# Ghost Threading: Helper-Thread Prefetching for Real Systems

Yuxin Guo
University of Cambridge
Cambridge, UK
yuxin.guo@cl.cam.ac.uk

Akshay Bhosale
University of Cambridge
Cambridge, UK
asb227@cam.ac.uk

Utpal Bora
University of Cambridge
Cambridge, UK
utpal.bora@cl.cam.ac.uk

Alexandra W. Chadwick
University of Cambridge
Cambridge, UK
alex.chadwick@cl.cam.ac.uk

Márton Erdős
University of Cambridge
Cambridge, UK
marton.erdos@cl.cam.ac.uk

Giacomo Gabrielli
Arm
Cambridge, UK
giacomo.gabrielli@arm.com

Timothy M. Jones
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

## Abstract

Memory latency is the bottleneck for many modern workloads. One popular solution from literature to handle this is helper threading, a technique that issues light-weight prefetching helper thread(s) extracted from the original application to bring data into the cache before the main thread uses it, hiding the long memory latency. Although prior work has reported promising results, many schemes are not available on real systems as they require hardware support for satisfying performance improvements.

To address this, we present Ghost Threading, a software-only helper-thread prefetching solution, which issues helper threads on idle Simultaneous Multithreading (SMT) contexts. The key challenge of prefetching is timeliness: data should not arrive too early or too late. Unlike prior work relying on proposed extra hardware synchronization or expensive OS synchronization, we develop a novel inter-thread synchronization approach based on an instruction supported by commercial processors, which enables cheap throttling of the helper thread. This ensures that it gets far enough ahead of the main thread, but not too far, for it to perform timely prefetching of data.

We evaluate Ghost Threading against state-of-the-art techniques on a modern Intel processor with memory-intensive workloads selected from graph analysis, database, and HPC domains. On an idle server, Ghost Threading achieves 1.33× geometric mean speedup over the baseline, 1.25× and 1.11× over state-of-the-art software prefetching and parallelization techniques, respectively. We also show that Ghost Threading maintains these benefits on a busy server where the memory bandwidth pressure is higher.

## Keywords

Helper Threading, Prefetch, Simultaneous Multithreading


*MICRO '25, Seoul, Republic of Korea*

ACM ISBN 979-8-4007-1573-0/2025/10
https://doi.org/10.1145/3725843.3756106

**ACM Reference Format:**
Yuxin Guo, Akshay Bhosale, Utpal Bora, Alexandra W. Chadwick, Márton Erdős, Giacomo Gabrielli, and Timothy M. Jones. 2025. Ghost Threading: Helper-Thread Prefetching for Real Systems. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3725843.3756106

## 1 Introduction

Memory latency heavily limits the performance of many modern workloads [3, 21, 34]. One tempting approach to solve this challenge is the *helper threading* technique [8, 19, 20, 23–25, 29, 41, 42, 49]. The key idea of helper threading is to execute a light-weight helper thread ahead of the main application thread, triggering data fetches in advance. These threads are extracted from the original application by either the programmer, the compiler, or the hardware. Compared to conventional prefetching techniques [2, 3, 6, 9, 17, 18, 28, 31, 37, 40, 47, 48], helper threading enjoys high prefetching accuracy by generating prefetches via similar instructions to those that the program will execute in the near future.

Timeliness is crucial for all prefetching [8, 19, 20, 23]; data fetched too early may be evicted from the cache before being used, and data fetched too late will not improve performance. To ensure timeliness in helper threading, schemes use inter-thread synchronization to throttle or accelerate the helper thread. Conventional software inter-thread synchronization mechanisms are prohibitively expensive (e.g. it may take up to 30K cycles to suspend a thread through the operating system (OS) [23]), and so existing helper threading techniques usually apply hardware modification [8, 23, 25, 41, 42], ISA extension [19], or both [29] to control the synchronization of the main thread and helper thread. These schemes control the distance between the main thread and the helper thread by (1) skipping some work in the helper thread when it is too close to (or behind) the main thread, and stopping/suspending the helper thread when it runs too far ahead of the main thread, or (2) launching short-lived helper threads with careful timing.

Although enjoying promising performance improvements, the hardware support for many helper-threading schemes has not seen

commercial deployment on real systems. Meanwhile, the few approaches that are available on real systems either fail to fully exploit the potential of helper threading due to inefficient synchronization mechanisms [23] or require excessive hardware resources (up to four processor cores) to achieve a satisfactory performance improvement [20]. Existing real-world thread-management mechanisms are simply not optimized to execute light-weight helper threads for prefetching purposes.

We propose Ghost Threading, a software-only prefetching technique that runs on a core with Simultaneous Multithreading (SMT) support, executing each helper thread in another SMT context on the same core as its corresponding main thread. The key idea behind Ghost Threading is a novel inter-thread synchronization mechanism that controls the speed of the helper thread in a timely and efficient manner. We observe that the real challenge in controlling the speed of the helper thread is to prevent it from running too far ahead, as our experimental results suggest that the helper threads typically tend to run excessively far into the future. Regardless, it is already straightforward and cheap to accelerate the helper thread if it is too slow by skipping instructions (e.g. several loop iterations).

Unlike prior work applying a pause-restart scheme [8, 19, 23], we utilize the `serialize` instruction supported by Intel processors [15] to slow down the helper thread when it runs too far ahead of the main thread, removing any need for an expensive system call or hardware modification. Although the `serialize` instruction may not be designed for inter-thread synchronization, it represents an almost ideal solution for our use case: it stops the pipeline from fetching and executing the next instruction in the thread until all modifications to flags, registers, and memory by instructions before the `serialize` finish [15]. This means that the helper thread only consumes modest backend resources when being decelerated by the `serialize` instructions, thus making all remaining hardware resources available to the main thread.

To construct the helper thread, we first select the target loads through profiling and then manually extract the instructions they depend on. The helper thread executes fewer instructions than the main one and naturally runs ahead, bringing the data into the L1 cache before the main thread needs them. When it detects that it is running too far ahead and risks making prefetches too early, it executes `serialize` instructions to slow itself down. In the following text, we call the helper thread extracted for Ghost Threading and synchronized by the `serialize` instructions the *ghost thread*.

In addition to an efficient inter-thread synchronization mechanism, selecting the correct target loads for prefetching is crucial for performance improvement. Since the ghost threads run in SMT contexts and compete with the main threads for hardware resources, there is a risk of performance degradation if the benefits of prefetching do not outweigh the negative impact of hardware-resource competition. Therefore, we propose a heuristic to select target loads by predicting if they will be profitable from Ghost Threading based on profiling results. If a target load selected by our heuristic is included in a parallelizable loop, we use our ghost thread to replace the thread for parallelization; otherwise, we use the existing parallel version of the loop.

We compare Ghost Threading with state-of-the-art techniques (a conventional software prefetching technique [3] and parallelization through SMT) by evaluating 11 benchmarks selected from graph analysis, database, and HPC domains on a modern Intel processor. On an idle server, Ghost Threading achieves 1.25× and 1.11× geometric mean speedup, and 11% and 5% geometric mean package energy saving over state-of-the-art software prefetching and parallelization techniques, respectively. We also show that Ghost Threading maintains these benefits on a busy server where the memory bandwidth pressure is higher: it provides 1.31× and 1.13× geometric mean speedup over state-of-the-art software prefetching and parallelization techniques, respectively. Ghost Threading requires significantly less implementation effort than conventional parallelization, thus compounding the benefit when considering existing sequential code bases.

We make the following key contributions:

- We present Ghost Threading, a software prefetching technique that utilizes SMT helper threading and is available on real systems.
- We develop a novel inter-thread synchronization mechanism to provide efficient and timely control of the speed of the helper thread.
- We propose a heuristic, which is summarized from our experimental results, to select target loads of Ghost Threading based on profiling information.
- We develop a prototype compiler pass that automatically extracts ghost threads based on user annotations.
- We have open-sourced our Ghost Threading implementation.

## 2 Background

### 2.1 Simultaneous Multithreading

Simultaneous Multithreading (SMT) [45, 46] is a technique that allows multiple independent threads to issue instructions to the functional units of a CPU core in a single cycle, increasing the core utilization when a thread stalls (e.g. due to a long-latency memory accesses). Compared with conventional multithreading techniques, SMT removes the requirement of a context switch, enabling multiple threads to execute in parallel in one core. Although originally designed for executing multiple independent threads, SMT can also be used to improve single-thread performance by helper threading.

SMT has been implemented by vendors such as Intel, AMD, and IBM. In this work, we utilize Hyper-Threading [30], the SMT implementation from Intel, to execute our ghost threads for prefetching. The Hyper-Threading technique enables two SMT threads per core, regarding one physical core as two independent logical cores. Compared with the original SMT technique, which shares almost all hardware resources between SMT threads, Intel Hyper-Threading evenly partitions several key buffers, such as the reorder buffer (ROB) and the load/store buffer, in out-of-order cores to enforce fairness and prevent deadlocks [15, 23, 30].

### 2.2 Memory-Level Parallelism

Unlike instruction-, thread-, and data-level parallelism, which ultimately describe the concurrent or parallel execution of multiple instructions, memory-level parallelism (MLP) improves the performance of modern processors by overlapping the latency of multiple memory-access operations. The foundation of MLP is that modern caches and memory chips support concurrent data access by techniques like memory banking. Meanwhile, modern processors

are equipped with structures, such as a load/store unit (LSU), used to manage in-flight memory operations. By executing multiple memory-access operations simultaneously, a processor hides the latency of memory accesses by overlapping them and hence reduces the possibilities of a full-window stall. MLP is becoming an increasingly important solution to the memory wall faced by modern computing systems, as memory latency is hard to reduce and may even increase to obtain other benefits [10].

## 2.3 Prefetching

Prefetching is a technique for extracting MLP by bringing data or instructions closer to the core before they are needed. Conventional hardware prefetching techniques generally rely on adding hardware components around caches to generate speculative memory accesses based on the history of cache misses [2, 6, 9, 18, 28, 37, 40, 47, 48]. Alternatively, software prefetching techniques utilize prefetching instructions inserted by programmers or compilers to prefetch data [1, 3, 17, 31]. Another distinct approach is execution-based prefetching, such as helper threading [8, 19, 20, 23–25, 29, 41, 42, 49] and runahead execution [7, 12, 13, 32–36, 38, 39]. These execution-based approaches trigger concurrent cache misses by speculatively executing instructions that are either likely or guaranteed to be executed later when the CPU pipeline is at risk of stalling.

## 2.4 Helper Threading

Helper threading [8, 19, 20, 23–25, 29, 41, 42, 49] is a powerful approach to improve MLP and performance. The basic mechanism of helper threading is to execute a subset of the original program ahead of the main application thread. The extracted subset (called p-slices by prior work [7, 8, 20]) only contains a few delinquent loads with high cache-missing ratios and the instructions that these loads depend upon, and thus usually executes fewer instructions than the main thread. Therefore, it naturally runs ahead of the main one, triggering cache misses through loads that the main thread will execute in the future and generating prefetch effects. The p-slices are executed as the helper thread either by an idle thread context of the same CPU (i.e. SMT context) [8, 23, 25], or another idle core [19, 20, 41, 42].

To identify the loads to be prefetched, existing techniques [8, 19, 20, 23, 24] usually rely on profiling to select the loads that have a high cache-miss ratio and coverage time. The helper thread can be extracted by either the programmer [20, 29, 49], compiler [23, 24], or hardware [25, 41, 42]. To control the speed of the helper thread, an inter-thread synchronization mechanism (either hardware or software) is required. Without hardware support, existing software synchronization techniques do not have access to the detailed state of the processor to properly determine the behavior of the helper threads at run time, requiring careful manual adjustment of hyper-parameters (e.g. synchronization period and the runahead distance of the helper thread) to obtain speedup [19, 20, 23].

The overhead of the hardware support required by helper threading techniques is not trivial. For example, two state-of-the-art techniques—Bootstrapping [25], which uses SMT to execute a look-ahead thread, and the Slipstream Processor [41], which uses an additional core to run an 'advanced stream'—both require new hardware components and changes in the design of the cache, ROB, and the fetch stage of the pipeline. In addition, Bootstrapping modifies the processor renaming stage, while the Slipstream Processor needs an inter-core communication buffer and modifications to the execution and commit stages.

```
0  for (int i = 0; i < NUM_OF_ITER; i++){
1    sum += hash(hash(array[index[i]]));
2  }
```

**(a) Original `Camel` benchmark.**

```
0  for (int i = 0; i < LARGER_NUM_OF_ITER/128; i++){
1    // may not be a constant iteration count
2    for (int j = 0; j < 128; j++){
3      sum += simple_hash(array[i][index[j]]);
4    }
5  }
```

**(b) `Camel` suitable for parallelization.**

```
0  for (int i = 0; i < NUM_OF_ITER/128; i++){
1    // may not be a constant iteration count
2    for (int j = 0; j < 128; j++){
3      sum += hash(hash(array[i][index[j]]));
4    }
5  }
```

**(c) `Camel` suitable for Ghost Threading.**

**Figure 1: Pseudo code of the `Camel` benchmark [3]. The target load for prefetching is at line 1 (figure 1(a)) or line 3 (figures 1(b) and 1(c)). Software prefetching works well on the original, parallelization works well when the target load's CPI is not too high and there is little computation performed with the loaded value, and Ghost Threading works well when the target load's CPI is high and there is more computation performed with the loaded value.**

# 3 Motivation

Ainsworth and Jones explored a space of loops for their software prefetching technique using two synthetic workloads [3]. We reuse their `Camel` benchmark to demonstrate loop characteristics that are favorable to different approaches, with code shown in figure 1.

`Camel` demonstrates a situation where a loop contains a load with a very high last-level cache (LLC) miss ratio, but where this load does not depend upon any results from prior iterations. When this load causes a long-latency main-memory access, modern out-of-order processors continuously fetch future instructions into the ROB and execute the instructions whose inputs are available. To maintain the sequential semantics of the original code, instructions in the ROB are committed in program order. When younger instructions complete their execution, they can not be removed from the ROB if there are any unfinished older ones (e.g. the load missing in the LLC). When the ROB is filled by instructions, the pipeline stalls as no more instructions can be fetched until space is freed in the ROB. This state is known as a full-window stall. Figure 2 illustrates how existing techniques and Ghost Threading boost single-core performance by increasing MLP when a full-window stall occurs.

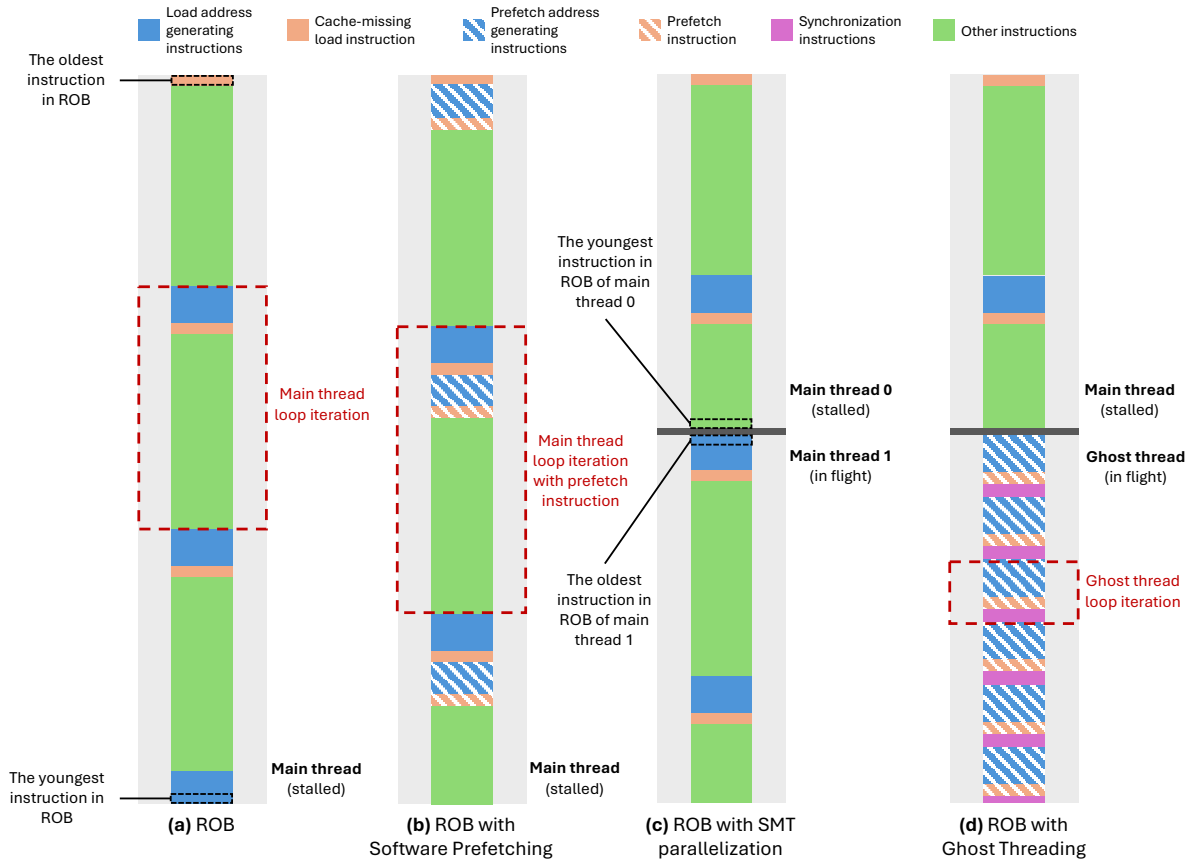



**Figure 2: ROB status of the single-threaded baseline, software prefetching technique, parallelization, and Ghost Threading (from left to right) when a full-window stall occurs. The upper portion of the ROB is occupied by older instructions, while the lower portion holds younger instructions. We assume the hardware resources are equally partitioned between the threads on the same physical core when SMT is applied.**

For simplicity, we assume that the SMT equally partitions hardware resources, like the ROB, for the threads issued.

*Out-of-Order Execution.* Figure 2(a) shows the state of a modern out-of-order processor when suffering from a full-window stall. When the ROB is blocked by the first cache-missing load, in this example three more loop iterations fit into the ROB and are fetched. As the cache-missing load is not self-dependent, the copies of the load in later iterations will be executed as soon as their inputs are ready. Here, two more loads are executed simultaneously with the first one and their latencies are overlapped to increase MLP.

*Software Prefetching.* Conventional software prefetching techniques improve MLP by inserting prefetch instructions in each loop iteration, increasing the size of the original loop to provide additional in-flight memory accesses within the ROB (figure 2(b)). This works well in the original `Camel` code, figure 1(a), because the number of instructions used to generate the address of the long-latency load is small compared to the number of instructions used for computation with the value loaded. Figure 3 shows that Ainsworth and Jones' software prefetching technique can obtain a significant speedup on this loop.

*Parallel execution via SMT.* If the loop is parallelizable, it obtains two benefits from SMT (illustrated by figure 2(c)). First, the alignment of instructions may mean that more loads are fetched into the ROB. Second, parallelization via SMT reduces the possibility of the whole pipeline stalling since threads are independent of each other. Altering the `Camel` loop to be favorable to SMT (figure 1(b)) shows that this works well when there is very little computation performed with the value loaded, meaning that a larger fraction of the loop body is used to calculate the address of the cache-missing load. An OpenMP-parallelized loop obtains modest performance improvements when the load sometimes hits and sometimes misses the cache (figure 3).

*Ghost Threading.* A more aggressive approach than the other techniques is Ghost Threading, which uses idle SMT contexts to execute ghost threads that include prefetching and inter-thread synchronization instructions. Unlike conventional software prefetching techniques, Ghost Threading decouples the prefetching instructions from the main thread and has the potential to improve the number of in-flight memory access operations in the ROB by a larger amount. As illustrated by figure 2(d), the number of in-flight

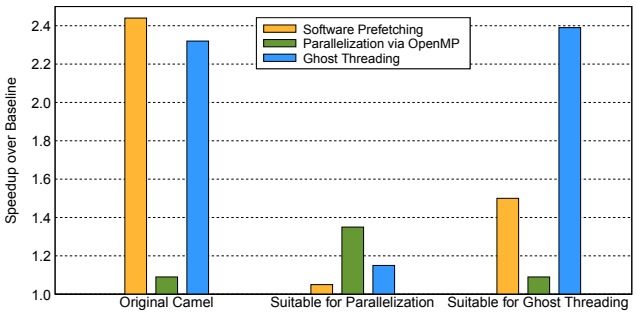



**Figure 3: Speedup of a state-of-the-art software prefetching technique [3], parallelization via SMT, and Ghost Threading over the baseline out-of-order core for the forms of `Camel` shown in figure 1. Section 5 discusses our experimental setup.**

loads is raised from three to eight. Additionally, the ghost thread replaces the blocking loads in the main thread with non-blocking prefetch instructions, enabling the instructions of the ghost thread to keep flowing in the ROB. In this case, the MLP exploited by Ghost Threading is limited by the load/store unit or miss status handling registers (MSHRs) instead of ROB size.

Ghost Threading works best when the cache-missing load has a high CPI and there is more computation performed with the value loaded than is profitable for SMT parallelization. Figure 1(c) shows the modified `Camel` code that demonstrates this, leading to substantial speedups again in figure 3. Ghost Threading is profitable as long as (1) the target load has a high CPI, (2) the innermost loop including this load has a larger dynamic size (i.e. instructions per iteration) than the extracted loop in the ghost thread, and (3) the ghost thread runs ahead at a suitable distance from the main thread. Otherwise, issuing the ghost thread could result in a slowdown, as the shared hardware resources are not profitably used for prefetching.

Ghost Threading is a more general-purpose technique for exploiting MLP than software prefetching or parallelization. Ainsworth and Jones' technique targets indirect loads and requires a flat loop structure to maximize benefits. For example, it cannot prefetch far enough ahead in figures 1(b) and 1(c) due to use of a nested loop with a low number of iterations. Parallelization may require significant rewriting of single-threaded applications and has limited benefits when code has cross-iteration dependences. In contrast, Ghost Threading does not require the whole loop to be parallel and can handle complex loop structures.

# 4 Ghost Threading

Ghost Threading is a software prefetching technique based on helper threading. For this work, we implement ghost threads by hand. In our experience, implementing Ghost Threading on an existing realistic code base requires significantly less effort than manual parallelization because the ghost threads do not modify any application state. To implement them, we identify target loads to be prefetched by profiling, then manually extract the p-slices for these loads. The ghost threads are activated at a time that minimizes the activation frequency and maximizes their life time. To ensure the timeliness of prefetches, the ghost threads run ahead of their main threads at a suitable distance, which is achieved by our novel inter-thread synchronization mechanism. We discuss how these are implemented in the following sections based on an example target loop from `bfs` in `GAP` [4] shown in figure 4.

## 4.1 Identifying the Target Load

To make Ghost Threading profitable, the target loads should meet the following three conditions. First, they should have high enough CPI values to make it worth overlapping their latency with other loads through prefetching. Second, they should be executed in loops with large dynamic sizes (instructions executed per iteration), which allows the ghost threads to increase MLP so it is higher than the original code. If the ghost thread does not have fewer instructions per iteration than the original loop, it may cause a slowdown due to the cost of thread management and synchronization. Last but not least, the target loads need to exist in a loop structure enabling a ghost thread with a long enough execution time to be extracted (section 4.2.2 gives an example). A long-running ghost thread amortizes the cost of activating the thread, and provides the space to run ahead of the main thread.

Like prior software prefetching work [8, 19, 20, 23, 31], we use profiling to obtain the information needed to identify the target loads for prefetching. In this work, we use OptiWISE [11], a profiling tool that generates instruction-level CPI values and loop metrics, to gather the required information.

Based on our experimental results, our heuristic for selecting target loads for prefetching is: (1) the load has a CPI higher than 21; (2) the size of the innermost loop that contains this load is larger than 10 instructions per iteration; (3) the load covers more than 15% of the whole-task execution time, or 80% of its function execution time. If there are multiple loads with CPI higher than 21 in a single loop, the aggregated coverage time should be considered instead.

If a target is identified by the heuristic in a parallelizable loop, we replace the thread for parallelization by our ghost thread. If there are no target loads found in a loop that can be parallelized, we keep the parallel version of the loop.

After profiling `bfs` with OptiWISE, we find the load on line 5 in figure 4(a) matches all three conditions above and is identified as the target load to be prefetched by Ghost Threading.

## 4.2 Constructing the Ghost Thread

A ghost thread comprises two parts: the p-slice and a synchronization segment. A p-slice contains one or more target loads to be prefetched and all the instructions that the target loads depend on (i.e. instructions used to generate the addresses of the loads). The synchronization segment is used to control the speed of the ghost thread by synchronizing it with the main thread, keeping it running at a suitable distance from the main thread. Section 4.3 describes the details of the synchronization mechanism.

*4.2.1 Extracting the P-slice.* The first step in constructing a ghost thread is to generate the p-slice for the corresponding target load(s). We build the p-slice manually by following the guidelines from prior helper-thread approaches [8, 20, 29, 49]. Compiler techniques [19, 23, 24] that automatically extract p-slices are available as well, and could be used in an automated deployment of Ghost Threading. Section 4.4 discusses details of our prototype compiler pass used to automatically extract ghost threads. To extract the p-slice we work

```
0  int TDStep(){
1    int count = 0;                                          P-slice
2    for (int* iter = queue.begin(); iter < queue.end(); iter++){
3      int u = *iter;
4      for (int v : graph.out_neigh(u)){
5        int curr_val = parent[v];          Target load
6        if (curr_val < 0){
7          if (compare_and_swap(parent[v], curr_val, u))
8            count += -curr_val;
9        }
10     }
11   }
12   return count;
13 }
```

**(a) The target load and its p-slice from `TDStep`.**

```
0  void Prefetch(){
1    for (int* iter = queue.begin(); iter < queue.end(); iter++){
2      int u = *iter;
3      for (int v : graph.out_neigh(u))
4        __builtin_prefetch(&parent[v]);
5    }
6  }
```

**(b) The p-slice from `TDStep` extracted into a ghost thread.**

```
0  atomic_int atomic_counter = 0;
1  int TDStep(){
2    int count = 0;
3    int tid = ActivateSmtThread(&Prefetch);
4    for (int* iter = queue.begin(); iter < queue.end(); iter++){
5      int u = *iter;
6      for (int v : graph.out_neigh(u)){
7        /* computations depend on parent[v] */
8        atomic_counter.add(1); // update the counter
9      }
10   }
11   DeactivateSmtThread(tid);
12   return count;
13 }
```

**(c) `TDStep` when Ghost Threading is implemented.**

```
0  void Prefetch(){
1    int local_counter = 0; bool serialize_flag = false;
2    for (int* iter = queue.begin(); iter < queue.end(); iter++){
3     int u = *iter;
4     for (int v : graph.out_neigh(u)){
5       __builtin_prefetch(&parent[v]);
6       local_counter++;
7       if (serialize_flag) do_serialize();
8       if (local_counter % SYNC_FREQ == 0){
9         int main_counter = atomic_counter.read();
10        if (local_counter <= main_counter){
11          serialize_flag = false;
12          SKIP_ITERATIONS(SKIP_STEP);
13        } else if (local_counter >= main_counter + TOO_FAR){
14          serialize_flag = true;
15        } else if (local_counter <= main_counter + CLOSE){
16          serialize_flag = false;
17        }
18      }
19    }                                   Synchronization segment
20   }
21 }
```

**(d) The full ghost thread for `TDStep` with synchronization.**

**Figure 4: Steps taken to extract a ghost thread from function `TDStep` in `bfs`. In (a), profiling shows that the load on line 5 has a high enough CPI and is in a loop (at line 2) with sufficient iterations to be amenable to Ghost Threading. The p-slice is also shown. In (b), the p-slice is extracted and placed in a new function for the ghost thread to execute. In (c), the original loop is updated to activate the ghost thread and track the iteration count to enable synchronization. In (d), the full ghost thread containing all synchronization code.**

backwards through the CFG from the target load, following the data-dependence chain until we reach an instruction that depends on itself (or an instruction we have already selected). If we haven't already, we also select instructions necessary to control the loop. Figure 4(a) shows an example taken from `bfs`. This is placed into a new function for the ghost thread to execute (figure 4(b)).

*4.2.2 Activating the Ghost Thread.* The generated p-slice will be issued as a ghost thread for prefetching when the target loop starts. As a software-only technique, Ghost Threading activates the ghost thread (e.g. spawns or wakes up a thread) by using a system call, which may take thousands of cycles. To amortize this cost, we activate the ghost thread at a time that maximizes its lifetime and minimizes its activation frequency. In our example, figure 4(c), the ghost thread is activated on line 3 just before the loop starts, then deactivated after the loop on line 11.

*4.2.3 Adding the Synchronization Segment.* In order to prefetch data that are useful to the main thread, the ghost thread should run ahead at a suitable distance from the main thread. To ensure the two threads do not get too far apart (or too close together), a counter records the main thread's iteration number, which is read in the ghost thread's synchronization segment. This is shown in figure 4(d). More details of how this synchronization mechanism is implemented are discussed in section 4.3.

## 4.3 Inter-thread Synchronization

A key novelty in Ghost Threading is the ability to timely and efficiently synchronize a ghost thread with its main thread so that it runs at a suitable distance ahead. Without this, prefetching may fail to yield speedups over the single-threaded implementation, or even cause a slowdown, due to running too far into the future, prefetching untimely data into the cache and wasting shared hardware resources.

A ghost thread executes a distilled version of a loop, containing only the instructions that the target loads depend on, which makes the ghost thread naturally run faster than the main thread and enjoy more MLP (illustrated by figure 2). But if the ghost thread keeps running ahead without limitation, the data it brings into the L1 cache are highly likely to be evicted before being accessed by the main thread, which means that all hardware resources, such as functional units, assigned to the ghost thread are wasted. Since we issue the ghost thread to the SMT context of the same core, all hardware resources wasted by unnecessary prefetching could otherwise have been utilized by the main thread. Additionally, data prefetched too early causes cache pollution, further harming the performance of the main thread. As a result, controlling the speed of prefetching is key to Ghost Threading to avoid this wastage.

Another less common scenario is when a ghost thread runs too slowly or even lags behind the main thread, resulting in unprofitable

prefetches. We observe two common situations where this occurs. First, since the main thread benefits from prefetches performed by the ghost thread, it executes much more efficiently overall. In this case, the main thread may catch up with the ghost thread whilst the latter is waiting for long-latency memory accesses. Second, the cost of spawning a thread causes an initial gap between the main and ghost thread. When the ghost thread is created, the main thread may have already processed many loop iterations. We experimented with skipping iterations when the ghost thread is spawned, but we found in practice that this does not make a significant difference to overall performance.

To summarize, in order to make it worth stealing hardware resources from the main thread of an application to execute the ghost thread, we need an inter-thread synchronization mechanism to control the speed of the ghost thread, preventing it from running too far ahead, too close, or even behind the main thread. It is straightforward to accelerate the ghost thread when it is behind the main thread by simply skipping some loop iterations. The challenge is therefore instead to prevent the ghost thread from running too far ahead without incurring vast overheads.

*4.3.1 Decelerating the Ghost Thread.* Prior work utilizing SMT contexts to issue helper threads [8, 19, 23] has applied a pause-restart scheme to synchronize the two threads: the helper thread is paused/-suspended when it goes too far ahead and restarted when the main thread catches up. Although this pause-restart mechanism is an ideal solution to synchronize the speed of the two threads, because it allows the main thread to utilize all spare hardware resources, unfortunately it is hard to develop a cheap thread-pausing mechanism in real systems. Kim et al. [23] proposed an experimental hardware synchronization mechanism, available on real Intel processors, which allows one thread to suspend or wake up another by executing a single instruction. However, this still takes around 1,500 cycles to suspend a thread, which is too long to achieve efficient inter-thread synchronization.

Therefore, instead of pausing the helper thread when it is too fast, we *slow it down*. The challenge of this approach is to slow the helper thread down without consuming too many hardware resources. To handle this, we use the `serialize` instruction provided by recent Intel processors, which prevents the CPU from fetching and executing any further instructions until all modifications to flags, registers, and memory by instructions before the `serialize` instruction are complete [15]. Unlike a typical memory fence that only stops executing instructions after the fence, the `serialize` instruction also stops fetching further instructions, saving frontend pipeline resources. Due to this feature, this instruction is almost an ideal approach to drop the speed of the helper thread, though it may not be designed for this purpose. With the `serialize` instructions inserted, the number of instructions from our ghost thread that can exist simultaneously in the ROB is equal to the number of instructions between two inserted `serialize` instructions in theory. By controlling the number of instructions between two `serialize` instructions, we indirectly control the amount of hardware resources assigned to the ghost thread.

Figures 4(c) and 4(d) show how we utilize the `serialize` instruction to control the speed of the ghost thread. The key idea is to periodically update a counter in the main thread to record the loop iteration number and let the ghost thread read it. By comparing the loop iteration numbers of the main and ghost threads, the mechanism determines whether to (1) execute a `serialize` instruction, or (2) skip some iterations in the ghost thread to speed it up. In this example, only the instructions from a single loop iteration of the ghost thread can exist in the ROB at the same time when `serialize` instructions are executed, which minimizes resource contention from the ghost thread on the main thread.

```
0  #pragma ghost_threading sync_frequency(14) skip_iter(10)
         serial_max_threshold(65) serial_min_threshold(63) sync_level(2)
1  for (int i = 0; i < NUM_OF_ITER/128; i++){
2    // may not be a constant iteration count
3    for (int j = 0; j < 128; j++){
4      __builtin_prefetch(&array[i][index[j]]);
5      sum += hash(hash(array[i][index[j]]));
6    }
7  }
```

**Figure 5: Camel loop nest from Figure 1(c), with the compiler directive for Ghost Threading (line 0) and annotated target load (line 4).**

Compared with prior pause-restart schemes that may take thousands of cycles to suspend/wake the helper thread, our novel inter-thread synchronization approach based on the `serialize` instruction enables a timely deceleration of the ghost thread after detecting it executing too far ahead. For Ghost Threading, the delay between recognizing that the ghost thread is running too far ahead and actually slowing it down is a handful of cycles. Similarly, resuming full-speed execution is trivial.

*4.3.2 Determining the Inter-Thread Distance.* The distance between the main thread and ghost thread is controlled by hyper-parameters for the synchronization, such as the synchronization frequency, how many iterations to skip if the ghost thread is behind the main thread, and when to execute `serialize` instructions. In order to ensure prefetching timeliness, the inter-thread distance should be neither too small nor too large.

The value of a suitable distance strongly depends on dynamic program characteristics and processor states, including (but not limited to) the number of instructions in the current loop iteration, the cache miss ratio of the target load, the number of entries in the ROB and load/store units, and the current cache occupancy. We believe that predicting such a suitable inter-thread distance accurately would require a model of the processor pipeline and cache hierarchy. This would likely require a large amount of reverse engineering and is beyond the scope of this work. Therefore, in this work we manually tune the hyper-parameters for synchronization by profiling, just like in prior work [19, 20, 23].

## 4.4 Automatic Ghost Thread Extraction

While we have extracted all ghost threads manually, automatic extraction provides an easier route to adoption, so we have developed a prototype pass within LLVM [26] to achieve this [1]. The work flow proceeds in a similar manner as described in this section. Target loads are identified as previously described via profiling (section 4.1). The programmer then annotates the target loads (via intrinsics)

[1] https://github.com/CompArchCam/GhostThreadingCompiler.git

and the loop that prefetching should be applied to via `#pragma` directives. The pragma is appended with clauses that specify the hyper-parameters for the compiler. An example of such a `#pragma` is shown in Figure 5, which illustrates the automatic ghost thread extraction of the loop nest in Figure 1(c). During compilation, a middle-end pass uses these directives as a starting point to identify the p-slice of IR instructions and extract them (section 4.2.1). The loop's control-flow structure is duplicated into a new function, the p-slice added, and then synchronization code inserted (section 4.3). Finally, the main loop's IR is altered to include a shared counter of the iteration number and activation of the ghost thread beforehand (section 4.2.2).

## 5 Methodology

We evaluate Ghost Threading on an Intel Core i7-12700 processor [14] (Ubuntu 24.04, performance-core base frequency 2.10GHz, efficient-core base frequency 1.60GHz, 94GB memory, 1/12/25 MiB L1/L2/L3 cache), which is composed of eight performance cores and four efficiency cores. We use the performance cores, which support Hyper-Threading, for our evaluation.

We evaluate 11 benchmarks from the domains of graph analysis, databases, and HPC, which are used by prior prefetching techniques [3, 31, 34, 36]. Six of them are from the GAP benchmark suite (`GAP`) [4]: Betweenness Centrality (`bc`), Breadth-First Search (`bfs`), Connected Components (`cc`), PageRank (`pr`), Single-Source Shortest Path (`sssp`), and Triangle Counting (`tc`). `GAP` has five built-in graphs available for each kernel: two synthetic graphs (`urand` and `kron`) and three real-world graphs (`twitter`, `road`, and `web`). Another five benchmarks are selected from HPC and database domains: `camel`, `kangaroo`, hashjoin with two and eight hashes (`hj2` and `hj8`), and `NAS-IS`. These benchmarks are compiled by either `gcc/g++` or `clang`, depending on their default build scripts, with `-O3` optimization enabled for all techniques to be evaluated. When automatically extracting ghost threads, the benchmarks are compiled using `clang/LLVM` with our implemented pass (part of the `-O3` pipeline). We run all benchmarks to completion and record the execution time of all functions except the ones for input generation and initialization. All experiments are repeated three times.

In order to select the target load to be prefetched, we profile the benchmarks using OptiWISE [11] with a reduced input dataset compared with evaluation. The cost mainly comes from two executions (one for sampling and another for instrumentation) of the application and was 3.6× geometric mean slowdown over the base run with the profiling input dataset. Table 1 summarizes the data inputs used for profiling and evaluation.

| Workload | Input for Evaluation | Input for Profiling |
|---|---|---|
| GAP | `kron` -g27 -k16 | `kron` -g26 -k15 |
| | `twitter` | `kron` -g26 -k24 (similar number of nodes and edges) |
| | `urand` -u27 -k16 | `urand` -u26 -k15 |
| | `road`: all roads in the USA | All roads in the central USA |
| | `web`: web-crawl of the .sk domain | Web-crawl of the .it domain |
| `camel` | 128MB synthetic input | 32MB synthetic input |
| `kangaroo` | 128MB synthetic input | 32MB synthetic input |
| `NAS-IS` | Input class 'B' | Input class 'W' |
| `hj2` | R=S=96MB | R=S=24MB |
| `hj8` | R=S=96MB | R=S=24MB |

**Table 1: Input datasets for profiling and evaluation.**

## 6 Evaluation

We evaluate the following four techniques in this work:

- Baseline: If not otherwise mentioned, the baseline is the single-threaded version of the original application.
- Software Prefetching (SWPF) [3]: The state-of-the-art software prefetching technique, which inserts prefetching instructions for the target indirect loads. We use the manually optimized version of SWPF in our evaluation.
- SMT Parallelization via OpenMP (SMT OpenMP): If not otherwise mentioned, SMT OpenMP is the OpenMP-parallelized version of the original application running on an SMT core. Two parallel threads are issued to the same physical core through SMT. To make a fair comparison with Ghost Threading, which does not require the code to be parallelizable, this option is ignored for workloads that require code rewriting for parallelization. `NAS-IS` and `kangaroo` cannot be parallelized without rewriting the code and are therefore excluded. The hash join workloads (`hj2` and `hj8`) have partially parallelized versions that do not require code rewriting and a fully parallelized version that does. We include the former in our results.
- Ghost Threading: The technique presented in this work. We profile the evaluated benchmarks as discussed in section 5, and extract the ghost thread by hand as discussed in section 4. The ghost thread is issued together with the main application thread on the same physical SMT core. If a loop can be parallelized and contains target loads selected by our heuristic, we replace the OpenMP thread with our ghost thread. If a loop does not contain target loads for Ghost Threading, we keep issuing its OpenMP thread (if applicable).
- Compiler Extracted Ghost Threads: The compiler driven, automated version of Ghost Threading. The compiler extracts ghost threads from `#pragma` annotated loops comprising target loads as discussed in section 4.4. Extracted ghost threads are issued using the same methodology as in the manual technique. Similar to Ghost Threading, we use our heuristic to decide if a loop will be annotated or not. If a parallelizable loop does not include target loads, we keep the default OpenMP pragmas (if applicable).

### 6.1 Single-core Performance

Figure 6 shows the speedup of SWPF, SMT OpenMP, Ghost Threading, and Compiler Extracted Ghost Threads over the baseline on an idle server across all 34 evaluated workloads. Here, we consider the same kernel from `GAP` with different graphs as individual workloads due to their varying behaviors. We make four key observations.

First, Ghost Threading provides 1.33× geometric mean speedup over the baseline, considerably out-performing the state-of-the-art conventional software prefetching technique (SWPF, 1.06×) and parallelization (SMT OpenMP, 1.22×). In workloads such as `camel`, `kangaroo` and `NAS-IS`, SWPF outperforms Ghost Threading (2.44× vs 2.32× for `camel`, 1.86× vs 1.50× for `kangaroo`, and 1.23× vs 1.00×

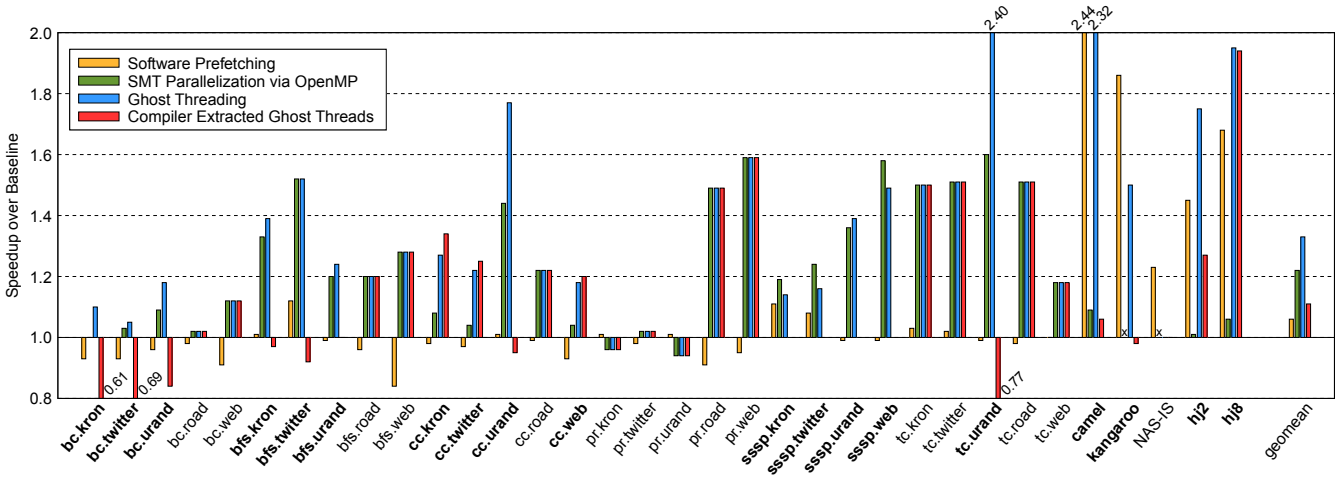



**Figure 6: Speedup of SWPF, SMT OpenMP, Ghost Threading, and Compiler Extracted Ghost Threads over the baseline on single physical core on an idle server. The bold x-label means that the corresponding workload uses ghost thread(s) to replace the OpenMP thread(s) for Ghost Threading as our heuristic predicts the former is profitable. The 'x' tick means that the corresponding number is unavailable.**

for `NAS-IS`[2]. Whereas, negligible speedup (or even slowdown) is observed for workloads from GAP, including the ones known to be memory intensive. SMT OpenMP gives good overall performance boosts but may not be the optimal solution to utilize SMT when the workloads are memory-intensive. The 1.22× geometric mean speedup provided by SMT OpenMP is raised to 1.33× by replacing the OpenMP threads with ghost threads when the latter is predicted to be profitable.

Second, our heuristic makes a good selection between the OpenMP threads (if applicable) and our ghost threads. Our heuristic identifies 19 out of 34 workloads to be suitable for Ghost Threading, and 16 of them obtain more (or equal) speedup from our ghost threads compared with SMT OpenMP ones. Although three workloads benefit less from Ghost Threading than SMT OpenMP, the speedup from Ghost Threading is still comparable to SMT OpenMP (1.14× vs 1.19× for `sssp.kron`, 1.16× vs 1.24× for `sssp.twitter`, 1.49× vs 1.58× for `sssp.web`).

Third, three workloads have no speedup or even slowdown when applying Ghost Threading: `pr.kron`, `pr.urand`, and `NAS-IS`. Our heuristic does not identify target loads from these workloads, so Ghost Threading on them is equivalent to SMT OpenMP, which causes slowdown on the `pr` workloads and is unavailable for `NAS-IS` without code rewriting. Therefore, `pr.kron` and `pr.urand` are slowed down by Ghost Threading, while `NAS-IS` obtains no benefit at all.

Finally, Compiler Extracted Ghost Threads achieve a 1.11× geometric speedup over the baseline, although being 22% slower than Ghost Threading. For several workloads, Compiler Extracted Ghost Threads follows the same performance trends as Manual Ghost Threading. However, significant slowdown is observed when using compiler ghost threads in the following workloads: `bc.kron`, `bc.twitter`, `bc.urand`, `bfs.kron`, `bfs.twitter`, `bfs.urand`, `cc.urand`, `tc.urand`, `camel`, `hj2` and `kangaroo`. This discrepancy stems from difficult-to-remove, unnecessary control flow in the extracted ghost-threading loop, resulting in the inclusion of several irrelevant instructions, adding to the execution time of the workloads. Such instructions (arising from the use of C++ STL constructs) lead to runtime memory issues in `sssp`, including segmentation faults. In three workloads—`cc.kron`, `cc.twitter` and `cc.web`—Compiler Extracted Ghost Threads achieves an average performance improvement of 4% over manual Ghost Threading.

Based on the above observations, it can be seen that Ghost Threading, when manually applied in particular, is a competitive replacement for parallelization when using SMT, since extracting a ghost thread from a loop requires much less effort than creating a parallel version of it.

## 6.2 Energy Consumption

To estimate the energy consumption of the evaluated techniques, we use Intel RAPL [22] with perf [27] as the interface. The server used for our evaluation does not support the measurement of DRAM energy consumption with RAPL, so we only measure the package-level energy consumption.

Figure 7 shows the measurement of package energy savings on the idle server. The geometric mean package energy savings of SWPF, SMT OpenMP, Ghost Threading, and Compiler Extracted Ghost Threads over the baseline are 6%, 12%, 16%, and 4% respectively. A strong correlation between the execution time and energy consumption can be observed: the more speedup a workload obtains from SWPF/SMT OpenMP/Ghost Threading, the less energy it tends to consume. Although these techniques tend to increase power consumption by executing additional prefetching instructions or spawning extra threads, the energy saved through performance improvements still outweighs the higher power usage.

[2] `camel` and `kangaroo` are synthetic workloads created by Ainsworth and Jones [3] to evaluate their technique.

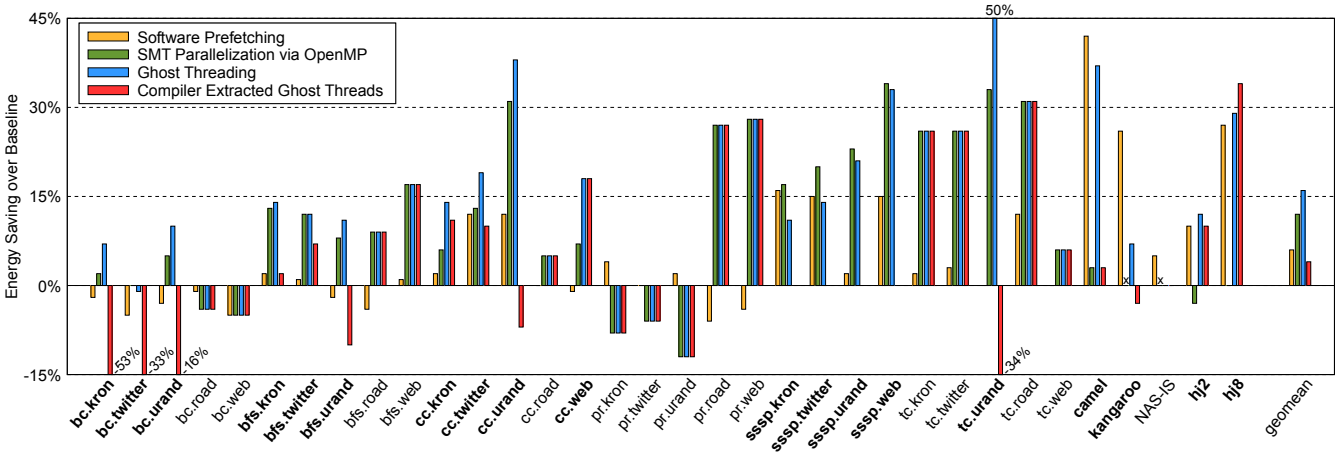



**Figure 7: Energy saving of SWPF, SMT OpenMP, Ghost Threading, and Compiler Extracted Ghost Threads over the baseline on the idle server. The bold x-label means that the corresponding workload uses ghost thread(s) to replace the OpenMP thread(s) for Ghost Threading based on our heuristic and profiling results. The 'x' tick means that the corresponding number is unavailable.**

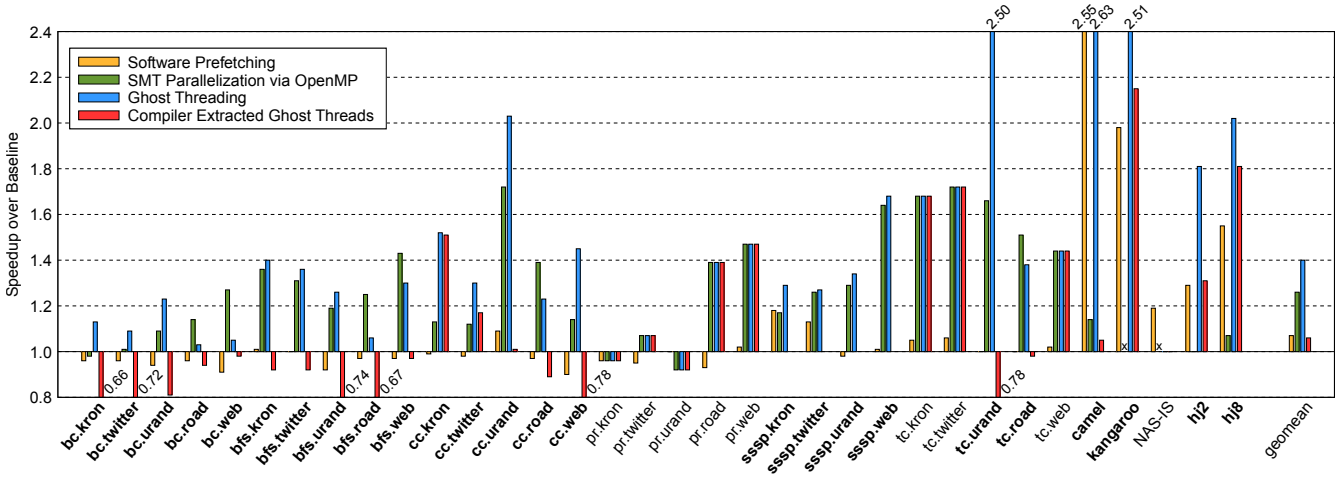



**Figure 8: Speedup of SWPF, SMT OpenMP, Ghost Threading, and Compiler Extracted Ghost Threads over the baseline on a busy server, where 21GB/s memory bandwidth pressure is added. The bold x-label means that the corresponding workload uses ghost thread(s) to replace the OpenMP thread(s) for Ghost Threading based on our heuristic and profiling results. The 'x' tick means that the corresponding number is unavailable.**

## 6.3 Single-core Performance in the Busy Server

Besides an idle server, we also evaluate the performance of Ghost Threading on a busy server. We use the Intel RDT Software Package [16] to generate synthetic memory bandwidth pressure on the server during our experiments. To simulate a busy state, we use seven performance cores on the server to generate memory bandwidth pressure, each with 3GB/s (totaling 21GB/s), and the remaining one performance core to run the evaluated benchmarks.

Figure 8 reports the performance of SWPF, SMT OpenMP, Ghost Threading, and Compiler Extracted Ghost Threads over the baseline on the busy server. The geometric mean speedups are 1.07×, 1.26×, 1.40×, and 1.06× respectively. The three observations we made from the results on the idle server also apply to the busy server, but there are two differences. First, Ghost Threading generates higher speedup in the busy server. On the one hand, 19 workloads enjoy higher performance improvements from Ghost Threading than SMT OpenMP (three more than the idle server). On the other hand, `camel` and `kangaroo`, which benefit less from Ghost Threading than SWPF, obtain higher speedup from Ghost Threading than SWPF on the busy server. Second, more workloads are selected by the heuristic to issue ghost threads in the busy server. Out of all 34 workloads, 25 workloads spawn ghost threads to replace OpenMP threads (six more than the idle server).

Compared with the idle server, the busy server provides less available cache space and memory bandwidth for a single processor core, increasing the CPI and coverage time of loads in the benchmarks to be evaluated. As a result, Ghost Threading becomes

suitable for more workloads and generates higher performance boosts on a busy server.

Compiler Extracted Ghost Threads do not follow the same performance trends as Ghost Threading on the busy server. Several workloads that benefit from Ghost Threading show significant slowdown. The reason for this discrepancy is as outlined in section 6.1. Improvements addressing the limitations and shortcomings of Compiler Extracted Ghost Threads are left for future work.

## 6.4 Multi-core Performance

From the discussion so far, it can be inferred that Ghost Threading is a viable alternative to parallelization when using SMT to boost the performance on a single core. In multi-core scenarios, Ghost Threading can be used together with parallelization. To illustrate the scalability of Ghost Threading when multiple cores are used, we evaluate GAP [4] with the parallelized versions of baseline, SWPF, SMT OpenMP, and Ghost Threading. In this case, these techniques are redefined as follows:

- Baseline: The OpenMP-parallelized version of the original application. The number of threads used for parallelization matches the number of physical cores so that each core executes a single thread (i.e. no SMT).
- SWPF: The OpenMP-parallelized version of the original SWPF. The number of threads used equals the number of physical cores (i.e. no SMT).
- SMT OpenMP: The OpenMP-parallelized version of the original application. SMT is enabled in this case with each physical core executing two threads.
- Ghost Threading: The OpenMP-parallelized version of the original implementation (ghost threads are extracted manually). Each main application thread is executed with a ghost thread on the same physical core via SMT. The number of main/ghost threads is equal to the number of physical cores used. The method used to determine if a ghost thread will replace an OpenMP thread is discussed below.

When Ghost Threading is deployed with multiple cores, the heuristic proposed in section 4.1 fails to make good decisions on selecting target loads, as it does not consider the shorter length of loops in the ghost threads. As discussed in section 4.1, a short-running ghost thread badly amortizes the thread spawning cost and has little space to run ahead of the main thread. As a result, the heuristic selects workloads that fail to provide performance boosts from Ghost Threading when they are parallelized. To avoid this, the decision to use Ghost Threading in multi-core scenarios is based on the profiling results of the workloads on training inputs. We run SMT OpenMP and Ghost Threading with the training input graphs described in table 1 and compare their execution time. If Ghost Threading is faster than SMT OpenMP when a specific number of cores are used, we use ghost threads instead of OpenMP threads in SMT contexts.

Figure 9 shows the performance scaling results for SMT OpenMP and Ghost Threading with increasing core count. We make three observations. First, Ghost Threading out-performs SMT OpenMP but needs a more careful selection of targets. Second, the performance improvements generated by SMT OpenMP and Ghost Threading drop when more than one core is used. This is due to the increased

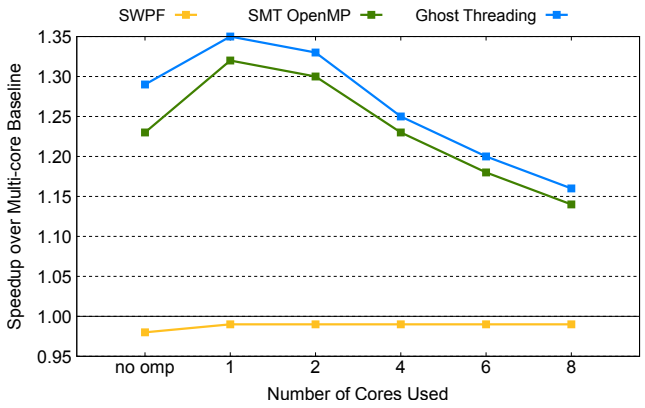


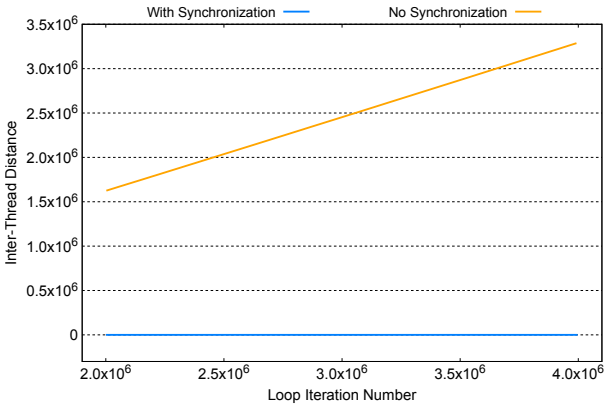

**Figure 9: Number of cores (and threads) used for baselines verses the geometric mean speedup of SWPF, SMT OpenMP, and Ghost Threading over multi-threading baselines. Both the 'no omp' and '1' columns in this figure use a single-threaded baseline and Ghost Threading, but they are not exactly the same: 'no omp' indicates that no OpenMP pragmas are inserted for either the baseline or Ghost Threading, whereas '1' signifies that OpenMP pragmas are enabled for them, with only one OpenMP thread used.**

competition between cores when accessing shared resources on the server, resources such as memory bandwidth and shared cache. Third, the initialization cost of OpenMP adds to the execution time of both the baseline and Ghost Threading (when ghost threads are spawned) but does not affect SMT OpenMP, wherein OpenMP is always enabled to issue SMT thread(s). This added cost limits the speedup obtained when using GhostThreading to 1.35× (from 'no omp' to '1' in figure 9). Whereas, for SMT OpenMP, we observe a performance uplift of 9% from 1.23× to 1.32×.

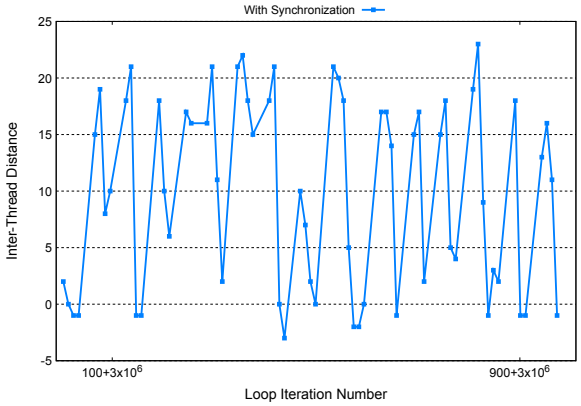
## 6.5 Effectiveness of Inter-thread Synchronization

One key contribution of this work is the novel inter-thread synchronization mechanism. To illustrate the importance of synchronization to Ghost Threading, we use `cc` (with `urand` as the input) from GAP as a case study by sampling the distance between the main thread and ghost thread when Ghost Threading is applied with and without the synchronization mechanism.

The key loop comprising target loads in this case appears in the function `Afforest`. Figure 10(a) shows the distance between the two threads with and without synchronization applied during a representative period (two million iterations) in this loop. Without synchronization, the inter-thread distance continually increases over time, failing to generate useful prefetches. But when synchronization is applied to control the speed of the ghost thread, inter-thread distance fluctuates within a specific range controlled by the synchronization hyper-parameters: when the ghost thread goes too far ahead of the main thread, it is slowed down by the `serialize` instruction; when the ghost thread is too close to (or even behind) the main thread, several loop iterations will be skipped to accelerate it. To illustrate this behavior clearly, figure 10(b) shows a shorter period (1000 iterations) of the target loop.




**(a) The inter-thread distance with and without the synchronization in two million loop iterations.**




**(b) The inter-thread distance when the synchronization is applied in a short (1000 iterations) but representative period.**

**Figure 10: Distance between the ghost thread and the main one of `cc.urand`. The inter-thread distance is measured by subtracting the main thread loop iteration number from that of the Ghost Threading (i.e. a positive distance means that the ghost thread is faster than the main thread).**

# 7 Related Work

## 7.1 Helper Threading

Speculative Precomputation [8] issues a prefetching thread on idle thread contexts of SMT CPUs and develops hardware structures used to prevent the helper threads from running too far ahead of the main thread. Jung et al. [19] present a helper threading scheme designed for loosely-coupled processors like CMP and a compiler algorithm that automatically extracts helper threads.

The Slipstream processor [41, 42] dynamically creates the helper thread, which is executed on another core to boost the main thread speed. The helper thread accelerates the main one by both branch prediction and prefetching. Although achieving promising speedup by issuing the helper thread, these approaches are all based on simulation and are unavailable on real systems. In contrast, Ghost Threading has shown promising results on real machines.

To evaluate the potential of helper threading on real machines, Kim et al. [23] also proposed a helper threading technique via Intel Hyper-Threading. To control of the speed of helper threads, they developed a hardware synchronization mechanism tested on real silicon, which takes around 1,500 cycles to suspend a thread via a single instruction. Although achieving a one-order-of-magnitude reduction compared to OS approaches, this hardware synchronization mechanism is unavailable on commercialized chips and still not fast enough to give timely control of the helper threads' speed, failing to generate significant performance improvements.

Kamruzzaman et al. [20] develop an inter-core prefetching approach that runs the main thread and prefetching thread on different cores of real-world multicore processors. The demand for explicit inter-thread synchronization is removed by switching the core used to execute the main and prefetching threads at a fixed period determined by the programmer. Every time the main thread migrates to a new core, the data it will use are ready in private caches due to the prefetching thread executed on this core before. Unlike Ghost Threading that targets improvements to single-core performance, the inter-core approach utilizes a multi-core system (up to four cores are needed for maximum performance).

## 7.2 Runahead and Other Precomputation Techniques

Runahead [7, 12, 13, 32–36, 38, 39] is an execution-based prefetching technique that generates concurrent cache misses by speculatively continuing execution when encountering a full-window stall caused by cache misses. The key idea is to enter a runahead mode, where the processor checkpoints the original architectural state and speculatively fetches and executes future instructions when the pipeline stalls. During runahead mode, no executed instructions update the architectural state but exist solely for prefetching in case of cache misses. The key difference between helper threading and runahead is that runahead executes the instructions for prefetching in a speculative execution mode instead of an idle thread context, enjoying more flexibility of starting/ending prefetching based on the dynamic program behavior. As a trade-off, the pipeline requires new hardware structures, and existing structures need to be modified.

Mutlu et al. [33] define several rules to determine the timing for entering and exiting runahead mode. Hashemi et al. [13] dynamically identify address generation chains for loads and only execute these chains in runahead mode, generating more concurrent cache misses in the same time interval. Continuous Runahead [12] applies an additional execution engine in the memory controller to execute the load chains identified dynamically. Precise Runahead [35] avoids the demand for pipeline state checkpoint/restore when entering/exiting runahead mode by utilizing the empty issue queues and registers when there is a full-window stall.

Decoupled Vector Runahead (DVR) [36] takes an approach more similar to helper threading: it offloads the runahead execution to a subthread created when indirect memory access patterns are detected. Compared to Ghost Threading, DVR requires hardware support for large vectors and additional logic to execute a runahead thread continuously.

Load Slice Core [5] extends the in-order, stall-on-use core with a second in-order pipeline, which executes memory accesses and address generation instructions in advance to increase MLP, boosting

performance and energy efficiency. SWOOP [44] develops a compiler and architecture co-design that transforms the body of critical loops into memory-bound access phases and compute-bound execute phases. SWOOP dynamically jumps to access phases in the future to bypass the stalling execute phase, hiding long latencies of memory accesses. Both schemes require changes to the underlying hardware, whereas Ghost Threading works on today's systems without hardware changes.

Decoupled access-execute schemes, like Clairvoyance [43], hoist loads to the beginning of a loop (the access phase), leaving the remaining computation towards the end (the execute phase). While this helps resolve loads early, these approaches are limited by register pressure or a lack of enough independent computation in the execute phase to hide the latency of the cache misses that occur in the access phase. In comparison, Ghost Threading decouples the prefetching code from the original code by executing them in different (and independent) threads, allowing the ghost threads to prefetch deeper into the future.

SPE [31] is an execution-based prefetching technique targeting real-world HBM systems. It extracts prefetching code in a similar way to Ghost Threading, but executes extracted prefetching code just before the loop containing the target load in the same thread. By strip-mining the main loop and creating the prefetching loop from the sub-loop, SPE avoids prefetching data from the far future and achieves an automatic tight-lock synchronization.

## 8 Discussion: Limitations of Software-only Techniques

As a software-only helper threading technique, Ghost Threading has achieved significant improvements in performance and energy efficiency. However, a key limitation of Ghost Threading (and other software-only techniques) is the restricted access to low-level processor states at run time (e.g. measuring the waiting time of a load and detecting an LLC miss or a full-window stall) without hardware support. Therefore, software approaches like Ghost Threading require manual tuning of inter-thread synchronization parameters and are unable to dynamically start or terminate helper threads when it is beneficial. Without the ability to start or terminate helper threads flexibly, software-only techniques rely on profiling to judiciously select targets for prefetching and fail to respond to dynamic program behaviors. On the one hand, they miss prefetching opportunities on loads that have low average cache-miss ratios but may still cause full-window stalls. On the other hand, they waste hardware resources on loads that have high average cache-miss ratios but may consistently hit the L1 cache in specific time periods. However, with a novel synchronization mechanism, like that developed for Ghost Threading, their main benefit is that they can be deployed immediately in real systems and provide significant speedups over parallelization alone.

## 9 Conclusion

We propose Ghost Threading, a software-only helper threading technique that issues helper threads for prefetching on idle SMT contexts. To guarantee the timeliness of prefetching, we develop a novel inter-thread synchronization scheme, which slows down the helper thread with modest hardware resource consumption. By exploiting higher MLP than conventional software prefetching and parallelization techniques, Ghost Threading provides promising improvements over these techniques. On an idle server, Ghost Threading reports 1.25× and 1.11× geometric mean speedup; and 11% and 5% geometric mean package energy saving over state-of-the-art software prefetching and parallelization techniques, respectively. On a busy server, it provides 1.31× and 1.13× geometric mean speedup over state-of-the-art software prefetching and parallelization techniques, respectively.

## Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), grant EP/W00576X/1, and Arm. Additional data related to this publication is available in the repository at https://doi.org/10.5281/zenodo.16732636.

## A Artifact Appendix

### A.1 Abstract

Our artifact provides the Ghost Threading source code and scripts necessary to reproduce the key results presented in the paper, including the single-core performance and energy efficiency of the baseline, a state-of-the-art software prefetching technique, a parallelization technique, and Ghost Threading (figure 3 and the first three bars of figures 6 to 8).

Our code targets x86-64 Intel machines that support Hyper Threading [30] and the `serialize` instruction [15]. The `serialize` instruction is available on 12th-generation (and newer) Intel Core processors and 4th-generation (and newer) Intel Xeon Scalable Processor families. We use an Intel Core i7-12700 processor [14] for our evaluation and strongly recommend Intel Core processor families to reproduce similar results.

The artifact is designed for Linux systems (we use Ubuntu 24.04). Reproducing the experiments in figures 7 and 8 requires perf [27] and Intel RDT Software Package [16], respectively. Compiling the source code requires GCC/G++ and CLANG. Python3 and gnuplot are used to analyze and plot the results.

### A.2 Artifact Check-list (Meta-information)

- **Algorithm:** Prefetching based on helper threading.
- **Program:** Source code of the baseline, software prefetching version, OpenMP parallelized version, and Ghost Threading version are provided for all benchmarks evaluated (GAP [4] and HPC workloads used by prior works [3, 31, 34, 36]).
- **Compilation:** GCC/G++ 13.0 or above and CLANG 17.0 or above. At least 94GB of memory is needed to build the input graphs of the benchmarks.
- **Binary:** Binaries are not included but can be compiled by our scripts.
- **Run-time environment:** Our code is for Linux (we use Ubuntu 24.04). Software dependencies include perf (whose version is related to the Linux kernel), Intel RDT Software Package, Python3, and gnuplot. Root access is not mandatory but highly recommended.
- **Hardware:** Our code only supports x86-64 Intel machines that support Hyper Threading [30] and the `serialize` instruction [15]. This needs 12th-generation (and newer) Intel Core processors or 4th-generation (and newer) Intel Xeon Scalable Processor families. At least 94GB of memory is needed to build the input graphs of the benchmarks. We strongly recommend Intel Core processor families.

- **Run-time state:** Ghost Threading is a helper threading technique available on real systems, so its performance is highly sensitive to the run-time system state. We strongly recommend minimizing background activity and avoiding other processes when reproducing our results.
- **Execution:** We strongly recommend minimizing background activity and avoiding other processes when reproducing our results.
- **Metrics:** The performance (measured by execution time) and energy efficiency of Ghost Threading.
- **Output:** The artifact aims to reproduce the result of figure 3 and the first three bars in figures 6 to 8. The CSV and PDF files will be generated for the corresponding figures.
- **Experiments:** Scripts are provided to run experiments.
- **How much disk space required (approximately)?:** 290GB is required after compiling our code (284GB for graphs of GAP).
- **How much time is needed to prepare workflow (approximately)?:** Less than 10 minutes to download dependent software and compile the source code. Around one hour to download and generate the graphs needed by GAP (this depends on the network state as well).
- **How much time is needed to complete experiments (approximately)?:** It takes around 85 hours on our system.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT license.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.16732636

## A.3 Description

*A.3.1 How to access.* The artifact DOI points to a zipped archive containing scripts for running experiments and source code of our tool. We also link the GitHub page of our artifact: https://github.com/CompArchCam/ghost-threading-eval-micro2025.git.

*A.3.2 Hardware dependencies.* Ghost Threading targets on x86-64 Intel machines that support Hyper Threading [30] and the `serialize` instruction [15]. This needs 12th-generation (and newer) Intel Core processors or 4th-generation (and newer) Intel Xeon Scalable Processor families. At least 94GB of memory is needed to build the input graphs of the benchmarks. We use an Intel Core i7-12700 processor [14] with 94GB memory for our evaluation and strongly recommend Intel Core processor families.

*A.3.3 Software dependencies.* Our code is for Linux systems (we recommend Ubuntu 22.04 or 24.04). GCC/G++ 13.0 or above and CLANG 17.0 or above are required for compilation. Software dependencies include perf (whose version is related to the Linux kernel), Intel RDT Software Package, Python3, and gnuplot. Root access is not mandatory but highly recommended.

*A.3.4 Data sets.* Several graphs, which are generated by our scripts, are needed to evaluate the GAP benchmark [4].

*A.3.5 Models.* No specific models are required.

## A.4 Installation

Install the software dependencies of the artifact on Ubuntu:

```
$ sudo apt install gcc g++ clang python3 gnuplot
```

We use perf to measure the energy consumption of our approaches for figure 7. It is a part of the Linux kernel and generally does not need installation. If perf does not exist on the system, it should be available in the `linux-tools-generic` package on Ubuntu:

```
$ sudo apt install linux-tools-generic
```

We use Intel RDT Software Package to generate memory bandwidth pressure to simulate a busy server for figure 8. To install:

```
$ git clone
↪   https://github.com/intel/intel-cmt-cat.git
$ cd intel-cmt-cat
# installation instructions from INSTALL file
$ make
$ sudo make install
# test installation
$ membw # this shows the usage instructions
```

All binaries required to reproduce our results will be compiled by the scripts provided once the software dependencies have been installed. Download and extract the artifact or clone the GitHub repository, then from within the directory of the artifact run:

```
$ ./build.sh
```

## A.5 Experiment Workflow

Our experiments are performed by running the binaries of the baseline, state-of-the-art software prefetching and parallelization technique, and Ghost Threading of various benchmarks and comparing their performance and energy efficiency. We provide scripts to reproduce our experimental results automatically, but the experiments can also be performed manually by running binaries with specific arguments.

## A.6 Evaluation and Expected Results

Our artifact aims to reproduce four key experiments: motivational comparison of techniques evaluated in this work (figure 3), single-core performance of Ghost Threading on an idle server (the first three bars of figure 6), single-core energy efficiency of Ghost Threading on an idle server (the first three bars of figure 7), and single-core performance of Ghost Threading on a busy server (the first three bars of figure 8).

There are three sub-folders for each experiment in our artifact: `./figure3`, `./figure6`, `./figure7`, and `./figure8`. Each folder includes a `README.md` file that describes how to run the experiments in detail.

We assume that the artifact evaluation will be conducted on systems similar to ours, so that the same (at least similar) workloads will be selected by our heuristic as the target of Ghost Threading. Therefore, our source code scripts use the same target loads and workloads selected in our system to generate results by default. If necessary, the users could manually select target loads and workloads themselves by profiling with the scripts in `./profile` folder.

The inter-thread synchronization hyper-parameters also vary with different system configurations. Again, our artifact uses the parameters tuned on our system by default. If needed, the users could tune the parameters based on their system via our scripts or source code.

As discussed above, our experiment configurations depend on system configurations: the target loads and target workloads of prefetching, and synchronization hyper-parameters could change. But our default source code and scripts should provide similar results as long as the system does not change significantly.

Since the behavior of the same binary could be quite different among different systems, it is hard for us to give an exact number of variations in the result. Meanwhile, workloads selected by our heuristic 4.1 could be different as well. But on similar systems, similar trends and geomean values should be observed.

## A.7 Experiment Customization

The `README.md` files in sub-folders describe how to customize the experiments by modifying the scripts. Meanwhile, the users can also modify our source code to make any changes they expect.

## A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

## References


[1] Sam Ainsworth and Timothy M Jones. 2017. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2017.7863749

[2] Sam Ainsworth and Timothy M. Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3173162.3173189

[3] Sam Ainsworth and Timothy M. Jones. 2019. Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective. *ACM Trans. Comput. Syst.* 36, 3 (2019). https://doi.org/10.1145/3319393

[4] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03619* (2015).

[5] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The Load Slice Core Microarchitecture. In *International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/2749469.2750407

[6] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. Comput.* 44, 5 (1995). https://doi.org/10.1109/12.381947

[7] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John Paul Shen. 2001. Dynamic Speculative Precomputation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2001.991128

[8] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher J. Hughes, Yong-Fong Lee, Daniel M. Lavery, and John Paul Shen. 2001. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/379240.379248

[9] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. 1992. Stride Directed Prefetching in Scalar Processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.1992.697004

[10] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. 2019. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 3 (2019). https://doi.org/10.1145/3366708

[11] Yuxin Guo, Alexandra W. Chadwick, Márton Erdos, Utpal Bora, Ilias Vougioukas, Giacomo Gabrielli, and Timothy M. Jones. 2024. OptiWISE: Combining Sampling and Instrumentation for Granular CPI Analysis. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO57630.2024.10444771

[12] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2016.7783764

[13] Milad Hashemi and Yale N. Patt. 2015. Filtered Runahead Execution with a Runahead Buffer. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2830772.2830812

[14] Intel. 2022. Intel® Core™ i7-12700 Processor. https://ark.intel.com/content/www/us/en/ark/products/134591/intel-core-i7-12700-processor-25m-cache-up-to-4-90-ghz.html.

[15] Intel. 2024. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. https://www.intel.com/content/www/us/en/content-details/825743/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[16] Intel. 2025. Intel(R) RDT Software Package. https://www.intel.com/content/www/us/en/content-details/789566/intel-resource-director-technology-intel-rdt-architecture-specification.html.

[17] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. 2024. Limoncello: Prefetchers for Scale. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3620666.3651373

[18] Doug Joseph and Dirk Grunwald. 1997. Prefetching Using Markov Predictors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/264107.264207

[19] Changhee Jung, Daeseob Lim, Jaejin Lee, and Yan Solihin. 2006. Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS.2006.1639375

[20] Md. Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2011. Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1950365.1950411

[21] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/2749469.2750392

[22] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Evaluation Comput. Syst.* 3, 2 (2018). https://doi.org/10.1145/3177754

[23] Dongkeun Kim, Shih-Wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John Paul Shen. 2004. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *International Symposium on Code Generation and Optimization (CGO 2004)*. https://doi.org/10.1109/CGO.2004.1281661

[24] Dongkeun Kim and Donald Yeung. 2002. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/605397.605415

[25] Sushant Kondguli and Michael C. Huang. 2019. Bootstrapping: Using SMT Hardware to Improve Single-Thread Performance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). https://doi.org/10.1145/3297858.3304052

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2004.1281665

[27] Linux. 2023. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[28] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: Critical Slice Prefetching. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery. https://doi.org/10.1145/3503222.3507745

[29] Chi-Keung Luk. 2001. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *International Symposium on Computer Architecture (ISCA)*, Per Stenström (Ed.). https://doi.org/10.1145/379240.379250

[30] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. 2002. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* 6, 1 (2002).

[31] Sanyam Mehta, Gary Elsesser, and Terry Greyzck. 2022. Software Pre-execution for Irregular Memory Accesses in the HBM Era. In *International Conference on Compiler Construction (CC)*. https://doi.org/10.1145/3497776.3517783

[32] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2005. Techniques for Efficient Processing in Runahead Execution Engines. In *International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2005.49

[33] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2003.1183532

[34] Ajeya Naithani, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. 2021. Vector Runahead. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA52012.2021.00024

[35] Ajeya Naithani, Josué Feliu, Almutaz Adileh, and Lieven Eeckhout. 2020. Precise Runahead Execution. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA47549.2020.00040

[36] Ajeya Naithani, Jaime Roelandts, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. 2023. Decoupled Vector Runahead. In *International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3613424.3614255

[37] Kyle J. Nesbit and James E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the International Conference on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2004.10030

[38] Stephen Pruett and Yale N. Patt. 2021. Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3466752.3480053

[39] Tanausú Ramírez, Alex Pajuelo, Oliverio J. Santana, and Mateo Valero. 2008. Runahead Threads to Improve SMT Performance. In *International Conference on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2008.4658635

[40] Sudhanshu Shukla, Sumeet Bandishte, Jayesh Gaur, and Sreenivas Subramoney. 2022. Register File Prefetching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3470496.3527398

[41] Vinesh Srinivasan, Rangeen Basu Roy Chowdhury, and Eric Rotenberg. 2020. Slipstream Processors Revisited: Exploiting Branch Sets. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA45697.2020.00020

[42] Karthik Sundaramoorthy, Zachary Purser, and Eric Rotenberg. 2000. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/378993.379247

[43] Kim-Anh Tran, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2017. Clairvoyance: Look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2017.7863738

[44] Kim-Anh Tran, Alexandra Jimborean, Trevor E. Carlson, Konstantinos Koukos, Magnus Själander, and Stefanos Kaxiras. 2018. SWOOP: Software-Hardware Co-design for Non-speculative, Execute-Ahead, In-Order Cores. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3192366.3192393

[45] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *International Symposium on Computer Architecture (ISCA*. https://doi.org/10.1145/232973.232993

[46] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/223982.224449

[47] Steven P. Vanderwiel and David J. Lilja. 2000. Data prefetch mechanisms. *Comput. Surveys* 32, 2 (2000). https://doi.org/10.1145/358923.358939

[48] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect Memory Prefetcher. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2830772.2830807

[49] Craig B. Zilles and Gurindar S. Sohi. 2001. Execution-based Prediction Using Speculative Slices. In *International Symposium on Computer Architecture (ISCA)*, Per Stenström (Ed.). https://doi.org/10.1145/379240.379246