

Software prefetching for unstructured mesh applications

Ioan Hadade

Oxford Thermofluids Institute
University of Oxford
Oxford, UK
ioan.hadade@eng.ox.ac.uk

Timothy M. Jones

Computer Laboratory
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

Feng Wang

Oxford Thermofluids Institute
University of Oxford
Oxford, UK
feng.wang@eng.ox.ac.uk

Luca di Mare

Oxford Thermofluids Institute
University of Oxford
Oxford, UK
luca.dimare@eng.ox.ac.uk

Abstract—Applications that exhibit regular memory access patterns usually benefit transparently from hardware prefetchers that bring data into the fast on-chip cache just before it is required, thereby avoiding expensive cache misses. Unfortunately, unstructured mesh applications contain irregular access patterns that are often more difficult to identify in hardware. An alternative for such workloads is software prefetching, where special non-blocking instructions load data into the cache hierarchy. However, there are currently few examples in the literature on how to incorporate such software prefetches into existing applications with positive results.

This paper addresses these issues by demonstrating the utility and implementation of software prefetching in an unstructured finite volume CFD code of representative size and complexity to an industrial application and across a number of processors. We present the benefits of auto-tuning for finding the optimal prefetch distance values across different computational kernels and architectures and demonstrate the importance of choosing the right prefetch destination across the available cache levels for best performance. We discuss the impact of the data layout on the number of prefetch instructions required in kernels with indirect-access patterns and show how to integrate them on top of existing optimisations such as vectorisation. Through this we show significant full application speed-ups on a range of processors, such as the Intel Xeon Skylake CPU (15%) as well as on the in-order Intel Xeon Phi Knights Corner (1.99 \times) architecture and the out-of-order Knights Landing (33%) many-core processor.

Index Terms—software prefetching, unstructured mesh, computational fluid dynamics, irregular memory access, memory parallelism, auto-tuning

I. INTRODUCTION

The growing disparity between the speed of the processor and that of the memory system [1] means that applications are dependent on their data being available in the fast on-chip caches for high performance. Data with high temporal or spatial locality is best suited to the cache hierarchy but even so, it is often difficult to avoid compulsory misses that occur when the data is accessed for the first time, or capacity misses that occur when the data is too large for a given cache. The traditional mechanism for dealing with these is prefetching, where the hardware anticipates, in the presence of common and simple access patterns, which data is likely needed by the processor in the near future and loads it into the caches prior to consumption.

As a result, applications that exhibit regular stream/stride memory access patterns usually benefit transparently, thanks to the hardware’s stream and stride prefetchers [2]. However, this is not the case for irregular applications (e.g., unstructured mesh CFD solvers) where the underlying irregular and indirect access patterns make prefetching more difficult to identify in hardware [3], [4]. A viable alternative for such applications is software prefetching, where the programmer or compiler inserts special non-blocking load instructions into the code to bring data into the cache hierarchy early, thereby avoiding expensive misses.

Although the concept and mechanics behind software prefetching are relatively straightforward, implementing and gaining benefit from it can be surprisingly challenging [5]. For software prefetching to be effective, the cost of generating the address and issuing the prefetch instruction must be outweighed by the latency saved from avoiding the cache miss [6]. This is difficult to achieve, because the prefetch instructions and all others required for the address calculation occupy valuable instruction slots within the processor, increasing the work it must perform and, for out-of-order machines, limiting the amount of non-prefetch instruction-level parallelism it can extract. In addition, the distance ahead, or number of loads in advance, to prefetch is hard to get right. If the prefetches are executed too far ahead, the data brought into the caches will likely be evicted before it is consumed thereby leading to cache pollution and an increase in traffic across the memory hierarchy. On the other hand, if prefetches are executed too late, the data will not be present in the cache by the time it is required, leading to sub-optimal performance.

Consequently, although software prefetching can be an ideal mechanism for improving the performance of unstructured mesh applications, there are few examples in the literature where this is demonstrated in applications of significant size and complexity with positive results.

This paper addresses this by demonstrating the implementation and utility of software prefetching in an unstructured finite volume CFD code of representative size and complexity to an industrial application. We present the benefit of auto-tuning for finding the optimal distance values for prefetches and demonstrate the importance of choosing the right prefetch destination depending on the cache hierarchy of the underlying

architecture. We discuss the impact of the data layout on the number of prefetch instructions required in kernels with indirect access patterns and show how the prefetch instructions can be integrated on top of existing optimisations such as vectorisation and abstracted away using generic language constructs. Through this, we show significant full application speed-ups on a range of processors, such as the Intel Xeon Skylake CPU (15%) as well as on the in-order Intel Xeon Phi Knights Corner architecture (1.99 \times) and the out-of-order Knights Landing many-core processor (33%).

II. RELATED WORK

There have been a number of previous studies that demonstrated the benefit and implementation of software prefetching in applications that contain irregular memory access patterns similar to those found in unstructured mesh solvers.

Lee et al. [4] present an evaluation of software prefetching across a number of SPEC CPU 2006 benchmarks. They observe best results in those that either contain irregular access patterns or a significant number of short array streams. This is in contrast to applications with regular/stream memory access patterns, where the addition of software prefetches has either a neutral or a negative effect on performance. The authors attribute this to the fact that: (1) indirect accesses are easier to compute in software rather than in hardware; (2) short array streams are often too short in duration for detection by the hardware prefetchers; (3) regular memory accesses are already prefetched by the hardware and software prefetches can interfere with the training of hardware prefetchers.

A similar study, although exclusively targeting applications with indirect access patterns, is presented by Ainsworth and Jones [6]. Compared to the previous paper, which relies on the manual insertion of prefetch instructions into sections of the program, the authors implement an automated approach via a compiler pass that automatically detects indirect accesses in the source code and generates sequences of instructions for prefetching the index and the data. Their implementation is evaluated across a number of benchmarks from the NAS parallel benchmark suite and across a number of different processor architectures, such as the out-of-order Intel Xeon Haswell and ARM Cortex-A57 CPUs and the in-order ARM Cortex-A53 and Intel Xeon Phi Knights Corner processors. Through their approach, they report positive results ranging between 10-30% on the out-of-order processors and between 2.1-2.7 \times on the in-order architectures averaged across the evaluated benchmarks. Unfortunately we could not use the compiler pass developed by Ainsworth and Jones because it is only incorporated into LLVM, whereas we use a proprietary compiler. The latter are usually preferred for scientific applications that run on high-performance computing systems since they tend to generate significantly faster code on modern processor architectures and usually include a more mature implementation of run-times such as OpenMP.

With regard to real applications (as opposed to benchmarks), Mudigere et al. [7] present the implementation of software

prefetching, among a number of other shared memory optimizations, in FUN3D, an unstructured vertex-centred finite-volume CFD code developed at NASA. Their approach consists of inserting prefetch instructions in edge-based loops at “carefully tuned” distances and locations which results in a 28% speed-up in the execution of these types of kernels on an Intel Xeon Ivy Bridge system. However, the authors provide no implementation details, which makes it impossible to reproduce their work in other unstructured CFD applications.

A similar story is found in the work of Al Farhan et al. [8] who also present a number of optimisations targeting the FUN3D application, albeit on a range of more up-to-date processors. Their implementation of software prefetching is based on exploiting the new instructions provided by the AVX-512 ISA such as `VGATHERPF0DPD` and `VGATHERPF1DPD` to prefetch the vertex/node data required in upcoming iterations of edge-based loops. A drawback to their approach is that the prefetch gather instructions they use are only available on architectures that support the AVX-512PF extensions. Furthermore, the authors provide no details regarding the actual impact that software prefetching has on the performance of edge-based kernels as these are presented together with other optimisations, such as vectorisation.

In conclusion, although there have been a number of studies that have investigated the utility and implementation of software prefetching for applications that contain indirect and irregular access patterns similar to unstructured mesh solvers, there are very few instances where this was performed in real applications. Moreover, where real applications were used, the provided details were scarce as the work usually focused on a broader selection of optimisations and not just software prefetching. The work presented herein aims to address this by describing in more detail than in the literature the steps required to implement software prefetches into an existing unstructured mesh application of significant size and complexity with positive results across multiple architectures.

III. UNSTRUCTURED FINITE-VOLUME CFD SOLVER

A. Overview

The test vehicle for this study is AU3X [9], [10], a cell-centred finite-volume code used for solving the unsteady Favre-averaged Navier Stokes equations on unstructured meshes in both time and frequency domains. The solver obtains steady solutions via pseudo-time marching and time-accurate solutions by dual-time stepping. The flow variables are stored at the cell centres while boundary conditions are applied to “ghost” cells, which mirror the position of the internal cell that is adjacent to the boundary. Inviscid and viscous fluxes are computed for every cell-to-cell and boundary-to-cell interface, whilst flow gradients are calculated at each cell centre using the weighted least-square method. A pictorial representation of the finite-volume scheme on unstructured grids is shown in Fig. 1.

The inviscid fluxes are computed by the upwind scheme using the approximated Riemann solver of Roe where second-order accuracy is obtained using the Monotonic Upwind

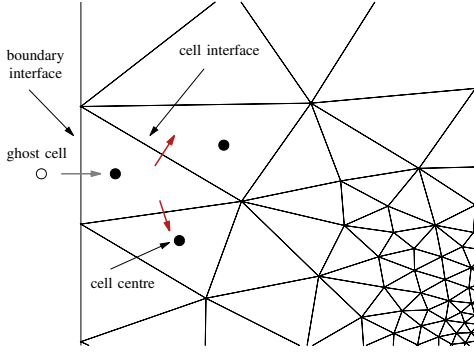


Fig. 1. Schematic of the cell-centred finite volume scheme on unstructured grids.

Scheme for Conservation Laws (MUSCL) and the van Albada limiter. Viscous fluxes are computed by a central differencing scheme using the inverse of the distance weighting from those evaluated at the cell centres on both sides of the interface. The solver supports a range of turbulence closures for high and low Reynolds flows such as mixing length, Spalart-Allmaras, $k-\epsilon$, $k-\omega$ and $k-\omega$ shear stress transport (SST), while convergence is reached either via Jacobi or Generalized Minimal Residual (GMRES) iterations. In this work, turbulent viscosity is computed via the Wilcox $k-\omega$ model while convergence is reached by means of Newton-Jacobi iterations.

The application is structured as a set of C++ classes embodying different types of gas and turbulence models and makes heavy use of polymorphism to execute them at run-time, depending on user selection. The code is optimised for both distributed and shared-memory systems and achieves good scalability on more than 10^4 MPI ranks. The core solver is fully vectorised based on a combination of OpenMP 4.0 directives and compiler intrinsics with support for AVX/AVX2/AVX-512 and IMCI ISA SIMD extensions. Readers are referred to Hadade et al. [11] for further details.

B. Computational kernels and access patterns

The solver spends 75% of its time per non-linear iteration in loops that iterate over the cell-to-cell interfaces (henceforth, faces) for computing numerical fluxes (see Fig. 1), gradients and limiters. An example of a simplified face-based loop can be seen in listing 1 where the unknowns q (i.e., flow variables) are gathered from the face end-points (i.e., cells) using indices $i1$ and $i2$ from the $indx$ connectivity array. The unknowns are then used together with face geometric properties (e.g., normals) to compute the flux residual f which is subsequently accumulated and scattered back to the face-end points in the res array.

The gather and scatter operations that arise from the indirect access in q and res via the $indx$ connectivity array can operate across large and irregular strides in memory and are determined by the mesh topology. As a result, the hardware prefetchers are often unable to anticipate which cells will be referenced in upcoming iterations due to the fact that these

```

1 for( ic=ics;ic<ice;ic++ )
2 {
3     // load indices
4     i1= indx[0][ic];
5     i2= indx[1][ic];
6     // gather the unknowns
7     u1= q[i1];
8     u2= q[i2];
9     // load face properties
10    wc= geo[ic];
11    // compute flux residual
12    f= wc*(u2-u1);
13    // accumulate and scatter-back
14    res[i1]-= f;
15    res[i2]+= f;
16 }

```

Listing 1. Example of a simple face-based loop (scalar) on unstructured grids.

are traversed in non-consecutive order and are referenced indirectly. On the other hand, the memory accesses in $indx$ and geo are contiguous and at unit-stride and therefore most likely in the remit of the hardware prefetchers. However, the benefit obtained from prefetching these regular accesses in face-based loops is most likely overwhelmed by the high cost in latency incurred from performing the gather and scatter operations. As a result, these types of kernels are good candidates for software prefetching since the indirect indexing arising from the gather and scatter operations can be computed and prefetched easily.

The solver spends the remaining time per non-linear iteration in kernels that iterate over the cells in the domain for computing source terms or for updating state vectors. These kernels usually exhibit a low arithmetic intensity and contain regular and unit-stride memory-access patterns. As a result, software prefetches for these loops would only be required in the absence of stream and stride hardware prefetchers or in the event that hardware prefetchers are not available across the entire cache hierarchy (e.g., in the Intel Xeon Phi Knights Corner).

C. Test case

The test case used in this study represents an aero-engine intake operating near ground. The computational domain contains 3.2 million degrees of freedom and is based on an unstructured mesh (Fig. 2) where near wall regions are discretised with hexahedral elements for boundary layer prediction whilst the free stream domain is discretised using prismatic elements.

IV. IMPLEMENTATION

Table I shows the systems that were used as experimental platforms in this study and provides details regarding their architectural characteristics and configuration.

A. Baseline

Listing 2 shows the structure and layout that is used for face-based kernels throughout the code. Since face-based loops exhibit a relatively large number of floating point operations per byte of data retrieved from main memory, vectorising them is crucial for extracting high performance out of modern processors. This is achieved in our code by a combination

TABLE I
DETAILS OF THE SYSTEMS USED AS EXPERIMENTAL PLATFORMS.

	Sandy Bridge (SNB)	Broadwell (BDW)	Skylake Sever (SKX)	Knights Corner (KNC)	Knights Landing (KNL)
Version	E5-2650	E5-2680	Gold 6140	7120P	7210
Year of release	2012	2016	2017	2013	2016
Execution	out-of-order	out-of-order	out-of-order	in-order	out-of-order
Sockets	2	2	2	1	1
Cores per socket	8	14	18	61	64
Clock (GHz)	2.0	2.4	2.3	1.2	1.3
L1 Cache (KB)	32	32	32	32	32
L2 Cache (KB)	256	256	1024	512	1024
L3 Cache (MB)	20	35	25	-	-
Memory (GB)	32	128	196	16	96/16
Memory type	DDR3	DDR4	DDR4	GDDR5	DDR4/MCDRAM
Compiler	icpc 17.0				

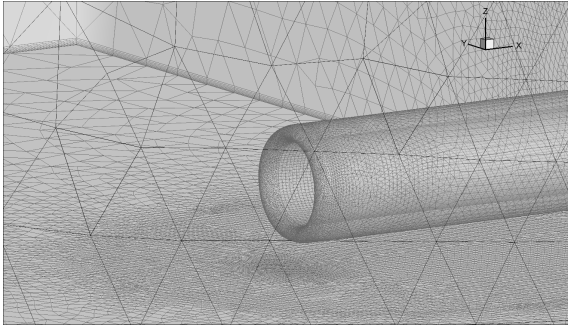


Fig. 2. Unstructured mesh of intake with hexahedra elements at near wall regions and prisms in the free stream domain.

of OpenMP 4.0 compiler directives, compiler intrinsics and by rewriting the face-based kernels in a format that is more suitable for exploiting vector-level parallelism. This can be observed in listing 2 where the main loop over the faces is divided into three distinct stages that map to the underlying gather, compute and scatter pattern and where each iteration processes a number of consecutive faces equal to the underlying vector register width as defined by the `VECLEN` macro (line 6). In the first stage, cell-centred data is gathered into local short-vector arrays declared on the stack using compiler intrinsics that are abstracted away in `vgather`. The compiler intrinsics are used to load the cell-centred data stored in an Array of Structures (AoS) layout using aligned vector loads and transpose it into the Structure of Arrays (SoA) format for efficient vector computations.

Since faces are accessed consecutively, loading face properties such as normals (line 12) is performed via the `vload` routine which uses aligned vector loads that are optimised for the underlying SIMD architecture. Once all data has been gathered or loaded, the next stage performs the actual computation via a nested loop that iterates over the lanes of the vector registers (line 15). This loop is easily vectorised by most modern compilers in part due to the use of the OpenMP 4.0 directive and the fact that all dependencies have been removed

```

1 double ql[NPDE][VECLEN],qr[NPDE][VECLEN];
2 double rl[NPDE][VECLEN],rr[NPDE][VECLEN];
3 double fl[NPDE][VECLEN],fr[NPDE][VECLEN];
4 double f[NPDE][VECLEN],wc[4][VECLEN];
5
6 for( ic=ics;ic<ice;ic+=VECLEN )
7 {
8     // gather from face end-points
9     vgather(q,n,&indx[0][ic],&indx[1][ic],ql,qr);
10    vgather(r,n,&indx[0][ic],&indx[1][ic],rl,rr);
11    // load face normals and area
12    vload(geo,nx+1,ic,wc);
13    // compute flux residual across all vector lanes
14    #pragma omp simd simdlen(VECLEN) safelen(VECLEN)
15    for( iv=0;iv<VECLEN;iv++ )
16    {
17        // assemble fluxes -- truncated
18        f[0][iv]= 0.5*(fr[0][iv]+fl[0][iv])*wc[3][iv];
19        // accumulate
20        rl[0][iv]-= f[0][iv];
21        rr[0][iv]+= f[0][iv];
22    }
23    // scatter-back to face end-points
24    vscatter(r,n,&indx[0][ic],&indx[1][ic],rl,rr);
25}

```

Listing 2. Original implementation of face-based kernels to aid vectorization

and computations are carried out on stack'ed vectors of sizes known at compile time.

Lastly, the third and final stage performs the scatter operation using the `vscatter` routine where the residuals at the cell-centre are transposed back from SoA into the AoS format and scattered to the face-end points using aligned SIMD stores. This is possible due to: (1) the inter-structure locality offered by the AoS format at the granularity of each cell where variables such as the residuals or unknowns are stored contiguously; (2) faces have been reordered at solver initialisation so that there are no dependencies at the end-points in groups of consecutive faces of size equal to `VECLEN`.

B. Inserting software prefetches

The `vgather`, `vload` and `vscatter` routines utilised in every face-based loop are not only useful for decoupling the

movement of data from the computation but also allow for the implementation of architecture-specific optimisations, such as using different compiler intrinsics depending on the supported SIMD ISA (e.g., AVX/AVX-512/IMCI). Furthermore, these routines are also the best place for inserting software prefetches. Thus, as data is gathered and transposed from the group of cells traversed in vector iteration i , prefetches are also issued for the data of all the cells that will be visited in the vector iteration $i + d$ where d is the distance parameter. Furthermore, prefetches are also executed for the indices in the connectivity arrays that are used for referencing the cell-centred data in `vgather`. Otherwise, cache misses that result from accessing the connectivity array would offset any benefits obtained from prefetching the actual data. Ainsworth and Jones [6] demonstrated that the optimal distance for prefetching the indices is twice that of the actual data. We use the same ratio in this work and prefetch indices stored in the connectivity arrays at twice the distance of the cell-centred data that is being referenced. Prefetches are also inserted for the regular and contiguous access patterns in `vload` in order to test whether we can outperform the hardware prefetcher.

Listing 3 shows the `prefetchi` routine used for prefetching the indices (lines 1–15), the `prefetchd` routine used for prefetching the cell-centred data (lines 17–38), and demonstrates how this is implemented within the `vgather` primitive (lines 40–47). Finally, listing 4 presents the structure of a face-based loop that includes software prefetching.

C. Impact of data layout

The efficiency of the software prefetching implementation is also dependent on the choice of data structures used for the cell-centred variables. A side effect of using the AoS data layout is that successive variables within the structure are loaded as well at cache line granularity. This means that compared to SoA, the AoS layout only requires a single prefetch instruction per cell data structure since consecutive entries will be loaded in the same cache line. For example, let us consider the flow and turbulence variables stored in `q`. In the AoS layout, these are stored for each cell-centre as $[u, v, w, t, p, k, \omega]$. In the SoA format, each component would be stored in a separate vector of length equal to the number of cells in the domain. As a consequence, prefetching all of the unknowns in SoA would require seven distinct prefetch instructions per cell-centred data structure since every component is located in a different array. This number increases quickly in the context of a vector iteration where four or eight faces are processed in parallel and where 56 or 112 prefetch instructions are required for the unknowns (seven variables \times two cells per face \times four/eight faces). In contrast, the AoS layout requires a single prefetch instruction referencing the first position (i.e., u) in the cell-centred data structure as all successive entries are loaded at a cache line granularity. Thus, prefetching all of the unknowns in one vector iteration using the AoS format requires either eight or sixteen prefetch instructions depending on the number of faces processed in parallel. This difference in the number of prefetch instructions required per loop iteration is important

```

1 void prefetchi(int *pos1, int *pos2)
2 {
3     # if defined L2_INDEX_PF
4     _mm_prefetch((char *)
5         &(pos1[L2_INDEX_PF]), _MM_HINT_T1);
6     _mm_prefetch((char *)
7         &(pos2[L2_INDEX_PF]), _MM_HINT_T1);
8     # endif
9     # if defined L1_INDEX_PF
10    _mm_prefetch((char *)
11        &(pos1[L1_INDEX_PF]), _MM_HINT_T0);
12    _mm_prefetch((char *)
13        &(pos2[L1_INDEX_PF]), _MM_HINT_T0);
14    # endif
15 }
16
17 template < typename type >
18 void prefetchd(type *data, int *pos1, int *pos2)
19 {
20     # if defined L2_DATA_PF
21     for( int iv=0; iv<VECLEN; iv++ )
22     {
23         _mm_prefetch((char*)
24             &(data[pos1[L2_DATA_PF+iv]]), _MM_HINT_T1);
25         _mm_prefetch((char*)
26             &(data[pos2[L2_DATA_PF+iv]]), _MM_HINT_T1);
27     }
28     # endif
29     # if defined L1_DATA_PF
30     for( int iv=0; iv<VECLEN; iv++ )
31     {
32         _mm_prefetch((char*)
33             &(data[pos1[L1_DATA_PF+iv]]), _MM_HINT_T0);
34         _mm_prefetch((char*)
35             &(data[pos2[L1_DATA_PF+iv]]), _MM_HINT_T0);
36     }
37     # endif
38 }
39
40 template < typename type >
41 void vgather(type *s, int n, int *p1, int *p2,
42             double d1[][VECLEN], double d2[][VECLEN])
43 {
44     // prefetch
45     prefetchd(s, p1, p2);
46     ...
47 }

```

Listing 3. Routines used for inserting software prefetches for the indices in the connectivity arrays and the referenced data together with an example of how they are executed.

since these instructions, although non-blocking, do take up valuable instruction slots and can therefore result in noticeable overheads with detrimental effects on performance.

D. Auto-tuning

Finding the optimal value of d across all kernels and for every distinct architecture can only be achieved by means of auto-tuning. In this work, the auto-tuning phase is executed once on every processor. This is based on a shell script that compiles the application with different distance values and destinations for the prefetches (i.e., L1 and L2, L2 only). In the context of face-based loops, the auto-tuner only has to search through multiple ranges of index values across both L1 and L2 cache levels since the values for prefetching the data are automatically set as twice less than the index values.


```

1 for( ic=ics;ic<ice;ic+=VECLEN )
2 {
3     // prefetch the indices, only once
4     prefetchi(&pos1[ic],&pos2[ic]);
5     // gather (implicit prefetch)
6     vgather(q,n,&pos1[ic],&pos2[ic],ql,qv);
7     // load (implicit prefetch)
8     vload(geo,nx+1,ic,c);
9     ...
10 }

```

Listing 4. Layout and structure of face-based kernel with software prefetching.

The starting value for prefetching the indices into the L1 cache is equal to the size of a vector iteration on the underlying processor (e.g., four on Sandy Bridge and Broadwell and eight on Skylake, Knights Corner and Knights Landing in double precision). The auto-tuner performs the first run executing software prefetches into the L1 cache only. After that, three more runs are performed that also include prefetches into the L2 cache starting at an index value that is twice that of the L1 value and that is increased in powers of two increments. After these runs are executed, the auto-tuner will then increase the distance value for L1 prefetches also in powers of two increments and start the same process again. For example, on the Skylake processor, the first run is performed executing prefetches into the L1 cache only and at a distance of eight for the indices and four for the actual data. This is then followed by three more runs where the indices and data are also prefetched into the L2 cache starting from a distance of sixteen for the indices and eight for the data and increasing in powers of two increments in every run. The value for prefetches into the L1 cache then gets incremented to 16 and the whole process starts again.

The distances used for prefetching the cell-centred data are also used for the regular access patterns in `vload` in order to reduce the amount of time required by the auto-tuning phase.

V. RESULTS AND DISCUSSIONS

We present the results of our implementation across the Intel Xeon Sandy Bridge, Broadwell and Skylake CPUs and the Intel Xeon Phi Knights Corner and Knights Landing processors. We show results for both the whole application (overall) and also for the top four face-based kernels (table II) where the largest percentage of run-time is spent in order to assert whether prefetch parameters such as distances vary between different architectures as well as computational kernels.

a) Sandy Bridge: On the Sandy Bridge CPU (Fig. 3), the advantage of software prefetching is minimal. The best results are obtained in the kernels computing the 2nd order MUSCL fluxes (`iflux`) where an 11% speed-up is achieved when prefetching the data at a distance of sixteen in the L1 cache (eight for the index) and no prefetches in the L2 cache. However, overall, software prefetching actually leads to a reduction in performance rather than an improvement.

b) Broadwell: The results on the Broadwell system are even worse than those on Sandy Bridge and are presented in

TABLE II
CHARACTERISTICS OF THE TOP FOUR FACE-BASED KERNELS IN THE APPLICATION BASED ON THEIR PERCENTAGE OF OVERALL RUN-TIME.

kernel	runtime (%)	flops/bytes	description
iflux	13	1.30	2nd order inviscid fluxes
vflux	8	0.80	2nd order viscous fluxes
diflux	32	0.84	linearised inviscid fluxes
dvflux	30	0.80	linearised viscous fluxes

Fig. 4. Software prefetching on Broadwell sees no improvement across any of the face-based kernels and leads to a drop in overall application performance for every configuration. The reason for the poor performance as a result of software prefetching for both Sandy Bridge and Broadwell systems is likely related to the smaller L2 cache compared to the other architectures which leads to capacity misses due to the large volume of data that is accessed per vector iteration in the face-based kernels. This is further attested by the fact that best performance on Broadwell and to some extent on Sandy Bridge are obtained in runs where prefetches are only executed for the L1 cache.

c) Skylake: The impact of software prefetching on Skylake is presented in Fig. 5. Compared to the Sandy Bridge and Broadwell systems, software prefetching results in a 15% full application speed-up on the Skylake system and between 15% and 36% improvement across the four face-based kernels. We attribute this to the larger L2 cache in the Skylake system which is four times greater than the L2 caches on the Sandy Bridge and Broadwell systems as well as the fact that the L3 cache on Skylake is configured as a victim cache. As a result, we performed an auto-tuning phase on Skylake where prefetch instructions were only executed for the larger L2 cache. This configuration obtained best performance thus validating our claims.

d) Knights Corner: Software prefetching exhibits substantial speed-ups on the Knights Corner coprocessor as presented in Fig. 6. This is to be expected since the in-order core design requires either more than one thread per core or software prefetching for avoiding pipeline stalls due as a result of a cache miss. Furthermore, the Knights Corner architecture can execute prefetch instructions on both pipes in the VPU and does not incorporate an L2 hardware prefetcher. As a result, our implementation leads to $1.99\times$ speed-up for the full application and between $1.81\times$ and $4.10\times$ in the four face-based kernels.

e) Knights Landing: For Knights Landing, software prefetching resulted in a 33% speed-up for the full application and between 54% and 98% improvement in the face-based kernels (Fig. 7). We attribute this to the fact that the L2 cache on the KNL, although shared by the two cores in a tile, is still twice as large as the one on Sandy Bridge and Broadwell. As a result, executing software prefetches only in the L2 cache on KNL obtains the best performance compared to prefetching across both the L1 and L2 caches. This is also due to the fact

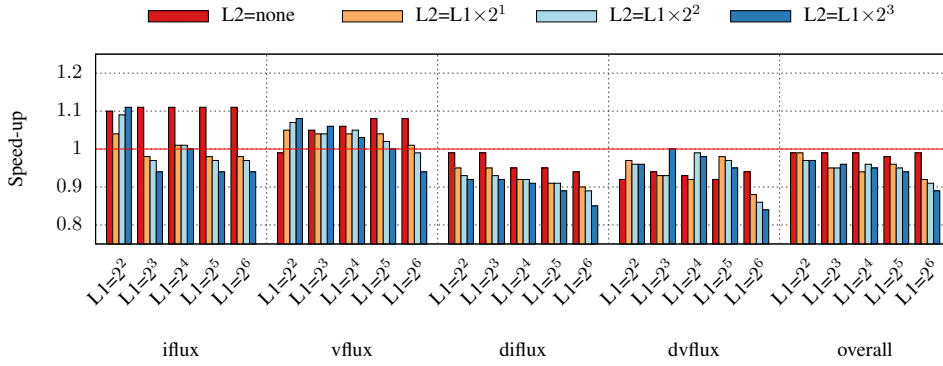


Fig. 3. Results of performing software prefetching on the Intel Xeon Sandy Bridge E5-2650 CPU on 16 MPI ranks. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at twice the distance of the indices.

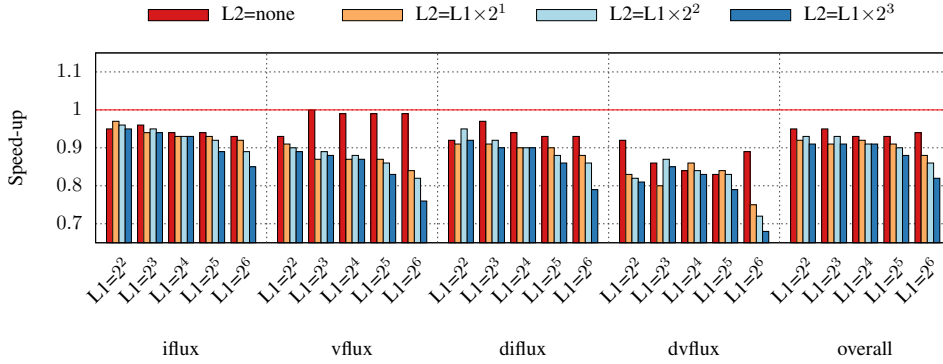


Fig. 4. Results of performing software prefetching on the Intel Xeon Broadwell E5-2680 CPU on 28 MPI ranks. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at twice the distance of the indices.

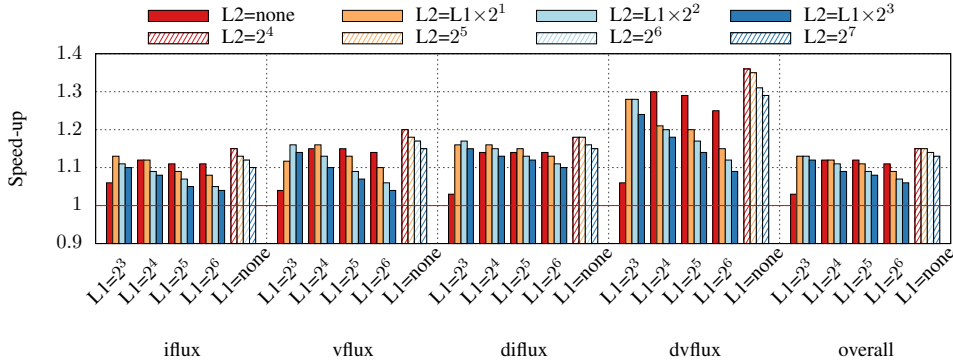


Fig. 5. Results of performing software prefetching on the Intel Xeon Skylake Gold 6140 CPU on 36 MPI ranks. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at twice the distance of the indices.

that the L1 prefetches hold critical hardware resources such as Line Fill Buffers until the cache line fill completes whereas L2 prefetchers do not [12].

VI. CONCLUSIONS

Although software prefetching can be an ideal mechanism for improving the performance of applications that contain indirect and irregular memory access patterns, implementing and gaining any performance from it in real applications can be surprisingly challenging. To address this, we have demonstrated the utility and implementation of software prefetching in an unstructured finite volume CFD code of representative

size and complexity to an industrial application. We have presented the importance of auto-tuning for searching and finding the optimal prefetch distance values across different computational kernels and processors and the potential benefit of only executing prefetches in the L2 cache. We have discussed the impact that the data layout can have on the performance and efficiency of software prefetching and presented ways through which prefetches can be integrated among existing optimisations such as vectorisation in kernels that contain indirect and irregular access patterns. Through this, we showed significant full application speed-ups of 15%

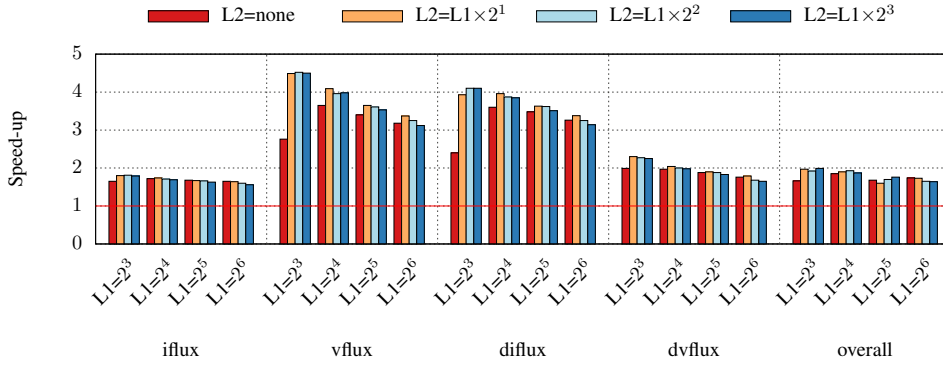


Fig. 6. Results of performing software prefetching on the Intel Xeon Phi Knights Corner 7120P coprocessor. The run was performed on 60 MPI ranks with one thread/rank per physical core (no hyper-threading). The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at twice the distance of the indices.

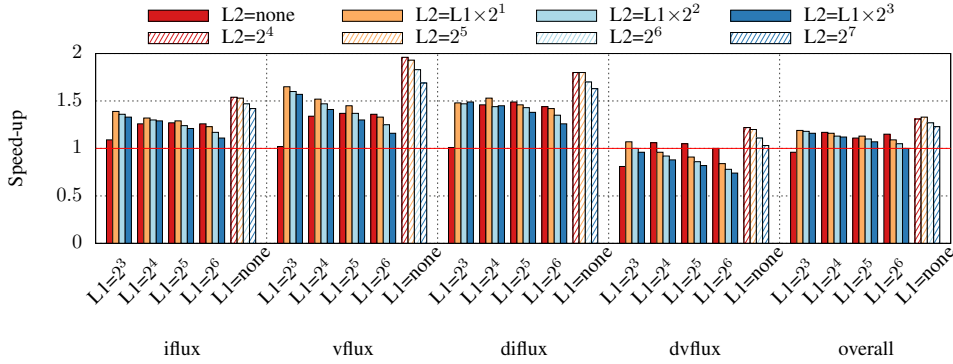


Fig. 7. Results of performing software prefetching on the Intel Xeon Phi Knights Landing 7210 CPU. The run was performed on 64 MPI ranks with one thread/rank per physical core (no hyper-threading) and using the Quadrant and Cache mode configuration. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at twice the distance of the indices.

on a Intel Xeon Skylake system, $1.99\times$ on an Intel Xeon Phi Knights Corner coprocessor and 33% on an Intel Xeon Phi Knights Landing CPU at full concurrency. However, our efforts resulted in no improvements on the Intel Xeon Sandy Bridge and Broadwell systems which we attribute to capacity misses due to the smaller L2 caches. This further emphasises the importance of investigating software prefetching across multiple processors and via automated processes such as auto-tuning which this paper does.

ACKNOWLEDGMENTS

Parts of this work were supported by the Engineering and Physical Sciences Research Council (EPSRC) and Rolls-Royce plc through the industrial CASE award 13220161 and grant EP/K026399/1. This work used the ARCHER KNL Testing and Development Platform part of the UK National Supercomputing Service (<http://www.archer.ac.uk>).

REFERENCES

- [1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.
- [2] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, July 2017, no. 248966-037.
- [3] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *ASPLOS*. New York, NY, USA: ACM, 1991, pp. 40–52.
- [4] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 2:1–2:29, Mar. 2012.
- [5] S. P. Vanderwielen and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, Jun. 2000.
- [6] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *CGO '17*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 305–317.
- [7] D. Mudigere, S. Sridharan, A. Deshpande, J. Park, A. Heinecke, M. Smelyanskiy, B. Kaul, P. Dubey, D. Kaushik, and D. Keyes, "Exploring shared-memory optimizations for an unstructured mesh cfd application on modern parallel systems," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 723–732.
- [8] M. A. A. Farhan and D. Keyes, "Optimizations of unstructured aerodynamics computations for many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2018.
- [9] L. Di Mare, D. Y. Kulkarni, F. Wang, A. Romanov, P. R. Ramar, and Z. I. Zachariadis, "Virtual gas turbines: Geometry and conceptual description," *Proceedings of ASME TurboExpo, Vancouver, Canada*, 2011.
- [10] F. Wang, M. Carnevale, G. Lu, L. di Mare, and D. Kulkarni, "Virtual gas turbine: Pre-processing and numerical simulations," in *ASME Turbo Expo 2016*. American Society of Mechanical Engineers, 2016.
- [11] I. Hadade, F. Wang, M. Carnevale, and L. di Mare, "Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures," *Computer Physics Communications*, 2018.
- [12] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming*. Morgan Kaufmann, 2016.

APPENDIX A

ARTIFACT DESCRIPTION APPENDIX: SOFTWARE PREFETCHING FOR UNSTRUCTURED MESH APPLICATIONS

A. Abstract

We provide source code implementing the software prefetching techniques discussed in the paper in a small application that resembles the main AU3X code and in one of the face-based kernels for which results are presented in the manuscript (iflux). We supply a shell script as an example of the auto-tuning infrastructure used in this work as well as the input files used in the experiments. This should allow the reader to: (1) examine in more detail the implementation of software prefetches in the code; (2) replicate the results across all of the processors used in this work for the iflux kernel; (3) see how auto-tuning can be integrated within the context of an application for finding optimal parameters for the software prefetches.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Auto-tuned software prefetching
- **Program:** au3x-ia3-reproduce
- **Compilation:** Intel icpc ≥ 17.0
- **Transformations:** Software prefetches implemented as inline routines in an unstructured CFD application and based on auto-tuned parameters.
- **Binary:** -
- **Data set:** unstructured mesh of aero-engine intake
- **Run-time environment:** Linux x86-64.
- **Hardware:** We recommend a system that has at least AVX support. The code has been tested on the systems presented in the paper.
- **Output:** Timings of iflux kernel.
- **Experiment workflow:** see instructions below
- **Publicly available?:** Yes

2) How software can be obtained: The source code can be downloaded from the following Bitbucket repository: <https://bitbucket.org/ioanhadade/au3x-ia3-reproduce/>

3) Hardware dependencies: The code runs on all of the hardware platforms used in the paper. With small modifications, it can be used on other architectures as well although this does not fall within the scope of this work.

4) Software dependencies: Intel icpc ≥ 17.0 and Make

5) Datasets: We provide the same data set as the one used in the paper which is an unstructured mesh of an aero-engine intake containing 3.2 million degrees of freedom.

C. Installation

In top-level directory, ‘make’ will present the user with a number of options such as building the code with software prefetching enabled using already known parameters, without software prefetching or both. The user also has to select the architecture he wants to build the binary for which dictates which version of the gather and scatter primitives implemented using compiler intrinsics are used. For example, for building all versions for a Skylake system, the user should type ‘make all TARGET=SKX’. Further information is provided in the README.

D. Experiment workflow

For auto-tuning, execute the shell script autotune.sh providing as argument the architecture on which the auto-tuning is to be performed such as SNB, BDW, SKX, KNC or KNL (i.e., the processor architectures evaluated in this paper). This will produce a log file ‘autotune.dat’ showing the speed-up based on running the application across the ranges of distance values presented in the paper. For verifying the results on one of the systems used in the paper, build all the executables on that given architecture (e.g., on Skylake ‘make all TARGET=SKX’) and run ‘./au3x-ia3-noswpf.exe’ and then ‘./au3x-ia3-swpf.exe’. The first executable does not execute any software prefetches whilst the latter does. A comparison between the two versions in terms of their run-time can then be performed which should match or exceed the ones presented in the paper.

E. Evaluation and expected result

The speed-ups between the version of the code without software prefetching and the one with software prefetching enabled should be similar to the speed-ups shown in the paper for the iflux kernel and on that specific architecture. The same applies to the results obtained from auto-tuning. However, we suspect that the results obtained by running this small mini application will be better than the ones of the main code used in the paper since the latter has a larger memory footprint and therefore exhibits more cache pollution.