Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective[*]

SAM AINSWORTH and TIMOTHY M. JONES, University of Cambridge, UK

Many modern data processing and HPC workloads are heavily memory-latency bound. A tempting proposition to solve this is software prefetching, where special non-blocking loads are used to bring data into the cache hierarchy just before being required. However, these are difficult to insert to effectively improve performance, and techniques for automatic insertion are currently limited.

This paper develops a novel compiler pass to automatically generate software prefetches for indirect memory accesses, a special class of irregular memory accesses often seen in high-performance workloads. We evaluate this across a wide set of systems, all of which gain benefit from the technique. We then evaluate the extent to which good prefetch instructions are architecture dependent, and the class of programs that are particularly amenable. Across a set of memory-bound benchmarks, our automated pass achieves average speedups of 1.3× for an Intel Haswell processor, 1.1× for both an ARM Cortex-A57 and Qualcomm Kryo, 1.2× for a Cortex-72 and an Intel Kaby Lake, and 1.35× for an Intel Xeon Phi Knight's Landing, each of which is an out-of-order core, and performance improvements of 2.1× and 2.7× for the in-order ARM Cortex-A53 and first generation Intel Xeon Phi.

CCS Concepts: • **Software and its engineering** → *Software performance*; *Compilers*;

ACM Reference Format:
Sam Ainsworth and Timothy M. Jones. 2019. Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective. *ACM Trans. Comput. Syst.* 1, 1, Article 1 (March 2019), 34 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many modern workloads for high-performance compute (HPC) and data processing are heavily memory-latency bound [19, 22, 27, 35]. The traditional solution to this has been prefetching: using hardware to detect common access patterns such as strides [10, 40], and thus bring the required data into fast cache memory before it is requested by the processor. However, these techniques do not work for irregular access patterns, as seen in linked data structures, and also in indirect memory accesses, where the addresses loaded are based on indices stored in arrays.

Software prefetching [8] is a tempting proposition for these data access patterns. The idea is that the programmer uses data structure and algorithmic knowledge to insert instructions into the program to bring the required data in early, thus improving performance by overlapping memory accesses. However, it is difficult to get right. To gain benefit, the cost of generating the address and issuing the prefetch load must be outweighed by the latency saved from avoiding the cache

[*] This is an extended version of the paper presented at CGO 2017 [2], featuring evaluation on four new systems, along with the presentation and analysis of two new configurable benchmarks designed to explain where, why, and to what extent software prefetching is useful for indirect memory accesses on in-order and out-of-order processors.

0734-2071/2019/3-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

miss. This is often not the case, as dynamic instruction count increases significantly, and any loads required for the address generation cause stalls themselves. Further, prefetching too far ahead risks cache pollution and the data being evicted before use; prefetching too late risks the data not being fetched early enough to mask the cache miss. Indeed, these factors can often cause software prefetches to under-perform, or show no benefit, even in seemingly ideal situations.

Therefore, to ease programmer effort it is desirable to automate the insertion of prefetches into code. Examples exist in the literature to do this for both stride [8, 33] and linked-list access patterns [26]. However, the former is usually better achieved in hardware to avoid instruction overhead, and the latter has limited performance improvement due to the lack of memory-level parallelism inherent in linked structures. Neither of these caveats apply to indirect memory accesses, which contain abundant memory-level parallelism. However, no automated approach is currently available for the majority of systems and access patterns.

To address this, we develop a novel algorithm to automate the insertion of software prefetches for indirect memory accesses into programs. Our compiler pass is able to deal both with more complex patterns than previous techniques [21, 32] that are commonly observed in code, and with the complexities of fault avoidance from loads at the compiler IR level. It can pick up access patterns that include computation (e.g., hashing) to calculate the indirection and ensure that no additional memory faults are generated by loads that are used in prefetch-address generation.

Within the compiler, we find loads that reference loop induction variables, and use a depth-first search algorithm to identify the set of instructions which need to be duplicated to load in data for future iterations. On workloads of interest to the scientific computing [5], HPC [28], big data [34] and database [38] communities, our automated prefetching technique gives an average 1.3× performance improvement for an Intel Haswell machine, 1.1× for an Arm Cortex-A57, 2.1× for an Arm Cortex-A53, and 2.7× for the first generation Intel Xeon Phi. We then consider reasons for the wide variance in performance attainable through software prefetching across different architectures and benchmarks, showing that look-ahead distance, memory bandwidth, dynamic instruction count and TLB support can all affect the utility of software prefetch.

In this extended version of the original paper [2], we further look at many other systems: the out-of-order Xeon Phi Knights Landing (1.35× speedup), Core i5 Kaby Lake (1.2× speedup), ARM Cortex-A72 (1.2× speedup) and Qualcomm Kryo (1.1× speedup). We discover that, while the overall strategy for prefetching remains generally the same, the magnitude of performance can differ greatly. We analyse the variety of reasons for this, and develop two new configurable benchmarks, Kangaroo and Camel, to explore a wide target application space. We use these to conclude that prefetching works best on out-of-order systems when the code is memory-bound and predictable, but when there is a significant amount of computation per loop, where speedups of up to 6× are achievable. We also discover that the techniques we use remain remarkably effective even under extreme situations.

2 RELATED WORK

Software prefetching has been studied in detail in the past, and we give an overview of techniques that analyse their performance, automate their insertion, and determine the look-ahead, in addition to those providing software prefetch though code transformations.

Prefetching Performance Studies Lee et al. [24] show speedups for a variety of SPEC benchmarks with both software and hardware prefetching. However, these benchmarks don't tend to show indirect memory-access patterns in performance-critical regions of the code, limiting observable behaviour. By comparison, Mowry [32] considers both Integer Sort and Conjugate Gradient from the NAS Parallel Benchmarks [5], which do. Both papers only consider simulated hardware.

However, we show the microarchitectural impact on the efficacy of software prefetching is important: Integer Sort gains a 7× improvement on an Intel Xeon Phi machine, but negligible speedup on an ARM Cortex-A57. In contrast, Chen et al. [9] insert prefetches for database hash tables by hand, whereas we develop an algorithm and automate insertion for this and other patterns.

Automatic Software Prefetch Generation Software prefetching for regular stride access patterns has been implemented in several tools, such as the Intel C Compiler [20]. These are particularly useful when they can beat the performance of the hardware stride prefetcher, such as in the Xeon Phi [30]. Methods for doing this in the literature directly insert software prefetches into loops, for example Callahan et al. [8]. Mowry [32] extends this with techniques to reduce branching, removing bounds checks for prefetches inside loops by splitting out the last few iterations of the loop. Wu et al. [41] use profiles to prefetch applications that are irregular but happen to exhibit stride-like patterns at runtime. Examples also exist in the literature for software prefetching of both recursive data structures, for example Luk and Mowry [26] prefetch linked lists, and function arguments, such as Lipasti et al. perform [25]. Cahoon and McKinley look at patterns in Java [6, 7], for regular arrays and linked structures.

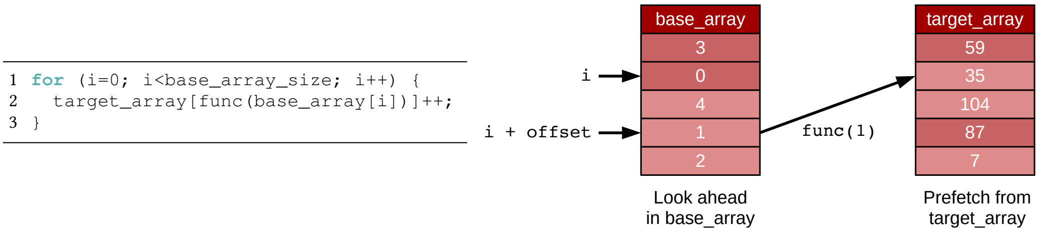
Mowry's PhD dissertation [32] discusses indirect prefetching for high level C-like code. In contrast, in this paper we give a full algorithm to deal with the complexities of intermediate representations, including fault avoidance techniques and value tracking. An algorithm for simple stride-indirect patterns is implemented in the Xeon Phi compiler [21], but it is not enabled by default and little information is available on its inner workings. Further, it picks up relatively few access patterns, and is comprehensively outclassed by our technique, as shown in section 6.

VanderWiel and Lilja [39] propose moving the software prefetches to a dedicated programmable hardware prefetch controller, to reduce the associated overheads, but their analysis technique also only works for regular address patterns without loads. Khan et al. [15, 16] choose instead to insert software prefetches dynamically using a runtime framework for code modification. This enables prefetching for applications where the source code is unavailable, and also gives access to runtime data, but limits access to static information such as types, and also adds overhead.

Scheduling Prefetches A variety of fine-grained prefetch scheduling techniques, to set the appropriate look-ahead distance, have been considered in the past. Mowry et al. [33] consider estimated instruction time against an estimated memory system time. The former is difficult to estimate correctly on a modern system, and the latter is microarchitecture dependent, which makes these numbers difficult to get right. Lee et al. [24] extend this by splitting instructions per cycle (IPC) and average instruction count, which are both determined from application profiling. As these are all small numbers, and errors are multiplicative, accuracy is challenging: the latter multiplies the distance by 4 to bias the result in favour of data being in the cache too early. In comparison, our algorithm schedule prefetches based on the number of loads required to generate an address.

Techniques Involving Software Prefetches Rather than directly inserting software prefetches within loops, some works have used them as parts of separate loops to improve performance or power efficiency. Jimborean et al. [14] use compiler analysis to duplicate and simplify code, to separate loads and computation, enabling different frequency-voltage scaling properties for different sections of the code.

Software prefetches can also be moved to different threads, to reduce the impact of the large number of extra instructions added to facilitate prefetching. Kim and Yeung [17] use a profile-guided compiler pass to generate "helper threads", featuring prefetch instructions, to run ahead of the main thread. Malhotra and Kozyrakis [29] create helper threads by adding software prefetch instructions to shared libraries and automatically detecting data structure traversals.



```
for (i=0; i<base_array_size; i++) {
  target_array[func(base_array[i])]++;
}
```

(a) Code containing stride-indirect accesses

(b) Prefetching from target_array

Fig. 1. Many workloads perform stride-indirect traversals starting from an array. We can look ahead in the base array and prefetch future values from the target array.

Hardware Prefetchers Hardware prefetchers in real systems focus on stride patterns [10, 12, 18, 37, 40]. These pick up and predict regular access patterns, such as those in dense-matrix and array iteration, based on observation of previous addresses being accessed. However, when a pattern is data-dependent, as is the case with indirect memory accesses, the address alone is insufficient.

Attempts have been made at prefetching irregular structures in hardware. This is desirable, as it does not require recompilation, and can potentially reduce overheads compared with software. However, though progress has been made, none have been implemented in commercial systems [12].

Pointer-fetching prefetchers [11], which fetch all plausible virtual addresses from cache lines read by a core, have been proposed in several schemes. The main downside to these approaches is the large over-fetch rate. In addition, these schemes are unable to deal with the array-indirect patterns seen in many workloads.

Attempts to extract dependence-graph streams at runtime, by detecting dependent loads, have been made [4, 31, 36]. These run dynamically detected load streams on programmable units on identification of the start of a set of loads, to prefetch the data. These require a large amount of analysis hardware to be added to the commit stage of the pipeline, and a large amount of processing power to run the detected streams.

Yu et al. [42] pick up stride-indirect patterns using runtime analysis of executed code to find the base array and size of each data element. Their approach successfully prefetches this single pattern, at the expense of complicated analysis hardware in the cache, which may affect the critical path of execution.

Ainsworth and Jones [3] design parallel hardware to run programmable prefetch kernels generated by the programmer or compiler. The compiler technique involved uses a similar data-flow technique to that presented here, but as this hardware is designed not to fault, and prefetch code is separate from the original program, the guarantees necessary to prevent software prefetches from causing faults are unnecessary, and scheduling can be performed using runtime information rather than statically at compile time.

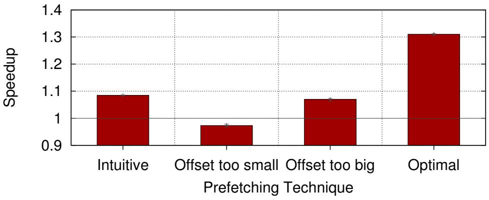
Summary Although Mowry has analysed indirect memory access in the past [32], nobody has yet performed a major study of software prefetching for them, nor developed an automated compiler pass to exploit them. The next section shows the potential for software prefetch for these access patterns, before developing our algorithm for automatic prefetch generation.

3 PREFETCH POTENTIAL

To show how software prefetches can aid common indirect access patterns, consider figure 1. Here, we have an access pattern that involves sequential movement through `base_array`, followed

```
1 for (i=0; i<NUM_KEYS; i++) {
2   // The intuitive case, but also
3   // required for optimal performance.
4   SWPF(key_buff1[key_buff2[i + offset]]);
5   // Required for optimal performance.
6   SWPF(key_buff2[i + offset*2]);
7   key_buff1[key_buff2[i]]++;
8 }
```

(a) An integer sort benchmark showing software prefetch locations



(b) Software prefetching performance for figure 2(a) on an Intel Haswell micro-architecture

Fig. 2. Inserting software prefetches for maximal performance is a challenge even in simple cases. In (a), the intuitive prefetch to insert is only at line 4, whereas optimal performance also requires that at line 6. In (b), intuitive schemes leave performance on the table, and choosing the correct offset is critical for high performance.

by access into `target_array`, based on a function `func` on the data from the first array. When using the identity function (i.e., `func(x) = x`), this represents a simple stride-indirect pattern. If `func(x)` is more complex, it represents a hashing access.

Both patterns are suitable for software prefetching, provided that `func(x)` is side-effect free. As the addresses accessed in `target_array` are data-dependent, a hardware stride prefetcher will be unable to discern any pattern, so will fail to accurately prefetch them. However, future memory access addresses can easily be calculated in software due to being able to look ahead in `base_array`.

Still, inserting the correct prefetch code, with suitable look-ahead distances (or offset from the current iteration), is challenging for an end user. Figure 2(a) shows the code required to prefetch a simple stride-indirect access pattern from an integer sort benchmark (as described in section 5.1), and figure 2(b) shows the performance of different schemes. The intuitive approach inserts only the prefetch at line 4, giving a speedup of 1.08×. However, for optimal performance, staggered prefetches to both the base array `key_buff2` and the indirect array `key_buff1` are required, even in the presence of a hardware stride prefetcher. Although the access to `key_buff2` is sequential, the hardware prefetcher can become confused by both the original code and the prefetch accessing the same data structure at staggered intervals, and so in every system we have observed the prefetcher is not sufficiently aggressive to fully prefetch the pattern. This means that the prefetch at line 6 is also required to give a speedup of 1.30×. Further, choosing a good prefetch distance is critical to avoid fetching the data too late (when the offset is too small), or polluting the cache (when the offset is too large).

Given these complexities, even for the simple example shown, we propose an automated software-prefetch-generation pass for indirect memory-access patterns within modern compilers. This avoids the programmer having to find suitable access patterns within their code, and allows the generation of good prefetch code without needing to be an expert in the properties of software prefetches.

4 SOFTWARE PREFETCH GENERATION

We present a pass which finds loads that can be prefetched based on look-ahead within an array, and generates software prefetches for those that will not be identified by a stride prefetcher. We first describe the analysis required, then the actual code generated. An overview is given in algorithm 1.

Algorithm 1 The software prefetch generation algorithm, assuming the intermediate representation is in SSA form.

```
DFS(inst) {
  candidates = {}
  foreach (o: inst.src_operands):
    // Found induction variable, finished this path.
    if (o is an induction variable):
      candidates ∪= {(o, {inst})}
    // Recurse to find an induction variable.
    elif (o is a variable and is defined in a loop):
      if (((iv, set) = DFS(loop_def(o))) != null):
        candidates ∪= {(iv, {inst}∪set)}

  // Simple cases of 0 or 1 induction variable.
  if (candidates.size == 0):
    return null
  elif (candidates.size == 1):
    return candidates[0]

  // There are paths based on multiple induction
  // variables, so choose the induction variable in
  // the closest loop to the load.
  indvar = closest_loop_indvar(candidates)

  // Merge paths which depend on indvar.
  return merge_instructions(indvar, candidates)
}

// Generate initial set of loads to prefetch and
// their address generation instructions.
prefetches = {}
foreach (l: loads within a loop):
  if (((indvar, set) = DFS(l)) != null):
    prefetches ∪= {(l, indvar, set)}

// Function calls only allowed if side-effect free.
remove(prefetches, contains function calls)
// Prefetches should not cause new program faults.
remove(prefetches, contains loads which may fault)
// Non-induction variable phi nodes allowed if the
// pass can cope with complex control flow.
remove(prefetches, contains non-induction phi nodes)

// Emit the prefetches and address generation code.
foreach ((ld, iv, set): prefetches):
  off = calc_offset(list, iv, load)
  insts = copy(set)
  foreach (i: insts):
    // Update induction variable uses.
    if (uses_var(i, iv)):
      replace(i, iv, min(iv.val + off, max(iv.val)))
    // Final load becomes the prefetch.
    if (i == copy_of(ld)):
      insts = (insts - {i}) ∪ {prefetch(i)}
  // Place all code just before the original load.
  add_at_position(ld, insts)
```

4.1 Analysis

The overall aim of our analysis pass is to identify loads that can be profitably prefetched and determine the code required to generate prefetch instructions for them. Target loads are those where it is possible to generate a prefetch with look-ahead: that is, we check whether we can generate a new load address by increasing the value of a referenced induction variable within the address calculation by a certain offset. Our analysis considers a function at a time and does not cross procedure boundaries.

We start with loads that are part of a loop (line 30 in algorithm 1). We walk the data dependence graph backwards using a depth-first search from each load to find an induction variable within the transitive closure of the input operands (line 1). We stop searching along a particular path when we reach an instruction that is not inside any loop. When we find an induction variable, we record all instructions that reference this induction variable (directly or indirectly) along each path to the load (lines 6 and 10). If multiple paths reference different induction variables, we only record the instructions which reference the innermost ones (line 21). This reflects the fact that these variables are likely to be the most fine-grained form of memory-level parallelism available for that loop.

Our recorded set of instructions will become the code to generate the prefetch address in a later stage of our algorithm. However, we must constrain this set further, such that no function calls (line 35) or non-induction-variable phi nodes (line 40) appear within it, because the former may result in side-effects occurring and the latter may indicate complex control flow changes are required. In these cases we throw away the whole set of instructions, and do not generate prefetches for the target load. Nevertheless, both could be allowed with further analysis. For example, side-effect-free function calls could be permitted, allowing the prefetch to call the function and obtain the same value as the target load. Non-induction phi nodes require more complicated control flow generation than we currently support, along with more complex control flow analysis. However, without this analysis, the conditions are required to ensure that we can insert a new prefetch instruction next to the old load, without adding further control flow.

4.2 Fault Avoidance

Though software prefetches themselves cannot cause faults, intermediate loads used to calculate addresses can (e.g., the load from `key_buff2` to generate a prefetch of `key_buff1` at line 4 in figure 2(a)). We must ensure that look-ahead values will be valid addresses and, if they are to be used by other intermediate loads, that they contain valid data.

To address this challenge, we follow two strategies. First, we add address bounds checks into our software prefetch code, to limit the range of induction variables to known valid values (line 49 in algorithm 1). For example, checking that `i + 2*offset < NUM_KEYS` at line 6 in figure 2(a). Second, we analyse the loop containing the load, and only proceed with prefetching if we do not find stores to data structures that are used to generate load addresses within the software prefetch code (line 37). For example, in the code `x[y[z[i]]]`, if there were stores to `z`, we would not be able to safely prefetch `x`. This could be avoided with additional bounds checking instructions, but would add to the complexity of prefetch code. We also disallow any prefetches where loads for the address-generating instructions are conditional on loop-variant values other than the induction variable. Together, these ensure that the addresses generated for intermediate loads leading to prefetches will be exactly the same as when computation reaches the equivalent point, several loop iterations later.

The first strategy requires knowledge of each data structure's size. In some cases, this is directly available as part of the intermediate representation's type analysis. For others, walking back through the data dependence graph can identify the memory allocation instruction which generated the array. However, in general, this is not the case. For example, it is typical in languages such as C for arrays to be passed to functions as a pointer and associated size, in two separate arguments. In these cases, and more complicated ones, we can only continue if the following two conditions hold. First, the loop must have only a single loop termination condition, since then we can be sure that all iterations of the loop will give valid induction values. Second, accesses to the look-ahead array must use the induction variable which should be monotonically increasing or decreasing.

Given these conditions, the maximum value of the induction variable within the loop will be the final element accessed in the look-ahead array in that loop and we can therefore use this value as a substitute for size information of the array, to ensure correctness. Although these conditions are sufficient alone, to ease analysis in our prototype implementation, we further limit the second constraint such that the look-ahead array must be accessed using the induction variable as a direct index (`base_array[i]` not `base_array[f(i)]`) and add a constraint that the induction variable must be in canonical form.

The software prefetch instructions themselves cannot change correctness, as they are only hints. The checks described in this section further ensure that address generation code doesn't create faults if the original code was correct. However, the pass can still change runtime behaviour if the program originally caused memory faults. While no memory access violations will occur if none were in the original program, if memory access violations occur within prefetched loops, they may manifest earlier in execution as a result of prefetches, unless size information comes directly from code analysis instead of from the loop size.

4.3 Prefetch Generation

Having identified all instructions required to generate a software prefetch, and met all conditions to avoid introducing memory faults, the next task is to actually insert new instructions into the code. These come from the set of instructions recorded as software prefetch code in section 4.1 and augmented in section 4.2.

We insert an `add` instruction (line 49 in algorithm 1) to increase the induction variable by a value (line 44), which is the offset for prefetch. Determining this value is described in section 4.4. We then generate an instruction (either a select or conditional branch, depending on the architecture) to take the minimum value of the size of the data structure and the offset induction variable (line 49). We create new copies (line 45) of the software prefetch code instructions, but with any induction-variable affected operands (determined by the earlier depth-first search) replaced by the instruction copies (line 49). Finally, we generate a software prefetch instruction (line 52) instead of the final load (i.e., the instruction we started with in section 4.1).

We only generate software prefetches for stride accesses if they are part of a load for an indirect access. Otherwise, we leave the pattern to be picked up by the hardware stride prefetcher, or a more complicated software stride-prefetch generation pass which is able to take into account, for example, reuse analysis [33]. Indeed, even in the case of stride-indirect accesses, the sequential accesses show locality based on cache lines. It is therefore the case that on unrolled loops, only a partial fraction of such prefetches are necessary. As the instruction overhead of a stride prefetch is minor, the extra prefetches do not affect performance significantly, and so we do not optimise them away here.

4.4 Scheduling

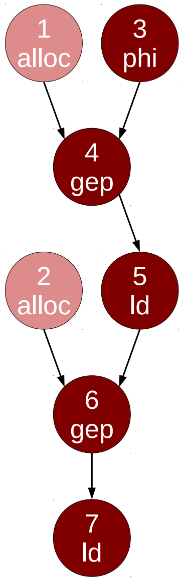
Our goal is to schedule prefetches by finding a look-ahead distance that is generous enough to prevent data being fetched too late, yet avoids polluting the cache and extracts sufficient memory parallelism to gain performance. Previous work [33] has calculated prefetch distance using a ratio of memory bandwidth against number of instructions. However, code featuring indirect accesses

```
1  start: alloc a, asize
2         alloc b, bsize
3  loop:  phi i, [#0, i.1]
4         gep t1, a, i
5         ld t2, t1
6         gep t3, b, t2
7         ld t4, t3
8         add t5, t4, #1
9         str t3, t5
10        add i.1, i, #1
11        cmp size, i.1
12        bne loop
```

(a) Original compiler IR code

```
add p1, i, #32
min p2, p1, asize
gep p3, a, p2
ld p4, p3
gep p5, b, p4
prefetch p5
add p6, i, #64
gep p7, a, p6
prefetch p7
```

(c) Generated prefetching code



(b) Depth-first search

Fig. 3. Our pass running on an integer sort benchmark.

is typically memory bound, so execution time is dominated by load instructions. We therefore generate look-ahead distances using the following formula.

$$offset = \frac{c(t-l)}{t} \tag{1}$$

where t is the total number of loads in a prefetch sequence, l is the position of a given load in its sequence, and c is a microarchitecture-specific constant, which represents the look-ahead required for a simple loop with few instructions, and is influenced by a combination of the memory latency and throughput (e.g., instructions-per-cycle (IPC)) of the system. High memory latency requires larger look-ahead distances to overcome, and high IPC means the CPU will move through loop iterations quickly, meaning many iterations will occur within one memory latency of time.

As an example of this scheduling, for the code in figure 2(a), two prefetches are generated: one for the stride on `key_buff2`, and one using a previously prefetched look-ahead value to index into `key_buff1`. This means $t = 2$ for these loads. For the first, $l = 0$, so $offset = c$ by eq. (1) so we issue a prefetch to `key_buff2[i+c]`. For the second, $l = 1$, so we issue a prefetch to `key_buff[i+c/2]`.

This has the property that it spaces out the look-ahead for dependent loads equally: each is prefetched $\frac{c}{t}$ iterations before it is used, either as part of the next prefetch in a sequence, or as an original load.

4.5 Example

An example of our prefetching pass is given in figure 3. From the load in line 7 in figure 3(a), we work backwards through the data dependence graph (DDG) using a depth-first search. The path followed is shown in figure 3(b). From the `gep` in line 6, we find an `alloc` that is not in a loop (line 2), and so stop searching down this path and follow the next. We next encounter the `ld` in line 5 and continue working through the DDG until reaching the `alloc` in line 1, which is also outside a loop, stopping search down this path. These two allocation instructions give the bounds of the a and b arrays.

Continuing along the other path from the gep is the phi in line 3, at which point we have found an induction variable. We take this set of instructions along the path from the phi node to the original load (dark red in figure 3(b)) and note that there are two loads that require prefetching. Therefore we calculate the offset for the original load as 32 and that for the load at line 5 as 64. From this, we generate the code shown in figure 3(c), where all references to i are replaced with min(i+32, a) for the prefetch at line 6 to avoid creating any faults with the intermediate load (line 4).

4.6 Prefetch Loop Hoisting

It is possible for analysed loads to be within inner loops relative to the induction variable observed. In this case, the inner loop may not feature an induction variable (for example, a linked list walking loop), or may be too small to generate look-ahead from. However, if we can guarantee control flow, and remove loop-dependent values for some iterations, it may be beneficial to add prefetches for these outside the inner loop.

We implement this by generating prefetches for loads inside loops where control flow indicates that any phi nodes used in the calculation reference a value from an outer loop. We then replace the phi node in the prefetch with the value from the outer loop, and attempt to make the prefetch loop invariant by hoisting the instructions upwards. This will fail if there are other loop invariant values on which the load depends. We must also guarantee the control flow occurs such that the loads generated by the software prefetches won't cause any new faults to occur. We can do this provided we can guarantee execution of any of the original loads we duplicate to generate new prefetches, or that the loads will be valid due to other static analyses.

4.7 Runtime Constraints

Our compiler pass is entirely static: it chooses whether or not to insert prefetches entirely based on compile-time properties. This means that, if the size of a loop at runtime is relatively small, the prefetch will not be useful, as it will target an address over the end of the accessed data structure. Similarly, if at runtime a data structure is small, it will fit in the cache, and therefore prefetching will not be useful despite the data-dependent memory-access pattern.

The ideal solution in both of these cases is not to generate any prefetches at all. Prefetching at a smaller offset will not work, as it will not fully hide the memory latency of the accesses, and thus the costs will typically outweigh the benefits In some cases this could be mitigated by using a runtime check to direct execution to different versions of the loop based on the dynamic number of iterations or the data size. In other cases, profile-guided optimization may be necessary to assess the overheads. Still, code that dominates execution time, and therefore necessitates speeding up, is more likely to feature a large number of iterations and large data sizes as a result, and we shall see that a static-only solution works well in practice.

4.8 Other Access Patterns

Our compiler pass targets only patterns featuring some amount of indirection that can be traced back to an induction variable. These are not the only memory accesses that can cause low performance. For example, linked data structures, and recursive functions upon them, cause unpredictable memory access patterns [26]. Our technique does not attempt to generate prefetches in these cases, as there is no induction variable to move ahead in the computation and use to generate future addresses. More generally, linked data structures do not directly exhibit large amounts of memory-level parallelism because it is difficult to move far ahead in the computation by, for example, looking ahead in an array. Instead, our pass would have to load several items ahead in the data structure before it could find an item worth prefetching, and these loads would reduce or remove all benefits.

4.9 Summary

We have described a pass to automatically generate software prefetch for indirect memory accesses, which are likely to miss in the cache, cannot be picked up by current hardware prefetchers, and are simple to extract look-ahead from. We have further provided a set of sufficient conditions to ensure the code generated will not cause memory faults, provided the original code was correct. We have also described a scheduling technique for these prefetches which is aimed at modern architectures, where despite variation in performance, the critical determiner of look-ahead distance is how many dependent loads are in each loop, rather than total number of instructions.

5 EXPERIMENTAL SETUP

We implement the algorithm described in section 4 as an LLVM IR pass [23], which is used within Clang. Clang cannot generate code for the Xeon Phi, so instead we manually insert the same prefetches our pass generates for the other architectures and compile using ICC. For Clang, we always use the O3 setting, as it is optimal for each program; however, for ICC we use whichever of O1, O2 or O3 works best for each program. We set $c = 64$ for all systems to schedule prefetches, as described in section 4.4, and evaluate the extent to which this is suitable in section 6.2. For all systems, we always prefetch into the L1 cache, with hints for maximum temporal locality (3 for `__builitin_prefetch`) and for a read access (`0`), and we found that these consistently worked best, though all settings gave very similar results. As our technique works for many workloads that do not feature thread-level parallelism, we initially look at single-threaded applications, before extending this to multi-threaded versions of the benchmarks where a parallel implementation is available.

5.1 Benchmarks

To evaluate software prefetching, we use a variety of benchmarks that include indirect loads from arrays that are accessed sequentially. We run each benchmark to completion, timing everything apart from data generation and initialisation functions, repeating experiments three times.

Integer Sort (IS) Integer Sort is a memory-bound kernel from the NAS Parallel Benchmarks [5], designed to be representative of computational fluid dynamics workloads. It sorts integers using a bucket sort, walking an array of integers and resulting in array-indirect accesses to increment the bucket of each observed value. We run this on the NAS parallel benchmark size B and insert software prefetches in the loop which increments each bucket, by looking ahead in the outer array, and issuing prefetch instructions based on the index value from the resulting load.

Conjugate Gradient (CG) Conjugate Gradient is another benchmark from the NAS Parallel suite [5]. It performs eigenvalue estimation on sparse matrices, and is designed to be typical of unstructured grid computations. As before, we run this on the NAS parallel benchmark size B.

The sparse matrix multiplication computation exhibits an array-indirect pattern, which allows us to insert software prefetches based on the NZ matrix (which stores non-zeros), using the stored indices of the dense vector it points to. The irregular access is on a smaller dataset than IS, meaning it is more likely to fit in the L2 cache, and presents less of a challenge for the TLB system.

RandomAccess (RA) HPCC RandomAccess is from the HPC Challenge Benchmark Suite [28], and is designed to measure memory performance in the context of HPC systems. It generates a stream of pseudo-random values which are used as indices into a large array. The access pattern is more complicated than in CG and IS: we look ahead in the random number array, then perform a hash function on the value to generate the final address for prefetching. Thus, prefetches involve more computation than in IS or CG.

Hash Join 2EPB (HJ-2) Hash Join [38] is a kernel designed to mimic the behaviour of database systems, in that it hashes the keys of one relation, and uses them as index into a hash table. Each bucket in the hash table is a linked list of items to search within. In HJ-2, we run the benchmark with an input that creates only two elements in each hash bucket, causing the access pattern to involve no linked-list traversals (due to the data structure used). Therefore, the access pattern is prefetched by looking ahead in the first relation's keys, computing the hash function on the value obtained, and finally a prefetch of this hashed value into the hash table. This is similar to the access pattern in RA, but involves more control flow, therefore, more work is done per element.

Hash Join 8EPB (HJ-8) This kernel is the same as HJ-2, but in this instance the input creates eight elements per hash bucket. This means that, as well as an indirect access to the hash table bucket, there are also three linked-list elements to be walked per index in the key array we use for look-ahead. It is unlikely that any of these loads will be in the cache, therefore there are four different addresses we must prefetch per index, each dependent on loading the previous one. This means a direct prefetch of the last linked-list element in the bucket would cause three cache misses to calculate the correct address. To avoid this, we can stagger prefetches to each element, making sure the previous one is in the cache by the time the next is prefetched in a future iteration. For example, we can fetch the first bucket element at offset 16, followed by the first linked-list element at offset 12, then offsets 8 and 4 for the second and third respectively.

This benchmark is complicated for a compiler pass to analyse, as the number of buckets accessed is essentially a runtime property rather than one defined by the code structure. This means that a manually written software prefetch is likely to be the best approach.

Graph500 Seq-CSR (G500) Graph500 [34] is designed to be representative of modern graph workloads, by performing a breadth-first search on a generated Kronecker graph in compressed sparse row format. This results in four different possible prefetches. We can prefetch each of the vertex, edge and parent lists from the breadth-first search's work list using a staggered approach, as for HJ-8. Further, as there are multiple edges per vertex, we can prefetch parent information based on each edge, provided the look-ahead distance is small enough to be within the same vertex's edges. The efficacy of each prefetch then depends on how many instructions we can afford to execute to mask the misses and, in the latter case, how likely the value is to be used: longer prefetch distances are more likely to successfully hide latency, but are less likely to be in the same vertex, and thus be accessed.

We run this benchmark on both a small, 10MiB Graph, with options `-s 16 -e 10` (**G500-s16**), and a larger 700MiB graph (**G500-s21**, options `-s 21 -e 10`), to get performance for a wide set of inputs with different probabilities of the data already being in the cache.

5.2 Systems

Table 1 shows the parameters of the systems we have evaluated. Each is equipped with a hardware prefetcher to deal with regular access patterns; our software prefetches are used to prefetch the irregular, indirect accesses based on arrays. Kaby, Haswell, Phi KNL, Kryo, A72 and A57 are out-of-order superscalar cores; Xeon Phi and A53 are in-order.

System	Specifications
Kaby	Intel Core i5-7500 CPU, 3.40GHz, 4 cores, 32KiB L1D, 256KiB L2, 6MiB L3, 16GiB DDR4
Haswell	Intel Core i5-4570 CPU, 3.20GHz, 4 cores, 32KiB L1D, 256KiB L2, 8MiB L3, 16GiB DDR3
Phi KNL	Intel Xeon Phi 7210 CPU, 1.30GHz, 64 cores, 32KiB L1D, 1MiB L2, 196GiB DDR4
Kryo	Inforce 6640, Qualcomm Kryo CPU, 2.2GHz, 4 cores, 32KiB L1D, 1MiB L2, 4GiB LPDDR4
A72	Juno R2, ARM Cortex-A72 CPU, 1.2GHz, 2 cores, 32KiB L1D, 2MiB L2, 8GiB DDR3L
A57	Nvidia TX1, ARM Cortex-A57 CPU, 1.9GHz, 4 cores, 32KiB L1D, 2MiB L2, 4GiB LPDDR4
Xeon Phi	Intel Xeon Phi 3120P CPU, 1.10GHz, 57 cores, 32KiB L1D, 512KiB L2, 6GiB GDDR5
A53	Odroid C2, ARM Cortex-A53 CPU, 2.0GHz, 4 cores, 32KiB L1D, 1MiB L2, 2GiB DDR3

Table 1. System setup for each processor evaluated.

6 EVALUATION

We first present the results of our autogenerated software prefetch pass across benchmarks and systems, showing significant improvements comparable to fine-tuned manual insertion of prefetch instructions. We then evaluate the factors that affect software prefetching in different systems.

6.1 Autogenerated Performance

Figure 4 shows the performance improvement for each system and benchmark using our compiler pass, along with the performance of the best manual software prefetches we could generate.

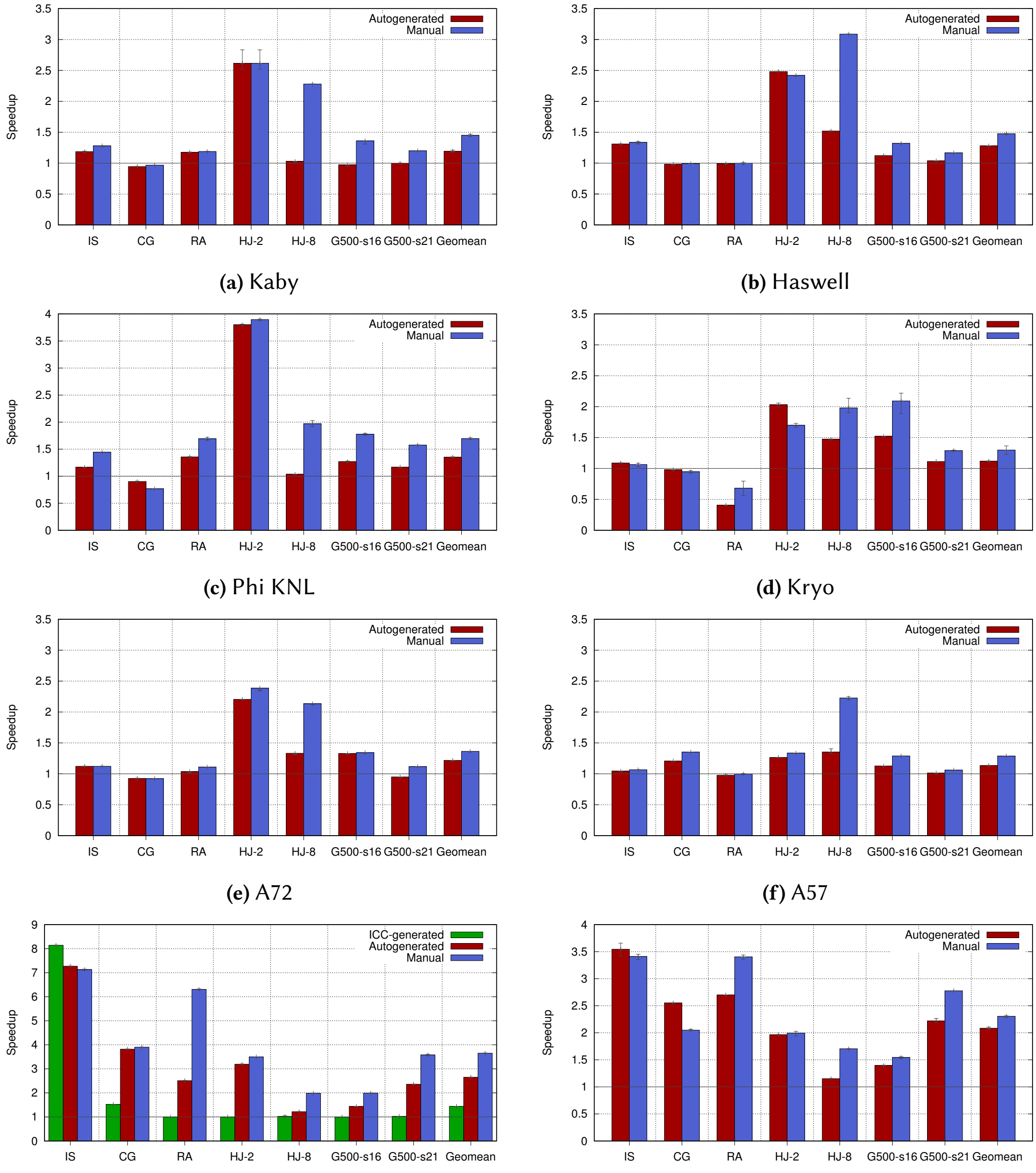
Haswell Haswell gets close to ideal performance on HJ-2, and IS, as the access patterns are fully picked up by the compiler pass. This is also true of CG but, as with RA, performance improvement with software prefetches is limited because the latency of executing the additional code masks the improvement in cache hit rates.

HJ-8 gets a limited improvement. The stride-hash-indirect pattern is picked up by the compiler, but the analysis cannot pick up the fact that we walk a particular number of linked-list elements in each loop. This is a runtime property of the input that the compiler cannot know, but manual prefetches can take advantage of the additional knowledge.

While G500 shows a performance improvement for both the s16 and s21 setups, it isn't close to what we can achieve by manual insertion of prefetch instructions. This is because the automated pass cannot pick up prefetches to the edge list, the largest data structure, due to complicated control flow. In addition, it inserts prefetches within the innermost loop, which are suboptimal on Haswell due to the stride-indirect pattern being short-distance, something only known with runtime knowledge.

Kaby The pattern here is largely similar to Haswell. Interestingly, RA changes from having a negligible improvement to being relatively significant: minor architectural changes have tipped performance in favour of software prefetching. IS gets some extra improvement from manual prefetching, due to using a larger lookahead distance. Incidentally, this is the only system-benchmark combination where a larger lookahead improves performance.

A57 The performance for the Cortex-A57 follows a similar pattern to Haswell, as both are out-of-order architectures. For IS, CG and HJ-2, differences between the automated pass and manual prefetches are simply down to different code generation. However, the A57 can only support one



(a) Kaby

(b) Haswell

(c) Phi KNL

(d) Kryo

(e) A72

(f) A57

(g) Xeon Phi

(h) A53

Fig. 4. Performance of our autogenerated software prefetching pass and the best manual software prefetches found. Also shown for the Xeon Phi is the performance of ICC-generated software prefetches.

page-table walk at a time on a TLB miss, limiting improvements for IS and HJ-2. CG's irregular dataset is smaller than for other benchmarks, so fewer page-table walks are required and a lack of parallelism in the TLB system doesn't prevent memory-level parallelism from being extracted via software prefetch instructions. The newer Cortex-A73 is able to support two page-table walks at once [13], likely improving prefetch performance.

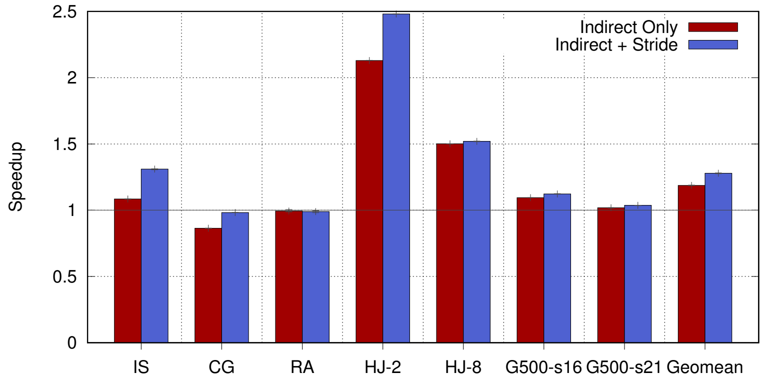


Fig. 5. Performance of inserting staggered stride software prefetches along with the indirect prefetch, compared to the indirect alone, for Haswell, with our automated scheme.

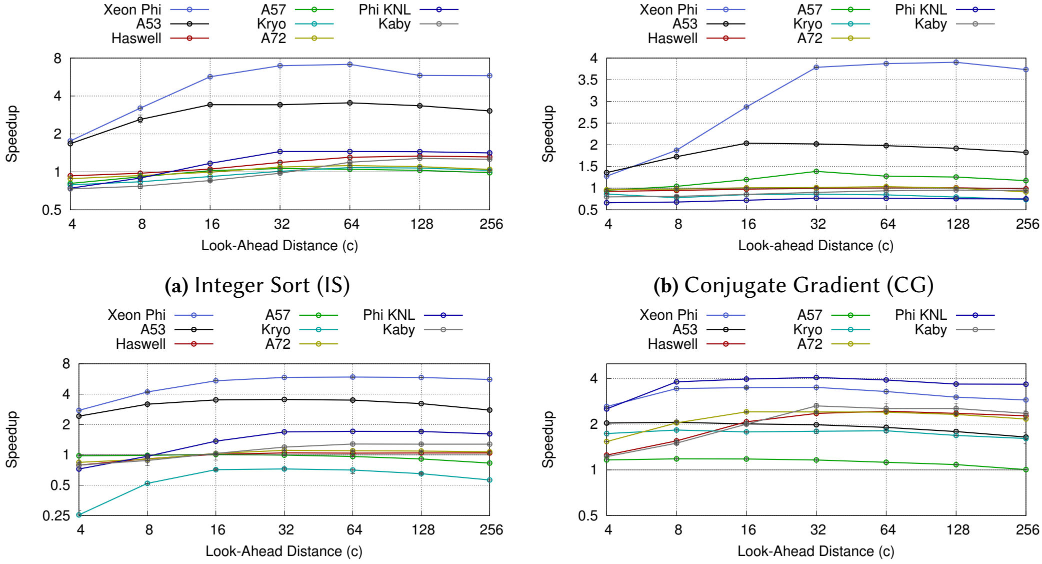
A72 The improvement gained on the A72 is higher than with the A57, with average speedups of 1.2× and 1.35× for automatic and manual prefetching respectively. This is mostly due to improved performance on HJ-2 and G500-s16, possibly because the system is more able to support concurrent TLB misses. Conversely, CG shows a negative improvement relative to A57 from prefetching: this is likely because the newer A72 is more able to extract memory-level parallelism through its out-of-order hardware, and so the added cost of prefetching for a benchmark for which irregular data fits in the last level cache is no longer overcome.

Kryo Kryo gives a more polarised result. Qualcomm's own implementation of the ARM ISA gains relatively large improvements for some benchmarks, particularly the Hash Join and Graph 500 workloads, which are relatively complicated access patterns. However, this is offset by the very poor performance for RA. Here, for the first time on any system and benchmark, we see a significant decrease in performance from both manual and automatic prefetching, despite the application being heavily memory bound. Indeed, as we show in section 6.2, it is possible to greatly reduce the performance of Kryo with badly configured prefetches, even though the added cost of the instructions does not appear to justify the penalty.

A53 As the Cortex-A53 is in-order, significant speedups are achieved across the board using our compiler pass. RA achieves a significant improvement in performance because the core cannot overlap the irregular memory accesses across loop iterations at by itself (because it stalls on load misses), so the comparatively high cost of the hash computation within the prefetch is easily offset by the reduction in memory access time. However, autogenerated performance for RA is lower than manual, as the inner loop is small (128 iterations). Though this loop is repeated multiple times, our compiler analysis is unable to observe this, and so does not generate prefetches for future iterations of the outside loop, meaning the first few elements of each 128 element iteration miss in the cache.

In the G500 benchmark, the edge to visited list stride-indirect patterns dominate the execution time on in-order systems, because the system does not extract any memory-level parallelism. Therefore, autogenerated performance is much closer to ideal than on the out-of-order systems.

Xeon Phi The Xeon Phi is the only system we evaluate for which the compiler can already generate software prefetches for some indirect access patterns, using an optional flag. Therefore, figure 4(g) also shows prefetches autogenerated by the Intel compiler's own pass, "ICC-generated".



(a) Integer Sort (IS)

(b) Conjugate Gradient (CG)

(c) RandomAccess (RA)

(d) Hash Join 2EPB (HJ-2)

Fig. 6. Varying look-ahead distance shows the best is usually the consistent across systems.

For the simplest patterns, IS and CG, which are pure stride-indirects, the compiler is already able to generate prefetches successfully. For IS, Intel's compiler is more optimal than ours, due to reducing overhead by moving the checks on the prefetch to outer loops.

As the Intel pass only looks for the simplest patterns, their algorithm entirely misses the potential for improvement in RA and HJ-2, as it cannot pick up the necessary hash computation. Its pass also misses out on any performance improvement for G500, despite the two simple stride-indirects present, from both work to vertex lists and edge to visited lists, likely because it is unable to determine the size of arrays and guarantee the safety of inserting loads to the work list and edge list structures.

We see dramatic performance improvements across the board on this architecture. The in-order Xeon Phi is unable to parallelise memory accesses by itself, so prefetching is necessary for good performance.

Phi KNL The Xeon Phi Knights Landing core is mildly out-of-order, but is seemingly more able to deal with multiple page table walks than the ARM cores. Therefore performance improvement is somewhere between Xeon Phi and Haswell. While for simple patterns the core is somewhat able to reorder accesses, for more complicated patterns, such as Hash Join, very large improvements in performance are attainable. Interestingly, this is the architecture where our prefetching pass is least likely to reach the improvements of manually generating code. This is down to the strategy of code generation in the automatic prefetches where we use value selection in our automated technique to avoid going over the array boundaries, rather than using conditional branching, as is generated manually. The former is more optimal for ARM systems as they feature conditional execution of instructions, whereas the latter seemingly works better for x86. This is merely an implementation choice rather than any limitation of automatic prefetching in most cases.

Stride Prefetch Generation As discussed previously in figure 1, performance for prefetching is optimal when, in addition to the prefetch for the indirect access, a staggered prefetch for the initial, sequentially-accessed array is also inserted. Figure 5 shows this for each benchmark on Haswell for our automated scheme: performance improvements are observed across the board, despite the system featuring a hardware stride prefetcher.

6.2 Microarchitectural Impact

Our compiler prefetch-generation pass creates the same code regardless of target microarchitecture. Given the significantly varying performance improvements attainable on the different machines we evaluate, this may not always be the optimal choice. Here, we consider how the target microarchitecture affects the best prefetching strategy, in terms of look-ahead distances, which prefetches we generate when there are multiple possibilities, and whether we generate prefetches at all. We evaluate this based on manual insertion of software prefetches, to show the limits of performance achievable across systems regardless of algorithm.

Look-Ahead Distance Figure 6 gives speedup plotted against look-ahead distance (c from eq. (1) in section 4.4) for IS, CG, RA and HJ-2 for each architecture. Notably, and perhaps surprisingly, the optimal look-ahead distance is relatively consistent, despite wide disparity in the number of instructions per loop, microarchitectural differences, and varied memory latencies. Setting $c = 64$ is close to optimal for every benchmark and microarchitecture combination. The A53 has an optimal look-ahead slightly lower than this, at 16–32, depending on the benchmark, as does the Xeon Phi on HJ-2, but performance doesn't drop significantly for $c = 64$, and we can set c generously. Kaby running IS is the only example where 128 is significantly better than 64: this is because the benchmark is particularly simple. The trends for other benchmarks are similar, but as there are multiple possible prefetches and thus multiple offsets to choose in HJ-8 and G500, we show only the simpler benchmarks here.

The reasons for this are twofold. First, the optimal look-ahead distance in general for a prefetch is the memory latency divided by the time for each loop iteration [33]. However, for memory bound workloads, the time per loop iteration is dominated by memory latency, meaning that high memory latencies (e.g., from GDDR5 DRAM), despite causing a significant overall change in performance, have only a minor effect on look-ahead distance.

Second, it is more detrimental to be too late issuing prefetches than too early. Although the latter results in cache pollution, it has a lower impact on performance than the increased stall time from the prefetches arriving too late. This means we can be generous in setting look-ahead distances in general, with only a minor performance impact.

RandomAccess on Kryo The case of RandomAccess on Kryo is interesting, as it's the only benchmark-microarchitecture combination which delivers significant slowdown across our test cases. Indeed, with a badly-set look-ahead distance of $c = 4$, where the indirect access is fetched 2 iterations ahead of the loop, we get a 4× slowdown. Though not shown on the graph, this continues for $c = 2$, where slowdown reaches almost 10×. This cannot be explained by the additional overhead of executing the extra instructions, which increase the dynamic amount of code by less than 2×. It also cannot be explained by fetching data we won't use, and thus causing contention in the memory system: we are only prefetching data that will be used, and doing so almost directly before actually using it. Our hypothesis is that this is caused by some pathological, unintended case in the cache system. As in practice, at such small look-ahead distances, data will be loaded while still being prefetched, this causes some sort of negative effect within the MSHRs or similar hardware within the cache, causing a significant temporary lockup.

As this only happens for one benchmark (and likely one of the least representative of real workloads), under badly generated prefetch code, it is likely unimportant for real code. However, it is notable and surprising that bad prefetches can slow down performance so significantly.

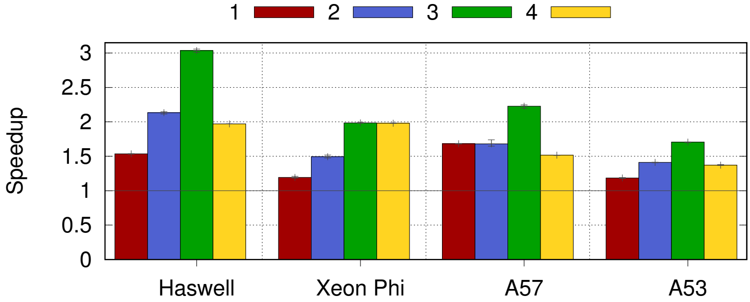


Fig. 7. Performance improvement for prefetching progressively more dependent loads, for HJ-8.

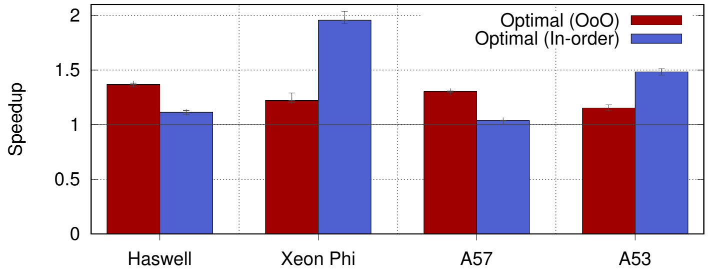
Prefetch Stagger Depth Even when it is possible to generate prefetches for all cache misses, it may not always be the optimal strategy. The extra instructions to prefetch more nested loads (see section 4.4) may outweigh the benefits from issuing the prefetches, because prefetches at one offset require a real load at another for the next prefetch in the sequence. This results in $O(n^2)$ new code, where n is the number of loads in a sequence. We may therefore choose to prefetch fewer loads in a sequence to quadratically reduce the additional code size.

For example, HJ-8 involves a stride-hash-indirect followed by three linked-list elements per bucket. This makes for four irregular accesses per loop iteration. However, as we see from figure 7, for all of the architectures tested, it is optimal to prefetch only the first three of these. This is because the cost of accessing the first three elements to find the address of the fourth, which must be done at a different offset to stagger it from the other prefetches, outweighs the benefits of performing the prefetch. Still, we get variation between the systems, in that the performance penalty for prefetching the fourth is lower for the two in-order systems. This is because, as the systems are more memory bound due to being entirely unable to overlap loop iterations themselves, there is more scope for increasing the instruction count to hide loads.

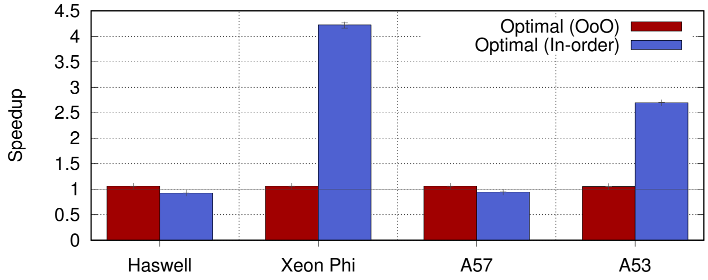
Indeed, for G500, the optimal scheme for in-order systems (figures 8(a) and 8(b)) features a further level of prefetching: as the systems are more memory bound, the extra overhead can be offset. A prefetching pass with more target-specific optimisations would likely have an architecture-specific maximum depth in terms of generated staggered prefetches.

Utility of Prefetches For some systems, generating certain prefetches is not useful. As an example, figures 8(a) and 8(b) show the performance improvements for two sizes of graph on G500 for two different schemes: one optimal for in-order systems, the other for out-of-order. As well as having a deeper stagger depth for the in-order code for prefetches in the outer loop, on in-order cores a large performance improvement is observed from inserting prefetches in the breadth first search's inner loop, where edges of each vertex are checked to see if they have been visited. This is an indirect, irregular access, but often the loops are short, and thus the prefetches are wasted. When systems cannot overlap memory accesses themselves, the prefetches are still useful, but when using out-of-order systems, the added overhead outweighs the benefits. This suggests that we should be more conservative when generating prefetches for targets with out-of-order execution: for example, only generating prefetches for loops we know to feature a large number of iterations and execute on large data structures.

Costs of Prefetching For some benchmarks, the expense of calculating the prefetches outweighs the benefit from reducing cache misses. Figure 9 shows the increase in dynamic instruction count for each benchmark on Haswell. For all but Graph500, dynamic instruction count increases dramatically



(a) Graph500 -s16 -e10



(b) Graph500 -s21 -e10

Fig. 8. Performance of Graph500 with a variety of prefetching schemes. The best manual strategy for Graph500 differs based on micro-architecture, with a more aggressive strategy being useful for in-order architectures.

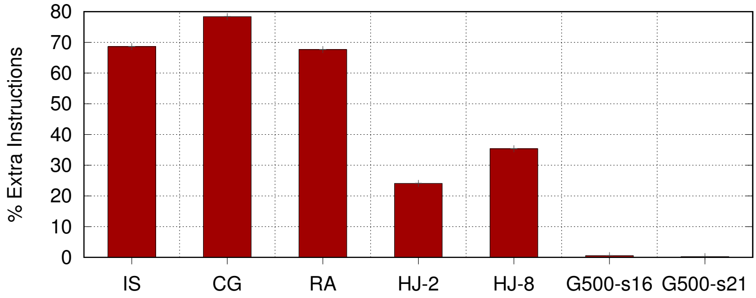


Fig. 9. Percentage increase in dynamic instruction count for Haswell as a result of adding software prefetches, with the optimal scheme chosen in each case.

by adding software prefetching, by almost 70% for IS and RA, and almost 80% for CG. In Graph500 workloads, prefetches reduce performance on Haswell within the innermost loop, and thus are only used on outer loops.

Bandwidth DRAM bandwidth can become a bottleneck for some systems and benchmarks. Out-of-order cores can saturate the bus by executing multiple loops at the same time. We demonstrate this in figure 10. IS running on multiple cores slows down significantly on Haswell, with throughput below 1 for four cores, meaning that running four copies of the benchmark simultaneously on

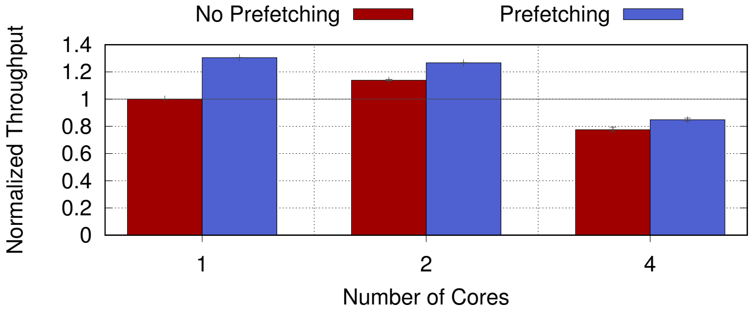


Fig. 10. Throughput for IS on Haswell, normalised to one task running on one core without prefetching. A value of 1 indicates the same amount of work is being done per time unit as the program running on one core without prefetching.

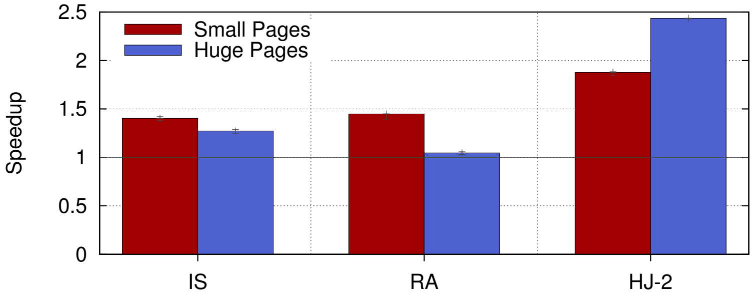


Fig. 11. Speedup for prefetching with transparent huge pages enabled and disabled, normalised to no prefetching with the same page policy.

four different cores is slower than running the four in sequence on a single core. This shows that, even with no overheads from parallelising a workload, the shared memory system is a bottleneck. However, even with four cores, software prefetching still improves performance.

TLB Support All architectures have 4KiB memory pages, but Haswell's kernel also has transparent huge pages enabled. Figure 11 shows the impact of prefetching with and without this support. Huge pages reduce the relative performance improvement attained by our scheme slightly for simpler benchmarks (like IS and RA), because we do not gain as much from bringing in TLB entries that would otherwise miss, as a side effect of software prefetching. However, it increases our performance improvement on page-table-bound benchmarks, such as HJ-2. All other benchmarks are unaffected and, overall, trends stay consistent regardless of whether huge pages are enabled or not.

6.3 Parallel Workloads

Up until this point we have considered only single-threaded applications, since our prefetching technique requires only memory-level parallelism, rather than thread-level parallelism, from code. Here we show that the two concerns are essentially orthogonal, and both can potentially be exploited concurrently to gain benefit.

Figure 12 shows how the same techniques developed here extend to parallel implementations of three of the benchmarks we considered earlier. We see that for HJ2 (figure 12(a)), all systems gain benefit from both prefetching and thread-level parallelism, regardless of the number of cores. Indeed, the speedup from prefetching is largely independent of how many threads are running simultaneously: the same technique works equivalently well in all cases. Similarly for CG (figure 12(b)), while performance is reduced on Haswell and A57 due to the small amount of data to be prefetched fitting in the last level cache, and thus the instruction overhead being too high, on A57 this overhead is made smaller with more cores, and A53 gives a consistent speedup for prefetching regardless of the number of cores.

The pattern for IS (figure 12(c)) is more interesting. While prefetching is always the best choice for performance regardless of the number of threads and system, this is not the case for thread-level parallelism, where on Haswell the optimal performance is using a single thread with prefetching optimisations, and for A53 the performance with prefetching on a single thread is comparable to that with prefetching on four threads (though both perform better than no prefetching with any number of threads). This is a similar pattern to that shown in figure 10, where we ran a single-threaded version of IS simultaneously on multiple cores, and observed the same slowdown compared to running the applications in sequence. This suggests that, rather than due to limitations in parallelism, this is caused by memory-system bottlenecks, and it is therefore interesting that prefetching still increases performance in all cases, rather than exacerbating the contention, likely due to the high accuracy afforded by the technique.

In summary, prefetching can be seen as an orthogonal technique to extracting thread-level parallelism. Single-threaded improvement is a good predictor of multi-threaded improvement for prefetching, and prefetching can improve performance when thread-level optimisation does not, and vice versa.

6.4 Summary

Our autogenerated pass generates code with close to optimal performance compared to manual insertion of prefetches across a variety of systems, except where the optimal choice is input dependent (HJ-8), or requires complicated control flow knowledge (G500, RA).

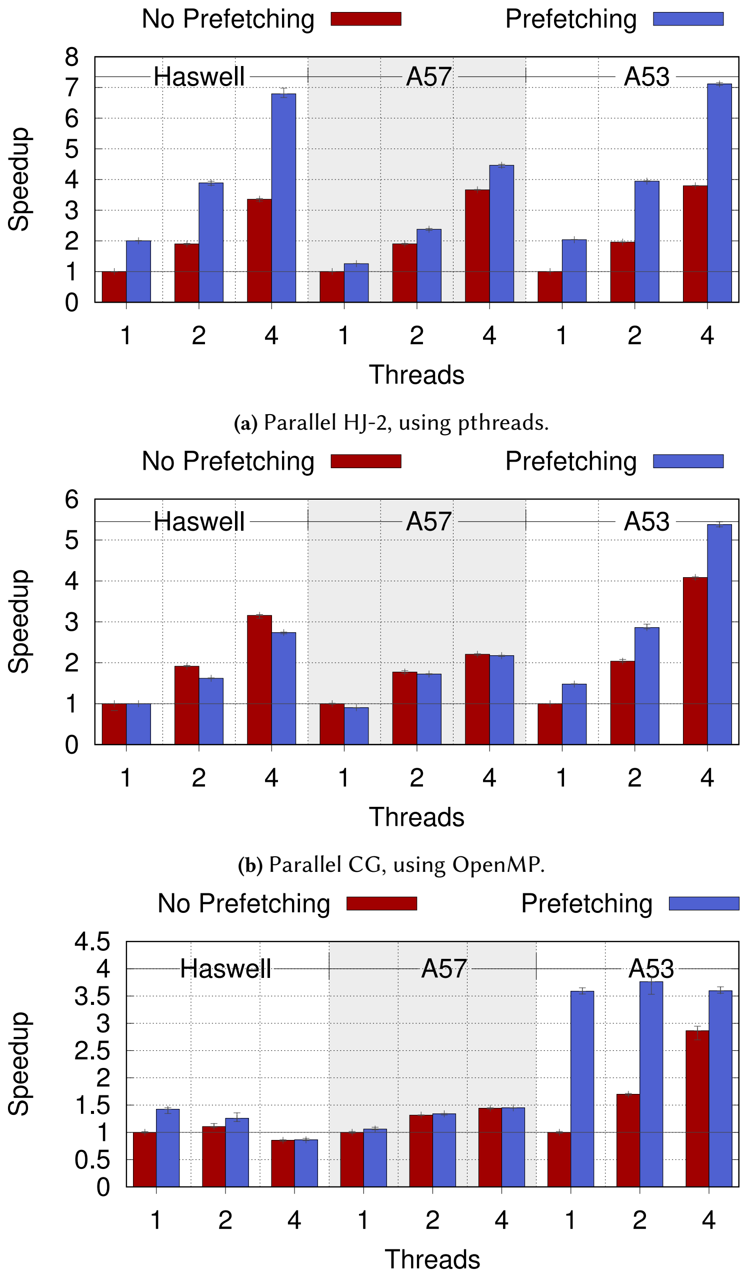
A compiler pass that is microarchitecture specific would only improve performance slightly: similar prefetch look-ahead distances are optimal for all the architectures we evaluate, despite large differences in performance and memory latency. Still, performance improvement can be limited by microarchitectural features such as a lack of memory bandwidth, an increase in dynamic instruction count, and a lack of parallelism in the virtual memory system. Despite these factors, every system we evaluate attains a net performance benefit from our technique.

7 EXPLORING THE WORKLOAD SPACE

Though we have shown performance holds up well for our automated pass on a variety of real-world benchmarks, we are interested to see the entire space of code that might be targeted by such a compiler technique. Specifically,

- does the pass perform well under a wide variety of input sizes, amount of data indirection, and amount of computation per loop;
- does the choice of a constant look-ahead distance hold up even under extreme cases; and
- how practical is attempting to fetch all layers of a multiple-indirect loop?

To answer these questions, we introduce a new configurable benchmark, Kangaroo, named because it jumps around memory in a variety of patterns. We first explore the structure of the code and its configurable parameters, then look at performance under a variety of settings and micro-architectures. We show that our technique holds up well even in extreme cases. We also use Kangaroo and another configurable benchmark we introduce, Camel, to make generalisations



(a) Parallel HJ-2, using pthreads.

(b) Parallel CG, using OpenMP.

(c) Parallel IS, using OpenMP.

Fig. 12. Speedup from combining software prefetching with thread-based parallelisation techniques, normalised to the time taken for the parallel implementation on a single thread.

```
1  #define SIZE_OF_DATA 33554432
2  #define c_0 64
3  for (n in NUMBER_OF_ARRAYS)
4    func_n = USE_HASH(n) ? hash : identity;
5
6  for (i=0; i<SIZE_OF_DATA; i++) {
7    SWPF(array_0[i+c_0]);
8    SWPF(array_1[func_0(array_0[i+c_1]]);
9    ...
10   array_n[...func_1(array_1[func_0(array_0[i])])]++;
11 }
```

Code listing 1. Pseudocode for Kangaroo.

about where prefetching works best: indirect loops with lots of computation per loop iterations on out-of-order architectures, and with little computation for in-order architectures.

7.1 Kangaroo

Our first configurable benchmark is based on Integer Sort from the NAS Parallel suite [5]. Pseudocode is given in code listing 1. This is the code from figure 1(a), generalised in the following ways.

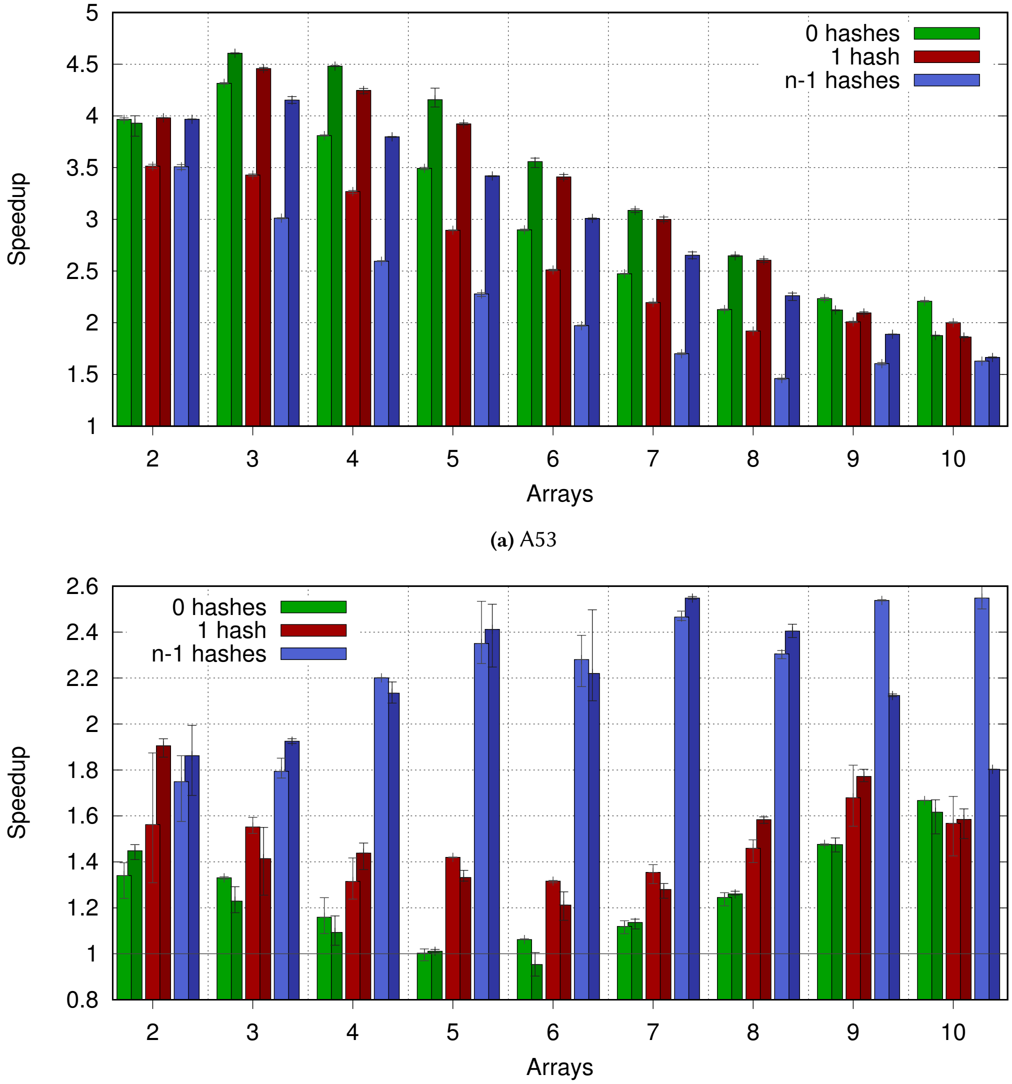
- Instead of a single array of indirection, a configurable number of arrays can be used (line 10), each initialised with random data, to complicate each loop and make prefetches more complex. The more extreme cases are designed to imitate the memory behaviour of linked data structures, as well as illuminate the relationship between prefetching complexity and performance.
- A variable number of manual prefetches can be configured by macro (lines 7–9). For large numbers of arrays, the $O(n^2)$ prefetching code required may become too expensive, and so only prefetching a partial number of the arrays may be more beneficial.
- The data input is allowed to vary (line 1, default 128MiB per array). Smaller inputs should fit in the cache, and so prefetching should be less necessary.
- Code to hash the output of each array [1] can be optionally inserted for each array lookup (line 4). This allows an increasing amount of computation to be performed in each loop.
- The look-ahead distance, c, is configurable (line 2). For configurations with lots of work per loop, 64 may not be the ideal choice.

Configuration is performed statically at compile time so as to ensure overheads are only those introduced by greater numbers of arrays and hash functions. We consider the impact of each of these in turn.

7.2 Number of Arrays

Figure 13 shows performance for manual (lighter) and automatic prefetching (darker), for increasing numbers of arrays and hashes per iteration. For A53, which is in-order, the optimal benefit, from both manual and automatic prefetching, tends to decrease as more arrays are accessed. This is explained by the increase in code required to prefetch: as more arrays access each other, more prefetches need to be inserted, and more loads to stagger increasingly-deep prefetches. Indeed, the code growth is $O(n^2)$ due to the prefetch-staggering, and thus prefetches become costlier.

However, under the same circumstances the story for Haswell is different. The optimal benefit using both manual and automatic prefetching with no hash code initially decreases with increasing numbers of arrays, which can be explained in a similar way to the A53. It then starts to increase again past five arrays, reaching over 1.5× by 10 arrays, despite the very large code increase at this

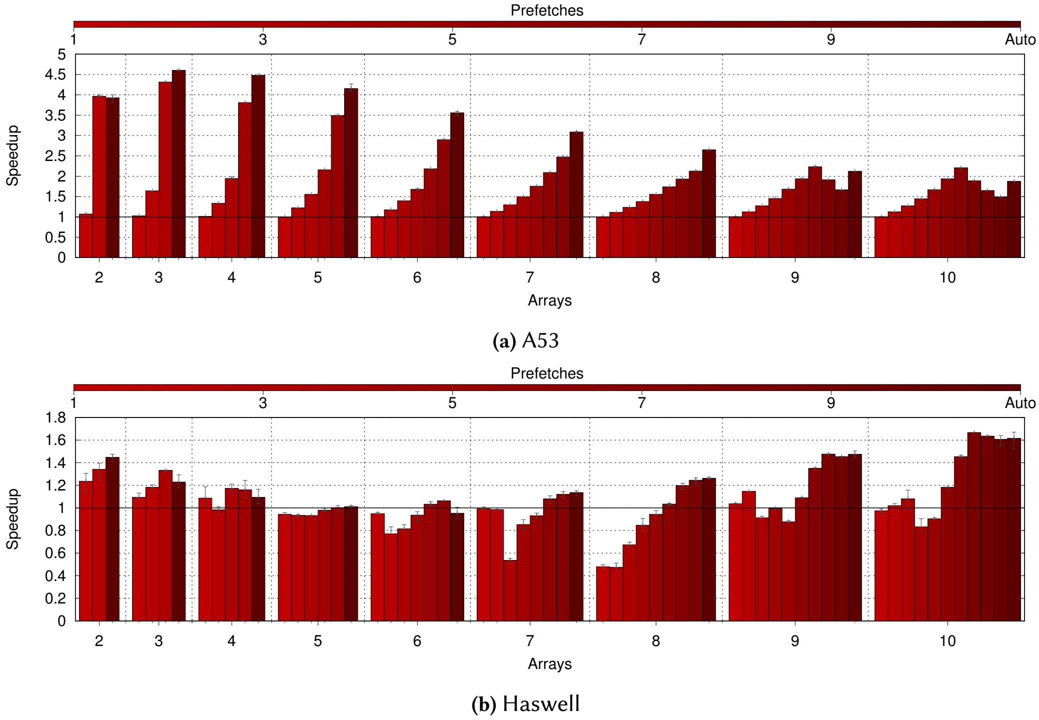


(a) A53

(b) Haswell

Fig. 13. Performance of Kangaroo with increasing numbers of arrays, with the best manual prefetches (lighter), along with automatic prefetching (darker) in each case, with and without hashing.

point: the inner loop increases in size by 7×. This is due to the out-of-order nature of the hardware. For small numbers of arrays, multiple loop iterations can fit in the reorder buffer at once. Thus, there is a tradeoff between extracting memory-level parallelism through prefetching, and through the reorder buffer, as extra prefetching code reduces the number of concurrent loop iterations. However, for larger numbers of arrays, the reorder buffer ends up full from a much smaller number of iterations. This reduces the hardware's innate exploitation of memory-level parallelism, and



(a) A53

(b) Haswell

Fig. 14. Performance of Kangaroo with increasing numbers of arrays and manual prefetches, along with automatic prefetching in each case, without hashing. Though the performance improvement differs, the pattern observed is largely the same with hashing.

thus opens up greater opportunities for prefetching, even at the exceptionally high cost associated with it at such extremes.

7.3 Compute Per Loop

To represent workloads with varying amounts of compute per loop, we add in optional hash functions on the data of every array access. In the case of one hash per loop iteration, this looks functionally very similar to a hash table walk (and thus the Hash Join benchmark).

Figure 13 also shows performance with increasing amounts of hash code per loop iteration. Again, for A53 we see a drop in performance when adding progressively more compute per loop, simply because the workload becomes more compute, and thus less memory, bound. In addition, the prefetches themselves become more expensive due to the added compute necessary to hash the inputs to the prefetches themselves.

However, we see a very different story with Haswell. As with Hash Join, performance with one hash and two arrays results in a larger performance improvement than the base, IS-like case, of two arrays and no hashing. Indeed, this is generally the case, and adding more hashes per loop increases the relative performance improvement further. What makes this highly surprising is that, by making the workload more compute-bound, and despite the prefetches more expensive to compute, the attainable benefits from prefetching actually increase!

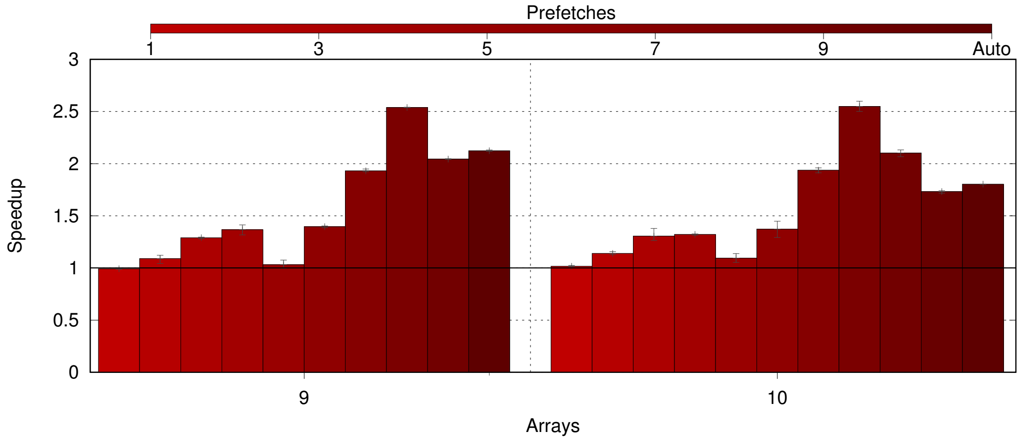


Fig. 15. Performance of Kangaroo for Haswell, with $n - 1$ hashes per loop iteration, for 9 and 10 arrays, and varying numbers of prefetches per loop. As with all other cases using 9 and 10 arrays, prefetching all of them is suboptimal, and this is more pronounced here than in the base case (no hashes) for Haswell.

Indeed, for indirect memory accesses, the traditional concepts of memory-bound vs compute-bound do not clearly apply for out-of-order processors. This is because of the limited reordering resources inside of the processor, which are sometimes able to extract some memory-level parallelism, and at other times are clogged with too much compute to be able to do so. In the latter case, even though prefetching adds to this contention, the prefetch instructions commit quickly (because they do not cause stalls), and allow true loads to also commit quickly through having already been prefetched, therefore the instruction window moves along much more quickly, even though the number of instructions per loop becomes much larger. So, the workloads are simultaneously compute and memory bound.

7.4 Number of Prefetches

The cost of increasingly nested prefetches grows as $O(n^2)$, because all loads preceding each level of prefetching must be repeated. We thus should expect that, at some point, issuing every possible prefetch is a suboptimal choice, particularly for workloads where the number of levels of indirection are very high. While such workloads are likely rare in practice, it is interesting from a theoretical point of view to observe where this occurs.

We do see this happen in Kangaroo, as shown in figure 14. However, this only occurs for 9 arrays of indirection onwards on both Haswell and A53. For 9 arrays, prefetching 8 of these is optimal on Haswell, and prefetching 7 is optimal on A53. This stays true in both cases for 10 arrays, and is also true regardless of the amount of extra compute hashing code: the pattern for Haswell becomes more pronounced with $n - 1$ hashes, as shown in figure 15. What is surprising is the sheer amount of indirection necessary before this occurs: in these cases, the prefetching code has already increased the size of the code by a factor of up to 8.5×. Indeed, automatic prefetching, which targets all of the arrays, still comes close to optimal performance even in these cases. This goes to show how incredibly inefficiently such workloads currently run on modern machines.

Perhaps more of note is the ability, on out-of-order cores, to aggressively slow down execution by only partially prefetching data structures. We see this on Haswell for 7 arrays and 3 prefetches, and 1 to 4 prefetches on 8 arrays. Give that, particularly for 1 prefetch on 8 arrays, this is a very

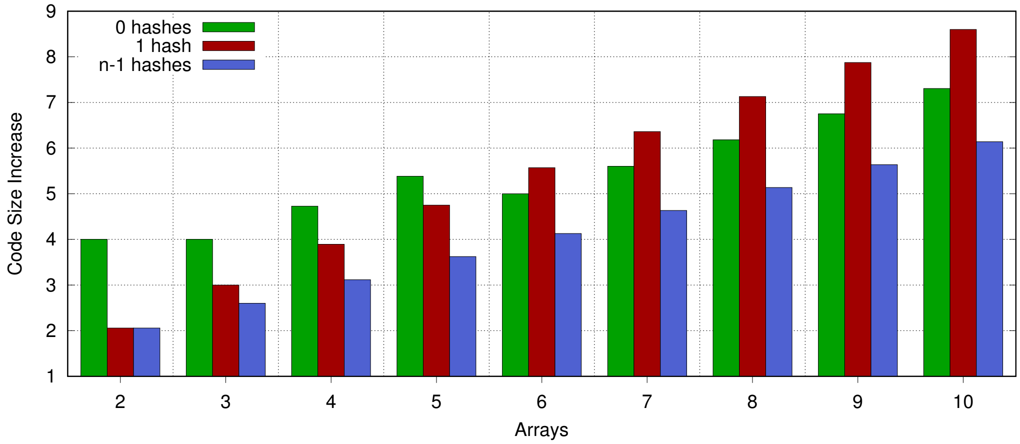


Fig. 16. Increase in static code size on x86-64 for automatic software prefetching, relative to the same code without prefetching, for a single iteration of Kangaroo's inner loop. As the amount of original code grows linearly, and the amount of prefetch code quadratically, the ratio increases as we add more arrays.

small amount of additional code, such a heavy performance impact is unexpected and appears to be related to the out-of-order hardware's instruction window. Without the prefetches, the instruction window is large enough to support two cache-missing memory accesses, from two consecutive loop iterations, simultaneously. However, in some cases, even a small amount of additional prefetch code reduces this such that only one of these loads fits in the instruction window at a time. Thus, we get a slowdown resulting in halved performance. Again, we see an example of code which is memory-bound being limited by the computation capabilities of the CPU itself.

The overall conclusion is that, even in extreme situations unlikely to be seen in regular code, prefetching all data structures in a set of indirections is a good policy to take. So, we again see that a fairly simple compile-time prefetching algorithm, without any microarchitecture-specific heuristics, is a good solution.

7.5 Amount of Code

Figure 16 shows the code size increase for a single innermost loop iteration for automatic prefetching for each test case in Kangaroo. The staggered prefetch code increases quadratically with number of indirections, and the original code only linearly, so relative code size increases as we add more arrays to the inner loop. For 2–5 arrays without prefetching or hashing, the loop is unrolled, so we divide the size of the code by the unroll factor to compensate. Incidentally, this is a potential cause of low performance with software prefetching: the compiler is less willing to unroll prefetched code, due to the size increase, resulting in more branches.

Even with large code size increases of up to 8.5×, we still gain performance improvement. This shows how significantly memory-bound the code is, so much so that even with an increasingly inefficient technique, performance is still improved.

7.6 Look-Ahead Constant

For our original benchmarks, the optimal choice of look-ahead constant c (section 4.4) was always around 64, regardless of workload or microarchitecture. However, the space of Kangaroo covers many more complicated cases, with more work and data per loop iteration. While all our previous

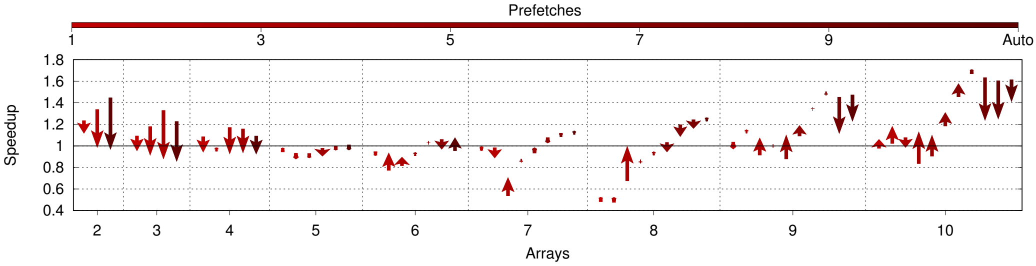


Fig. 17. Performance of Kangaroo for Haswell, with look-ahead constant (c) set to 16 compared with the default 64. For arrows pointing downwards, the smaller look-ahead reduces performance.

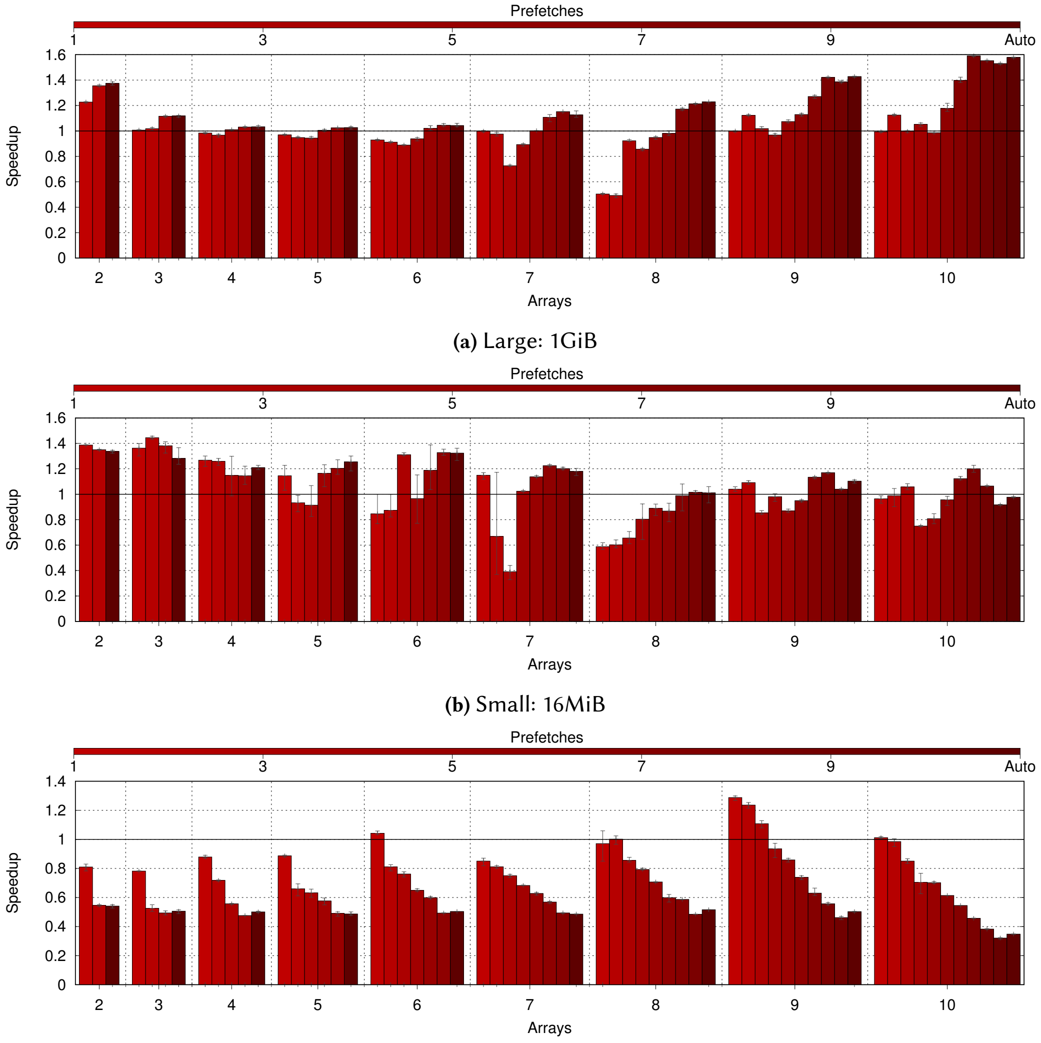
graphs in this section have had c set at 64, figure 17 sets c to 16 instead. Interestingly, performance isn't improved significantly in any case. For setups with smaller amount of work per loop (fewer arrays and/or less computation) $c = 16$ is vastly worse, and for setups with more work, performance is strikingly similar even when the loops are extremely complicated (e.g., 10 arrays per loop iteration). We have previously argued that this is because the amount of loads per loop is most important in setting the ideal look-ahead distance per loop, as these are what cause the code's dominant latency, and our formula in section 4.4 takes this into account. In the extreme cases (lots of arrays), there are more loads within the new prefetches, which aren't taken into account in the scheduling formula directly. This means that lower look-aheads have less of a negative impact. However, as prefetching too late is vastly worse than prefetching too early, performance isn't negatively impacted.

Therefore, we can justify the choice of a very simple scheduling algorithm for prefetches, even in extreme cases. That this constant value continues to work so well even in cases far from those initially intended to be covered by our scheme is surprising, yet explainable by the dominance of cache load misses for such workloads' execution times.

7.7 Different Data Sizes

Previous experiments with Kangaroo are at the default data size of 128MiB per array. However, real applications feature a variety of input sizes. Since the smaller the data, the more can fit in the cache, prefetching is less likely to be useful with smaller data sizes. Indeed, if all the data fits in the cache, we should expect to only pay the computation penalty, and get none of the memory system benefits from prefetching. In extreme cases, with many dependent arrays, and thus a large growth in code size, we should expect a large slowdown as a result.

Figure 18 shows the performance for increasing numbers of arrays and prefetches, as in figure 14, but for larger and smaller datasets. For the large arrays (1GiB), performance improvement is very similar to the default, as the data doesn't fit in the TLB or cache. The performance improvement with 16MiB arrays is lower, as the code is less memory bound, though still usually positive. For the smallest (0.5MiB) arrays, where the data all fits in the L3 cache, performance is usually reduced compared to the baseline due to the code overhead, reaching as low as 0.4×. This is still less overhead than the added compute would suggest, as the code is still somewhat bound by the cache. Indeed, partial prefetches of some data structures sometimes show improvement. For A53 (figure 19), the 16MiB arrays result in similar performance to the previous larger dataset, as the last-level cache is relatively small. However, with 0.5MiB arrays we see significant slowdown of up to 5×, in a similar pattern to Haswell. That the slowdown is so high when the code is less memory bound is



(a) Large: 1GiB

(b) Small: 16MiB

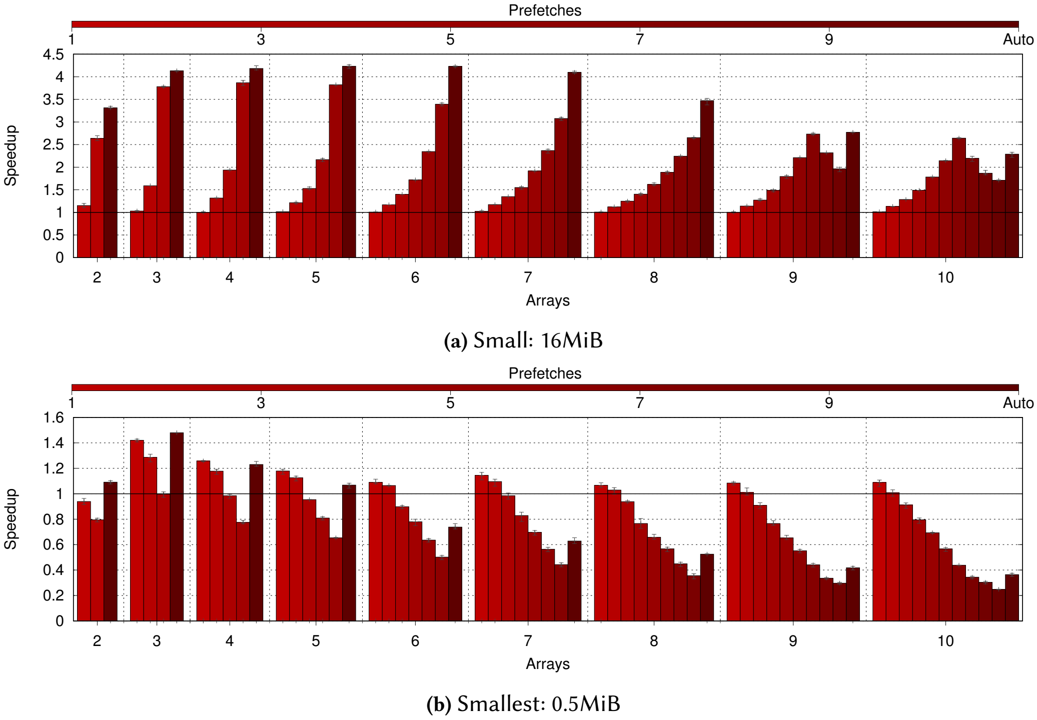
(c) Smallest: 0.5MiB

Fig. 18. Performance of Kangaroo on Haswell with large, small and smallest data set (1GiB, 16MiB and 0.5MiB per array respectively).

an indicator of just how much performance is unrealized with such code using larger datasets, so extremely expensive prefetching strategies still show significant benefit.

7.8 Cost of Prefetching

An interesting question is how much performance is left on the table due to the extra compute cost of the prefetch instructions, which, as we have previously shown in figure 16, results in a large amount of extra code. Figure 20 shows the performance of Kangaroo when prefetch code is added but does nothing useful. To achieve this, the index of the array is anded with 1 for the purposes



(a) Small: 16MiB

(b) Smallest: 0.5MiB

Fig. 19. Performance of Kangaroo on A53 with small and smallest data set (16MiB and 0.5MiB per array respectively).

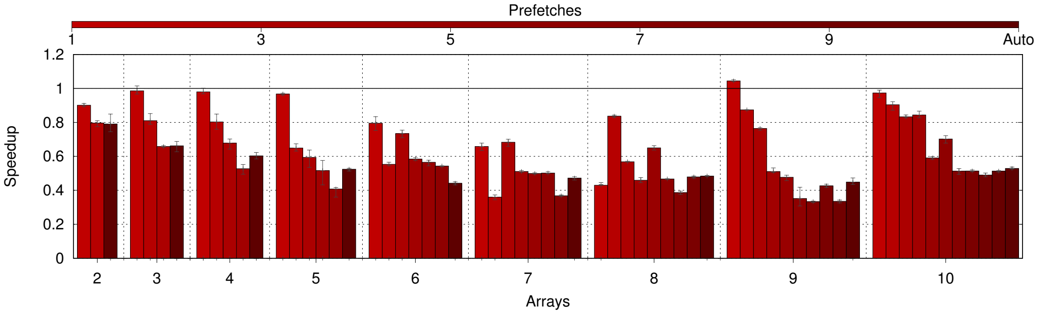
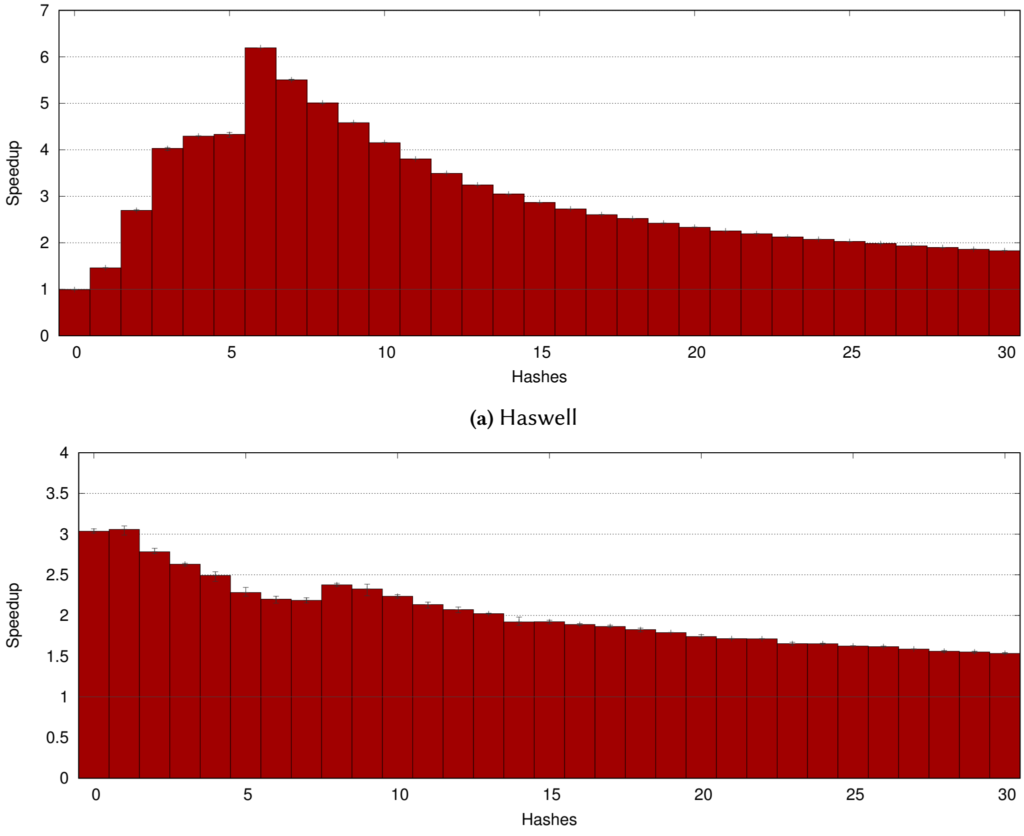


Fig. 20. Performance of Kangaroo for Haswell, with the prefetch calculation code generated, but without useful prefetches.

of prefetching, so that the code cannot be removed by dead code elimination, but the prefetched addresses are always in the cache and the prefetches do not improve performance.

While figure 20 shows a performance reduction, it is not likely to be the true cost of the prefetches: as Haswell is an out-of-order superscalar and the code is memory bound without prefetches, the extra compute should not be too expensive. It is more likely that, as discussed previously, the extra code instead limits the amount of memory-level parallelism that can be extracted from the reorder buffer. Another way of looking at the extra compute cost is by looking at code which isn't memory



(a) Haswell

(b) A53

Fig. 21. Speedup for Camel as a result of prefetching, when varying the number of hash computations done on each data element, but keeping the memory access pattern the same.

```
#define SIZE_OF_DATA 33554432
#define c_0 64
#define c_1 32

for (i=0; i<SIZE_OF_DATA; i++) {
  SWPF(array[i+c_0]);
  SWPF(*array_0[i+c_1]);
  sum += hash(hash(hash...(*array_0[i])));
}
```

Code listing 2. Pseudocode for Camel.

bound, as previously shown in figure 18(c), but this ignores the fact that even with prefetching, for memory-bound workloads the code will still be limited to some extent by the memory system. The true cost of the prefetches, therefore, likely lies somewhere between figure 20 and figure 18(c).

7.9 Exploring Compute Per Loop Further: Camel

It is clear from these experiments that, for loops with more compute between irregular memory accesses, the improvements for software prefetching increase on out-of-order architectures. However, it is not clear to what extent this is true. To explore this further, we introduce another configurable benchmark, Camel, which carries an increasing amount of compute per loop iteration, shown in code listing 2. The memory access itself is the simplest indirect access, being just an array of pointers. This means the prefetch is cheap, and the compute code expensive.

Figure 21 shows the performance improvements attainable. For Haswell (figure 21(a)), what is surprising is the sheer magnitude of performance available from prefetching: for six rounds of hashing, a speedup of over 6× can be achieved, significantly higher than the maximum on the in-order A53. For the exact same memory access pattern with no extra compute code, no speedup is observed. On the flip side, even with 30 hash functions dominating the number of instructions executed, an almost 2× speedup can still be observed from prefetching the memory access pattern. It is extremely clear from these results that making a memory-bound indirect memory access workload more compute-bound paradoxically makes the workload more memory-bound on out-of-order cores, because it prevents them from reordering memory accesses themselves. By comparison, the pattern for A53 (figure 21(b)) is much more intuitive, in that improvement is maximised with a high memory access to compute ratio. However, improvements still stay significant even with large amounts of computation.

7.10 Summary

To see how our pass deals with a wide space of workloads, and further see how different workloads affect different microarchitectures, we have introduced two configurable benchmarks: Kangaroo and Camel. We see that, for in-order architectures, the simpler the pattern is to prefetch, the better the performance improvement, though as code is so heavily memory bound, most patterns can be adequately prefetched with significant performance improvement. For out-of-order architectures, the opposite is true: with complicated access patterns, or lots of compute per loop iteration, the out-of-order hardware alone is less able to extract memory-level parallelism, and thus software prefetching becomes necessary for good performance. Even under extreme use cases, our automatic pass remains remarkably resilient: the look-ahead constant used in previous examples is still a sensible choice, and even for complicated patterns, the overhead of the prefetch instructions, despite growing as n^2, doesn't outweigh the benefits provided reasonably sized data structures.

8 CONCLUSION

While software prefetching appears to make sense as a technique to reduce memory latency costs, it is often the case that prefetching instructions do not improve performance, as good prefetches are difficult to insert by hand. To address this, we have developed an automated compiler pass to identify loads suitable for prefetching and insert the necessary code to calculate their addresses and prefetch the data. These target indirect memory-access patterns, which have high potential for improvement due to their low cache hit rates and simple address computations. Across eight different in-order and out-of-order architectures, we gain average speedups between 1.1× and 2.7× for a set of memory-bound benchmarks. We have investigated the various factors that contribute to software prefetch performance, and developed two new benchmarks, Kangaroo and Camel, to explore the space of code featuring indirection. We discover that, provided the data doesn't fit in the cache, for in-order architectures the simpler the prefetch, the greater performance gained, though such architectures are highly amenable to software prefetching in general. For out-of-order cores, the story is more complicated: prefetching is more useful for complex code with lots of indirection or computation within a loop, as the out-of-order hardware is less able to parallelise the memory accesses itself. Even for benchmarks where code size increases by up to 8.5× in inner loops, we still gain significant performance improvement from software prefetching.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/M506485/1, and ARM Ltd. Additional data related to this publication is available in the data repository at https://doi.org/10.17863/CAM.37731 and at https://github.com/SamAinsworth/reproduce-tocs2019-paper.

REFERENCES

[1] 2012. http://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key#12996028. (2012).

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

[3] Sam Ainsworth and Timothy M. Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[4] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. 2001. Data Prefetching by Dependence Graph Precomputation. In *ISCA*. 10. https://doi.org/10.1145/379240.379251

[5] 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 *SC*.

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

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

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

[9] 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).

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

[11] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A Stateless, Content-directed Data Prefetching Mechanism. In *ASPLOS*.

[12] Babak Falsafi and Thomas F. Wenisch. 2014. A Primer on Hardware Prefetching. *Synthesis Lectures on Computer Architecture* 9, 1 (2014). http://dx.doi.org/10.2200/S00581ED1V01Y201405CAC028

[13] Andrei Frumusanu. 2016. The ARM Cortex A73 – Artemis Unveiled. http://www.anandtech.com/show/10347/arm-cortex-a73-artemis-unveiled/2. (2016).

[14] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. 2014. Fix the Code. Don't Tweak the Hardware: A New Compiler Approach to Voltage-Frequency Scaling. In *CGO*.

[15] M. Khan and E. Hagersten. 2014. Resource conscious prefetching for irregular applications in multicores. In *SAMOS*.

[16] Muneeb Khan, Michael A. Laurenzano, Jason Mars, Erik Hagersten, and David Black-Schaffer. 2015. AREP : Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance. In *PACT*.

[17] Dongkeun Kim and Donald Yeung. 2002. Design and Evaluation of Compiler Algorithms for Pre-execution. *SIGPLAN Not.* 37, 10 (Oct. 2002).

[18] 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*.

[19] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *MICRO*.

[20] 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).

[21] 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*.

[22] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. 2014. SQRL: Hardware Accelerator for Collecting Software Data Structures. In *PACT*.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.

[24] 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.

[25] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. 1995. SPAID: Software Prefetching in Pointer- and Call-intensive Environments. In *MICRO*.

[26] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-based Prefetching for Recursive Data Structures. In *ASPLOS*. 12.

[27] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. CHALLENGES IN PARALLEL GRAPH PROCESSING. *Parallel Processing Letters* 17, 01 (2007).

[28] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) Benchmark Suite. In *SC*. Article 213.

[29] V. Malhotra and C. Kozyrakis. 2006. *Library-Based Prefetching for Pointer-Intensive Applications*. Technical Report. Computer Systems Laboratory, Stanford University.

[30] John D. McCalpin. 2013. Native Computing and Optimization on the Intel Xeon Phi Coprocessor. https://portal.tacc.utexas.edu/documents/13601/933270/MIC_Native_2013-11-16.pdf. (2013).

[31] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. 2001. Slice-processors: An Implementation of Operation-based Prediction. In *ICS*. 14. https://doi.org/10.1145/377792.377856

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

[33] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *ASPLOS*.

[34] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. May 5, 2010. Introducing the Graph 500. *Cray User's Group (CUG)* (May 5, 2010).

[35] Karthik Nilakant, Valentin Dalibard, Amitabha Roy, and Eiko Yoneki. 2014. PrefEdge: SSD Prefetcher for Large-Scale Graph Traversal. In *SYSTOR*. Article 4, 12 pages.

[36] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence Based Prefetching for Linked Data Structures. In *ASPLOS*.

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

[38] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. 2013. Main-memory Hash Joins on Multi-core CPUs: Tuning to the Underlying Hardware. In *ICDE*.

[39] S.P. VanderWiel and D.J. Lilja. 1999. A compiler-assisted data prefetch controller. In *ICCD*.

[40] 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).

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

[42] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect Memory Prefetcher. In *MICRO*.