Software Prefetching for Indirect Memory Accesses

Sam Ainsworth Timothy M. Jones
University of Cambridge, UK
{sam.ainsworth,timothy.jones}@cl.cam.ac.uk

Abstract

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. Across a set of memory-bound benchmarks, our automated pass achieves average speedups of 1.3× and 1.1× for an Intel Haswell processor and an ARM Cortex-A57, both out-of-order cores, and performance improvements of 2.1× and 3.7× for the in-order ARM Cortex-A53 and Intel Xeon Phi.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Compilers, Optimization

Keywords Software Prefetching, Compiler Analysis

1. Introduction

Many modern workloads for high-performance compute (HPC) and data processing are heavily memory-latency bound [10, 13, 18, 25]. The traditional solution to this has been prefetching: using hardware to detect common access patterns such as strides [4, 28], 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 [2] 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 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 [2, 23] and linked-list access patterns [17]. 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. 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 [1], HPC [19], big data [24] and database [26] 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 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.

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. [15] 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

978-1-5090-4931-8/17$15.00

in performance-critical regions of the code, limiting the observable behaviour. By comparison, Mowry [22] considers both Integer Sort and Conjugate Gradient from the NAS Parallel Benchmarks [1], 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 a negligible speedup on an ARM Cortex-A57. In contrast, Chen et al. [3] 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 [11]. These are particularly useful when they can beat the performance of the hardware stride prefetcher, such as in the Xeon Phi [21]. Methods for doing this in the literature directly insert software prefetches into loops, for example Callahan et al. [2]. Mowry [22] 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. [29] 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 [17] prefetch linked lists, and function arguments, such as Lipasti et al. perform [16].

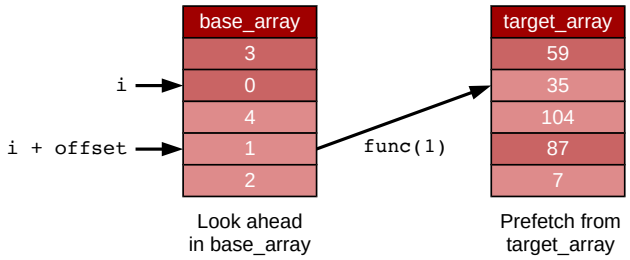
Mowry's PhD dissertation [22] discusses indirect prefetching for high level C-like code. In contrast, in this paper we give a full algorithm to deal with the complexities of intermediate representations, including fault avoidance techniques and value tracking. An algorithm for simple stride-indirect patterns is implemented in the Xeon Phi compiler [12], 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 [27] 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. [7, 8] choose to instead insert software prefetches at runtime using a runtime framework for code modification. This allows prefetching to be done 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. [23] 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. [15] 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.

```
1 for (i=0; i<base_array_size; i++) {
2   target_array[func(base_array[i])]++;
3 }
```

(a) Code containing stride-indirect accesses



(b) Prefetching from target_array

Figure 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.

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. [6] use compiler analysis to duplicate and simplify code, to separate loads and computation. This is to enable 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 [9] use a profile-guided compiler pass to generate "helper threads", featuring prefetch instructions, to run ahead of the main thread. Malhotra and Kozyrakis [20] create helper threads by adding software prefetch instructions to shared libraries and automatically detecting data structure traversals.

Summary Although Mowry has analysed indirect memory access in the past [22], 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

```
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 }
```

Code listing 1: An integer sort benchmark showing software prefetch locations. The intuitive prefetch to insert is only at line 4, whereas optimal performance also requires that at line 6.

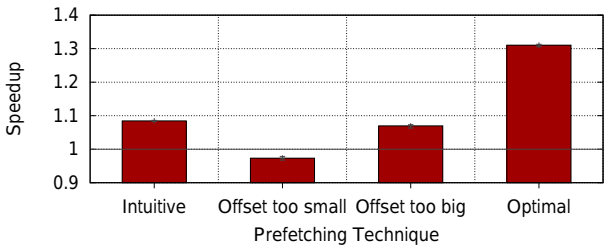


Figure 2: Software prefetching performance for code listing 1 on an Intel Haswell micro-architecture. Inserting software prefetches for maximal performance is a challenge even in simple cases, because intuitive schemes leave performance on the table, and choosing the correct offset is similarly critical for high performance.

`base_array`, followed 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. Code listing 1 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 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, meaning 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 large), or polluting the cache (when the offset is too small).

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.

```
1  DFS(inst) {
2    candidates = {}
3    foreach (o: inst.src_operands):
4      // Found induction variable, finished this path.
5      if (o is an induction variable):
6        candidates U= {(o, {inst})}
7      // Recurse to find an induction variable.
8      elif (o is a variable and is defined in a loop):
9        if (((iv, set) = DFS(loop_def(o))) != null):
10         candidates U= {(iv, {inst}Uset)}
11
12   // Simple cases of 0 or 1 induction variable.
13   if (candidates.size == 0):
14     return null
15   elif (candidates.size == 1):
16     return candidates[0]
17
18   // There are paths based on multiple induction
19   // variables, so choose the induction variable in
20   // the closest loop to the load.
21   indvar = closest_loop_indvar(candidates)
22
23   // Merge paths which depend on indvar.
24   return merge_instructions(indvar, candidates)
25 }
26
27 // Generate initial set of loads to prefetch and
28 // their address generation instructions.
29 prefetches = {}
30 foreach (l: loops within a loop):
31   if (((indvar, set) = DFS(l)) != null):
32     prefetches U= {(l, indvar, set)}
33
34 // Function calls only allowed if side-effect free.
35 remove(prefetches, contains function calls)
36 // Prefetches should not cause new program faults.
37 remove(prefetches, contains loads which may fault)
38 // Non-induction variable phi nodes allowed if the
39 // pass can cope with complex control flow.
40 remove(prefetches, contains non-induction phi nodes)
41
42 // Emit the prefetches and address generation code.
43 foreach ((ld, iv, set): prefetches):
44   off = calc_offset(list, iv, load)
45   insts = copy(set)
46   foreach (i: insts):
47     // Update induction variable uses.
48     if (uses_var(i, iv)):
49       replace(i, iv, min(iv.val + off, max(iv.val)))
50     // Final load becomes the prefetch.
51     if (i == copy_of(ld)):
52       insts = (insts - {i}) U {prefetch(i)}
53   // Place all code just before the original load.
54   add_at_position(ld, insts)
```

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

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.

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

While 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 code listing 1). We therefore need to ensure that any look-ahead values will be valid addresses and, if they are to be used for 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 code listing 1. 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 generated 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 stride software prefetch generation pass which is able to take into account, for example, reuse analysis [23].

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 [23] has calculated prefetch distance using a ratio of memory bandwidth against number of instructions. However, code featuring indirect accesses 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, 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 code listing 1, 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]`.

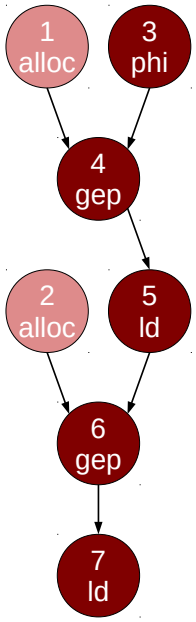
This has the property that it spaces out the look-ahead for dependent loads equally: each is prefetched $\frac{c}{t}$ iterations

```
1  start: alloc a, asize
2         alloc b, bsize
3  loop:  phi i, [#0, i.1]
4         gep t1, a, i
5         ld l2, 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

```
1 add p1, i, #32
2 min p2, p1, asize
3 gep p3, a, p2
4 ld p4, p3
5 gep p5, b, p4
6 prefetch p5
7 add p6, i, #64
8 gep p7, a, p6
9 prefetch p7
```

(c) Generated prefetching code



(b) Depth-first search

Figure 3: Our pass running on an integer sort benchmark.

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 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 [14], 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.

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 [1], 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 [1]. 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 [19], 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, in that we look ahead in the random number array, then perform a hash function on the value to generate the final address for prefetching. Thus, each prefetch involves more computation than in IS or CG.

Hash Join 2EPB (HJ-2) Hash Join [26] 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,

System	Specifications
Haswell	Intel Core i5-4570 CPU, 3.20GHz, 4 cores, 32KB L1D, 256KiB L2, 8MiB L3, 16GiB DDR3
Xeon Phi	Intel Xeon Phi 3120P CPU, 1.10GHz, 57 cores, 32KiB L1D, 512KiB L2, 6GiB GDDR5
A57	Nvidia TX1, ARM Cortex-A57 CPU, 1.9GHz, 4 cores, 32KiB L1D, 2MiB L2, 4GiB LPDDR4
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.

followed by the first linked-list element at offset 12, then offsets 8 and 4 for the second and third respectively.

Graph500 Seq-CSR (G500) Graph500 [24] 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. Haswell and A57 are out-of-order superscalar cores; A53 and Xeon Phi are in-order.

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.

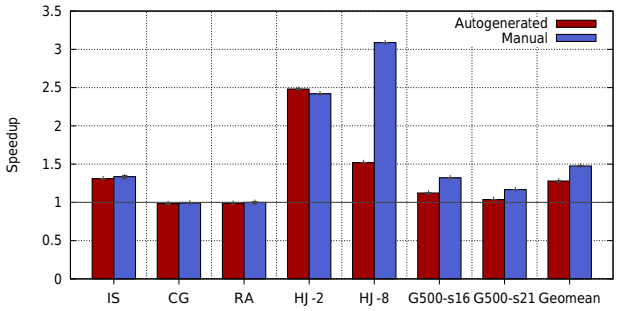
HJ-8 gets a limited improvement. The stride-hash-indirect pattern is picked up by the compiler, but the analysis cannot pick up the fact that we walk a particular number of linked-list elements each time in a loop. This is a runtime property of the input that the compiler could never know, but manual prefetches can take advantage of this 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.

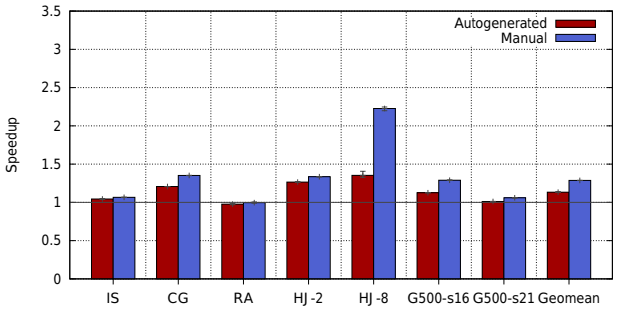
A57 The performance for the Cortex-A57 follows a similar pattern to Haswell, as both are out-of-order architectures. For IS, CG and HJ-2, differences between the automated pass and manual prefetches are simply down to different code generation. However, the A57 can only support one page-table walk at 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 [5], likely improving prefetch performance.

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.

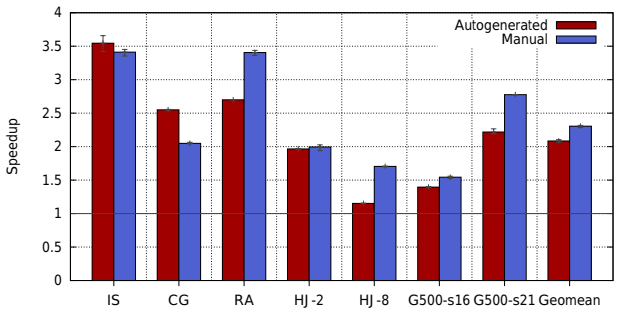
In the G500 benchmark, the edge to visited list stride-indirect patterns dominate the execution time on in-order systems, because the system does not extract any memory-



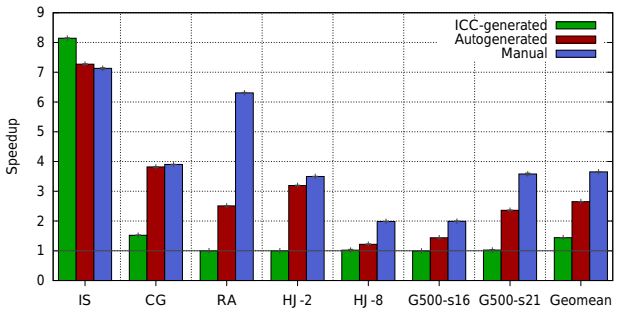
(a) Haswell



(b) A57



(c) A53



(d) Xeon Phi

Figure 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.

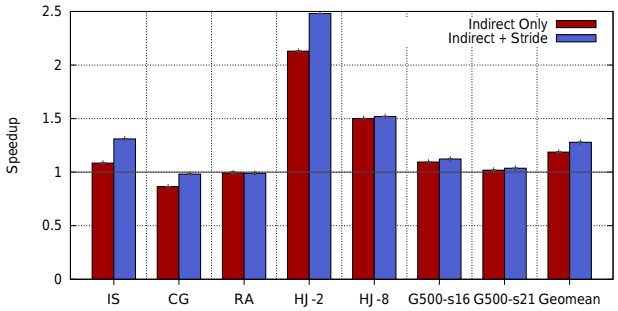


Figure 5: Performance of inserting staggered stride software prefetches along with the indirect prefetch, compared to the indirect alone, for Haswell, with our automated scheme.

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(d) also shows prefetches autogenerated by the Intel compiler's own pass, "ICC-generated".

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.

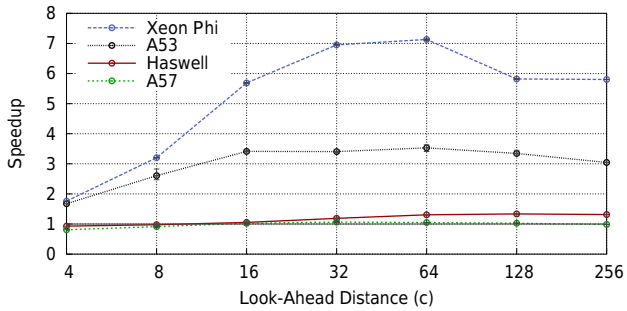
As the Intel pass only looks for the simplest patterns, their algorithm entirely misses the potential for improvement in RA and HJ-2, as it cannot pick up the necessary hash computation. Its pass also misses out on any performance improvement for G500, despite the two simple stride-indirects present, 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.

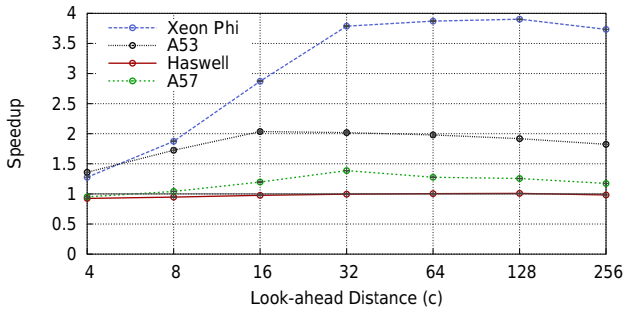
Stride Prefetch Generation As discussed previously in figure 1, performance for prefetching is optimal when, in addition to the prefetch for the indirect access, a staggered prefetch for the initial, sequentially-accessed array is also inserted. Figure 5 shows this for each benchmark on Haswell for our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

6.2 Microarchitectural Impact

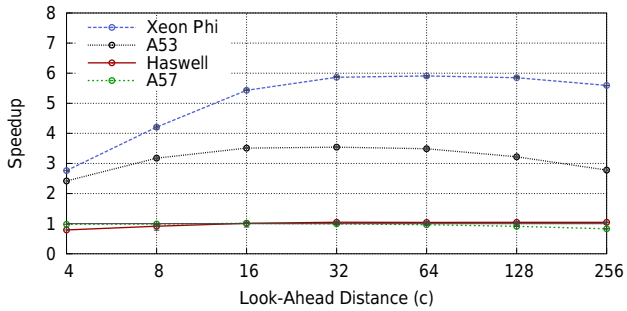
Our compiler prefetch-generation pass creates the same code regardless of target microarchitecture. Given the significantly varying performance improvements attainable on the different machines we evaluate, this may not always be the optimal choice. Here, we consider how the target microarchitecture 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



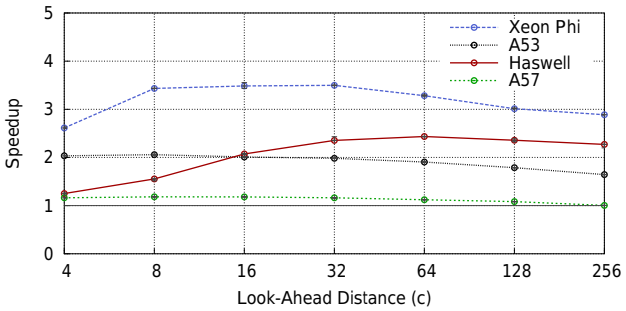
(a) Integer Sort (IS)



(b) Conjugate Gradient (CG)



(c) RandomAccess (RA)



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

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

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. 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 [23]. 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, has 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.

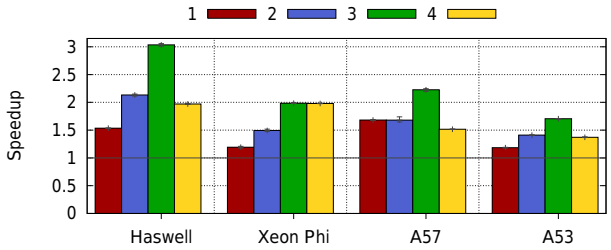


Figure 7: Performance improvement for prefetching progressively more dependent loads, for Hash Join 8EPB (HJ-8)

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.

Costs of Prefetching For some benchmarks, the expense of calculating the prefetches outweighs the benefit from a

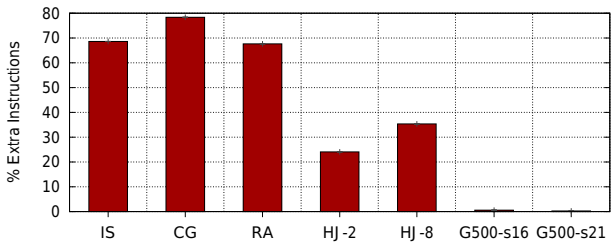


Figure 8: Percentage increase in dynamic instruction count for Haswell as a result of adding software prefetches, with the optimal scheme chosen in each case.

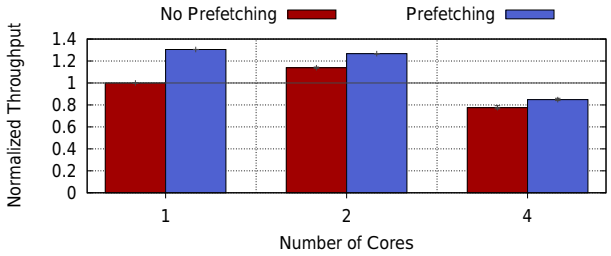


Figure 9: Throughput for IS on Haswell, normalised to one task running on one core without prefetching. A value of 1 indicates the same amount of work is being done per time unit as the program running on one core without prefetching.

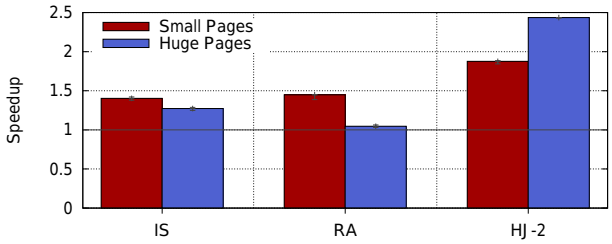


Figure 10: Speedup for prefetching with transparent huge pages enabled and disabled, normalised to no prefetching with the same page policy.

reduction in cache misses. Figure 8 shows the increase in dynamic instruction count for each benchmark on Haswell. For all but the Graph500 benchmarks, dynamic instruction count increases dramatically 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 9. 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 four different cores is slower than running the four in sequence on a single core. This shows that 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 10 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 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 which 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. 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 four different in-order and out-of-order architectures, we gain average speedups between 2.1× and 2.7× for a set of memory-bound benchmarks, and then investigate the various factors that contribute to software prefetch performance.

Acknowledgements

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 `http://dx.doi.org/10.17863/CAM.6349`.

References

[1] 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. The NAS Parallel benchmarks – summary and preliminary results. In *SC*, 1991.

[2] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS*, 1991.

[3] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), Aug. 2007.

[4] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS*, 1992.

[5] A. Frumusanu. The ARM Cortex A73 – Artemis Unveiled. `http://www.anandtech.com/show/10347/arm-cortex-a73-artemis-unveiled/2`, 2016.

[6] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras. Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *CGO*, 2014.

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

[8] M. Khan, M. A. Laurenzano, J. Mars, E. Hagersten, and D. Black-Schaffer. AREP : Adaptive resource efficient prefetching for maximizing multicore performance. In *PACT*, 2015.

[9] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. *SIGPLAN Not.*, 37(10), Oct. 2002.

[10] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the Walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[11] R. Krishnaiyer. 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.

[12] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin, and H. Saito. Compiler-based data prefetching and streaming non-temporal store generation for the Intel(R) Xeon Phi(TM) coprocessor. In *IPDPSW*, 2013.

[13] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips. SQRL: Hardware accelerator for collecting software data structures. In *PACT*, 2014.

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[15] J. Lee, H. Kim, and R. Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1), Mar. 2012.

[16] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *MICRO*, 1995.

[17] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ASPLOS*, 1996.

[18] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01), 2007.

[19] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC challenge (HPCC) benchmark suite. In *SC*, 2006.

[20] V. Malhotra and C. Kozyrakis. Library-based prefetching for pointer-intensive applications. Technical report, Computer Systems Laboratory, Stanford University, 2006.

[21] J. D. McCalpin. 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.

[22] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Computer Systems Laboratory, 1994.

[23] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS*, 1992.

[24] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the Graph 500. *Cray User's Group (CUG)*, May 5, 2010.

[25] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki. Prefedge: SSD prefetcher for large-scale graph traversal. In *SYSTOR*, 2014.

[26] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

[27] S. VanderWiel and D. Lilja. A compiler-assisted data prefetch controller. In *ICCD*, 1999.

[28] V. Viswanathan. 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.

[29] Y. Wu, M. J. Serrano, R. Krishnaiyer, W. Li, and J. Fang. Value-profile guided stride prefetching for irregular code. In *CC*, 2002.

A. Artefact Description

A.1 Abstract

Our artefact provides x86-64 and AArch64 binaries for all of our evaluated benchmarks, along with scripts to use these benchmarks to regenerate the time data for the graphs in the paper. This will allow evaluation of our results on any of the Haswell, Cortex-A57 or Cortex-A53 systems described in the paper (or similar micro-architectures).

We also provide source code for all of our benchmarks and our evaluation pass itself, along with scripts to compile binaries for both x86-64 and ARM machines.

We further provide Collective Knowledge (CK) integration to automatically compile and run benchmarks, and generate graphs.

A.2 Description

A.2.1 Check-List (Artefact Meta Information)

- **Algorithm:** Automated software prefetching.
- **Program:** Graph 500, NAS, HPCC, Hash Join (all sources and binaries included).
- **Compilation:** Clang 3.9 or above.
- **Transformations:** Software prefetch insertion implemented as an LLVM pass.
- **Binary:** Included for Linux (Ubuntu 16.04 recommended) for both x86-64 and AArch64. Source code and scripts included to regenerate binaries.
- **Run-time environment:** Provided binaries are for Linux (Ubuntu 16.04) x86-64 and AArch64, but source code given.
- **Hardware:** We recommend a Haswell i5-4570 for verifying x86 results, or an ARM Cortex-A57 and/or Cortex-A53 for verifying ARM results. Similar systems should give comparable results.
- **Output:** Speedup graphs are output by the CK framework. For scripts without CK, the time is written to files along with program output. These are generated for each figure in the original paper.
- **Experiment workflow:** Collective Knowledge, or manual Linux shell scripts.
- **Publicly available?:** Yes.

A.2.2 How Delivered

Our benchmarks, source code, scripts and CK integration are available on Github: `https://github.com/SamAinsworth/reproduce-cgo2017-paper`.

A.2.3 Hardware Dependencies

We recommend testing on an Intel Haswell (an i5-4570 was our test system, but any should give similar results), an ARM Cortex-A57 (we used an Nvidia TX1), or an ARM Cortex-A53 (we used an Odroid C2). Any similar micro-architectures should give comparable results.

A.2.4 Software Dependencies

Our binary files assume Ubuntu 16.04 on both x86-64 and AArch64 systems, with 64-bit executable support on both. Similar Linux distributions should also work. We assume transparent huge pages are enabled on x86: however, figure 10 shows the expected difference in performance when huge pages are disabled.

No further software requirements are necessary for evaluating the binaries provided. However, if you would like to recompile our binaries from source without using CK, an x86-64 system along with Clang 3.9 or above, and aarch64-gnu-linux-gcc (from the Ubuntu package gcc-aarch64-linux-gnu) are required: the former for both binaries, and the latter for linking AArch64 binaries.

The CK tools may require Ubuntu 16.04 or above (14.04 can sometimes cause issues).

A.3 Installation

Using CK To install CK, do:

```
$ sudo apt-get install python python-pip git
$ sudo pip install ck
```

You can install our repository via CK as follows:

```
$ ck pull repo --url=https://github.com/SamAinsworth
    ↪/reproduce-cgo2017-paper
```

Manual You can pull our scripts and benchmarks from Github:

```
$ git pull https://github.com/SamAinsworth/reproduce
    ↪-cgo2017-paper
```

A.4 Experiment Workflow

Using CK Once you have installed the above packages, simply run:

```
$ ck run workflow-from-cgo2017-paper
```

To run benchmarks continuously without prompts, run

```
$ ck run workflow-from-cgo2017-paper --quiet
```

Manual Navigate to `script/reproduce-cgo2017-paper`.

To run the experiments on an x86-64 machine, run `./run_x86.sh` while in the root directory of our provided evaluation directory tree. Tests should take around 20 minutes on a modern machine.

To run the experiments on an AArch64 machine, run `./run_arm.sh` while in the root directory of our provided evaluation directory tree. Tests should take around two hours on an out-of-order ARM system, and around four hours on an in order ARM system.

To recompile all of our benchmarks (not necessary) on an x86-64 machine, run `./compile_x86.sh` and `./compile_aarch64.sh` while in the root directory of our provided evaluation directory tree.

For low-variance results, we advise running the experiments on a system with no other compute- or memory-intensive applications running in the background.

A.5 Evaluation and Expected Result

Using CK Once you have run the CK workflow, you can then run:

```
$ ck dashboard workflow-from-cgo2017-paper
```

This will include the generated time data from running the experiments, along with reference results for comparison for either Haswell or A57 depending on the architecture.

Manual Move directory to `script/reproduce-cgo2017-paper`. Files for the relevant figures in the paper will be output by the `./run_x86.sh` and `./run_arm.sh` scripts. These will contain the outputs of each program, including a time value. These time values can be compared with the speedups given in the corresponding figures of the paper for similar micro-architectures. Depending on the similarity of the micro-architecture being used for evaluation, speedups of comparable magnitude and trends should be observed.

Example output taken from a Haswell machine is given in the folder `example_data`.

A.6 Experiment Customisation

Scripts are all customisable to allow different fetch distances to be chosen at compile time: check the relevant directories for each benchmark, or `module/workflow-from-cgo-paper/module.py` using CK. Similarly, source and a shared object are provided for the software prefetching pass, to allow it to be applied to other code or with other offsets, if desired. These can be used in the same way exemplified in the currently provided scripts.

Experiments can also be run on different micro-architectures from those we have evaluated: either different x86-64 or AArch64 systems with the provided scripts, or others with slight modification.

If you would like to recompile our software prefetching pass without using CK, you can do so by navigating to `package/plugin-llvm-sw-prefetch-pass` and using make. The Makefile within this folder will have to be reconfigured with `LLVM_DIR` set to the directory of your LLVM Clang source and build.

The software prefetching shared object pass can also be compiled and installed using CK, then used separately:

```
$ ck install package:plugin-llvm-sw-prefetch-pass
 $ clang -Xclang -load -Xclang $(ck find package:
     ↪plugin-llvm-sw-prefetch-pass)/lib/
     ↪SwPrefetchPass.so -O3 ...
```

A.7 Submission and Reviewing Methodology

The artefact for this paper was reviewed according to the guidelines at `http://cTuning.org/ae/submission-20161020.html`.