Prefetching in Functional Languages

Sam Ainsworth
University of Cambridge
Cambridge, UK
sam.ainsworth@cl.cam.ac.uk

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

Abstract

Functional programming languages contain a number of runtime and language features, such as garbage collection, indirect memory accesses, linked data structures and immutability, that interact with a processor's memory system. These conspire to cause a variety of unintuitive memory-performance effects. For example, it is slower to traverse through linked lists and arrays of data that have been sorted than to traverse the same data accessed in the order it was allocated. We seek to understand these issues and mitigate them in a manner consistent with functional languages, taking advantage of the features themselves where possible. For example, immutability and garbage collection force linked lists to be allocated roughly sequentially in memory, even when the data pointed to within each node is not. We add language primitives for software-prefetching to the OCaml language to exploit this, and observe significant performance improvements a variety of micro- and macro-benchmarks, resulting in speedups of up to 2× on the out-of-order superscalar Intel Haswell and Xeon Phi Knights Landing systems, and up to 3× on the in-order Arm Cortex-A53.

CCS Concepts: • **Software and its engineering** → **Functional languages**; *Language features*; • **Computer systems organization** → *Superscalar architectures.*

Keywords: Software Prefetching, Hardware Prefetching, Functional Programming, OCaml

ACM Reference Format:
Sam Ainsworth and Timothy M. Jones. 2020. Prefetching in Functional Languages. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM '20), June 16, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3381898.3397209

1 Introduction

High-level, functional programming languages are becoming increasingly popular as ways to write terse, bug-resistant code, even in high performance settings [23, 54]. The memory access patterns of these codes differ from those in more traditional languages, such as C, due to core language features, such as immutability [41] and garbage collection [7, 37]. This tends to result in code that is very linked-list heavy, features significant reuse of existing, immutable objects, and large numbers of pointer-based memory accesses [11].

Studies exist on how more traditional programming languages affect the memory system and techniques to improve performance [3, 33, 56]. However, due to the very different memory-access patterns observed in functional languages [41], little is known about their resulting performance and memory-access times. The extent to which such code is memory bound has not been evaluated, and techniques to mitigate any issues functional programming causes within the memory system focus on theoretical analysis [41, 49].

We find that similar code patterns in functional programming languages can result in wildly different performance from the memory system. For example, computing over lists or arrays of sorted data is typically highly memory bound compared to running the same code over unsorted allocation-order data. Therefore, even when writing code in high-level languages, for good performance it is necessary to understand and consider the detail of the memory-access pattern. We address this by providing techniques to mitigate performance deficits in badly performing cases via software prefetching, which has been employed successfully in other languages [3, 18]. This improves functional-language performance, because although memory accesses are irregular, they are often predictable, and so these non-blocking loads can be used to bring data into the cache before it is required.

We add software-prefetching primitives to the OCaml language and standard library [1], and evaluate these on list and array code from the OCaml standard library, sorting algorithms, and a new benchmark suite covering diverse memory-bound behaviour, showing speedups of up to 3×. We also analyse the utility of hardware prefetchers for functional languages, and discover the surprising result that they are more useful for linked-list workloads than the array workloads that are more typical in imperative languages, as no other part of the system can effectively speculate on their structure, making hardware prefetchers essential for efficient high-level language implementations.

2 Background

This paper makes observations and develops techniques based on microarchitectural behaviour of programs in functional languages. The relevant features are summarised here.

2.1 Functional Programming Languages

The features and implementations of functional languages have important implications on the memory system. When we give examples, we use OCaml, but the concepts also apply to languages such as Haskell, Standard ML and Lisp.

Linked Lists Linked lists are more common in functional languages than imperative [37], and are usually directly built into the language. This is because such languages typically feature immutable data [41], and so arrays are unsuitable. Linked lists cause several issues from a memory perspective [28]. The pointer after each data item results in a significant memory overhead, and potentially allows data to be spread across memory, resulting in poor performance [21, 29, 47]. Linked lists also only allow sequential, rather than random, access [48], which reduces the memory-level parallelism available: we can only access the next element once the current one has been loaded.

Garbage Collection High-level languages typically use garbage collection to avoid manual memory deallocation by the programmer, and this can impact memory layout and thus performance. Implementations of OCaml and many other languages use generational garbage collectors [7, 37], which allocate data in a small, minor heap, where allocation is simply decrementing a pointer. Once this is filled, data is moved into a larger major heap, reordering it in memory.

Objects For many languages, including OCaml (and Java [38], where due to generics [12, 45] even primitive values such as integers are often accessed via object indirection [46]), in structures such as arrays and linked lists, primitive types such as integers are allocated in-line, whereas compound objects are allocated separately, and indirected to via a pointer. The added indirection can be a cause of poor performance [12], depending on the details of how these are allocated.

Immutability Functional languages often enforce immutability on objects, to make programs easier to reason about, and allow partial reuse of recursive data structures [41]. This can make memory accesses more regular. In languages with mutable linked lists, elements can be inserted in the middle of existing linked lists, resulting in irregular accesses when the list is traversed. However, in languages such as OCaml [37], while we can reuse the tails of old lists, anything before an inserted object must be reallocated. This results in fewer discontinuities, and thus more effective traversal.

Still, the reuse of objects allowed by immutability can be a double-edged sword. The layout of such objects may be suboptimal for the traversal of the data we are to use (for example, if the data is sorted after being allocated). We shall see that this can cause very poor memory performance.

2.2 Computer Architecture

The hardware that applications run on is as important as their programming-language features. Here we describe architectural techniques that affect memory accesses.

Prefetching Regular memory accesses, where data is accessed with spatial locality, are good for performance on modern systems. This is because the memory system fetches a cache line, typically 64 bytes of data [26], at a time, and also because the hardware prefetcher [20, 21, 27, 50, 53] can pick up and predict such patterns from address sequences. However, when objects are scattered throughout memory, performance can still be improved by software prefetching [18]: we can insert non-blocking load instructions to access data that the programmer predicts will soon be accessed.

Out-of-Order Superscalar Processors Modern processors can reorder memory accesses to an extent, provided they are independent. An out-of-order superscalar processor features a reorder buffer [32, 51], where currently active instructions reside. If these instructions don't feature dependencies between each other, then they can execute concurrently.This allows form memory-level parallelism, which can help hide high main-memory latencies. However, it is dependent on there being enough independent loads in the reorder buffer at once. If the reorder buffer is small, if data dependencies prevent parallelism, or if other instructions within the reorder buffer restrict the loads available to run simultaneously, then performance can be limited. This is why prefetching is also needed for the highest performance [3, 20, 21, 53].

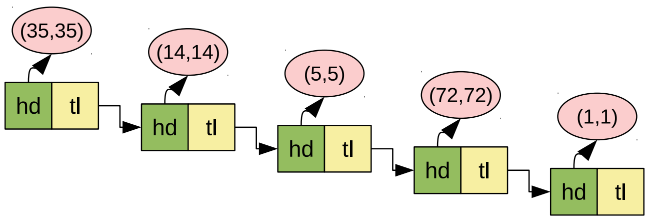
3 Motivation

Fundamental features of functional languages can result in a lack of spatial locality in the memory accesses of programs written in them, causing severe performance losses. Here we explore where these factors appear, and the extent to which performance is reduced, before moving on to consider how we can mitigate the problem in the next section.

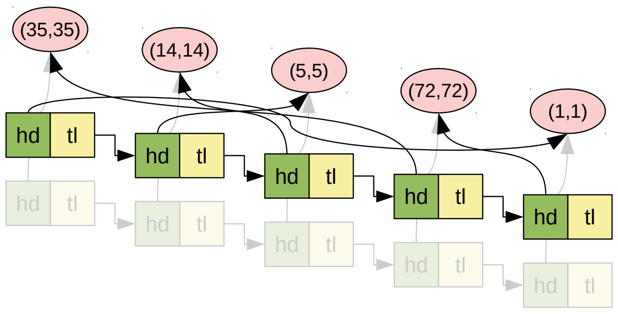
3.1 Spatial Locality

Figure 1 shows graphically how the memory system can cause sub-optimal performance for a functional language, for two data structures: linked lists and arrays of tuples. Boxes that are joined together indicate consecutive memory locations, which feature spatial locality and so are likely to perform well with respect to the memory system. Arrows indicate pointer indirections, whose targets could be placed anywhere in memory. However, a good allocator should place objects that have been created at similar times close to each other. Hence in practice we may find that the pointer indirections have spatial locality and also perform well.

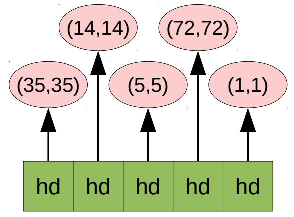
Figure 1(a) shows a linked list of tuples just after creation. Each tuple is located close to its list node, and list nodes are located close to one another. Traversal of the list follows a regular pattern of pointer accesses and memory locations. Figure 1(c) shows the same concept for an array. However,



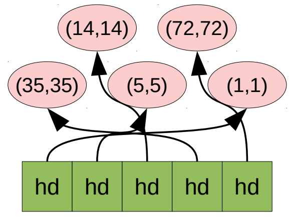
(a) List of tuples in allocation order



(b) Sorted tuple list; old garbage-collected list grayed out



(c) Tuple array (alloc. order)



(d) Sorted array of tuples

Figure 1. The memory structures of both linked lists and arrays of tuple pairs, before and after sorting.

after sorting the list, shown in figure 1(b), new list nodes are created but the tuples are unchanged, so the pointers target memory locations seemingly at random. Traversal of this list's data now no longer follows a regular pattern, and thus memory performance will be poor even with a good allocation strategy. In the second example (figure 1(d)), no new array is allocated, since arrays are mutable, but the pointers to tuples again no longer follow a regular pattern.

These irregular access patterns cannot be predicted by the memory system, causing significant bottlenecks in relation to their allocation-ordered counterparts.

3.2 Analysis

Figure 2 shows how long adding together all of the elements of sorted versus allocation-ordered lists and arrays takes, using a left fold: the two-operand sum operation is applied to both the current list or array element, and all elements seen so far summed together. We see that, despite performing the same work, it takes over 6× the time to calculate the sum of either lists or arrays when sorted. Surprisingly, this is still the case when we make the workload more compute-intensive by adding in a hash calculation per element (*hash-sum* bars). While we may expect the computation to become

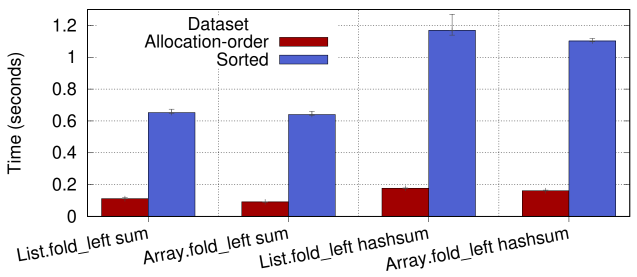


Figure 2. The time taken for adding together large lists and arrays of tuples in Ocaml, sized to fit in DRAM but not the cache, using a left-fold operator, and varying the sum function to either add the numbers together, or to perform a hash of them, involving more computation.

less memory-bound, due to the extra ~0.06s of compute performed, in fact this compute expands to add an additional 0.5–0.6s of time compared to the same sorted-data benchmarks without the hash computation.

The code run is identical. The performance drop is caused by a low instructions per cycle (IPC) metric, in turn caused by a large proportion of the program's loads missing in all caches, forcing a high-latency access to RAM. When the data is sorted, the array and linked-list accesses are still roughly sequential, as the array is reused and the linked-list elements reallocated for immutability. However the data pointed to by each array or linked-list element is now accessed non-sequentially in memory, with a large performance hit.

For simple workloads, the out-of-order hardware in the processor can mask some of the overheads, by fetching multiple cache misses simultaneously, but when we add in extra computation, even a simple hash function, the out-of-order hardware's issue queue becomes clogged with instructions, leaving less space for many concurrent loads, and so memory-level parallelism, and performance, go down by far more than the extra time to perform the additional instructions.

Still, we can put the low IPC of the cores to good use. Instead of accepting the high cache miss rates, we can use the opportunity to issue non-blocking loads, to anticipate the cache misses before they occur, and reduce the performance impact associated with reordered data. The next section introduces software prefetching primitives into the OCaml compiler to achieve this, before demonstrating how we can apply this technique to arrays and lists of data.

4 Software Prefetching

A solution to memory-bound code that is traditionally used in imperative languages is software prefetching [18], where non-blocking loads for addresses that will be accessed in a few iterations' time are issued early. To explore the utility of this within a functional language, we have designed primitives and composite techniques for prefetching and added them to the OCaml compiler, using *prefetch* intrinsics. These have no side effects save for performance, and are lowered

```
1  val prefetch : 'a -> unit
2  val Array.prefetch: 'a array -> unit
3  let fold_left f x a =
4    let r = ref x in
5    for i = 0 to length a - 1 do
6      prefetch (unsafe_get a
7         (min (i+16) ((length a) - 1)));
8      Array.prefetch a (i+32);
9      r := f !r (unsafe_get a i)
10   done;
11   !r
```

Code listing 1. An example of prefetching on *Array.fold_left*. We prefetch the object at a given array index (line 6), and the array index itself before we load from it (line 8).

to software-prefetch assembly instruction for compound objects, or no-ops for base types. Using a basic *prefetch* function, and two further intrinsics specialised for typical memory layouts, we design high-level constructs for linked lists and arrays to improve the performance of memory-bound code.

4.1 Prefetch Intrinsic

We add the *prefetch* function as a builtin intrinsic into OCaml. The prefetch function takes in any type. If the input is a boxed (indirected to by pointer) object, it is used as input to an assembly prefetch instruction. In the example given in code listing 1 in blue (line 6), we load the pointer we will use in 16 elements' time, then prefetch the referenced cache line.

The intrinsic is lowered through the bytecode as a `Prefetch` primitive. This is ignored by interpreters, but for the optimizing compiler, Ocamlopt, the backend generates as assembly a `prefetcht0` instruction for X86, and a `prfm pldl1keep` for Aarch64. Both of these are prefetches that hint that the data should be 1) brought into the L1 cache, and 2) it is temporally local, and so should be kept in the cache rather than immediately evicted once used. Neither of these assumptions may apply in many common use cases, but the performance difference between these and other versions is negligible, and so a single intrinsic simplifies the programmer interface without affecting speedup. On a fault both are ignored.

Our other techniques, array prefetching and offset prefetching, are similar but more specialised. Array prefetching takes two arguments, which are lowered into a prefetch for an offset into an array address, and offset prefetching is lowered into an offset from an arbitrary pointer; the former allows more type checking at higher levels of the compiler.

4.2 Prefetching from Arrays

We first discuss looking ahead in the array and prefetching the pointers found using the prefetch function, before giving a more complicated technique involving access to the array representation itself. The relevant patterns to prefetch from arrays have already been explored in detail for other languages [3]. For functional languages, we need to consider the additional impact the overheads introduced by pointer indirection on all objects [11] causes, resulting in increased irregularity of memory access and thus prefetching potential.

Basic Prefetching The standard prefetch function on an array works similarly to in imperative languages [3]. In the blue line (6) in code listing 1, we first look ahead in the array to the element we will access in 16 iterations' time. This value, determined experimentally, is set such that the data will have arrived from main memory by the time we need it, but will not yet have been evicted from the cache due to capacity misses; still, the same "look-ahead" values work across workloads and systems, so the programmer need not tune these directly. The "min" function should be implemented directly for integers, rather than the default polymorphic OCaml comparator, to avoid unnecessary computation. As we iterate over the entire list, we also include a check to make sure we don't load over the end of the array: while a prefetch over the end would not cause any issues, a true load would fault, and we do perform a true load to access the element pointer which we will prefetch. In this example, we load the minimum element of the lookahead value and the array length, as this can be lowered down into branch-free execution in assembly, but `if` expressions to conditionally call the prefetch intrinsic could also be used.

Array-specialised Prefetching Basic prefetching does not target all data structures, and so some performance is left on the table. To gain maximal benefit, we also need to prefetch the array indices we will access in a few iterations' time, staggered so that they will also be available for the basic prefetch. For OCaml, we only have load/store access to arrays, and cannot take the address of each element, as addresses cannot be directly computed. Even though the hardware prefetcher partially achieve this, as we shall later see in figure 11, it is insufficiently aggressive for the patterns targeted here, being tuned for dense, regular access patterns.

To solve this we extend the Array library with a further intrinsic. Rather than taking the address of an element within an array and passing that to our existing prefetch function, we instead implement *Array.prefetch* in the compiler, which works similarly to *Array.get* in OCaml, but returns `unit`, and is lowered to a non-blocking prefetch in the assembly instead of a blocking load. This means the programmer isn't exposed to the underlying data representation in the process. An example is shown in red (line 8) in code listing 1.

4.3 Prefetching from Linked Lists

While in general linked-list nodes may be distributed throughout memory [28], in OCaml they tend to be laid out almost sequentially due to a combination of the copying garbage collector and immutability preventing significant reordering. This gives an opportunity to prefetch in code using linked lists where it wouldn't normally exist, since walking the list is relatively cheap, whereas when code is memory bound, fetching data pointers referenced by the list is expensive. We

```
let prefetch_list l = match l with
 | x::y::z::aa::ab::ac::ad::t -> prefetch(ad)
 | _ -> ()
let rec fold_left f accu l =
  match l with
   | [] -> accu
   | a::l -> prefetch_list l; fold_left f (f
        accu a) l
```

Code listing 2. Simple linked-list prefetch for *List.fold_left*.

```
val prefetch_offset : 'a -> unit
let prefetch_prefind l = match l with
 | x::y::z::aa::ab::ac::ad::t ->
     prefetch(ad); t
 | _ -> []
let prefetch_prefound m = match m with
 | x::ys -> prefetch(x);ys
 | _ -> []

let fold_left f accu l =
  let rec flpf f accu l pfl =
   match l with
    | [] -> accu
    | a::l -> prefetch_offset l (-256);
      (let pfr = prefetch_prefound pfl in
      flpf f (f accu a) l pfr) in
      flpf f accu l (prefetch_prefind l)
```

Code listing 3. Complex linked-list prefetch code, along with offset prefetching, applied to *List.fold_left*.

can look ahead in the list to increase memory-level parallelism by prefetching multiple data pointers at once.

As with arrays, there is a trade-off between performance and implementation complexity. We first present a *simple* scheme that can be inserted in-place, before presenting a *complex* scheme that changes function arguments, then finally presenting an *offset* technique that speculates assumptions about the allocator to maximise performance benefit.

Simple List-walking Prefetching Code listing 2 gives an example of prefetching objects within each list. On each recursive call, in addition to the original computation, we walk the list to prefetch several elements down the chain.

Complex List-storage Prefetching The simple scheme for linked-list prefetching, while unobtrusive to add to code, is wasteful. We load multiple elements of the linked list each time because we must walk sequential elements to look ahead, unlike with arrays, and could avoid doing this work. Code listing 3 shows how to do this. Before executing the function, we walk ahead in the linked list and return the tail. Then, during execution, we need only prefetch the head of the tail, and return the new tail for the next prefetch. This alters function arguments, but reduces redundant list walks.

Table 1. System setup for the three processors evaluated.

System	Specifications
Haswell	Intel Core i5-4570, 3.20GHz, 4 cores, 32KB L1D, 256KiB L2, 8MiB L3, 16GiB DDR3
A53	Odroid C2, Arm Cortex-A53, 2.0GHz, 4 cores, 32KiB L1D, 1MiB L2, 2GiB DDR3
KNL	Intel Xeon Phi 7210 CPU, 1.30GHz, 64 cores, 32KiB L1D, 1MiB L2, 196GiB DDR4

Offset Prefetching For arrays, prefetching array locations as well as objects improves performance. However, we can't directly apply the same strategy to linked lists as we can't look ahead past the next element without loading all previous elements. Still, linked lists in OCaml typically end up being allocated almost sequentially in memory. This brings an opportunity to speculatively prefetch addresses close to the one being accessed. In a language like C, where direct access to the addresses of objects is available, it would be possible to do this with a simple prefetch instruction. However, since this isn't allowed in OCaml for memory safety reasons, we must add a new prefetching primitive that prefetches with an offset from a given location. This function, which takes in an object and an integer offset value, issues a prefetch to the address at `offset` words away from the object. This is also shown in code listing 3, and while it may seem slightly low-level for a functional language, the programmer is unable to directly access memory addresses, and it is impossible to cause memory faults using this techniques, since prefetches are speculative and non-faulting.

When walking through a linked list we prefetch backwards from the current element. This is down to the way OCaml allocates memory, and has the potential to change based on specifics of memory allocation and garbage-collection style. This means offset prefetching makes more assumptions about the runtime. Still, offset prefetching is low cost as it requires no loads to be performed, unlike simple or complex prefetching, and so can be achieved in two instructions: one arithmetic offset and one prefetch instruction.

4.4 Summary

We have introduced three new primitives—object prefetching, array prefetching, and object offset prefetching, and associated high-level techniques using them—and now consider how these apply to speed up memory-bound workloads.

5 Experimental Setup

We add software prefetching capability to the OCaml optimizing compiler, Ocamlopt [1]. This uses a two-phase generational garbage collector, with default 512kiB minor heap and variable-sized larger major heap. The minor heap uses bump allocation with copy collection, whereas the major heap uses a free-list allocator and stop-the-world mark-and-sweep collector [37]. We use the default next-fit policy in the major heap allocator – the most likely to cause fragmentation, and least favourable to offset prefetching out of the three options (next-fit, first-fit and best-fit). We shall see that offset prefetching still succeeds, because the prefetches do not always need to be correct to improve performance. Incorrect prefetches are innocuous to performance provided they bring in few new cache lines, since arithmetic and prefetch instructions are cheap (.25 and .5 cycle throughput on Haswell, respectively [22]), and the evaluated memory-latency bound workloads are unhindered by additional cycles and memory bandwidth consumed by mild inaccuracy.

We are interested in the behaviour of memory-bound workloads, and so we execute on 100MiB of raw data (excluding overheads from data structures, such as linked-list pointers or pair data), sized to fit in DRAM but not in the last-level cache. On A53, this does not fit in the on-chip DRAM in some cases, so data sizes are reduced until they fit. In general, all data sizes that do not fit in the cache show similar improvements from prefetching, so the overall results are unchanged, and results apply for any reasonably-sized dataset. This is also larger than the minor heap, and so the data will go through some reordering before execution from the generational garbage collector. Input data is generated using a pseudorandom generator [42], such that it will be significantly reordered when sorted.

We evaluate on Intel Haswell, Intel Knights Landing (KNL) and Arm Cortex-A53 processors, all of which are shown in table 1. Haswell is an aggressive out-of-order superscalar core; KNL is mildly out-of-order superscalar, and A53 is in-order superscalar. All feature hardware prefetchers, which are enabled unless stated. We run benchmarks ten times, taking the mean, maximum and minimum speedup.

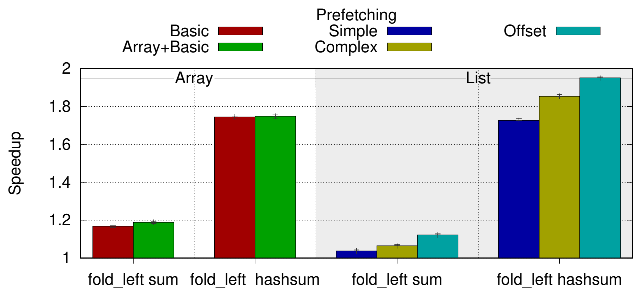
6 Evaluation

Having developed prefetching techniques on arrays and linked lists, in this section we apply these techniques to code from the OCaml list and array libraries, before looking at more complex examples including sorting algorithms and a benchmark suite of diverse memory-bound workloads. We then finally look at tuning information, performance in circumstances that are less memory bound, and evaluate the utility of the hardware prefetcher for functional programming languages, by customising the code input to the list and array libraries to a variety of different setups.

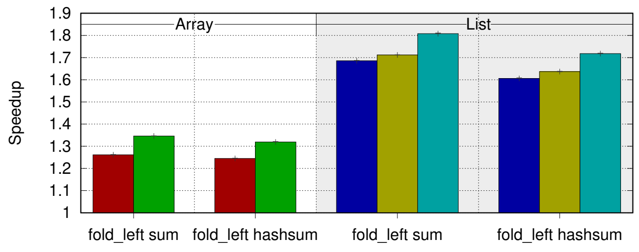
6.1 Fold_Left

First we look at speedup on the left fold examined in section 3; the techniques from section 4 are shown in figure 3.

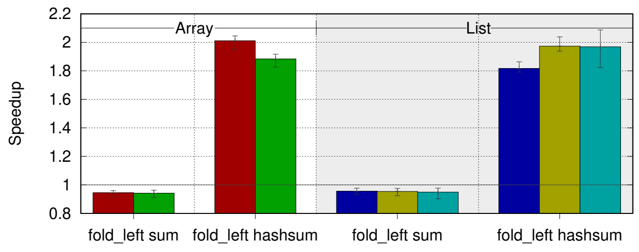
Haswell For *Array.fold_left sum* in figure 3(a), despite being heavily memory bound, basic prefetching adds a small speedup (1.17×), increasing to 1.19× when we use our array specialisation that also fetches the sequential array element. While this is still a notable improvement, it is limited because



(a) Haswell



(b) A53



(c) KNL

Figure 3. Performance improvement for prefetching fold_left on both arrays and lists using each of the techniques described in section 4, for the systems in table 1. Results for all systems and benchmarks are based on tuning results optimal for fold_left on Haswell.

the extra prefetching code is relatively large, and thus expensive to run, and also because the out-of-order hardware can already parallelise the memory accesses to an extent. However, the same code with some added computation (i.e., *Array.fold_left hashsum*) tells a very different story. Without software prefetching the extra compute code significantly increases execution time compared to *Array.fold_left sum* (0.64s to 1.1s in figure 2); with prefetching the difference between the two is negligible, with speedup of 1.75×. Software prefetching allows us to run more compute code in between each cache-missing load, while still allowing us to parallelise loads. Since the out-of-order hardware reduces in effectiveness with more compute code, as it is able to execute fewer loads together, prefetching shows larger benefits.

Performance improvements for the list-based left fold are comparable to those for the array-based functions, despite very different access patterns and implementation. The performance improvement for complex prefetching, where the prefetched list is only walked once, relative to the simple scheme, where it is walked many times, is slight. The irregular-object memory accesses dominate execution time, and thus the repeated walking of the list isn’t much more expensive than the complex technique. Offset prefetching improves linked-list performance more than array prefetching does for arrays. Although offset prefetching depends on the allocator and is less reliable, the hardware prefetcher and out-of-order scheduler are less capable of predicting the pattern, and so software techniques deliver more improvement.

Arm Cortex-A53. Results for the in-order superscalar Arm Cortex-A53 (table 1) are given in figure 3(b). We see that the performance improvements are largely comparable despite the very different systems, but with software prefetching being more useful in more cases on the A53.

Software prefetching is useful on A53 even when the computation between each recursive call is minimal. This is because there is no out-of-order execution to dynamically overlap memory accesses. The same offsets used on Haswell are close to optimal, despite the very different architectures.

The data shows that the prefetching techniques developed for out-of-order superscalar systems are widely applicable across different architectures. Indeed, in-order systems require these techniques even more for high performance.

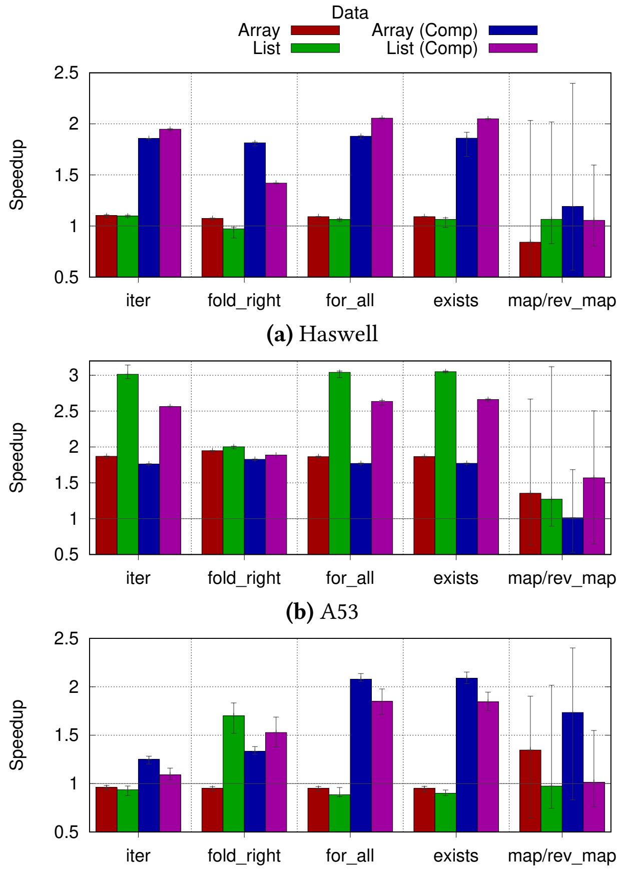
Intel Xeon Phi Knights Landing (KNL). The many-core Xeon Phi Knights Landing system (also given in table 1) is based on small, albeit out-of-order, superscalar cores. This places each core somewhere between the A53 and Haswell in size and performance, and this halfway house exposes a number of interesting effects from prefetching.

The results of performing the same experiments on KNL are shown in figure 3(c). For the cases with added hash computation, we see higher performance improvements than on both of the other systems: a memory system designed to allow lots of memory-level parallelism combined with a processor only able to extract small amount of it by itself combine to create high utility for software prefetching.

However, we also see that in other cases, prefetching can significantly reduce performance, unlike on A53 and Haswell. This is because the system is both less memory bound than A53, due to being mildly out-of-order, and yet more compute bound than Haswell, due to being not out-of-order enough. Hence the compute cost of working out prefetch addresses is a dominant factor on such an architecture. Whereas smaller systems are so memory bound that almost any intervention improves performance, and larger systems are able to deal the overheads though scheduling from a large instruction window, the benefits and penalties are magnified on KNL.

6.2 Library Prefetching

We now look at using the best techniques (array and offset prefetching) on further workloads. We extend the OCaml list and array libraries to add prefetching by using OCaml’s



(c) KNL

Figure 4. Speedup for functions from the OCaml list and array libraries, with prefetching. Error bars show the range of results: high variance (e.g. map/rev_map) is caused by the garbage collector, independent of prefetching.

dynamic type-checking feature to see if the data is a pointer or not at run time, at the start of a library call, then use that to direct to the appropriate prefetching function. Through this, programmers automatically obtain prefetching benefits.

Figure 4 shows performance for each technique on functions from the OCaml list library applied to sorted lists of pairs. As previously, we add extra computation based on a hash (displayed in the graphs as the *Comp* bars), to model cases with more instructions per memory access.

Performance improvement is broadly similar to that for *fold_left*. Lists typically receive a larger speedup than arrays do, as even though the prefetching technique is more expensive, the functions are more memory bound. Fold_right is an exception here on Haswell: as the function accesses the data pointed to by its list elements in reverse order, we must prefetch it by storing the elements we will access, then replay them backwards. This is expensive, reducing performance gains on Haswell, though increasing them on KNL compared to other functions, due to how memory bound the code is.

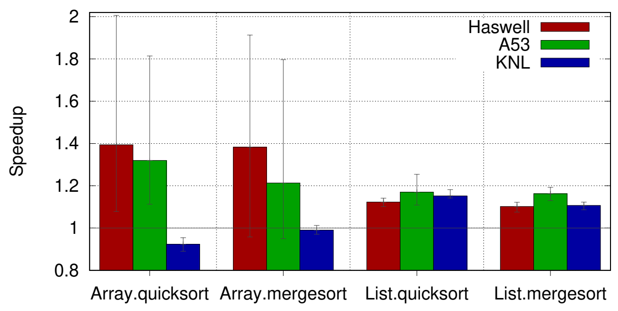


Figure 5. Sorting algorithm speedup on the test systems.

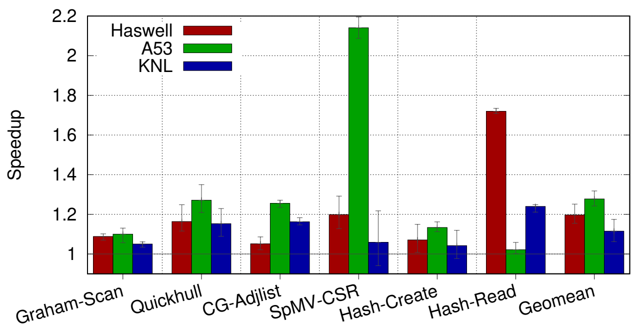


Figure 6. Speedup for our OCaml memory-bound suite.

6.3 Sorting Algorithms

It is not only functions that execute on sorted data that access data in a different order from its allocation. The sorting process itself reorders data multiple times in intermediate stages. This makes such algorithms a good target for prefetching. We focus on two particular sorting algorithms: the OCaml standard library's [1] Mergesort functions on both linked lists and arrays, our own, fast implementation of Quicksort on linked lists, and an implementation of Quicksort for arrays from the OCaml compiler's test suite [1].

Mergesort repeatedly combines sorted lists or arrays. We can prefetch within the list merge function, looking ahead in the sublists to bring in future elements. A notable feature here is that, unlike in previous examples, we have to prefetch from two lists or arrays at the same time. As the list we access next in each case is data dependent, we prefetch the element of the list we have just accessed. This means that the next few elements of both lists are always prefetched, and that we only have to prefetch from one list for each recursive call of the function. Though Quicksort also operates on lists or arrays, the complex memory accesses exist within a partition function, which instead of reading multiple separate lists at once, creates multiple lists from a single list.

Figure 5 shows the performance improvement from using prefetching in each case. Though the improvement is significant for something as fundamental as sorting, it is smaller than the speedup previously gained on some library functions. This is for two reasons. First, only a fraction of the time of the Quicksort and the Mergesort are spent actually doing partitioning and merging, and thus the proportion we are attempting to speed up is smaller than in the other list-library functions. Second, there is limited computation between each recursive call, and thus Haswell and KNL are partially able to parallelise memory accesses themselves.

6.4 OCaml Memory-bound Benchmark Suite

Having seen how software prefetching performs on a variety of kernels, we now look at its behaviour on workloads solving real-world problems. Due to a lack of publicly available code in OCaml, we have developed a new set of benchmarks: the OCaml memory-bound benchmark suite, described in table 2. This is designed to reflect imperative and functional use-cases, be based on existing, widely applicable code but with timeable outcomes and with defined inputs, and represent workload classes that are memory-latency bound.

Figure 6 shows the speedups attainable through prefetching on each system. Though the same configurations are used for all systems, every workload consistently is sped up by the addition of software prefetching. However, the magnitude to which this is the case can vary strikingly between systems, such that workloads with the highest speedup on one system may have the lowest on another.

In terms of list-based benchmarks, Graham-Scan [52] is improved in performance just from its use of List.mergesort, described in section 6.3, which accounts for a large proportion of its execution time. Quickhull, by comparison, gains a larger speedup on each workload; although it requires custom prefetching to do so (not being based directly on the list library), almost the entire execution can be prefetched.

The CG-Adjlist [44] workload is sped up most significantly on A53, with a very small speedup on Haswell. This is because Haswell with its out-of-order execution is able to reorder and parallelise the simple indirect memory array accesses created by using the adjacency list representation's edges as indices, whereas the A53 is unable to do this. This is magnified in SpMV-CSR, where A53 gets a very large benefit (2.15×), for an indirection on a more compressed CSR representation of a graph. KNL can only gain limited benefit because it is less memory-bound than A53, yet the extra software prefetching computation is expensive, unlike for Haswell, which is superscalar enough to hide the overheads.

Hash-Create [25] features a small but positive increase in performance in all cases. This is because adding elements to a hash table is dominated by the cost of memory allocation, which prefetching neither helps nor hinders, though the ability to prefetch the memory accesses we will store to within the hash table itself can still extract some benefit. By comparison, the reading back of elements, which requires no memory accesses, results in a significantly larger speedup on Haswell and KNL than on A53, reversing the trend seen for SpMV-CSR. This is caused by the nature of hash indexing,

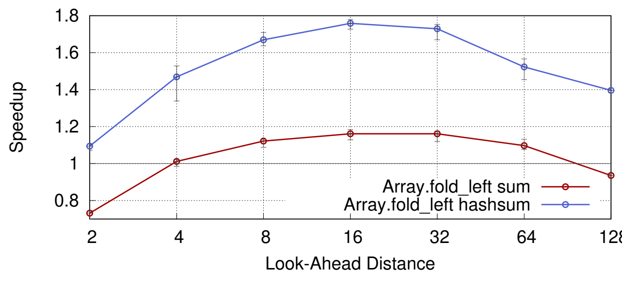
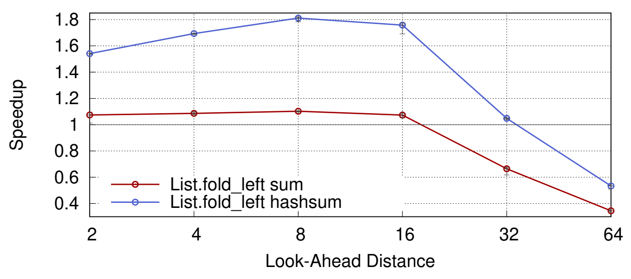
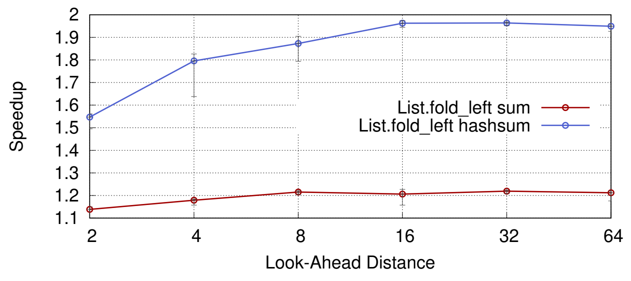


Figure 7. Performance impact of varying look-ahead for the basic prefetch to the tuple, for the array-based left folds.



(a) Simple Prefetching



(b) Complex Prefetching

Figure 8. Performance impact of varying the look-ahead distance for both basic and complex prefetching to the tuple, for the list-based left folds.

which requires a relatively large amount of computation per memory access. On A53, this results in a simple outcome: the program is less memory-bound as it does more computation. By comparison, for the out-of-order Haswell and KNL, this computation blocks up the reorder buffer, meaning the cores are less able to use this out-of-order resource to issue multiple memory accesses at once. On Haswell, we can regain this parallelism through software prefetching, which is also true on KNL, though the benefits are less pronounced due to KNL having fewer functional units, resulting in compute-bound behaviour not exhibited by Haswell.

6.5 Look-ahead Distances

Prefetching can be sensitive to the look-ahead distance chosen: after how many future elements should we start to prefetch? Too low a look-ahead and misses will already have occurred before data arrives in the cache. Too high a look-ahead, and prefetched data will be evicted before use.

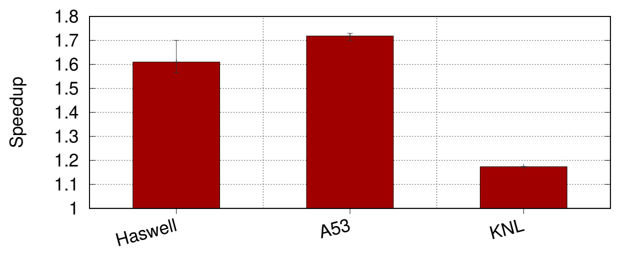


Figure 9. Speedup for prefetching *List.nth* on each system. Performance improvement remains consistent regardless of the linked list type, as the linked list data is not accessed.

For the array workloads, in figure 7 we observe an optimal distance of 16, which is similar to results on C-like languages [3]. Still, values of 8, 16, 32, and 64 give positive improvements, and so we have leeway in setting this value.

Results for the list workloads are given in figure 8. Speedup for complex prefetching is considerably flatter: all values improve performance, with an optimum at 32. By comparison, simple prefetching gives a smaller optimal of 8, followed by rapid performance degradation. This is because the latter represents a different trade-off, in that for simple prefetching on lists, a longer look-ahead distance implies more work, as each prefetch requires walking through every linked-list node within the look-ahead distance.

Here we show tuning for left folds on Haswell: the same values optimised here are used throughout the paper on all workloads and benchmarks. This is because numbers presented here are relatively consistent regardless of microarchitecture and benchmark, as previously observed [3].

6.6 Other Prefetching

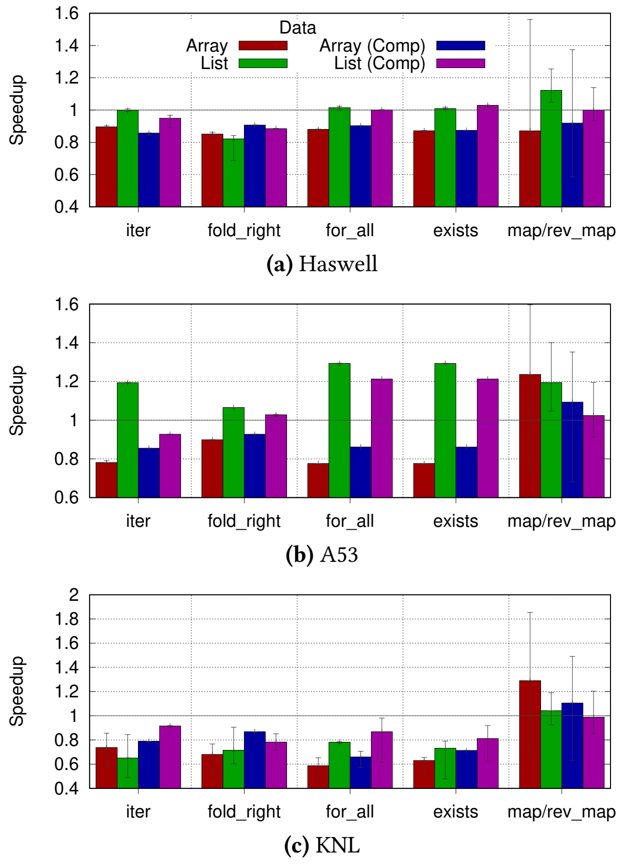
We have considered how prefetching can be used to improve the performance of sorted data in the OCaml list and array libraries, along with the sorting process itself, and a variety of more macro-scale workloads. We now consider cases that involve only linked-list walking, and no extra data access, along with how using the same prefetching code targeted at sorted data affects unsorted allocation-ordered data, and how the hardware prefetcher affects performance.

List.nth Within the list library, *nth* is somewhat of an outlier, as it doesn't access the data items it iterates through, it only walks through the linked list itself. This means there is very little computation between a list node being accessed and moving on to the next node, and computation time does not depend on the datatype of the list elements.

While the memory-access pattern of walking the linked list will be approximately sequential as we have argued previously, this limited computation means that the hardware prefetcher will typically be too conservative. We can achieve higher performance by using software prefetching, via offset prefetching alone into the linked list itself. This is shown in figure 9, where we achieve speedups up to 1.7×.

Table 2. The workloads of the OCaml memory-bound benchmark suite.

Workload	Source	Prefetching	Description
Graham Scan	[52]	List Complex, Offset (via sorting algorithm)	Calculates the convex hull of a list of integer pairs, by sorting the points then categorizing them.
Quickhull	[10]	List Complex, Offset	Calculates the convex hull of a list of integer pairs, using a quicksort-style divide-and-conquer approach.
CG-Adjlist	[44]	Array, Basic, Offset	Conjugate-gradient solving for graphs in adjacency-list format.
SpMV-CSR	PageRank [43], NAS-CG [9]	Array, Basic	Performs sparse matrix-vector multiplication on graphs based on an efficient compressed sparse-row (CSR) representation.
Hash-Create	[25]	Array	Times the creation and filling of a large hash table.
Hash-Read	[25]	Array, Basic	Times reading all the elements of a large hash table.



(a) Haswell

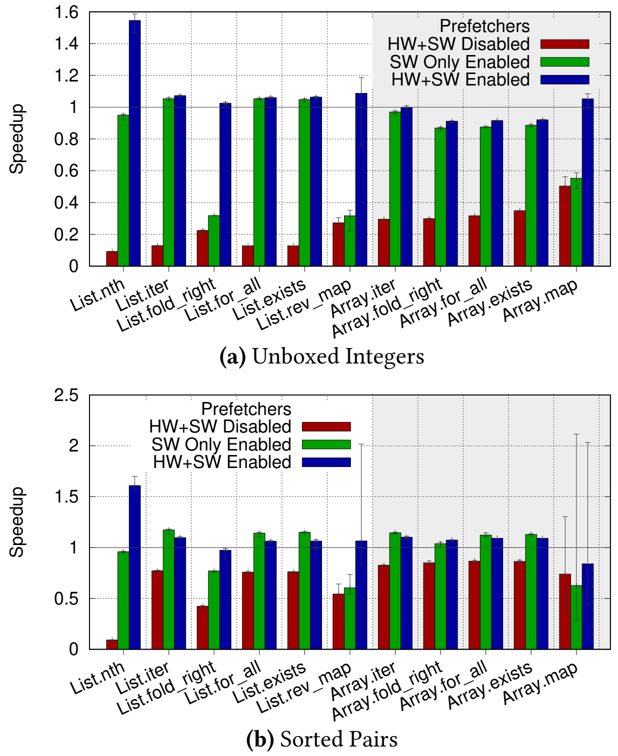
(b) A53

(c) KNL

Figure 10. Speedup for prefetching used on unsorted allocation-ordered data, using the same techniques as previously used on sorted data (figure 4).

Allocation-order Data The extra cost associated with prefetching is acceptable because it is outweighed by the cost of cache misses when data is spread throughout memory. However, when data is accessed more sequentially, cache miss rates are lower, and prefetching less useful. To measure the impact in these cases, we evaluate our prefetching techniques on unsorted allocation-order tuples.

We see in figure 10 that on the A53 for linked lists, performance is still improved. This is because the in-order system is so heavily memory bound that the extra computation is still worthwhile, even though the majority of memory accesses are anticipated by the hardware prefetcher. In other cases, performance is typically reduced slightly, as the high cost of the prefetching reduces the effective instructions per cycle for the rest of the code. In practical terms, this means that different library functions should be used for sorted versus allocation-order lists, except when using in-order cores.



(a) Unboxed Integers

(b) Sorted Pairs

Figure 11. Speedup for prefetching with the hardware prefetcher turned on and off for Haswell, relative to the default (hardware prefetcher on, no software prefetching).

Disabling the Hardware Prefetcher Every system we evaluate features hardware prefetchers, which are enabled by default. On Haswell, we can disable this to observe performance [53]. Relative metrics are shown in figure 11.

There are a number of surprising results in this data. Hardware prefetching gives an enormous performance improvement on many of these workloads, including those for which software prefetching also improves performance. While either software or hardware prefetching alone is sufficient to get the majority of the potential performance benefit, both are necessary for the best speedup in many cases. This is even true when, for example on the unboxed-integer list benchmarks, the hardware and software prefetchers fetch the same data. For some workloads (*List.fold_right*, *List.rev_map*, *Array.map*) the hardware prefetcher captures other access patterns, such as the stack frame and intermediate lists, and so more software prefetches would be necessary for maximal performance in a software-only setup than used here, where we tune with the hardware prefetcher enabled.

Another surprising result is the speedup for lists versus arrays from the hardware prefetcher, which is even more necessary for linked lists in functional languages than for arrays, despite arrays being a simpler pattern to pick up. This is due to the out-of-order microarchitecture. In a sequence of array loads, each address can be calculated independently, and so the CPU can issue multiple loads in parallel. However, in a linked list, the data dependencies prevent the out-of-order hardware from exploiting memory-level parallelism even if the data is laid out regularly in memory. In effect, the hardware prefetcher is serving as a value predictor [34] for addresses, allowing the system to overcome the data dependencies of the linked list and dramatically increasing performance. While without a hardware prefetcher we can achieve a similar effect with software prefetching, with neither performance drops by up to an order of magnitude. This also has implications for systems without hardware prefetchers, such as in embedded environments, or systems with less sophisticated hardware-prefetching mechanisms, such as the Xeon Phi [31], where software prefetching is vital to achieving high performance from functional languages.

6.7 Summary

Significant speedups are attainable both on sorting algorithms and within the OCaml list and array libraries from software prefetching. Up to 3× speedup can be achieved from adding prefetch-aware code into libraries. Our memory-bound benchmark suite for OCaml displays geomean improvements of 20% for Haswell, 28% for A53, and 11.5% for KNL, with improvements of up to 2.1×. The hardware prefetcher is vital for functional code: up to 10× performance loss can be observed without it, though software prefetching can make up much of this loss in its absence.

7 Analysis

We have shown that adding software prefetching to OCaml can bring about significant speedups. We now consider wider applications of prefetching for functional languages and how our techniques fit into this landscape.

7.1 Suitability for Functional Languages

Abstracting away precise memory layout, and avoiding giving the programmer access to memory addresses, is an advantage of functional languages compared with the low-level access granted by C-like languages, in terms of ease of writing correct code. Our software-prefetching extensions are unable to cause memory faults, don't give direct memory access, and don't affect program semantics. This makes them highly desirable as a performance optimisation whereby the correct code can be written first, then improved by adding prefetch instructions which cannot alter correctness. With the exception of offset prefetching, none of our techniques require a specific memory-access layout. None conflict with garbage collectors that are allowed to move data in memory. Indeed, while the general concept of prefetching is perhaps somewhat imperative, that it is so unobtrusive in its implementation suggests that it is a nice fit even in a pure functional language. That the improvements are so significant (60% speedup for something as fundamental as accessing the nth element of a linked list, for example) means that it is hard to argue against adding such a feature.

7.2 How Does This Apply to Other Languages?

Though we have implemented our prefetching scheme in OCaml, the performance issues, and thus improvements from prefetching, are not limited to this language. Many functional language implementations, including the Glasgow Haskell Compiler, feature generational garbage collectors [36], make heavy use of linked lists, and use pointer indirection to implement list elements. While GHC features a prefetch intrinsic, this feature is undocumented, and we were unable to compile programs using it; we suspect it is intended for backend code such as garbage collection. The need to prefetch boxed objects that have been reordered is common to all languages without value types. While C# allows compound objects to either be indirected to via pointer, or stored directly in an array or list [11], as do C and C++, many high-level languages only allow reference indirection [46]. OCaml allows built-ins such as integers to be unboxed, thus stored directly within an array or list, whereas Java's templated classes such as ArrayLists use costly indirection even for base types [12, 45].

For our linked-list schemes to work, walking ahead in lists must be cheap, and thus linked-list nodes must be allocated in an approximately sequential order. Having both a compacting garbage collector, and immutable objects (which prevents multiple insertions into the middle of a previously allocated linked list) is sufficient to achieve this. We should therefore expect this technique to be widely applicable to functional languages, but less applicable to those where mutability of data structures is a widely used feature of the language. In the latter case, list nodes that are contiguous may be widely distributed throughout memory.

7.3 Automation

In OCaml, the process of walking through a linked list is easily observed in the syntax: whenever *x::ys* appears within a pattern match, we can infer that a list is likely being iterated through, and can insert prefetching. This pattern matching could be done in the compiler; we could generate complex and offset prefetching, by altering the arity of functions to take the partially pre-walked list, and by specialising the prefetch lookup pattern to the targeted data type and memory access pattern. By doing this in the backend, the output prefetch code could be specialised for each datatype even on polymorphic code. While automation is unnecessary for library code, where implementations can be tuned manually, for custom code an automatic technique would be favourable.

While the algorithm is relatively simple, we do not have an implementation due to the unavailability of complex middle-end optimisers, such as LLVM, for functional languages. While many compilers, such as GHC and MosML, can output LLVM backend code, too much semantic information is lost by this point to reliably pick up linked-list code. Still, with further development of optimising compilers for functional languages, this analysis will be simple to implement, though as we see in the comparison of figure 4 and figure 10, the same code with prefetching can result in very different performance profiles based on input, so profile-guided optimisation may be necessary for ideal performance.

8 Related Work

Okasaki [41] designs data structures that work well with the features of functional languages, particularly immutability. Our work is in some sense the converse: we observe how the data structures used in functional languages create opportunities to improve memory-system performance, and thus allow prefetching in hardware and software for structures such as linked lists, where normally this would not make sense. Previous work on prefetching linked structures in imperative languages has focused on the links themselves, such as Luk et al. [35], particularly the single next element of the list [14] or through storing jump pointers to move ahead through structures [15] whereas we demonstrate that such patterns are less significant in terms of performance degradation than the items they point to, due to memory layouts, in functional languages, and so we can instead fetch many elements ahead, achieving more memory-level parallelism. Prefetching in functional languages so far has been limited to the back-end such as within the garbage collector [2, 40], where prefetching has been both used to target existing algorithms [13] and those rearchitected for effective prefetching [24], rather than in the language itself.

Ainsworth and Jones [3, 5] look at prefetching within languages such as C. They observe a similar pattern for arrays that we observe in OCaml, where prefetching workloads with indirection brings large benefits on in-order cores, and benefits on out-of-order cores provided a reasonable amount of work occurs between accesses. This has been previously observed for arrays of pointers in Java [16, 17], which is also a common pattern in OCaml due to the extensive use of pointer indirection in the language. However, in OCaml, not all code with pointer indirection is memory bound, due to the memory-allocation strategy and generational garbage collector. Further, these features mean linked lists behave similarly to arrays in OCaml, in that they are mostly sequential in memory and so are targets for software prefetching.

Wu et al. [56] utilise profile-guided optimisation to insert software prefetches for linked lists in imperative languages, when the linked lists are allocated approximately sequentially in memory. Lee et al. [33] and Mowry [39] look at prefetching imperative languages for array structures in simulation. Chen et al. [19] use prefetching for hash tables. These feature a moderate amount of hash computation between each access, so large performance improvements should be observed, as we observe in OCaml. Some automated compiler schemes exist for imperative languages [3, 18, 30, 31, 39]. As the memory patterns in functional languages differ considerably, the same techniques rarely apply directly.

In today's systems, hardware prefetchers are designed primarily to find stride patterns in addresses [8]. We have shown that these also pick up the linked-list and array patterns found in OCaml. Attempts to combine the strengths of hardware and software prefetching via codesign, to achieve flexibility without instruction cost, have been proposed [4, 6, 55]. These are also likely to yield benefits in functional languages.

9 Conclusion

The features typical within functional languages can result in highly memory-bound programs. Pointer indirection within data structures can result in irregular memory accesses, and linked lists can make memory accesses unpredictable.

However, there is an opportunity to improve the execution time of poorly performing code by using software prefetching. We have presented a variety of prefetching techniques and have integrated these into the OCaml compiler and standard library. Significant speedup is attainable for code featuring irregular memory accesses, using both arrays and linked lists, particularly in data that is either being, or has been, sorted, and even simple patterns can be vastly improved via software prefetching. We expect the techniques developed in this paper to be widely applicable to functional languages, as the features we exploit are not particular to OCaml. The improvements are large enough that such techniques should be made widely available even in high-level languages.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1, EP/P020011/1 and EP/M506485/1, and ARM Ltd. Additional data related to this publication is available in the repository at https://doi.org/10.17863/CAM.51790.

References

[1] 2020. OCaml Compiler. https://github.com/ocaml/ocaml. (2020).

[2] Arbob Ahmad and Henry DeYoung. 2009. *Cache performance of lazy functional programs on current hardware.* Technical Report. CMU.

[3] S. Ainsworth and Timothy M. Jones. 2017. Software prefetching for indirect memory accesses. In *CGO*.

[4] Sam Ainsworth and Timothy M. Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *ASPLOS*.

[5] Sam Ainsworth and Timothy M. Jones. 2019. Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective. *ACM Trans. Comput. Syst.* 36, 3 (June 2019), 8:1–8:34.

[6] Hassan Al-Sukhni, Ian Bratt, and Daniel A. Connors. 2003. Compiler-Directed Content-Aware Prefetching for Dynamic Data Structures. In *PACT*.

[7] A. W. Appel. 1989. Simple Generational Garbage Collection and Fast Allocation. *Softw. Pract. Exper.* 19, 2 (Feb. 1989).

[8] Jean-Loup Baer and Tien-Fu Chen. 1995. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. Comput.* 44, 5 (1995).

[9] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks – Summary and Preliminary Results. In *Supercomputing*.

[10] C. Bradford Barber, David P. Dobkin, David P. Dobkin, and Hannu Huhdanpaa. 1996. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.* 22, 4 (Dec. 1996).

[11] Maira Wenzel Bill Wagner. 2016. Value Types (C# Reference). https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types. (Dec. 2016).

[12] Joshua Bloch. 2008. *Effective Java, 2nd Edition.* Addison-Wesley.

[13] Hans-J. Boehm. 2000. Reducing Garbage Collector Cache Misses. In *ISMM*.

[14] Brendon Cahoon and Kathryn S McKinley. 1999. Tolerating latency by prefetching Java objects. In *Workshop on Hardware Support for Objects and Microarchitectures for Java*.

[15] B. Cahoon and K. S. McKinley. 2001. Data flow analysis for software prefetching linked data structures in Java. In *PACT*.

[16] Brendon Cahoon and Kathryn S. McKinley. 2002. Simple and Effective Array Prefetching in Java. In *JGI*.

[17] Brendon Cahoon and Kathryn S McKinley. 2005. Recurrence analysis for effective array prefetching in java. *Concurrency and Computation: Practice and Experience* 17, 5-6 (2005).

[18] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software Prefetching. In *ASPLOS*.

[19] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance Through Prefetching. *ACM Trans. Database Syst.* 32, 3, Article 17 (Aug. 2007).

[20] Tien-Fu Chen and Jean-Loup Baer. 1992. Reducing Memory Latency via Non-blocking and Prefetching Caches. In *ASPLOS*.

[21] Babak Falsafi and Thomas F. Wenisch. 2014. A Primer on Hardware Prefetching. *Synthesis Lectures on Computer Architecture* 9, 1 (2014).

[22] Agner Fog. 2019. Instruction tables. https://github.com/tsoding/convex-hull. (2019).

[23] Anders Fugmann, Jonas B. Jensen, and Mads Hartmann Jensen. 2015. Four years of OCaml in production. http://engineering.issuu.com/2015/09/17/ocaml-production.html. (2015).

[24] Robin Garner, Stephen M. Blackburn, and Daniel Frampton. 2008. Effective Prefetch for Mark-Sweep Garbage Collection. In *ISMM*.

[25] Jon Harrop. 2009. F# vs OCaml vs Haskell: hash table performance. http://flyingfrogblog.blogspot.com/2009/04/f-vs-ocaml-vs-haskell-hash-table.html. (2009).

[26] D. Kanter. 2012. Intel's Haswell CPU Microarchitecture. http://www.realworldtech.com/haswell-cpu/. (Nov. 2012).

[27] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. 2016. Path confidence based lookahead prefetching. In *MICRO*.

[28] KjellKod.cc. 2012. Number crunching: Why you should never, ever, EVER use linked-list in your code again. https://www.codeproject.com/Articles/340797/Number-crunching-Why-you-should-never-ever-EVER-us. (Aug. 2012).

[29] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. 2001. Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes. In *PACT*.

[30] Rakesh Krishnaiyer. 2012. Compiler Prefetching for the Intel Xeon Phi coprocessor. https://software.intel.com/sites/default/files/managed/54/77/5.3-prefetching-on-mic-update.pdf. (2012).

[31] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin, and H. Saito. 2013. Compiler-Based Data Prefetching and Streaming Non-temporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor. In *IPDPSW*.

[32] Gurhan Kucuk, Dmitry Ponomarev, and Kanad Ghose. 2002. Low-complexity Reorder Buffer Architecture. In *ICS*.

[33] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9, 1, Article 2 (March 2012), 29 pages.

[34] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *ASPLOS*.

[35] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-based Prefetching for Recursive Data Structures. In *ASPLOS*.

[36] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. 2008. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *ISMM*.

[37] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml.* O'Reilly.

[38] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. 2000. Java Programming for High-performance Numerical Computing. *IBM Syst. J.* 39, 1 (Jan. 2000).

[39] Todd C. Mowry. 1994. *Tolerating Latency Through Software-Controlled Data Prefetching.* Ph.D. Dissertation. Stanford University, Computer Systems Laboratory.

[40] Nicholas Nethercote and Alan Mycroft. 2002. The Cache Behaviour of Large Lazy Functional Programs on Stock Hardware. *SIGPLAN Not.* 38 (June 2002).

[41] Chris Okasaki. 1998. *Purely Functional Data Structures.* Cambridge University Press, New York, NY, USA.

[42] Stack Overflow. 2012. http://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key#12996028. (2012).

[43] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab.

[44] Andrew Ray. 2014. OCaml conjugate gradient solver. https://gist.github.com/andrewray/0be90c852343d79c723a. (2014).

[45] W. Clay Richardson, Donald Avondolio, Scot Schrager, Mark W. Mitchell, and Jeff Scanlon. 2007. *Professional Java JDK 6 Edition.* Wiley.

[46] John Rose, Brian Goetz, and Guy Steele. 2014. State of the Values. http://cr.openjdk.java.net/~jrose/values/values-0.html. (April 2014).

[47] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence Based Prefetching for Linked Data Structures. *SIGOPS Oper. Syst. Rev.* 32, 5 (Oct. 1998).

[48] Amir Roth and Gurindar S. Sohi. 1999. Effective Jump-pointer Prefetching for Linked Data Structures. In *ISCA*.

[49] Jérémie Salvucci and Emmanuel Chailloux. 2016. Memory Consumption Analysis for a Functional and Imperative Language. *Electronic Notes in Theoretical Computer Science* 330, Supplement C (2016), 27 – 46. RAC 2016 - Resource Aware Computing.

[50] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. 2015. Efficiently prefetching complex address patterns. In *MICRO*.

[51] James E. Smith and Andrew R. Pleszkun. 1988. Implementing Precise Interrupts in Pipelined Processors. *IEEE Trans. Comput.* 37, 5 (May 1988).

[52] Tsoding. 2015. Convex Hull. https://github.com/tsoding/convex-hull. (2015).

[53] Vish Viswanathan. 2014. Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors. (Sept. 2014).

[54] Keith Waclena. 2006. OCaml for the Skeptical: Why OCaml? The University of Chicago Library DLDC, http://www2.lib.uchicago.edu/keith/ocaml-class/why.html. (2006).

[55] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. 2003. Guided Region Prefetching: A Cooperative Hardware/Software Approach. In *ISCA*.

[56] Youfeng Wu, Mauricio J. Serrano, Rakesh Krishnaiyer, Wei Li, and Jesse Fang. 2002. Value-Profile Guided Stride Prefetching for Irregular Code. In *CC*.