Software prefetching for dynamic data structures

Gabriel Dehame

08 May - 28 July 2023

Supervised by:
Alexandra Jimborean
Team CAPS, University of Murcia

Contents

1 Introduction **2**

2 Preliminaries and background **3**
2.1 Coroutines . . . 3
2.2 Helper threads (HT) . . . 3
2.3 LLVM and the LLVM IR . . . 4

3 State of the Art **5**
3.1 Hardware prefetching . . . 5
3.2 Software prefetching . . . 6
3.3 Hybrid prefetching . . . 7

4 Benchmark description **8**

5 Our approach **8**

6 Benchmarks profiling and characterization **9**
6.1 Benchmark 1 . . . 9
6.2 Benchmark 2 . . . 10
6.3 Benchmark 3 . . . 11

7 Target identification heuristic **12**

8 Benchmarks prefetching: space exploration **12**
8.1 Coroutines . . . 12
8.1.1 Benchmark 1 . . . 12
8.1.2 Benchmark 2 . . . 13
8.1.3 Benchmark 3 . . . 14
8.2 Helper thread . . . 15

9 Implementation **16**
9.1 General presentation . . . 16
9.2 Detailed overview of the algorithm . . . 16

10 Methodology **18**

11 Results **18**

12 Conclusion **19**

A Sensitivity studies **21**
A.1 Optimal prefetching distance for benchmark 3 21
A.2 Experimental study for benchmark 1's helper thread's optimal chunk size 21
A.3 Empirical study on the chunk size for benchmark 2's helper thread 21

B Complementary graphs **22**

1 Introduction

Usually, CPUs have three cache levels L1 (level 1), L2 (level 2), and LLC (last level cache), accessing the L1 cache takes around 1 nanosecond. Accessing the LLC takes around 10 nanoseconds and accessing the main memory around 100 nanoseconds. When data is required, for instance when a variable is read, the CPU has to fetch that data. If it finds it in the cache, called a cache hit, it uses it, otherwise, called a cache miss, it fetches it from the main memory and stores it in the caches then uses it. Prefetching is a method used to reduce the execution time of a program by bringing the data or instructions in advance in the CPU cache while the program is executed so that this data is available in the cache when the execution of the program needs it. Prefetching therefore effectively reduces the memory latency of a program as accessing the caches is significantly faster than accessing the main memory. Thus, prefetching can have a huge impact on the speed of a program's execution by drastically reducing the time needed to access data, data accesses being extremely frequent in programs.
Data prefetching focuses on prefetching the data stored in the structures used in a program so that when the structure is read, the data is already in the cache. Programs often use different kinds of structures to store their data and iterate over them, which we call a traversal, to work with these data. These structure traversals need to load addresses to access and work with the data. These loads vary in latency depending on whether the data has been used recently or not. Therefore, it is interesting to prefetch these data in parallel with the main execution so that when the program requires it, it can be found in the cache. Such prefetches are easily computable by hardware for regular structures such as simple arrays. It becomes more difficult for indirect accesses such as $a[b[i]]$ but it can be tackled with software prefetching. However, more complex structures such as dynamic structures, for instance, linked lists or trees, need to store one or more pointers to the rest of the structure holding the data, therefore we cannot easily compute in advance the address of the next element(s).
To solve this, we present a profiling solution. We observe which lines of the code lead to too many misses and try to prefetch the addresses accessed in these lines. These prefetches can be done in multiple manners and we investigated two main solutions, the first is through the use of helper threads and the second through coroutines.
Therefore, our main contributions are the following:

- Study a set of benchmarks representative of real-world applications
- Design and compare a solution based on helper threads and one based on coroutines
- Implementat an LLVM pass to automatically prefetch dynamic data structures
- Evaluate the performance benefits of each of these software prefetching techniques

Everything that has been done is available in a reproduction repository.
Finally, this paper is organized as follows: Section 2 defines the notions used in this report, Section 3 is a bibliographical report on related works, Section 4 contains descriptions of the real-world applications, Section 5 outlines our approach, Section 6 characterizes the benchmarks, Section 7 describes a heuristic to identify the loads that are interesting to prefetch, Section 8 showcases a space exploration to find prefetching solutions, Section 9 details our implementation of the selected solution, Section 10 our methodology for running the benchmarks, Section 11 the results of our benchmarks, and finally Section 12 concludes the report.

2 Preliminaries and background

In this first section, we define the domain-specific concepts required to understand the report. We start by defining what are coroutines, then what is a helper thread and finally what is LLVM and its IR.

2.1 Coroutines

Coroutines are functions that can remember a state between the calls, allowing them to hold their execution and then be resumed upon being called another time.
Such functions have been used recently in prefetching papers [1] to optimize dynamic structure traversals that do not modify the structure but only fetch some information in it.

For instance, figure 1 showcases a hash probe using coroutines. To do so, a coroutine is used to do the probing of a given key in the corresponding linked list. Upon loading the next node of the list it prefetches it and suspends. The caller of the probe maintains a circular buffer with all the probing coroutines called and when the head of the buffer suspends, it calls the next one. Finally, when a coroutine has found the right element or has finished parsing the list, it returns, and the caller removes the coroutine from the buffer.

```
1:  function HASH-PROBE-CORO(key, hash)
2:      node = hash.get(key)
3:      prefetch node
4:      co_await suspend_always{ }
5:      while node do
6:          if key == node→key then
7:              co_return node→value
8:          else
9:              node = node→next
10:             prefetch node
11:             co_await suspend_always{ }
12:         end if
13:     end while
14:     co_return null
15: end function
```

Figure 1: Coroutines based hash index probe algorithm from [1]

In this work, we will use the concept of coroutines to prefetch nodes of dynamic structures parsed inside loops that do not necessarily operate in read-only.

2.2 Helper threads (HT)

A helper thread is simply a thread that will be started in parallel with a main execution, which we will call the main thread. This new helper thread will do the minimal computations required to compute the next addresses to prefetch for a set of targeted loads during a loop iteration. Then, after computing these addresses it will prefetch them, this way the helper thread, in parallel with the main thread, prefetches the data that the main thread will use.

This is a relatively straightforward concept both to understand and to implement. However, several issues come with helper threads and make it hard to fine-tune especially when the structure parsed is modified.

Firstly, the main thread can store data in memory that would affect the execution of the helper thread and the converse too. The converse situation can be solved by using local variables instead of the real values but the other way is more complex.

Secondly, the helper thread takes time to initialize and thus starts its execution after the main thread which can be an issue if the main thread modifies the dynamic structure we iterate on. For instance, if it removes elements from the structure or adds new ones.

Finally, the helper thread can be slower than the main thread. This is because it prefetches and thus stalls while loads are being issued. However, when the main thread runs the matching iteration of its loop after the helper thread (for instance, runs the 100-th iteration of the loop after the helper thread runs the 100th of the loop) then it doesn't stall as much as the helper thread or even not at all. This leads to sometimes the main thread catching up on the helper thread. Then, depending on what the main thread does, the application can crash if it frees memory that would be used by the helper thread for instance.

These issues can be tackled by making sure that the helper thread is always ahead of the main thread. To achieve such a goal one can start a helper thread multiple iterations of the loop in advance and choose the number of iterations such that the main thread can never catch up on the helper thread.

However, this is architecture-dependent as the architecture affects how fast a loop iteration is and how the application interacts with the memory.

In any case, we tried during this work to apply this method to prefetch dynamic structures parsed inside loops.

2.3 LLVM and the LLVM IR

LLVM (Low Level Virtual Machine) is a compiler framework usable for any high-level language and any target architecture. It is split into three parts as illustrated in figure 2, the LLVM frontend which compiles a high-level language into an LLVM-specific language called the LLVM IR (Intermediate Representation). Then there are the LLVM passes which transform or analyze the LLVM IR to optimize it, and finally, the LLVM backend which compiles the LLVM IR into an assembly language.

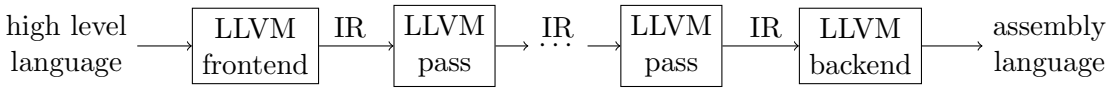


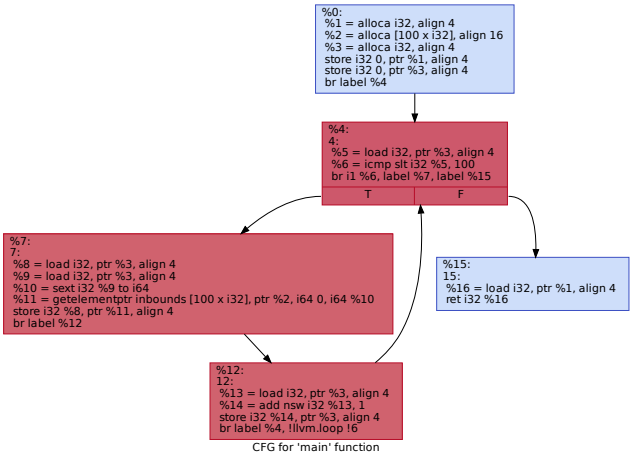
Figure 2: LLVM compilation steps

The LLVM IR itself is an instruction-based language meant to be close to assembly languages but more readable for a human. The instructions are grouped in **basic blocks** which are small sequences of instructions ending with either a return (end of a function) or a jump instruction, either a conditional one for *if* branches or loop termination or an unconditional one to the following block.

These basic blocks are thus linked to other blocks through the jumps and this can be used to create a directed graph whose nodes are the basic blocks and edges are the links between blocks created by the jumps, for instance, if B1 ends by a jump to B2 then the graph has a directed link from B1 to B2. Such a graph is known as a control flow graph, CFG.
An example of LLVM IR and CFG for a simple program is given in figure 3.

```c
int main(){
  int a[100]
  for (int i=0; i<100; i++){
    a[i]=i;
  }
  return 0;
}
```

(a) simple C program



(b) example of CFG

```llvm
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca [100 x i32], align 16 ; a
  %3 = alloca i32, align 4 ; i
  store i32 0, ptr %1, align 4
  store i32 0, ptr %3, align 4 ; int i=0
  br label %4

4:
  %5 = load i32, ptr %3, align 4
  %6 = icmp slt i32 %5, 100 ; i<100 ?
  br i1 %6, label %7, label %15 ; T:7, F:15

7:
  %8 = load i32, ptr %3, align 4 ; get i
  %9 = load i32, ptr %3, align 4 ; get i
  %10 = sext i32 %9 to i64
  %11 = getelementptr inbounds [100 x i32], ↩
      ptr %2, i64 0, i64 %10 ; get a[i]
  store i32 %8, ptr %11, align 4 ; a[i]=i
  br label %12

12:
  %13 = load i32, ptr %3, align 4 ; i
  %14 = add nsw i32 %13, 1 ; i+1
  store i32 %14, ptr %3, align 4 ; i=i+1
  br label %4, !llvm.loop !6

15:
  %16 = load i32, ptr %1, align 4
  ret i32 %16
}
```

(c) example of LLVM IR

Figure 3: Example of a simple C program 3a, its LLVM IR 3c and its CFG 3b

3 State of the Art

3.1 Hardware prefetching

First of all, as prefetching is a technique that consists of bringing data from the memory to the cache, it is deeply hardware-related. So, the first idea is to implement a hardware mechanism. Such mechanisms are known as hardware prefetchers.

The first hardware prefetching approach, called stream buffer, was proposed by Jouppi in 1990 [2], it consists in storing in a buffer, implemented as a queue after the L1 cache, a sequence of addresses that follow an address that led to a cache miss. Then, upon missing in the L1 cache, the stream buffers are checked, popping elements, and if an element of the buffer matches the searched address, it is used and the buffer is filled with addresses following its last address. This way, the buffer remains ahead of the current instruction and as it is not stored in a cache it does not use cachees space, thus adding fast memory in the architecture. More recently, this technique was improved by Kim et al. [3] adding a stride matching prefetcher that tries to match the last missed address with 2 previously missed addresses such that there exists a constant stride separating one of them from another and that other from the last. Then, if such previous misses exist, the stride that provides the longest sequence of misses is chosen, heuristically considered as the most likely to lead to an address of interest, and the next element of the sequence is prefetched.
These stream buffer techniques have the advantage of not affecting the cache space but rather adding new fast memory as well as handling that memory smartly. However, they only catch simple access patterns, trying to match the patterns to stride access patterns which only represent a small part of existing patterns, such as $a[i + x]$ in a for loop iterating on i, but do not handle $a[i * x]$ on a similar loop for instance.

More recently, Wu et al. [4] developed a temporal data hardware prefetcher called Triage. Temporal prefetchers focus on prefetching data that will be fetched soon after a given trigger data fetch. Triage stores metadata in the LLC of correlated addresses, when two addresses are accessed by the same instruction pointer then they are considered correlated and an entry is added or updated in the table. To save space in that table, they also store a stride and length of the correlation so that if A and B, B+1, B+2 are correlated they only store that A is correlated with B with stride 1 and length 3, saving space compared to previous temporal prefetchers. Then, upon accessing an address, the prefetcher looks in the table for all correlated addresses and moves them in a prefetch queue then prefetches the top d addresses of the queue, with d a chosen prefetch distance. If the address isn't correlated to enough addresses, then the most distant correlated address is used to look for more correlated addresses in the metadata table. Storing this metadata in the LLC allows the prefetcher to fetch metadata 7.5 to 20 times faster than previous temporal prefetchers who would add a new hardware component. This also allows the method to be used on any architecture.
Even though it makes this solution more universal as it doesn't need to modify the architecture by adding a new component. The use of the LLC to store metadata imposes a tradeoff between metadata storage and cache size because storing as much metadata as possible means the LLC is unusable for storing the data used by the application. Moreover, their prefetch queue-filling method leads to many metadata queries if the prefetching distance gets too high (the notion of "too high" depends on the number of correlated addresses).

These temporal prefetchers consider addresses interesting when they are temporally close but another notion of proximity is the spatial locality, when two data are stored close to one another in memory. This approach generally requires less space storage than temporal locality prefetchers. Navarro-Torres et al. [5] developed *Berti*, a spatial prefetcher located in the L1 data cache allowing it to see all virtual addresses generated by the CPU. *Berti* computes for each instruction pointer the optimal local delta, defined as the difference between two addresses accessed from this instruction pointer, they are optimal in the sense that they lead to the most timely prefetches. Then they can use those computed local deltas to generate optimal prefetches for each instruction.
This new way of considering delta prefetching, computing them per instruction pointer, offers a new perspective on spatial prefetching and is more power-saving, and more accurate as it more rarely prefetches too soon. However, the benchmarks also show that the method isn't to be chosen naively as it sometimes

is less efficient than previous methods.

A major issue with hardware prefetching generally is that it has trouble prefetching complex data accesses as it tries to find a regularity in the accessed addresses that might not exist. Especially when considering indirect memory accesses or dynamic data structures in which the next element's location in memory is independent of the previous one. Therefore, we need to investigate other ways of prefetching data, such as through software prefetching.

3.2 Software prefetching

Software prefetching consists of analyzing the program at compilation time identifying which data accesses in the code are likely to cause misses and generating prefetch instructions for these data accesses with a heuristically computed prefetch distance.

One of the first papers about software prefetching for dynamic data structures is from Luk and Mowry [6], in which they develop three methods to overcome the pointer chasing problem, that is the traversal of a recursive data structure such as linked lists or trees. Their first idea is to prefetch all the successors of an element of the structure upon iterating over that element. This is of very little use for linked lists as there is only one successor, but for a tree traversal for instance, if the program recursively traverses in the left node then in the right, one can at least partially hide the latency of the right successor. Their second method consists of adding a history pointer for each element of the structure. Upon traversing the structure one maintains a queue of the elements of a given size s, chosen to be the prefetch distance. When the queue is full, then, for each node one iterates on, one pops the queue and adds a history pointer to the current node in the popped node. Then on future traversals, one can prefetch the element pointed to by the history pointer of the node, hoping it is still the s-th next element. Finally, they also showcase a scheme to linearize data structures by mapping the structure to an array, which is extremely inefficient if the structure changes a lot. So it is to consider only for slowly changing structures.
These methods are merely preliminary ways of thinking about prefetching dynamic data structures and are not very efficient. The cost of the last one is high, as it requires us to create a new array each time we modify the structure. The second one as well as the last one works well only if the data structure follows certain rules. The first one is a bit more applicable but only partially covers the cache latency as it needs the prefetch distance to match the length of the previous traversals, which is not generally true.

Ainsworth and Jones [7] proposed a method to automatically compute at compile-time the addresses to prefetch for indirect memory accesses such as $a[b[i]]$. These indirect accesses are rarely picked up by hardware prefetchers and so require to use a software prefetching method. Their method consists of walking backward the dependency graph of a given load using a depth-first search algorithm and stopping when they leave the loop or end up on the induction variable. Meanwhile, they record the required instructions to compute the address and then introduce the recorded instructions at the beginning of the loop adding an offset to compute the address used in a further iteration of the loop. They also provide a formula to compute the offset based on the number of loads to prefetch, the position of the target load in the set of loads, and the instructions per cycle of the CPU.
Despite being more expressive than previous prefetching methods, as it can prefetch indirect memory accesses, this method still doesn't work on dynamic data structures as they target only loads that can be prefetched by adding an offset to the loop induction variable. Additionally, it lacks a way to compute optimally the offset, only giving a heuristic formula. Moreover, it also lacks a way to distinguish whether a given load should be prefetched or not, as sometimes prefetching a load leads to an even higher overhead if the given load isn't likely to miss.

To solve the last two issues, Jamilan et al. [8] proposed a profile-guided approach to improve the performance of the previous method. They reuse their algorithm but with slight modifications to take as input a profile of the target program, which is a record of the memory interactions of the program previously ran, to identify the delinquent loads, which are loads that often miss in the cache. They also use this profile to compute an optimal prefetch distance and an optimal point in the program where to introduce the prefetches. Especially, if a loop is identified as only running a few iterations. In such cases, they prefetch before the loop and not inside as otherwise it will mostly prefetch unused data. To

compute the optimal prefetch distance, they use the profiles to identify the memory and instructions latencies and to define the optimal prefetch distance as $\frac{MemLat}{InstLat}$. The loop iteration cannot be faster than the instructions latency and prefetching optimally would allow hiding the memory latency so that if we prefetch k iterations in advance, by the time the k iterations are done, the data arrived in the cache.
Through this approach, Jamilan et al. improved the previous approach presented by Ainsworth and Jones but at the cost of requiring to profile the program. This is something that must be done specifically for each target architecture. Therefore it is not very convenient for the user, especially as it is not done automatically. Moreover, they still rely on the same prefetching method developed by Ainsworth and Jones and thus do not handle dynamic data structures.

3.3 Hybrid prefetching

Depending on the situation, software prefetching or hardware prefetching is more efficient than the other, therefore we are led to investigate combining both to cover all situations, which leads to hybrid prefetching.

In particular, Lee et al. [9] analyzed the situations in which the two kinds of prefetching were efficient and why. They concluded that hardware prefetching is not able to cover complex structures nor irregular access patterns whereas software prefetching can. Additionally, hardware prefetchers are limited in resources and thus cannot consider every access pattern of a given execution whereas software prefetching can insert prefetch instructions independently for each. Moreover, software prefetchers can avoid prefetching out-of-loop bounds elements whereas at the hardware level, the hardware prefetcher cannot anticipate when the loop will end and thus makes costly useless prefetches. However, software prefetchers increase the instruction count which can slow down the execution if the application isn't memory-bound enough and hardware prefetchers unlike software ones can adapt their behavior to the runtime of the application and thus be more precise. Lee et al. also point out that using both prefetchers allows one to gain from both advantages: working for a large variety of access patterns and the software prefetcher can make the hardware prefetcher even more efficient. Nonetheless, they can also interact negatively and lead to worse results than if only one is used.

A few years later, Mehta et al. [10] complemented that previous study by analyzing how the CPU architecture affects the prefetching strategies. They noted that on a SandyBridge, it is possible to combine a stream hardware prefetcher and an L1 software prefetcher to bring data in the L1 cache overcoming the L1 hardware prefetcher's limitations. On a Xeon Phi, if we use an L1 software prefetcher then the L2 stream hardware prefetcher doesn't prefetch. However, it is possible to combine software prefetchers for LLC and L1 cache to bring data in the L1 cache. Then the said researchers expose techniques to prefetch in the L1 cache on both Xeon Phi and SandyBridge. For the first, they have to use a software prefetcher to insert prefetches for all levels of cache whereas for the second the hardware prefetcher handles L2 cache and LLC so they only insert prefetches for L1 cache. They also compute the ideal prefetch distance differently depending on the architecture.

These two studies show that it is interesting to consider combining hardware and software prefetching to cover more access patterns and to prefetch more efficiently. However, this requires architecture-specific analyses and so poorly generalizes.

Finally, Talati et al. [11] recently proposed a hybrid prefetching technique to optimize indirect and irregular memory accesses. They created a compile-time analysis of a target program that represents the data accesses of the application into a graph called DIG (data indirection graph). This graph encodes the relations between the data accesses, the nodes are the data structures access, and the edges are the dependencies between those. Then, after building that graph, they feed it to a programmable hardware prefetcher that they created to generate the prefetches according to the DIG. This especially allows the hardware prefetcher to use the DIG to be aware of indirect memory accesses and thus prefetch them, unlike native hardware prefetchers. The DIG also contains trigger edges (edges from a node to itself) that contain information to initialize a prefetch sequence such that upon a trigger event. Typically a given load of an element in the structure represented by the node, the hardware prefetcher can then use this information to create the correct prefetches.
Talati et al. managed to make use of both software compile-time analysis and hardware prefetchers to

provide a more energy and time efficient prefetching scheme compared to previous works on average. It fails to beat two of the previous approaches on two specific workloads but generally, it does better. However, this method specifically targets two kinds of data accesses, the single-valued indirection, and the ranged indirection which are illustrated in figure 4, thus they do not handle dynamic structure accesses.

```
for (int i=0; i<N; i++) {
 do(b[a[i]]);
}
```

```
for (int i=0; i<N; i++) {
 for (int j=a[i]; j<a[i+1]; j++) {
  do(b[j]);
 }
}
```

Figure 4: Illustration of single-valued indirection (left) and ranged indirection (right)

4 Benchmark description

The software prefetching methods described in this report have been tested on three benchmarks that have been extracted from industrial applications and are representative of real-life workloads. The benchmarks use dynamic data structures that are updated during the execution of the application to resemble a real-life scenario. Thus, the accessed memory becomes fragmented, accesses are irregular and highly difficult to predict. This behavior is notoriously difficult for both traditional and state-of-the-art prefetchers.

After a brief description of the benchmarks, we characterize the benchmarks' behavior through profiling, attempting to identify the loads that miss mostly in the cache (delinquent loads). Next, we try to build a heuristic to identify such loads statically, through the compile-time analysis. Finally, we develop automatic methods to prefetch these loads and hide their latency.

Benchmark 1 This first benchmark application implements a double-linked list and a for loop iterating on the list. First, the application initializes and shuffles the linked list. Next, the for loop processes its elements. In this work, we skip the initialization and shuffling phases and target the main loop for optimization.

Benchmark 2 The second benchmark iterates in a while loop over a double-linked list following the approach described next. The linked list's head is stored in another structure, $MainList2$, and at each iteration of the loop, the head is removed from the list and the element of the structure which points to the head of the list is changed to point to the new head of the list. Moreover, the loop doesn't necessarily jump to the next element of the linked list. This jump is done by setting the pointer in $MainList2$ to the next element of the list based on computations done in the loop iteration itself. Depending on those computations, the next element is either set to $NULL$ (in which case the loop will break on the next iteration) or to the next element of the list. This loop is the main computation and our optimization target. Before this, there is an initialization step similar to the one of benchmark 1.

Benchmark 3 This final benchmark is significantly easier in principle because it is simply a for loop iterating over an array. The special point for this benchmark is only that the array is stored in an auxiliary structure but that structure isn't modified at any point.

5 Our approach

This section details the phases of our approach.

In the first phase, we profile the three previously described benchmarks gathering information about their execution. In particular, we measure the number of cache hits and misses caused by a load separately for every cache level and every line of the source code. We also gather the number of cycles spent stalling by the CPU due to a cache miss, these are cycles during which the CPU does nothing

while waiting for data to be loaded from the cache. Again we measure them for every level of cache and line of source code separately. With these profiles measured, we identify the delinquent loads, which are loads that are mostly slowing down the applications. These are the ones that are responsible for most of the misses or stalls. We therefore refer to them as **target loads** in the rest of the report because they are the loads we want to prefetch.

In the second phase, we try to build a heuristic to statically identify these target loads by analyzing the source code and trying to find what makes the target loads identified in the first phase distinguish themselves from the others.

In the third phase, we investigate two methods to prefetch these target loads manually. The first one is by using coroutines and the second one is by using helper threads.

Finally, in the fourth phase, we select the most efficient of the two previous methods and automatize it in an LLVM pass to be able to apply it to any application.

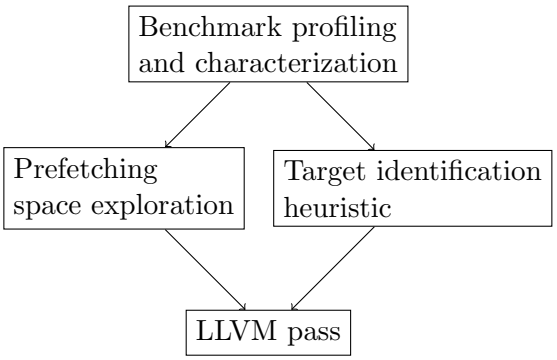


Figure 5: Dependency among the four phases of our approach

6 Benchmarks profiling and characterization

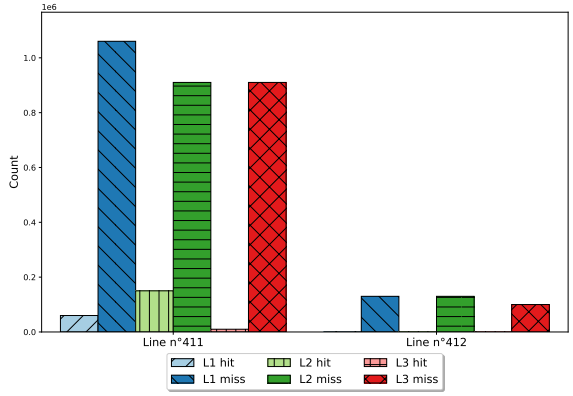
First of all, to try and improve the performances of these benchmarks, we profiled the applications with Intel's vTune profiling application to characterize them by measuring their cache interactions and looking for the pieces of code that were mainly responsible for cache misses and stalls. This way we would be able to know which elements we have to prefetch to obtain the best possible performance improvements with the lowest number of prefetches. In this part of the report, we will showcase the profiling we have done and the conclusions we deduced from them.

To find which elements were the most important to prefetch, we ran the applications compiled with GCC with options **O3** and **gdwarf-4** and measured the cache misses and hits for each line of the source code as well as the number of cycles during which the CPU stalls due to a miss, each of these three measurements being done for all three levels of the cache. Then, we analyze the results to keep only the delinquent loads of the source code. We define these as the lines that have at least a 30% miss rate in L1 and lead to at least 5% of the total number of stalls during an L1 miss or 5% of the total number of L1 misses, we consider only L1 here because upon missing in L1, the CPU will fetch in L2 then if it misses again it will fetch in L3 and then, in theory, the stalls in L1 are higher than in L2 which are higher than in L3 and same for the numbers of misses. This can be affected by the measurement method of vTune (the profiling application we use) which cannot be completely accurate because it would take too many resources, it uses PEBS (precise event-based sampling), however, the general scope of the measurements remains the same.

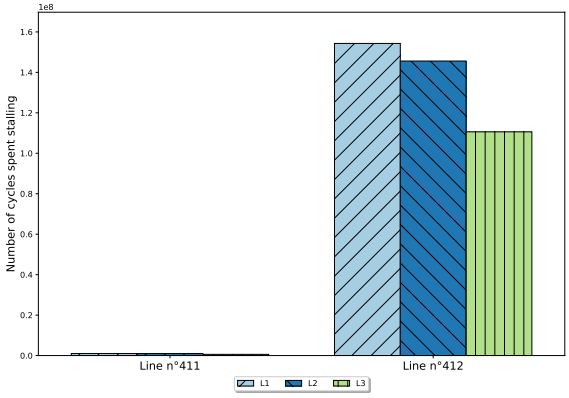
6.1 Benchmark 1

Our profiling of the application leads to two figures, the first one being figure 6a which shows the number of cache hits and misses at the three levels of cache for each delinquent load. Then we also illustrate in figure 6b the number of cycles spent stalling for each cache level and each delinquent load. We also illustrate in figure 7 the source lines containing the delinquent loads. We can see from these figures that the benchmark has two lines of code that mainly cause the stalls and misses, and thus are mostly responsible for slowing down the code. We also see that the first (line 410) leads to the most misses whereas the second (line 411) leads to the most stalls. This is because the second one stalls due to the misses of the previous line as line 410 only reads data to store it somewhere and so the CPU can execute the following instructions while waiting for the data to be fetched ($Symbol\_377$ here) but line 411 needs that data for $Symbol\_236$'s computation and thus it stalls, we also see that those two lines

have very high miss rates at all levels of cache.



(a) Cache hits and misses for delinquent loads in benchmark 1



(b) CPU stalls due to cache misses for delinquent loads in benchmark 1

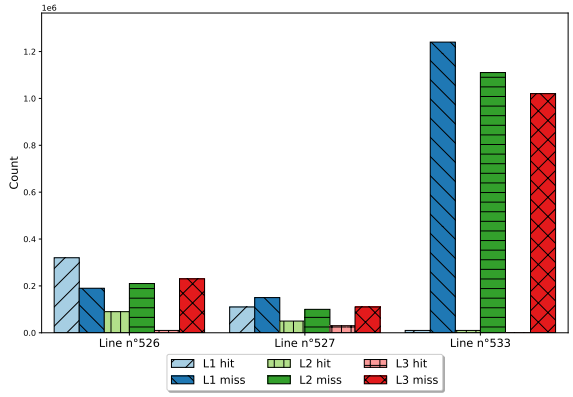
Figure 6: Measurements for benchmark 1

```
for (uint32_t i = 0; i < num_iter; i++) {
    [...]
    Symbol_304 *Symbol_685 = (Symbol_304*)(Symbol_377->ListNode.next); //line 411
    Symbol_69 *Symbol_879 = Symbol_236(Symbol_377->position); //line 412
    [...]
    Symbol_377 = Symbol_685; // getting next element
  }
```

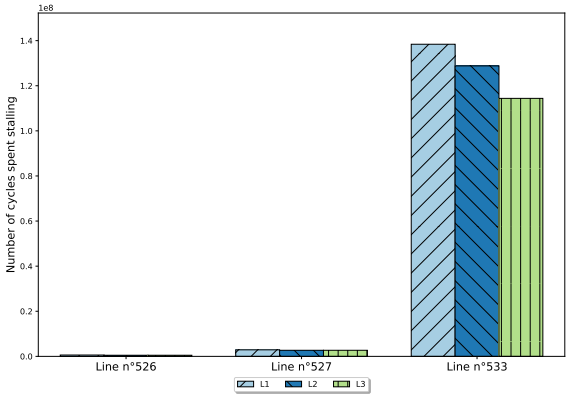
Figure 7: Extract of the source code illustrating the delinquent loads in benchmark 1

6.2 Benchmark 2

Similarly to the case of benchmark 1, we drew two graphs to find out which lines of the code were mostly slowing down the application, drawing in figure 8a the misses and hits for all three levels of cache and then in figure 8b the number of cycles spent stalling during misses for each level of cache, all these for each delinquent load of the source code. The source lines containing the delinquent loads are illustrated in figure 9. We see in these two figures that there are only three delinquent loads in this benchmark. One of which, line 531, again gathers most of the misses and this time it is also the one that gathers almost all of the stalling cycles. We also see that this line has almost 100% miss rate at all levels of cache whereas the other two are lower especially for L1 and L2, being below 70% but still above 50%.



(a) Cache hits and misses for delinquent loads in benchmark 2



(b) CPU stalls due to cache misses for delinquent loads in benchmark 2

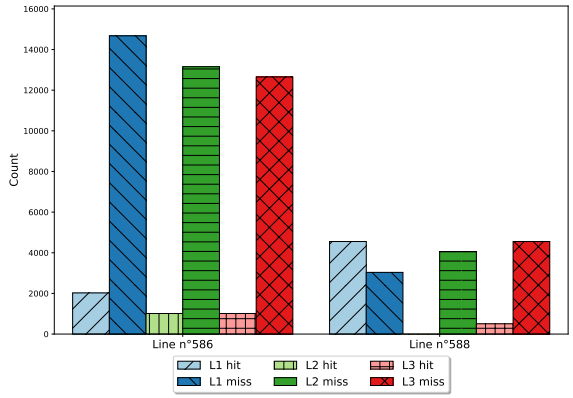
Figure 8: Measurements for benchmark 2

```
while (ListNode != ((void *)0)) {
    Symbol_1164 = (Symbol_308 *)ListNode;
    uint32_t position = Symbol_1164->position; //line 526
    uint16_t Symbol_352 = Symbol_1164->Symbol_352; //line 527
    [...]
    Symbol_69 *Symbol_879 = Symbol_236(position); //definition of Symbol_879
    [...]
    {if ((Symbol_879->Symbol_857.Symbol_474 != Symbol_0)) continue;}; //line 533
    [...]
    ListNode = MainList2->ListNode;
  }
```

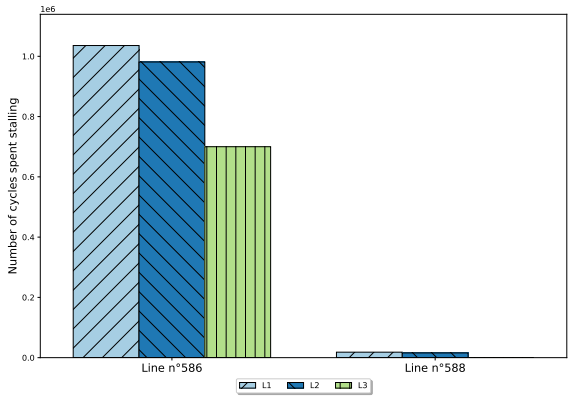
Figure 9: Extract of the source code illustrating the delinquent loads in benchmark 2

6.3 Benchmark 3

For this last benchmark, we found out by profiling the application that the delinquent loads pointed out by the hits and misses of figure 10a and the cycles spent stalling during misses of figure 10b were on addresses that could be computed given the address of the array and the value of the loop's induction variable. As one can see in figure 11, the two lines containing the delinquent loads are loads of the structure element pointed to by *Symbol_879*, and this pointer is computed in the function *Symbol_236* using as only parameter an attribute of the current element of the array that we are considering in the current iteration of the loop, that is $array3sub[i]$. This means that there is no need for unusual prefetching techniques and we can simply compute the value of the addresses of the delinquent loads multiple iterations ahead through already existing techniques. Moreover, we again have one line which concentrates most of the stalls and most of the misses with another line having a high miss rate and many misses but far less than the first one. We thus again have to focus on prefetching one line specifically, here line 586.



(a) Cache hits and misses for delinquent loads in benchmark 3



(b) CPU stalls due to cache misses for delinquent loads in benchmark 3

Figure 10: Measurements for benchmark 3

```
for (uint32_t i = 0; i < num_iter; i++) {
    uint16_t position = array3->array3sub[i].position;
    Symbol_69 *Symbol_879 = Symbol_236((uint32_t)position);
    {if ((((__builtin_expect((!!((Symbol_879) == ((void *)0))), 0))))) continue;};
    array_aux[i].Symbol_652 = Symbol_879->Symbol_858.Symbol_699; //line 586
    array_aux[i].Symbol_556 = position;
    array_aux[i].Symbol_364 = Symbol_879->Symbol_858.Symbol_364; //line 588
    [...]
  }
```

Figure 11: illustration of delinquent loads in benchmark 3

7 Target identification heuristic

After profiling the benchmarks, we wanted to find a way to heuristically identify the target loads so that such profilings are not necessary to apply our prefetching method and thus have a more easily portable solution. To do so, we analyzed the source code of the benchmarks and specifically what made the delinquent loads differ from the others.

We found out that the target loads were always loads that fetch data from an element of a structure that isn't constant among loop iterations for the first time inside the loop iteration. For instance in benchmark 1 illustrated in figure 7, line 410 is the first line dereferencing the induction variable pointer *Symbol_377* and line 411 is the first to use the data behind which thus requires line 410 to finish loading the element pointed to by *Symbol_377* and thus stalls. In benchmark 2 illustrated in figure 9, the same happens, lines 524 and 525 are the first lines dereferencing the induction variable *ListNode* and are used for the first time in the definition of *Symbol_879* which makes it non-constant among loop iterations and is first dereferenced in line 531. Finally, in benchmark 3, *Symbol_879* is again defined based on the value of the induction variable i and is accessed for the first time in line 584 which is the most delinquent.

We also analyzed the rest of the source code and figured that there were no other data structures that are non-constant among loop iterations and are loaded in the loop. Therefore, the heuristic is sound and complete for these 3 benchmarks. Moreover, this heuristic makes sense with the way the CPU works. Effectively, the first time an address is loaded, it is brought from the main memory to the cache filling an entire cache line with surrounding addresses which thus covers not only the target attribute of a structure element but all its attributes, for example, $p \rightarrow a$ and $p \rightarrow b$ are two attributes of the structure element pointed to by p. Thus, further accesses to attributes of that structure element will be fast, and we only need to prefetch the first one. We also ignore constant structures because they do not vary with loop iterations, they will be slow to load only in the first iteration of the loop.

8 Benchmarks prefetching: space exploration

Once we identify for each benchmark which lines contain delinquent loads, the next step is to find a method to insert prefetch instructions. To this end, we investigate two concepts: the helper threads and the coroutines. The general concepts are described in the section 2 *Preliminaries and background*. This section presents how they are employed to optimize our target benchmarks.

8.1 Coroutines

8.1.1 Benchmark 1

For benchmark 1 (see figure 7), we prefetch both loads identified as causing either a significant number of cache misses or CPU stalls, as reducing their cache misses has the potential of greatly improving the performances. We take advantage of the fact that the first delinquent load is required for the second to happen, line 411 needs line 410 in figure 7 from the previous loop execution, and we also take advantage of the fact that the delinquent load of line 411 is passed as an argument to a function. This way, we focus on turning the given function, here *Symbol_236*, into a coroutine which instead of computing the actual value that would originally be computed will use its state to remember the argument it received at the last iteration (or equivalently, the last call) and use its actual argument to prefetch the target and then make the original computation based on the previous argument, which is in the state. For this, we initially, before the loop, call once our coroutine for it to prefetch the initial *Symbol_377* $\rightarrow$ *position* and store *Symbol_377* in its state. Then upon calling the coroutine in the loop, we pass it as argument *Symbol_685* which is the next value of *Symbol_377*, the coroutine will prefetch *Symbol_685* $\rightarrow$ *position*. Then based on its state, which is the previous value of *Symbol_377*, it will do the computation *Symbol_236* would have done and return the result after storing *Symbol_685* (the next *Symbol_377*) in its state. This coroutine is illustrated in figure 13 and function *Symbol_236*, which has been replaced by the coroutine is illustrated in figure 12. We can see on the coroutine code that there is a variable called *state* and a switch based on *state*, this is known as Duff's device and is the technique

used in C to implement coroutines because they are not natively implemented unlike in Python. The static variable *previousParam* is what we use to actually implement our state, generally speaking, a coroutine's state is all its static variables.

```
Symbol_69 *Symbol_236(uint32_t Symbol_880)
{
  do{
    if (((((__builtin_expect((!!((structure_of_interest) == ((void *)0))), 0))))
          || ((Symbol_880) >= getNumIter()))) { return((((void *)0))); }
    } while (0);
  return (structure_of_interest + Symbol_880);
}
```

Figure 12: Function *Symbol_236*

```
Symbol_69 * coroutine1(Symbol_304* Symbol_377){
  static int state = 0;
  static Symbol_304* previousParam = NULL;
  switch(state){
    case 0: //initialization case
      __builtin_prefetch(&(Symbol_377->position),0,0);
      __builtin_prefetch(&(Symbol_377->ListNode.next),0,0);
      state = 1;
      previousParam = Symbol_377;
      return NULL;
    case 1: // loop case
      __builtin_prefetch(&(Symbol_377->position),0,0);
      __builtin_prefetch(&(Symbol_377->ListNode.next),0,0);
      //Symbol_236's computation filling res and previousParam before returning
      if (structure_of_interest && previousParam->position < num_iter) {
        Symbol_69* res = structure_of_interest + previousParam->position;
        previousParam = Symbol_377;
        return res;
      }
      previousParam = Symbol_377;
      return NULL;
  }
  return NULL;
}
```

Figure 13: Prefetching coroutine for benchmark 1

8.1.2 Benchmark 2

For benchmark 2 (see figure 9), we applied a similar idea but as there is only one load causing both the most stalls and the most misses, we focus on prefetching only this one. For this, we turn again *Symbol_236* from figure 9 into a coroutine because it is the function that defines the pointer used as the initial source of the most delinquent load. We applied the same method as for benchmark 1, the next value of *ListNode* is given as a parameter to the function because the *position* argument passed to *Symbol_236* is computed using *ListNode* only which is the iteration variable of the loop and so we cannot go further in the dependence graph of the variables. However this time, to prefetch the target delinquent load, we need the result of the call to *Symbol_236* from the next loop iteration unlike in benchmark 1. Thus it changes a bit the coroutine, this time we store in its state the previous result of *Symbol_236* and the coroutine will use its argument to make the computation *Symbol_236* would have done in the next iteration, then prefetch the target load, store the result in its state and return the previous result which was in its state when it was called.
This is again illustrated in figure 14 with an initialization case called before the loop and a loop case called in the loop and which replaces the call to *Symbol_236*.

```
Symbol_69 * coroutine2(NodeC* ListNode){
  static int state = 0;
  static Symbol_69* previousResult = NULL;
  Symbol_69* res = NULL;
  switch(state){
    case 0: // initialization case
      if (structure_of_interest){
        previousResult = structure_of_interest + ((Symbol_308 *)ListNode)->position;
        __builtin_prefetch(&(previousResult->Symbol_857.Symbol_474),0,0);
      }
      state = 1;
      return NULL;
    case 1: // loop case
      res = previousResult;
      //modified computation of Symbol_236 to add a prefetch and store the result in the state
      if (!structure_of_interest || !ListNode) previousResult = NULL;
      else{
        previousResult = structure_of_interest + ((Symbol_308 *)ListNode)->position;
      __builtin_prefetch(&(previousResult->Symbol_857.Symbol_474),0,0);
      }
      return res;
  }
  return 0;
}
```

Figure 14: Prefetching coroutine for benchmark 2

8.1.3 Benchmark 3

For benchmark 3 (see figure 11), things are different because we iterate over an array in a for loop, so we can simply use the induction variable, i, to prefetch elements that are defined based on the array, which is the case of our delinquent loads. Therefore a coroutine isn't necessary at all, we can simply add a basic prefetching line at the beginning of the loop as shown in figure 15

```
for (uint32_t i = 0; i < num_iter; i++) {
    __builtin_prefetch(
      &(((array3->array3sub[i+16].position)
         + structure_of_interest)->Symbol_858.Symbol_699),0,0);
    uint16_t position = array3->array3sub[i].position;
    Symbol_69 *Symbol_879 = Symbol_236((uint32_t)position);
    {if ((((__builtin_expect((!!((Symbol_879) == ((void *)0))), 0))))) continue;};
    array_aux[i].Symbol_652 = Symbol_879->Symbol_858.Symbol_699; //line 584
    array_aux[i].Symbol_556 = position;
    array_aux[i].Symbol_364 = Symbol_879->Symbol_858.Symbol_364; //line 586
    [...]
  }
```

Figure 15: Basic prefetching for benchmark 3

If we want to conceptually test a coroutine, we would, following the same method as before, walk up the dependence graph of our target load until we reach variables defined outside the loop, and redefine the first variable defined inside the loop in that walk. This variable is *position*. To define the variable, we need the induction variable i and a variable defined outside the loop, $array3$. Our coroutine will thus take $array3$ and the next value of i as arguments, prefetch the next value of the target delinquent load, compute the current position based on its state, and store in its state the actual value of i. This mimics a basic prefetching with a prefetch distance of 1. But doing so in a more complex manner and should thus be less efficient, especially as we usually prefetch such access patterns multiple loop iterations in advance. This leads to the coroutine of figure 16, in which we have an initialization case performed before the loop in an initialization call to the coroutine. Then we have the loop case which prefetches if the global variable *structure_of_interest* is not null because otherwise, the iteration stops at line 583 as *Symbol_236* would return *NULL*.

```
uint16_t coroutine3(Array3 *array3, uint32_t i){
  static int state = 0;
  static uint16_t previousI = 0;
  switch(state){
    case 0: // initialization case
      state = 1;
      previousI = i;
      if (structure_of_interest)
          __builtin_prefetch(&(((array3->array3sub[i].position) + structure_of_interest)->↩
              Symbol_858.Symbol_699),0,0);
    case 1: // loop case
      if (structure_of_interest)
        __builtin_prefetch(&(((array3->array3sub[i].position) + structure_of_interest)->↩
            Symbol_858.Symbol_699),0,0);
      uint16_t prevI = previousI;
      previousI = i;
      return array3->array3sub[i].position;
  }
}
```

Figure 16: Coroutine prefetching for benchmark 3

8.2 Helper thread

General scheme to add a helper thread To implement our helper thread, we initialize it, as soon as we have defined all the variables required for the loop computation, by sequentially prefetching an initial chunk of the structure, whose size is to be adjusted heuristically, and store the address of the element following that chunk. Then we start our helper thread passing the previously stored element so that it starts ahead of the main thread to be timely, and then we continue the main thread's execution in parallel to the recently created helper thread. This scheme is represented on the figure 17:

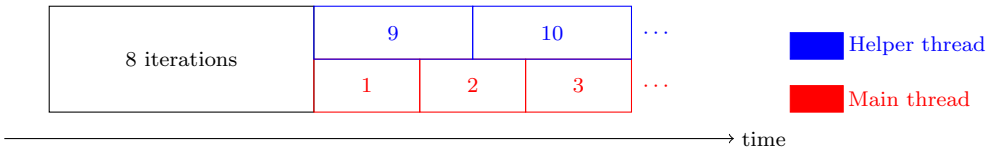


Figure 17: An application running with our helper thread scheme for a chunk size of 8. Each block corresponds to a loop iteration whose number is written inside the block.

This requires to properly adjust the chunk size for optimal performance. Therefore we performed such a sensitivity study for benchmark 1 presented in appendix A.2 and it led us to choose a chunk size of 8 elements. However, while doing so for benchmark 2, we figured that during the loop iteration, the element of the list we are using in this iteration is removed from the list. Therefore if at some point the helper thread isn't ahead of the main thread then it will not have access to the next element anymore because its current element will have been removed from the list and so its next pointer will not point anywhere. In figure 17, the Main thread's iterations last $\frac{2}{3}$ of the Helper thread's, thus the 24th iteration will be done by the Main thread before the Helper thread and prevent the Helper thread from doing the 25th. This is especially problematic when the initialization of the helper thread is not adjusted well enough. This happens if the chunk size is too small, in this case, the helper thread will stop after its first execution and thus only add the initialization overhead and consequently decrease the application's performance.

We didn't push further on benchmark 3 because benchmark 2 raised the mentioned issues and because we can apply existing techniques on benchmark 3 as the target structure is an array and thus the target load can be computed multiple iterations in advance without requiring complex methods such as helper threads.

9 Implementation

This section details the implementation of the coroutine-based prefetching method elaborated during space exploration, with a few adaptations so that we can automatize it.

9.1 General presentation

First of all, in the method applied during the space exploration, functions were redefined into coroutines. This is no longer possible at the LLVM IR level because functions are inlined. Therefore, we need to change our course of action. However, before doing such a transformation we were searching the dependency graph of our target load upward to find the loop iteration dependent values used to compute the target, we can still do such computation.

After this computation, we checked whether the load of the target address is always executed or not among a loop iteration. This happens when the target is inside a *if* branch for instance. Then we look for what we called a "**diamond**", because of its shape, containing the target, which is a sequence of basic blocks. It starts with a block, called entry, ending with another block, called exit, both executed at each loop iteration as well and contains all blocks in between. This is illustrated in figure 18, on which we drew a loop header, which marks the beginning of a loop iteration followed by a sequence of blocks, then we have our diamond containing the target load, preceding another sequence of blocks ending with a loop latch marking the end of a loop iteration.

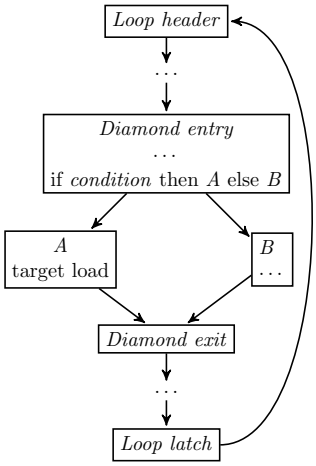


Figure 18: Illustration of a diamond

Then, depending on whether the target is inside a diamond or not we implemented two methods. These two methods will be detailed below.

The target load is not always executed Then we extract the diamond, except the diamond exit, from the loop and create a coroutine to execute that diamond but instead of loading the target, it will prefetch the target address, address that the target load is loading, and then hold its execution. Then upon reaching the original diamond entry in the loop, the coroutine will be resumed to load the target and continue the diamond's execution. If multiple targets are in the same diamond then multiple hold-resume will be generated.

The target load is always executed Then, we have two distinct sub-cases.
The first is when the target uses a value of a variable that itself depends on a *if* branch. In which case we move the target load in the corresponding *if* branches with its dependencies, and use the value that the used variable would have if defined from that branch. Therefore the target load ends up duplicated in multiple *if* branches which are part of a diamond and so we apply the previous method.

The second sub-case is when there is no such dependency and then we identify the dependencies among the previous loop iteration and duplicate the target's computation in the previous loop. We also reorder the instructions to do this computation as early as possible and replace the load with a prefetch.

9.2 Detailed overview of the algorithm

We assume in this part that the targets have already been identified, either using the previously described heuristic (section 7) or using hand-written annotations in the LLVM IR. This way, we will be able to later modify the heuristic if needed or to add a profile generation to annotate the IR to extend our method. Additionally, it will allow us to measure the efficiency of the heuristic comparing the pass when it uses it and the pass when it uses annotations of the profiled delinquent loads.

The target load is not always executed As the overview explains, we extract the diamond containing the target load from the loop to define the coroutine (line 13 in algorithm 1). However, some values can be referenced inside that diamond that were defined before the diamond entry. Similarly, others can be defined inside the diamond and referenced after. For the ones defined before the diamond, we pass them as arguments to the coroutine (lines 11 and 14 in algorithm 1). For the ones used after the diamond, we define global variables to which the coroutine will store the values used outside the diamond and that will be loaded in the diamond exit (lines 12 and 15 in algorithm 1).

Additionally, we need to take care of memory writing instructions. We will reorder the execution thus some writes can affect the execution. The writes which affect the execution are those that precede the diamond and write in an address used inside the diamond, and those inside the diamond that write in an address used before the diamond.
Those before the diamond will be considered as explained in the next paragraph.
For the others, as we want to call the coroutine as soon as possible, we might execute memory writing instructions earlier than they should have been. To avoid this we verify that none are neither necessary to execute before the target load nor write in an address used before the diamond. Otherwise, we abort the coroutine creation: we cannot prefetch the target, in algorithm 1 we assume that this treatment is done before calling the function **Prefetch**. Then, for the ones that are not mandatory for the target load, we move them to the resuming part of the coroutine (line 18 in algorithm 1). This part of the algorithm also moves instructions that are executed before the load inside the diamond and that are not required to know if the load is to be executed so that the prefetch is done sooner.

Once the coroutine is created, we need to add calls to it in the loop (line 20 in algorithm 1). And to prefetch the target, we need to call it as soon as possible, ideally sooner than upon reaching the diamond entry otherwise it is equivalent to not having built a coroutine. Therefore, we use the values passed as arguments to the coroutine as an indication of where we can call the coroutine for the first time so that it prefetches. And, we create the first call after the last argument is defined and every instruction that writes in pointers used inside the coroutine is executed because it can affect the execution. Then, we replace the diamond entry from the original loop, that we removed to put in the coroutine, with a block containing a call to the coroutine, which will thus resume its execution starting from the target load if it was to be executed or otherwise do the memory writing instructions that were in the diamond.

The target is always executed Again, we have two distinct sub-cases. The first is when the target uses a value of a variable that itself depends on a *if* branch. In this case, we move the target load to the corresponding *if* branches with its dependencies, possibly duplicating computation to be in both branches of the *if* and using, instead of the variable that depends on the *if* branch, the corresponding ones that would have been stored in that variable. This way, we duplicated the targets but put them inside a diamond, so we can apply the previous method. In the algorithm 1 we do not detail this transformation, we assume that this is handled before calling **Prefetching** by filling *alwaysExecuted* and *notAlwaysExecuted* accordingly.

The second case is when there is no such dependency. Then, a coroutine is not necessary, we simply move the instructions that define the next value of the induction variables used to compute the target address as early as possible and we duplicate the computation of the address to prefetch it. To clarify, we work on two successive iterations of the loop. We consider the target address of a given iteration i, and we find the instructions computing the associated value of the inductions variables (line 2 in algorithm 1). These instructions are part of iteration $i-1$ and we move them as early as possible to then copy the instructions from iteration i that define the target address right after the values of the induction variables of iteration i are defined (line 5 in algorithm 1) and add a prefetch of the target (line 6 in algorithm 1).

An important aspect is what we call "as soon as possible". When we move an instruction to an "as soon as possible" position, we move the instructions it depends on as soon as possible too, recursively. Then we move it after the latest of those and of the memory writing instructions that write in an address used by the given instruction, if any. This is done in line 4 in algorithm 1 after computing the insertion point in line 3.

Algorithm 1 Dynamic structure prefetching pass pseudo-code

Prefetching(Loop, alwaysExecuted, notAlwaysExecuted):

```
for all target ∈ alwaysExecuted do
    depsInPrevIteration = findDepsInPrevIteration(target, Loop)
    insertionPoint = soonestInsertionPoint(depsInPrevIteration, Loop)
    moveDefinitions(depsInPrevIteration, insertionPoint)
    cloneComputation(target, insertionPoint)
    addPrefetch(target, insertionPoint)
end for
diamondToTargetsMap = groupTargetsByDiamond(notAlwaysExecuted, Loop)
for all diamond ∈ diamondToTargetsMap.keys() do
    (diamondEntry, diamondExit) = getEntryAndExit(diamond, Loop)
    coroArgs = usedValuesDefinedOutside(diamond, Loop)
    coroReturns = valuesUsedOutside(diamond, Loop)
    (coro, args) = extractDiamondInFunction(diamond, coroArgs)
    replaceUses(coro, coroArgs, args)
    addReturnLoads(diamondExit, coroReturns)
    for all target ∈ diamondToTargetsMap[diamond] do
        resumingBlock = addPrefetch(target, coro)
        moveNonControlInstructionsInResume(diamondEntry, target, resumingBlock)
    end for
    addCoroutineCalls(coro, coroArgs, diamond, diamondEntry, Loop)
end for
```

10 Methodology

All experiments are run on a laptop under Ubuntu 22.04 with an Intel Core i5-10300H Comet Lake (2.5 GHz, 4 processors), 2x8 Go of DDR4 RAM at 2933 MHz and an M.2 NVMe PCIe SSD.

All runs of benchmark 1 and benchmark 2 are done on a list of $1,000,000$ elements and so the loops do 1 million iterations, all runs of benchmark 3 are done with an array of $20,000$ elements. The lists are randomly generated according to a seed that is given as an argument and is 0.5 for all experiments. To do our measurements, we ran the benchmarks 100 times measuring the execution times of the target function with $CLOCK\_MONOTONIC$, and computed an average execution time of these 100 runs.

To measure the execution times, we compile with clang and option **O3** to emit an LLVM IR then use opt with option **O3** on the IR, and finally recompile with clang and option **O3** to obtain a binary. For the measurements of the passes, after using opt, we use it a second time to apply our pass on the IR and then reuse opt a third time with option **O3**.

11 Results

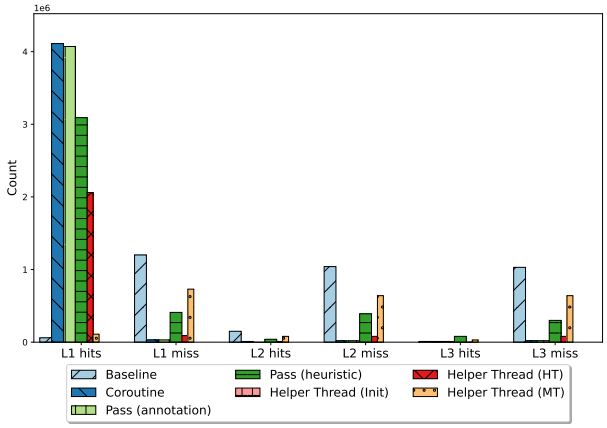
In this section, we present the results obtained through our manual prefetching generated during the space exploration and the results obtained through our automatic pass. We also added in appendix A the three experimental studies we performed related to the optimal values for the prefetch distance of benchmark 3's basic prefetching, the optimal chunk size for benchmark 1's helper thread, and the limit values for which benchmark 2's helper thread starts before the main thread and doesn't stop before the main thread. We then used the values obtained through these side studies for our measurements presented in this section.

Figure 19 showcases the results of our experiments. We decided to put forward mainly the execution time and the hits and misses and add in appendix B the CPU stalls because CPU cycle stalls are time lost and thus are covered by the comparison of the execution times.

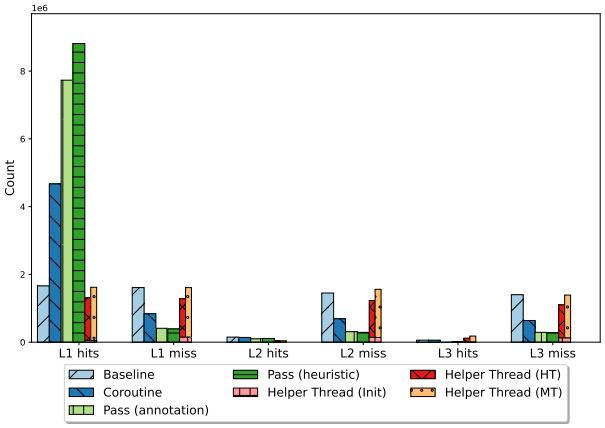
These figures point out that the coroutine method is more efficient than the helper thread, with better performances in every aspect. It brings more L1 hits, reduces the execution time more than helper threads, and leads to fewer misses. However, we can see on benchmark 3 that the coroutine is less efficient than the basic prefetching. That is because the basic prefetching is prefetching multiple iterations in advance whereas the coroutine only does one. Therefore, our coroutine-based prefetching seems good for prefetching dynamic structures but not interesting for non-dynamic ones.

Moreover, we can see that our pass is even more efficient than the manual version reducing the misses when using the IR annotations. However, it has a bigger overhead, benchmark 2 leads to a coroutine unlike the other two benchmarks but the manual version always implements coroutines so it has the overhead of a coroutine in any case. Therefore, benchmark 2 shows that the pass leads to more efficient coroutines but we do not see it on benchmark 1 because it doesn't create a coroutine.

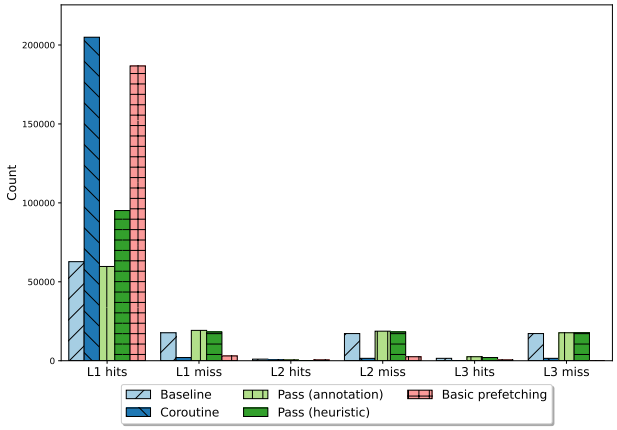
Finally, the last thing to analyze is the efficiency of our heuristic by comparing the pass with annotations and with the heuristic. The results put forward that the heuristic is working well on these three benchmarks as the execution times are very close to the pass with annotation. However, the hits and misses differ more. The pass with heuristic misses more than with annotation on benchmark 1 but has less overhead overall as the sum of L1 hits and misses is less than when using annotations. Additionally, on the other 2 benchmarks, the numbers of misses are similar between the two versions of the pass, but the pass with heuristic leads to more hits, which means that on these benchmarks, the overhead is higher. This means that the heuristic catches more targets than it should on these benchmarks and thus leads to more significant overhead.



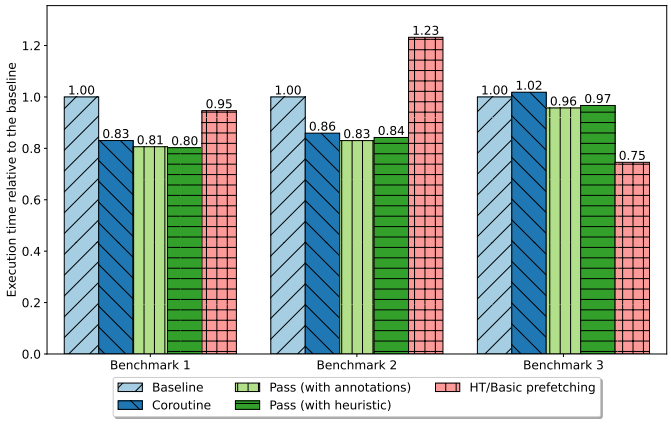
(a) Hits and misses for all versions of benchmark 1



(b) Hits and misses for all versions of benchmark 2



(c) Hits and misses for all versions of benchmark 3



(d) Relative execution time for all benchmarks

Figure 19: Result figures for all versions of each benchmark. "HT/Basic prefetching" in figure 19d refers to Helper threads for benchmarks 1 and 2 and Basic prefetching for benchmark 3. Additionally, CPU stalls measurements are provided in appendix B

12 Conclusion

To conclude, we developed a method to prefetch at compile-time the elements of dynamic structures parsed inside loops that may modify the structure and we obtained performance improvements ranging from 3% to 20% averaging at 13%. We have also implemented that method in an LLVM pass and tested it on the three benchmarks used as study cases for space exploration. Future work will focus on further

validating our method by testing the compiler implementation on a wider selection of benchmarks. Moreover, we experimented with a solution using a helper thread and presented the issues that this raised to prefetch dynamic structures that may be modified in the considered loop. We also proposed a heuristic to identify statically the delinquent loads of a loop iteration but once again did not have time to test it on more than the set of benchmarks used to elaborate it.

This work opens the way to develop methods to prefetch dynamic structures parsed in loops that can modify it using coroutines and an option on whether to use a heuristic (which in our pass can be changed easily) or use a profiling technique. Our work can also be perfected through experimental validation on more representative benchmarks. We also believe that there can be more relevant prefetching methods in cases where the loop is inside a function called multiple times and where the calls are independent in terms of execution order. In which case we could apply a method similar to the one presented in the figure 1. Another future work possible is considering dynamic structures parsed through the use of recursive functions.

References

[1] Christopher Jonathan et al. "Exploiting coroutines to attack the" killer nanoseconds"". In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1702–1714.

[2] Norman P Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers". In: *ACM SIGARCH Computer Architecture News* 18.2SI (1990), pp. 364–373.

[3] Taesu Kim, Dali Zhao, and Alexander V Veidenbaum. "Multiple stream tracker: a new hardware stride prefetcher". In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. 2014, pp. 1–10.

[4] Hao Wu et al. "Practical Temporal Prefetching With Compressed On-Chip Metadata". In: *IEEE Transactions on Computers* 71.11 (2021), pp. 2858–2871.

[5] Agustín Navarro-Torres et al. "Berti: an Accurate Local-Delta Data Prefetcher". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 975–991.

[6] Chi-Keung Luk and Todd C Mowry. "Compiler-based prefetching for recursive data structures". In: *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. 1996, pp. 222–233.

[7] Sam Ainsworth and Timothy M Jones. "Software prefetching for indirect memory accesses". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 305–317.

[8] Saba Jamilan et al. "Apt-get: Profile-guided timely software prefetching". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 747–764.

[9] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. "When prefetching works, when it doesn't, and why". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.1 (2012), pp. 1–29.

[10] Sanyam Mehta et al. "Multi-stage coordinated prefetching for present-day processors". In: *Proceedings of the 28th ACM international conference on Supercomputing*. 2014, pp. 73–82.

[11] Nishil Talati et al. "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 654–667.

A Sensitivity studies

A.1 Optimal prefetching distance for benchmark 3

Prefetching techniques that can compute the target load multiple iterations in advance require setting a suitable prefetch distance. In the case of benchmark 3, as the target load is loading an array cell, it is possible to do such computation in advance. Therefore, we study here which prefetch distance provides the best performance. Since identifying the prefetch distance statically is hard, we performed a sensitivity study of the execution times of the application and the target code regions, by varying the prefetch distance. The execution times are represented in figure 20 on which we can see that the best performance for the target function is obtained for a prefetch distance of 16. Similar results are obtained with prefetch distance 32.

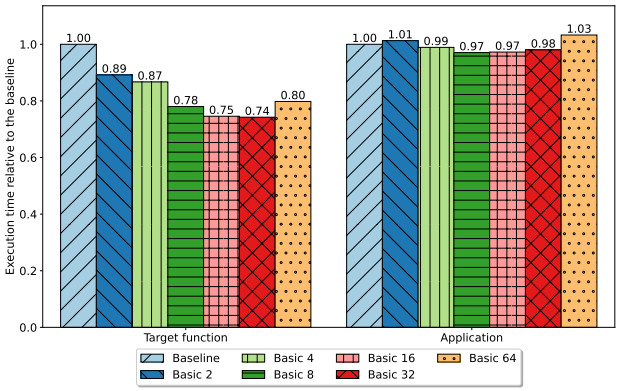


Figure 20: Execution time according to prefetch distance for benchmark 3

A.2 Experimental study for benchmark 1's helper thread's optimal chunk size

We issued also an experimental study to find the optimal chunk size. We, therefore, ran benchmark 1 with a helper thread for a range of chunk sizes from 2^3 to 2^{15}, taking only powers of two and jumping some. Figure 21 showcases our result and we can see that the chunk size has a fairly low influence for a large spectrum of values, from 2^3 to 4096 at least. We also see that choosing a too large size leads to an inefficient helper thread as the prefetching will not be timely and will simply slow down the code because of the initialization of the thread. Thus we chose to keep a chunk size of 8.

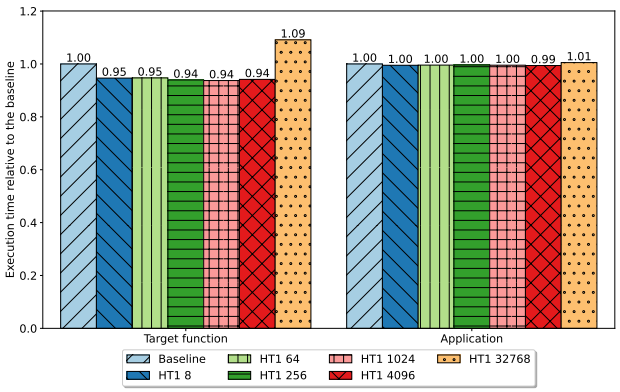


Figure 21: Execution time according to the chunk size for benchmark 1

A.3 Empirical study on the chunk size for benchmark 2's helper thread

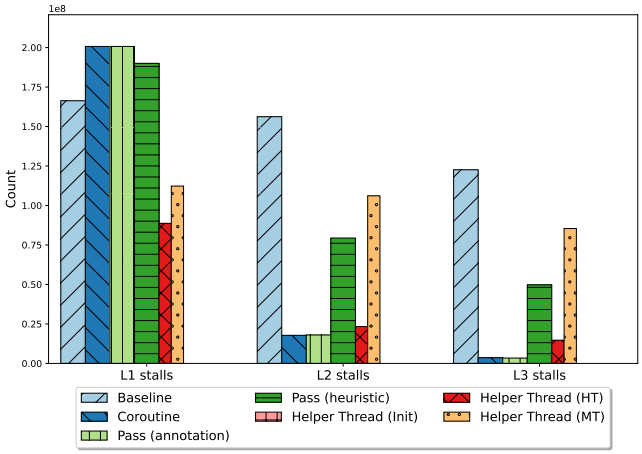
After finding out about the issues we mentioned in the space exploration section, we worked on empirically finding the minimal chunk size for which the helper thread starts its loop before the main thread in benchmark 2 as well as the minimal chunk size for it to complete all its iterations. In this case, the main thread never catches up on the helper thread because otherwise it would remove the element

from the list and the helper thread would stop as it considers having finished traversing the list.

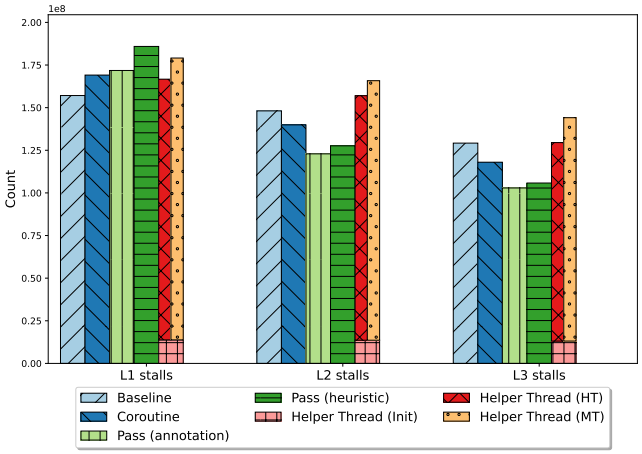
In the first study, we ran the application on different chunk sizes and printed the number of iterations done by the helper thread. This way we can see if it does more than one or not. Doing so means it starts its loop before the main thread computes the elements that have been sequentially prefetched during the helper thread initialization. This way we found out, by binary splitting on the chunk size, that the minimal chunk size is 263, for our architecture.

Doing the same experiment but until the number of loop iterations done by the helper thread is the one done by the main thread minus the chunk size because it starts after the initial chunk, we obtain a minimal chunk size of 60056.

B Complementary graphs



(a) Number of CPU stalls for all versions of benchmark 1



(b) Number of CPU stalls for all versions of benchmark 2

Figure 22: CPU stalls graphs for benchmarks 1 and 2

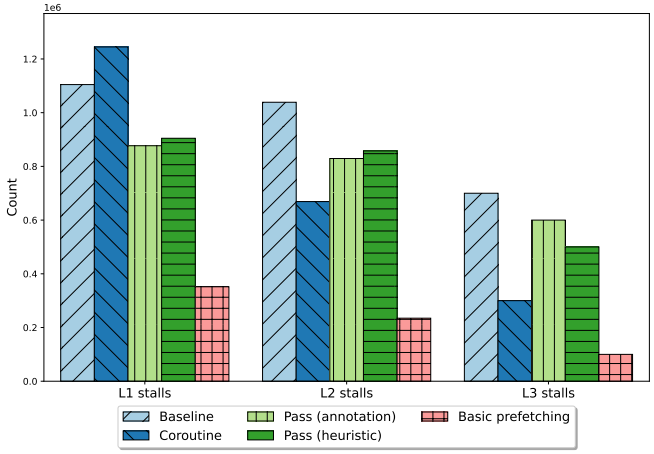


Figure 23: Number of CPU stalls for all versions of benchmark 3