: Struct containing information related to dynamic loops.
- `void *copyPrivateData`: Pointer to the `copyprivate` data to be broadcast to the thereads in the team.
- `Task implicitTask`: Task representing the implicit task of the team.

Relevant Methods

- `void Team::run()`: This function is used to launch the team and is called by the master thread. It distinguishes between two cases:
  1. If there are multiple teams, it iterates sequentially over the threads with a global ID in the range $[\texttt{masterGtid}, \texttt{masterGtid} + \texttt{numThreads} - 1]$, sets their team local thread ID, and, if they are not the master thread, sets the team pointer and wakes them up.
  2. If there is only one team, it iterates over all threads in the team in the same way and sets their team pointer. Unlike in the previous case, now all threads can be woken up at once using `wake_up_all`, which is slightly more efficient.

Listing A.21: void Team::run

```
1  inline void run() {
2    if (runtime::numTeams > 1) {
3      for (kmp_int32 i = masterGtid + 1;
4            i < masterGtid + numThreads; i++) {
5        auto &thread = runtime::getThread(i);
6        thread.setTid(i - masterGtid);
7
8        if (i != masterGtid) {
9          thread.setCurrentTeam(this);
10         thread.wakeUp();
11       }
```

```
12      }
13    } else {
14      for (kmp_int32 i = masterGtid + 1;
15            i < masterGtid + numThreads; i++) {
16        auto &thread = runtime::getThread(i);
17        thread.setCurrentTeam(this);
18      }
19
20      wake_up_all();
21      mempool_wfi();
22    }
23  }
```

- `void Team::forStaticInit(ident_t *loc, kmp_int32 gtid, kmp_sched_type schedtype, T *plastiter, T *plower, T *pupper, SignedT *pstride, SignedT incr, SignedT chunk)`: This function is used to determine the static loop parameters for the `for` and `distribute` constructs. It is called once by every participating thread.

  We assume that the loop increment is always 1, since this is the case for the code generated by LLVM 14.

  It first distinguishes between the `static` and `static_chunked` scheduling types. In the former case, the chunk size is not provided and is therefore calculated as the ceiling of the loop size divided by the number of threads, since the OpenMP standard specifies that the work should be distributed as evenly as possible. The switch case then falls through to the `static_chunked` case, where we assume that the chunk size is already provided.

  We define the `span` like on Qiu's work as the distance between two consecutive iterations of the loop, possibly running on different threads. With this, we can calculate the `stride`, which just accounts for the number of threads in between to iterations running on the same thread.

  The `plower` bound is calculated by shifting the `span` `tid` times starting from the global lower bound, and `pupper` is that plus the `span` minus the increment. The `plastiter` flag is set to `true` if the calling thread is the last one to execute the loop.

  The scheduling related to the `distribute` construct is almost the same, except that the number of teams is used instead of the number of threads.

Listing A.22: void Team::forStaticInit

```
1  template <typename T,
2            typename SignedT = typename std::make_signed<T>::type,
3            typename UnsignedT =
4                        typename std::make_unsigned<T>::type>
```

```
 5 void forStaticInit(ident_t * loc, kmp_int32 gtid,
 6                    kmp_sched_type schedtype, T *plastiter,
 7                    T *plower, T *pupper, SignedT *pstride,
 8                    SignedT incr, SignedT chunk) const {
 9
10   assert(incr == 1 && "Loop increment is not 1");
11
12   switch (schedtype) {
13   case kmp_sch_static: {
14
15     // Calculate chunk size
16     chunk = static_cast<SignedT>(*pupper - *plower + 1) /
17               numThreads +
18             (static_cast<SignedT>(*pupper - *plower + 1) %
19               numThreads != 0);
20
21     // Fall through to static chunked
22   }
23   case kmp_sch_static_chunked: {
24     assert(incr != 0 && "Loop increment must be non-zero");
25     assert(chunk > 0 && "Chunk size is not positive");
26     assert((static_cast<T>(chunk) <= *pupper - *plower + 1) &&
27            "Chunk size is greater than loop size");
28
29     kmp_int32 tid = runtime::getThread(gtid).getTid();
30
31     SignedT numChunks =
32         (static_cast<SignedT>(*pupper - *plower) + chunk) / chunk;
33
34     SignedT span = incr * chunk;
35     *pstride = span * static_cast<SignedT>(numThreads);
36     *plower = *plower + static_cast<T>(tid) *
37                 static_cast<T>(span);
38     *pupper = *plower + static_cast<T>(span - incr);
39     *plastiter = (tid == (numChunks - 1) % numThreads);
40
41     break;
42   }
43
44   // Distribute (teams)
45   case kmp_distribute_static: {
46
47     // Calculate chunk size
```

```
48      chunk =
49          static_cast<SignedT>(*pupper - *plower + 1) /
50            runtime::numTeams +
51          (static_cast<SignedT>(*pupper - *plower + 1) %
52            runtime::numTeams != 0);
53
54      // Fall through to static chunked
55    }
56    case kmp_distribute_static_chunked: {
57      assert(incr != 0 && "Loop increment must be non-zero");
58      assert(chunk > 0 && "Chunk size is not positive");
59      assert((static_cast<T>(chunk) <= *pupper - *plower + 1) &&
60             "Chunk size is greater than loop size");
61
62      SignedT numChunks =
63          (static_cast<SignedT>(*pupper - *plower) + chunk) / chunk;
64
65      SignedT span = incr * chunk;
66      *pstride = span * static_cast<SignedT>(runtime::numTeams);
67      *plower = *plower + static_cast<T>(teamId) *
68                  static_cast<T>(span);
69      *pupper = *plower + static_cast<T>(span - incr);
70      *plastiter = (teamId == (numChunks - 1) % runtime::numTeams);
71
72      break;
73    }
74    default: {
75      assert(false && "Unsupported scheduling type");
76      break;
77    }
78    }
79  }
```

- `void Team::void dispatchInit(ident_t * loc, kmp_int32 gtid, kmp_sched_type schedtype, T lower, T upper, SignedT incr, SignedT chunk)`: This function is called once by every thread participating in a dynamic loop to initialize the dynamic schedule struct, which is shown in Listing A.23. This contains all of the loop parameters as well as a `valid` flag indicating whether the schedule has been initialized, a counter `numDone` to count how many threads have finished executing the loop, and a mutex to lock access to the struct since it is accessed concurrently by multiple threads.

  Our implementation of the function currently only supports the `kmp_sch_dynamic_chunked` scheduling type, which is used by dynamic `for` loops containing either `schedule(dynamic,`

`chunk)` or `schedule(dynamic)` clauses.

After locking the schedule struct, it first checks if the schedule has already been initialized by another thread and immediately returns if that is the case, since this only needs to be done once.

Otherwise, `lower`, `upper` and `chunk` are directly stored, while the remaining parameters are calculated as in Listing A.22. Finally, the `valid` flag is set to `true`.

Listing A.23: struct Team::DynamicSchedule

```
1  struct DynamicSchedule {
2    kmp_uint32 lowerNext = 0;
3    kmp_uint32 upper = 0;
4    kmp_uint32 chunk = 0; // Chunk size assumed to be positive
5    kmp_int32 incr = 0;
6    kmp_int32 stride = 0;
7
8    bool valid = false;
9    kmp_int32 numDone = 0;
10
11   Mutex mutex;
12 };
```

Listing A.24: void Team::dispatchInit

```
1  template <typename T,
2            typename SignedT = typename std::make_signed<T>::type,
3            typename UnsignedT =
4                        typename std::make_unsigned<T>::type>
5  void dispatchInit(ident_t * loc, kmp_int32 gtid,
6                    kmp_sched_type schedtype, T lower, T upper,
7                    SignedT incr, SignedT chunk) {
8
9    assert(incr == 1 && "Loop increment is not 1");
10   assert(chunk > 0 && "Chunk size is not positive");
11   assert((static_cast<T>(chunk) <= upper - lower + 1) &&
12          "Chunk size is greater than loop size");
13
14   switch (schedtype) {
15   case kmp_sch_dynamic_chunked: {
16     std::lock_guard<Mutex> lock(dynamicSchedule.mutex);
17
18     if (dynamicSchedule.valid) {
19       return;
20     }
```

```
21
22      SignedT span = incr * chunk;
23
24      dynamicSchedule.lowerNext = static_cast<kmp_uint32>(lower);
25      dynamicSchedule.upper = static_cast<kmp_uint32>(upper);
26      dynamicSchedule.chunk = static_cast<kmp_uint32>(chunk);
27      dynamicSchedule.incr = incr;
28      dynamicSchedule.stride = span *
29        static_cast<SignedT>(numThreads);
30
31      dynamicSchedule.valid = true;
32      break;
33    }
34    default: {
35      assert(false && "Unsupported scheduling type");
36      break;
37    }
38    };
39  }
```

- `bool Team::dispatchNext(ident_t *loc, kmp_int32 gtid, SignedT *plastiter, T *plower, T *pupper, SignedT *pstride)`:
  This function is called by a thread executing a dynamic loop to obtain the next chunk of iterations to execute. It uses the type variable `T` to account for the fact that there are both signed and unsigned versions of the entrypoint (Appendix A.1.3) associated with this function.

  First, it locks the dynamic schedule object so that it can be safely modified, since other threads will try to access it concurrently. Then, it asserts that the schedule is initialized by checking that `valid` is `true`.

  Next, it checks if there is any work left to be done by seeing if `lowerNext` is greater than `upper`. If that is the case, then that means that there are no more work chunks available, so it increments the number of threads that are done. If all are done, it sets the `valid` flag to `false` and resets `numDone` so that the schedule can be reused. It returns `false` to indicate that there was no chunk left to execute.

  If there is work left, it sets the lower bound of the current chunk to `lowerNext` and increments `lowerNext` by the chunk size. `lowerNext` is checked again against `upper` to know if the current chunk is the last one. If that is the case, the upper bound is set to the global upper bound and `plastiter` is set to `true`. Otherwise, the upper bound is set to `lowerNext - 1` and `plastiter` is set to `false`. Finally, the stride is set to the stride of the schedule without modification and `true` is returned to indicate that there was a chunk to execute.

Listing A.25: void Team::dispatchNext

```
template <typename T,
          typename SignedT = typename std::make_signed<T>::type>
bool dispatchNext(ident_t *loc, kmp_int32 gtid,
                  SignedT *plastiter, T *plower,
                  T *pupper, SignedT *pstride) {

  std::lock_guard<Mutex> lock(dynamicSchedule.mutex);
  assert(dynamicSchedule.valid
         && "Dynamic schedule is not valid");

  if (dynamicSchedule.lowerNext > dynamicSchedule.upper) {
    if (++dynamicSchedule.numDone == numThreads) {
      dynamicSchedule.valid = false;
      dynamicSchedule.numDone = 0;
    }

    return false;
  }

  *plower = static_cast<T>(dynamicSchedule.lowerNext);

  dynamicSchedule.lowerNext += dynamicSchedule.chunk;
  if (dynamicSchedule.lowerNext > dynamicSchedule.upper) {
    *pupper = static_cast<T>(dynamicSchedule.upper);
    *plastiter = true;
  } else {
    *pupper = static_cast<T>(dynamicSchedule.lowerNext - 1);
    *plastiter = false;
  }

  *pstride = dynamicSchedule.stride;

  return true;
};
```

- `void Team::copyPrivate(ident_t *loc, kmp_int32 gtid, size_t cpy_size, void *cpy_data, void (*cpy_func)(void *, void *), kmp_int32 didit)`: This function is used to implement the `copyprivate` clause of a `single` section.

  If the calling thread executed the `single` section, as indicated by `didit` being unequal to 0, the pointer to the private data is stored in the team's `copyPrivateData` field. Otherwise, the private data is copied by calling `cpy_func`.

A barrier is executed in between to ensure that the pointer is set before the data is copied. The one at the end makes sure that all thread have copied the data before continuing.

Listing A.26: void Team::copyprivate

```
1  inline void copyPrivate(ident_t *loc, kmp_int32 gtid,
2                          size_t cpy_size, void *cpy_data,
3                          void (*cpy_func)(void *, void *),
4                          kmp_int32 didit) {
5    if (didit != 0) {
6      copyPrivateData = cpy_data;
7    }
8
9    barrier.wait();
10
11   if (didit == 0) {
12     cpy_func(cpy_data, copyPrivateData);
13   }
14
15   barrier.wait();
16 };
```

- `Team::~Team`: This is the team's destructor. In order to avoid threads getting stuck in the barrier associated with the team object by spinning on the `generation` variable (Appendix A.2.2), which would be deallocated when the team object is destroyed, the destructor waits for all threads in the team to go to sleep before destroying the team.

Listing A.27: Team::~Team

```
1  inline ~Team() {
2    for (kmp_int32 i = masterGtid + 1; i < masterGtid + numThreads;
3          i++) {
4      while (runtime::getThread(i).isRunning()) {
5        // Wait for thread to finish
6      }
7    }
8  }
```

A.3. Supporting Code

A.3.1. Runtime Namespace

The `kmp::runtime` namespace contains global variables used throughout our implementation as well as some helper functions.

Global Variables

1. `threads`: Array of `Thread` objects, where each one is initialized with the corresponding core ID.
2. `defaultTeam`: Default team used if no `teams` construct is used.
3. `requestedNumTeams`: Number of teams requested by the `num_teams` clause of a `teams` construct.
4. `requestedThreadLimit`: Number of threads requested by the `thread_limit` clause of a `teams` construct.
5. `numTeams`: Number of teams currently running.
6. `teamsBarrier`: Barrier used to synchronize the end of a `teams` construct.

Listing A.28: Global Variables

```
template <kmp_int32... Is>
constexpr std::array<Thread, sizeof...(Is)>
sequencetoArray(std::integer_sequence<kmp_int32, Is...> /*unused*/) {
  return {{Is...}};
}

std::array<Thread, NUM_CORES> threads =
    sequencetoArray(
        std::make_integer_sequence<kmp_int32, NUM_CORES>{});

Team defaultTeam(0, 0);

std::optional<kmp_int32> requestedNumTeams;
std::optional<kmp_int32> requestedThreadLimit;
kmp_int32 numTeams = 1;

Barrier teamsBarrier(NUM_GROUPS);
```

Helper Functions

The helper functions are used to access and run threads more conveniently.

Listing A.29: Helper Functions

```
1  static inline void runThread(kmp_int32 core_id) {
2    threads[static_cast<kmp_uint32>(core_id)].run();
3  };
4
5  static inline Thread &getThread(kmp_int32 gtid) {
6    return threads[static_cast<kmp_uint32>(gtid)];
7  };
8
9  static inline Thread &getCurrentThread() {
10   return threads[mempool_get_core_id()];
11 };
```

Appendix B

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

TASK ASSIGNMENT FOR A
SEMESTER THESIS AT THE DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

SPRING SEMESTER 2024

Diego de los Santos

Implementing an OpenMP runtime for MemPool in LLVM

March 18, 2024

Advisors: Samuel Riedel, ETZ J69.2, Tel. +41 44 632 65 69, sriedel@iis.ee.ethz.ch
Sergio Mazzola, ETZ J76.2, Tel. +41 44 632 81 49, smazzola@iis.ee.ethz.ch

Professor: Prof. Dr. L. Benini

Handout: March 18, 2024
Due: July 01, 2024

1 Introduction

Striving for high image quality, even on mobile devices, has led to an explosion in the pixel count of smartphone cameras over the last decade [1]. These image sensors, boasting tens of millions of pixels, create a massive amount of data to be processed on a tight power envelope as quickly as possible. Computational photography, computer vision, augmented reality, and machine learning are only a few of the possible applications. Highly specialized image signal processors (ISPs) harness the features of these workloads, which are highly parallelizable, to meet timing and power constraints. Google offers a prominent example: the Pixel Visual Core, employed for image processing in Google smartphones [2].

At ETH, we are developing our own manycore system that can be used as an ISP called MemPool [3]. It boasts 256 area-optimized 32-bit Snitch [4]. Snitch, developed at ETH as well, implements the RISC-V instruction set architecture (ISA), which is an open ISA targeting modularity and scalability [5]. Despite its size, MemPool gives all 256 cores low-latency access to the shared L1 memory, with a maximum latency of only five cycles when no contention occurs. This implements efficient communication among all cores, making MemPool suitable for various workload domains and easy to program.

A common framework for parallel programming is OpenMP [6]. It implements a task-based or fork-join approach to parallelization through annotations in the C code. The compiler will generate routines and runtime calls to distribute the parallel execution across the specific platform correctly. To this end, each system supporting OpenMP needs to implement a corresponding OpenMP Runtime. Specifically, there are two major runtimes, the GOMP and KMP runtime for GCC and LLVM, respectively. MemPool has minimal support for the GOMP runtime. However, since LLVM has become the standard compiler infrastructure for our research group, a KMP runtime to support LLVM is crucial.

Goal The goal of this thesis is to implement the KMP runtime for MemPool. To this end, the student will first identify the most important OpenMP functions for MemPool and then implement the runtime in conjunction with functional tests for verification. For benchmarking, the student will implement and evaluate digital signal processing (DSP) kernels and potentially full applications from the image processing or communications domain.

2 Milestones

The following are the milestones that we expect to achieve throughout the project:

- Become familiar with MemPool and the OpenMP runtime.

- Investigate the most useful features of the OpenMP standard.
- Implement the key features of the KMP runtime for MemPool and functionally verify them.
- Benchmark the newly implemented runtime on several DSP kernels.

2.1 Stretch Goals

Should the above milestones be reached earlier than expected and the student is motivated to do further work, we propose the following stretch goals to aim for:

- Advance the KMP implementation by implementing features beyond the basic set determined during the initial project phase.
- Build a full demonstrator application using OpenMP. The application can be chosen during the project, but examples include an HDR image processing pipeline, a ray tracing algorithm, or a 5G communication pipeline.
- Tune your benchmarks or demonstrator application for Heartstream, a 64-core chip implementing the latest MemPool architecture.

3 Project Realization

3.1 Time Schedule

The time schedule presented in Table 1 is merely a proposition; it is primarily intended as a reference and an estimation of the time required for each required step.

Project phase	**Time estimate**
Familiarization with MemPool and OpenMP to determine key features	2 weeks
Implementation and verification of key features	4 weeks
Evaluation of your work and implementation of DSP kernels in OpenMP	4 weeks
Write report	2 weeks
Prepare presentation	2 weeks

Table 1: Proposed time schedule and investment

3.2 Meetings

Weekly meetings will be held between the student and the assistants. The exact time and location of these meetings will be determined within the first week of the project in order to fit the student's and the assistants' schedule. These meetings will be used to evaluate the status and progress of the project. Beside these regular meetings, additional meetings can be organized to address urgent issues as well.

3.3 Weekly Reports

The student is advised, but not required, to write a weekly report at the end of each week and to send it to his advisors. The idea of the weekly report is to briefly summarize the work, progress and any findings made during the week, to plan the actions for the next week, and to bring up open questions and points. The weekly report is also an important means for the student to get a goal-oriented attitude to work.

3.4 Coding Guidelines

HDL Code Style Adapting a consistent code style is one of the most important steps in order to make your code easy to understand. If signals, processes, and modules are always named consistently, any inconsistency can be detected more easily. Moreover, if a design group shares the same naming and formatting conventions, all members immediately *feel at home* with each other's code. At IIS, we use lowRISC's style guide for SystemVerilog HDL: `https://github.com/lowRISC/style-guides/`.

Software Code Style We generally suggest that you use style guides or code formatters provided by the language's developers or community. For example, we recommend LLVM's or Google's code styles with `clang-format` for C/C++, PEP-8 and `pylint` for Python, and the official style guide with `rustfmt` for Rust.

Version Control Even in the context of a student project, keeping a precise history of changes is *essential* to a maintainable codebase. You may also need to collaborate with others, adopt their changes to existing code, or work on different versions of your code concurrently. For all of these purposes, we heavily use *Git* as a version control system at IIS. If you have no previous experience with Git, we *strongly* advise you to familiarize yourself with the basic Git workflow before you start your project.

3.5 Report

Documentation is an important and often overlooked aspect of engineering. A final report has to be completed within this project.

The common language of engineering is de facto English. Therefore, the final report of the work is preferred to be written in English.

Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Inkscape or any other vector drawing software (for block diagrams) is strongly encouraged by the IIS staff.

If you write the report in LaTeX, we offer an instructive, ready-to-use template, which can be forked from the Git repository at `https://iis-git.ee.ethz.ch/akurth/iisreport`.

Final Report The final report has to be presented at the end of the project and a digital copy needs to be handed in and remain property of the IIS. Note that this task description is part of your report and has to be attached to your final report.

3.6 Presentation

There will be a presentation (15 min presentation and 5 min Q&A) at the end of this project in order to present your results to a wider audience. The exact date will be determined towards the end of the work.

4 Deliverables

In order to complete the project successfully, the following deliverables have to be submitted at the end of the work:

- Final report incl. presentation slides
- Source code and documentation for all developed software and hardware
- Testsuites (software) and testbenches (hardware)
- Synthesis and implementation scripts, results, and reports

References

[1] S. Skafisk, *This is How Smartphone Cameras Have Improved Over Time*, 2017 (accessed August 18, 2020). [Online]. Available: https://petapixel.com/2017/06/16/smartphone-cameras-improved-time/

[2] J. Redgrave, A. Meixner, N. Goulding-Hotta, A. Vasilyev, and O. Shacham, "Pixel Visual Core: Google's fully programmable image, vision and AI processor for mobile devices," in *2018 IEEE Hot Chips 30 Symposium (HC30)*. Cupertino, US: IEEE Technical Committee on Microprocessors and Microcomputers, Aug. 2018.

[3] S. Riedel, M. Cavalcante, R. Andri, and L. Benini, "MemPool: A scalable manycore architecture with a low-latency shared L1 memory," *IEEE Transactions on Computers*, vol. 72, no. 12, pp. 3561–3575, 2023.

[4] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Trans. Comput.*, vol. 70, no. 11, pp. 1845–1860, Feb. 2021.

[5] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, A. Waterman, Y. Lee, and D. Patterson, "The RISC-V instruction set manual," 2014. [Online]. Available: https://github.com/riscv/riscv-isa-manual

[6] O. A. R. Board, "OpenMP Application Programming Interface," 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

Zurich, March 18, 2024 Prof. Dr. L. Benini

List of Acronyms

API . . . . . Application Programming Interface
AXI . . . . . Advanced eXtensible Interface

CSR . . . . . Compresed Sparse Row

DMA . . . . . Direct Memory Access

ELF . . . . . Executable and Linkable Format

GCC . . . . . GNU Compiler Collection

HAL . . . . . Hardware Abstraction Layer

MMM . . . . . Matrix-Matrix Multiplication

OS . . . . . Operating System

SoC . . . . . System on Chip
SPVM . . . . . Sparse Matrix-Vector Multiplication

WFI . . . . . Wait For Interrupt

List of Figures

List of Tables

Bibliography

[1] S. Riedel, M. Cavalcante, R. Andri, and L. Benini, "Mempool: A scalable manycore architecture with a low-latency shared l1 memory," *IEEE Transactions on Computers*, vol. 72, no. 12, p. 3561–3575, Dec. 2023. [Online]. Available: http://dx.doi.org/10.1109/TC.2023.3307796

[2] M. Bertuletti, S. Riedel, Y. Zhang, A. Vanelli-Coralli, and L. Benini, "Fast shared-memory barrier synchronization for a 1024-cores risc-v many-core cluster," 2023.

[3] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers*, vol. 70, no. 11, p. 1845–1860, Nov. 2021. [Online]. Available: http://dx.doi.org/10.1109/TC.2020.3027900

[4] *OpenMP: The OpenMP API specification for parallel programming*. [Online]. Available: https://ww.openmp.org

[5] W. Qiu, "Design and verification of llvm openmp runtime library on pulp/hero," 2019.

[6] A. Kurth, A. Capotondi, P. Vogel, L. Benini, and A. Marongiu, "Hero: an open-source research platform for hw/sw exploration of heterogeneous manycore systems," in *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ser. ANDARE '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3295816.3295821

[7] *LLVM OpenMP Runtime Library Reference*. [Online]. Available: https://raw.githubusercontent.com/llvm/llvm-project/main/openmp/runtime/doc/Reference.pdf

[8] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, jul 2012. [Online]. Available: https://doi.org/10.1145/2185520.2185528

[9] J. McGee, *C/C++ Runtime Startup*. [Online]. Available: https://etherealwake.com/2021/09/crt-startup/

[10] W. Snyder, P. Wasson, D. Galbi, and et al, *Verilator*, 2023. [Online]. Available: https://verilator.org

[11] Jera, *MinUnit – a minimal unit testing framework for C*. [Online]. Available: https://jera.com/techinfo/jtns/jtn002

[12] S. Riedel, F. Schuiki, P. Scheffler, F. Zaruba, and L. Benini, "Banshee: A fast LLVM-based RISC-V binary translator," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, Nov. 2021, pp. 1105–1113.

[13] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, *Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines*, 1st ed. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3596711.3596751