Software prefetching for unstructured mesh applications

IOAN HADADE*, Oxford Thermofluids Institute, University of Oxford, United Kingdom
TIMOTHY M. JONES*, Computer Laboratory, University of Cambridge, United Kingdom
FENG WANG, Oxford Thermofluids Institute, University of Oxford, United Kingdom
LUCA DI MARE, Oxford Thermofluids Institute, University of Oxford, United Kingdom

This paper demonstrates the utility and implementation of software prefetching in an unstructured finite volume computational fluid dynamics code of representative size and complexity to an industrial application and across a number of modern 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 addressing patterns and show how to best implement them in an existing large-scale computational fluid dynamics application. Through this we show significant full application speed-ups on a range of processors and realistic test cases in both single core/tile and full socket configurations, such as 1.14× on the Intel Xeon Sandy Bridge, 1.09× on the Intel Xeon Broadwell, 1.29× on the Intel Xeon Skylake, 1.99× on the in-order Intel Xeon Phi Knights Corner coprocessor and 1.51× on the out-of-order Intel Xeon Phi Knights Landing many-core processor.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → **Software performance**; • **Theory of computation** → *Shared memory algorithms*;

Additional Key Words and Phrases: software prefetching, unstructured mesh, irregular memory access, auto-tuning, performance optimisation

ACM Reference Format:
Ioan Hadade, Timothy M. Jones, Feng Wang, and Luca di Mare. 0. Software prefetching for unstructured mesh applications. *ACM Trans. Parallel Comput.* 0, 0, Article 0 (0), 23 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

The growing disparity between the speed of the processor and that of the memory system [Wulf and McKee 1995] 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 or capacity cache misses. The traditional mechanism for dealing with these is prefetching, where the hardware anticipates,

*Corresponding authors.

Authors' addresses: Ioan Hadade, ioan.hadade@eng.ox.ac.uk, Oxford Thermofluids Institute, University of Oxford, Oxford, United Kingdom; Timothy M. Jones, timothy.jones@cl.cam.ac.uk, Computer Laboratory, University of Cambridge, Cambridge, United Kingdom; Feng Wang, feng.wang@eng.ox.ac.uk, Oxford Thermofluids Institute, University of Oxford, Oxford, United Kingdom; Luca di Mare, luca.dimare@eng.ox.ac.uk, Oxford Thermofluids Institute, University of Oxford, Oxford, United Kingdom.

1539-9087/0/0-ART0 $15.00
https://doi.org/0000001.0000001

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 [Intel Corporation 2017]. However, this is not the case for irregular applications (e.g., unstructured mesh solvers) where the underlying irregular and indirect access patterns make prefetching more difficult to identify in hardware [Callahan et al. 1991; Lee et al. 2012]. A viable alternative for such applications is for prefetching to be implemented in software, 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 [Vanderwiel and Lilja 2000]. 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 [Ainsworth and Jones 2017]. 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. The challenge of selecting an optimal distance value for the prefetches is exacerbated by the fact that this might differ across computational kernels and processors [Mowry et al. 1992]. 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.

We aim to address this by demonstrating the implementation and utility of software prefetching in an unstructured finite volume computational fluid dynamics (CFD) code of representative size and complexity to an industrial application and on realistic test cases. The CFD application used in this study was previously optimised for efficient execution on modern multicore and manycore processors [Hadade et al. 2018b] with a limited study on software prefetching presented in Hadade et al. [2018a].

This paper is an extension to the previous work and makes the following contributions:

- We demonstrate the use of auto-tuning for finding the best distance values for the software prefetches as well as their optimal target with respect to the cache hierarchy. We show how the search space in the auto-tuning phase can be reduced significantly by prefetching the indices used in the indirect accesses at twice the distance of the referenced data and provide a skeleton implementation for the auto-tuner.
- We empirically demonstrate the importance of prefetching both the irregular and regular access patterns, rather than just the former, in loops that exhibit both. We observe that prefetching only the irregular accesses in software results in marginal improvements and attribute this to the potential interference of the software prefetches with the underlying hardware prefetchers [Lee et al. 2012]. As a result, software prefetching should either target all access patterns in a loop or none at all.

- We demonstrate, on processors with inclusive cache hierarchies, that best performance is achieved by targeting either the L2/L3 caches or the L1 cache only, since overlapping prefetches across all levels leads to a higher number of evictions in L1 therefore degrading performance. Alternatively, our experiments show that if prefetches in L1 at a given distance give positive results, targeting the prefetches to the L2/L3 levels at twice that distance should perform even better.
- We show how on in-order architectures, such as the Knights Corner coprocessor, best performance is obtained by both hyperthreading and software prefetches, although the latter on its own can result in considerable speed-ups (1.81× and 4.10×) in loops with indirect access patterns.
- We discuss the impact of the underlying data layout on the number of prefetch instructions required in kernels with indirect access patterns. We demonstrate how the Array of Structures (AoS) layout is best suited for storing indirectly accessed data provided that this is aligned, padded and transposed accordingly so as to allow vectorisation.
- We provide examples on how we integrate the prefetch instructions in our application and show how these can interact with other existing optimisations, such as vectorisation.
- We observe that the optimal distance values for the prefetches remain constant across computational kernels, even though they exhibit different flop per byte ratios and number of indirect accesses, and across different inputs (i.e., computational meshes). As a result, auto-tuning could be performed on smaller micro-benchmarks rather than the full application, although this has not been pursued in this work and is left as a mere hypothesis.
- We provide an example implementation and source code [Hadade 2018] that can be used to reproduce the results presented in this paper.
- We report significant significant full application speed-ups on a range of processors and realistic test cases in both single core/tile and full socket configurations, such as 1.14× on the Intel Xeon Sandy Bridge, 1.09× on the Intel Xeon Broadwell, 1.29× on the Intel Xeon Skylake, 1.99× on the in-order Intel Xeon Phi Knights Corner coprocessor and 1.51× on the out-of-order Intel Xeon Phi Knights Landing many-core processor.

2 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. [2012] 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 [2017]. 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 [Bailey et al. 1991] 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× 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. Proprietary compilers 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. Furthermore, they include a more mature implementation of the OpenMP run-time and are generally more likely to be available to users on high-performance computing systems. For example, at the time of writing, LLVM 8 did not support the OpenMP 4.0 standard on which our application relies heavily.

With regard to real applications (as opposed to benchmarks), Mudigere et al. [2015] present the implementation of software prefetching, among a number of other shared-memory optimisations, in FUN3D [FUN 2017], 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 Farhan and Keyes [2018] 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.

3 UNSTRUCTURED FINITE-VOLUME CFD SOLVER

3.1 Overview

The test vehicle for this study is AU3X [Di Mare et al. 2011; Wang et al. 2016], 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

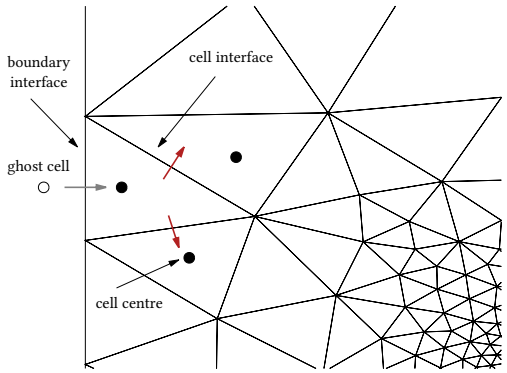

Fig. 1. Schematic of the cell-centred finite volume scheme on unstructured grids. Black-filled circles represent cell-centres, white-filled circles ghost cells, red arrows cell-to-cell interfaces and gray arrows boundary-to-cell interfaces. State variables are stored at the ghost and cell-centres while fluxes are computed for every cell-to-cell and boundary-to-cell interfaces where the cell end-points are referenced indirectly.

centre using the weighted least-square method [Mavriplis 2003]. 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 [1981] where second-order accuracy is obtained using the Monotonic Upwind Scheme for Conservation Laws (MUSCL) [van Leer 1979] and the van Albada limiter [Hirsch 1990]. 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-ϵ, k-ω and k-ω 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 [Wilcox 1988] 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 [Ope 2016] directives and compiler intrinsics with support for AVX/AVX2/AVX-512 and IMCI ISA SIMD extensions. Readers are referred to Hadade et al. [2018b] for further details.

3.2 Computational kernels and access patterns

Similar to all well-tuned unstructured CFD codes [Gropp et al. 2001], our solver spends the majority of its time per non-linear iteration (75%) in loops that iterate over the cell-to-cell interfaces (henceforth, faces) for computing numerical fluxes (see fig. 1), gradients and limiters. These face-based loops are structured as a sequence of gather, compute and scatter operations where values are gathered from pairs of cells sharing a face followed by the calculation and scatter back of results to the respective face end-points. An example of a scalar 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 the

```
for( ic=ics;ic<ice;ic++ )
{
    // load indices
    i1= indx[0][ic];
    i2= indx[1][ic];
    // gather the unknowns
    u1= q[i1];
    u2= q[i2];
    // load face properties
    wc= geo[ic];
    // compute flux residual
    f= wc*(u2-u1);
    // accumulate and scatter-back
    res[i1]-= f;
    res[i2]+= f;
}
```

Listing 1. The anatomy of a face-based loop (scalar) on unstructured grids. The array q represents the flow variables stored at the cell-centre while indx is the connectivity array between a face and the two cells that share it. The array wc stores the normals to the face at iteration ic and res represents the residuals that are scattered back to the face end-points.

indices from the indx connectivity array. These are then used together with the geometric properties of the face (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 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 in software.

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).

3.3 Test cases

The test cases used in this study represent real-world aerospace applications such as an aero-engine intake operating near ground and a single passage of the NASA Rotor 37.

The computational domain of the intake contains 3.2 million degrees of freedom and is based on an unstructured mesh (fig.2) where near wall regions are discretised with hexahedra elements for boundary layer prediction whilst the free stream domain is discretised using prismatic elements. A

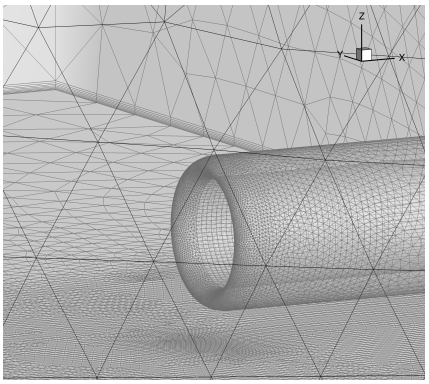

Fig. 2. Unstructured mesh of aero-engine intake.

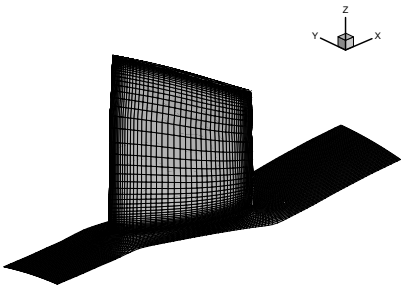

Fig. 3. Block-structured mesh of rotor passage.

numerical study using the AU3X code was previously presented by Carnevale et al. [2014, 2017] based on experimental data provided by Murphy and MacManus [2011].

The computational domain of the NASA Rotor 37 single passage contains 2.4 million degrees of freedom and is based on a block-structured mesh (fig. 3) with hexahedra elements used in the entire domain. The test case was designed and experimentally studied at the NASA Glenn Research Center by Reid and Moore [1978] and is one of most popular rotor test cases for validating CFD codes that are used in the aerodynamic design and analysis of turbomachines.

The main difference between the two test cases is that the first is discretised on a fully unstructured mesh whereas the latter on a block-structured mesh. Consequently, the indirect memory accesses present in face-based loops will exhibit larger and more irregular strides on the unstructured mesh whereas the block-structured grid will allow for a more regular and streaming access pattern despite the use of indirection. We therefore expect software prefetching to have a higher impact on the unstructured test case.

4 HARDWARE SETUP

Table 1 shows the systems that were used as experimental platforms in this study and provides details regarding their architectural characteristics and configuration. These are the Intel Xeon Sandy Bridge (SNB), Broadwell (BDW) and Skylake Server (SKX) multicore CPUs, the Intel Xeon Phi Knights Corner (KNC) coprocessor and the self-hosted Intel Xeon Phi Knights Landing (KNL) processor. With regards to KNL, this was configured in Quadrant clustering mode and Cache memory mode throughout this work.

5 IMPLEMENTATION

5.1 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 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

	SNB	BDW	SKX	KNC	KNL
Version	E5-2650	E5-2695	Gold 6152	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
Cores	8	18	22	61	64
Threads	2	2	2	4	4
Clock (GHz)	2.0	2.1	2.1	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	30.25	-	-
Memory (GB)	32	128	192	16	96/16
Memory type	DDR3	DDR4	DDR4	GDDR5	DDR4/MCDRAM
Stream (GB/sec)	35	64	98	181	82/452
L1 HW Prefetcher	Yes	Yes	Yes	No	Yes
L2 HW Prefetcher	Yes	Yes	Yes	Yes	Yes
Compiler	icpc 17.4	icpc 17.4	icpc 18.3	icpc 17.4	icpc 17.0
MPI Library	Intel MPI 2017	Intel MPI 2017	Intel MPI 2018	Intel MPI 2017	Intel MPI 2017

Table 1. Details of the systems used as experimental platforms.

underlying vector register width as defined by the `VECLEN` macro. 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, face variables are stored in a SoA layout. Thus, loading geometrical properties such as normals (line 24) 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 27). 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 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` [Löhner 2010].

5.2 Inserting software prefetches

The `vgather` and `vload` routines present 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). As a result, these routines are also the best place for inserting software prefetches.

```
#if defined __MIC__
  #define VECLEN 8
#elif defined __AVX512F__
  #define VECLEN 8
#elif defined __AVX__
  #define VECLEN 4
#else
  #define VECLEN 1
#endif

#define MAXNPDE 8

double ql[MAXNPDE][VECLEN],qr[MAXNPDE][VECLEN];
double rl[MAXNPDE][VECLEN],rr[MAXNPDE][VECLEN];
double fl[MAXNPDE][VECLEN],fr[MAXNPDE][VECLEN],
double f[MAXNPDE][VECLEN], wc[4][VECLEN];

for( ic=ics;ic<ice;ic+=VECLEN )
{
  // gather from face end-points
  vgather(q,n,&indx[0][ic],&indx[1][ic],ql,qr);
  vgather(r,n,&indx[0][ic],&indx[1][ic],rl,rr);
  // load face normals and area
  vload(geo,nx+1,ic,wc);
  // compute flux residual across all vector lanes
  #pragma omp simd simdlen(VECLEN) safelen(VECLEN)
  for( iv=0;iv<VECLEN;iv++ )
  {
    // assemble fluxes -- truncated
    f[0][iv]= 0.5*(fr[0][iv]+fl[0][iv])*wc[3][iv];
    // accumulate
    rl[0][iv]-= f[0][iv];
    rr[0][iv]+= f[0][iv];
  }
  // scatter-back to face end-points
  vscatter(r,n,&indx[0][ic],&indx[1][ic],rl,rr);
}
```

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

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 [2017] demonstrated that the optimal distance for prefetching the indices is twice that of the actual data. This is due to the fact that the amount of work performed on each iteration of the loop is fairly constant and so prefetching the indices and data will take approximately the same amount of time, assuming the loads to the indices hit in the L1 cache.

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

```
// prefetch data at cell data at half the distance of the indices
# if defined L1_INDEX_PF
  # define L1_CELL_DATA_PF (L1_INDEX_PF >> 1)
# endif
# if defined L2_INDEX_PF
  # define L2_CELL_DATA_PF (L2_INDEX_PF >> 1)
# endif

// prefetch indices in connectivity array
 void prefetchi(int *pos1, int *pos2)
{
  // prefetch in L2
  # if defined L2_INDEX_PF
  _mm_prefetch((char *)&(pos1[L2_INDEX_PF]),_MM_HINT_T1);
  _mm_prefetch((char *)&(pos2[L2_INDEX_PF]),_MM_HINT_T1);
  # endif
  // prefetch in L1
  # if defined L1_INDEX_PF
  _mm_prefetch((char *)&(pos1[L1_INDEX_PF]),_MM_HINT_T0);
  _mm_prefetch((char *)&(pos2[L1_INDEX_PF]),_MM_HINT_T0);
  # endif
}

// prefetch cell-centred data (irregular access)
template < typename type >
void prefetchcd(type *data, int *pos1, int *pos2)
{
  # if defined L2_CELL_DATA_PF
  for( int iv=0;iv<VECLEN;iv++ )
  {
    _mm_prefetch((char *)&(data[pos1[iv+L2_CELL_DATA_PF]]),_MM_HINT_T1);
    _mm_prefetch((char *)&(data[pos2[iv+L2_CELL_DATA_PF]]),_MM_HINT_T1);
  }
  # endif
  # if defined L1_CELL_DATA_PF
  for( int iv=0;iv<VECLEN;iv++ )
  {
    _mm_prefetch((char *)&(data[pos1[iv+L1_CELL_DATA_PF]]),_MM_HINT_T0);
    _mm_prefetch((char *)&(data[pos2[iv+L1_CELL_DATA_PF]]),_MM_HINT_T0);
  }
  # endif
}

// prefetch face data (regular access)
template < typename type >
void prefetchfd(type *data, int offset)
{
  # if defined L2_FACE_DATA_PF
   _mm_prefetch((char *)&(data[offset+L2_FACE_DATA_PF],_MM_HINT_T1);
  # endif
  # if defined L1_FACE_DATA_PF
   _mm_prefetch((char*)&(data[offset+L1_FACE_DATA_PF],_MM_HINT_T0);
  # endif
}
```

Listing 3. Routines for prefetching different kinds of data.

```
template < typename type >
void vgather(type *s, int n, int *p1, int *p2,
          double d1[][VECLEN], double d2[][VECLEN])
{
  // prefetch cell data using
  // indices in p1 and p2 (irregular)
  prefetchcd(s,p1,p2);
  ...
}

void vload(double **s, int nv, int offset,
          double d[][VECLEN])
{
  // prefetch face data (regular)
  for( int iv=0;i<nv;iv++ )
    prefetchfd(&s[iv][0],offset);
    ...
}

// face-based loop with software prefetching
for( ic=ics;ic<ice;ic+=VECLEN )
{
  // prefetch the indices, only once
  prefetchi(&pos1[ic],&pos2[ic]);
  // gather (implicit prefetch of cell data)
  vgather(q,n,&pos1[ic],&pos2[ic],ql,qr);
  // load (implicit prefetch of face data)
  vload(geo,nx,ic,wc);
  // computation
  ...
}
```

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

the hardware prefetcher as well as the claim of both Lee et al. [2012] and Ainsworth and Jones [2017] that prefetching the irregular accesses in software can negatively impact the operation of the stream/stride hardware prefetchers.

Listing 3 shows the implementation of routines used for prefetching the indices, cell-centred and face data respectively. Listing 4 demonstrates how all these are implemented within the `vgather` and `vload` primitives along with the structure of a face-based loop with software prefetching which is followed throughout the application.

5.3 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 and face variables. A side effect of using the AoS data layout for the cell-centred values 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 as long as their number is not greater than eight in double precision or sixteen in single precision. For example, let us consider the flow and turbulence variables stored in `q`. In the AoS layout (fig. 4), these are stored for each cell-centre as $[u, v, w, t, p, k, \omega]$ where u, v and w are the velocity vector components, t is temperature, p is pressure and k and ω are the additional variables

Fig. 4. Flow variables stored in an Array of Structures (AoS) layout with padding.

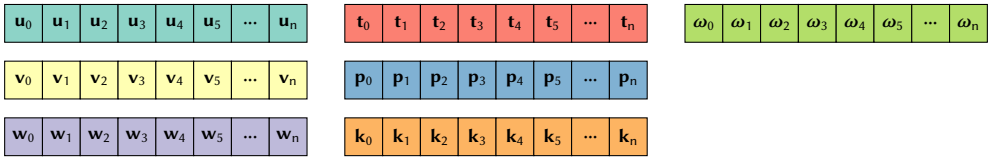

Fig. 5. Flow variables stored in a Structure of Arrays (SoA) layout.

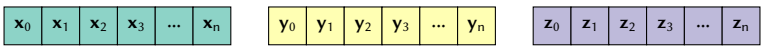

Fig. 6. Face normals stored in a Structure of Arrays (SoA) layout.

used to compute the turbulent viscosity. In the SoA format (fig. 5), each of these variables 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. The latter is true if the per-cell data structure is aligned at the correct byte boundary for the underlying architecture and padded to occupy the entire cache line if necessary. 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.

The difference in the number of prefetch instructions required per face-based loop iteration is important since these instructions, although non-blocking, do take up valuable instruction slots and can therefore result in noticeable overheads with detrimental effects on performance.

With regard to face data structures such as normals, these are accessed using regular unit stride loads since faces are visited in consecutive order. As a result, these variables are best stored in the SoA layout (fig. 6) as this allows for efficient vector load/store operations even though it requires more in-flight memory streams. However, when prefetching these values, it is important to remember that prefetches bring data into the cache hierarchy at cache line granularity. As a result, the prefetches should execute such that they do not bring the data into the caches more than once at any given time. For example, on AVX/AVX2 architectures such as Sandy Bridge and Broadwell, the face-loops are unrolled by four since this is the number of double precision floating-point values that can fit on the underlying 256-bit vector registers. However, a cache-line on these architectures can hold eight double precision variables. As a result, executing a software prefetch for the face data in every vector iteration on these architectures is not optimal since the same can be accomplished with a single prefetch instruction every other iteration. As earlier mentioned, minimizing the prefetch degree as much as possible is imperative since these instructions do go through the entire pipeline.

Instruction	L1	L2	L3	Macro
PREFETCHNTA	No	No	Yes	_MM_HINT_NTA
PREFETCH0	Yes	Yes	Yes	_MM_HINT_T0
PREFETCH1	No	Yes	Yes	_MM_HINT_T1
PREFETCH2	No	Yes	Yes	_MM_HINT_T2

Table 2. Characteristics of prefetch instructions in the x86 ISA. The macro value is the second argument that is passed to the _mm_prefetch compiler intrinsic.

5.4 Auto-tuning

Finding the optimal value for the prefetch distance 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 and is an automated process integrated within the build system of the application. The auto-tuner is implemented as a shell script that compiles the application with different distance values and destinations for the prefetches (i.e., L1, L2, or L1 and L2) as supported by the x86 ISA (table 2) and saves the combination of values that exhibit best application performance so that it can be used subsequently by the build system for the production binary. All data obtained from the parametric study for that particular architecture is also stored for later analysis. A simplified example of the auto-tuning script used in this work is illustrated in listing 5 for reference.

The search space for the auto-tuner is considerably reduced by the fact that it only has to iterate through multiple ranges of index values across the different cache levels. This is because the values for prefetching the indirectly referenced data are automatically set to half those of the index values as previously discussed in section 5.2.

The auto-tuner performs the first run executing software prefetches into the L1 cache (prefetch0) followed by a run where prefetches target the L2 and L3 levels only (prefetch1). After that, three more runs are performed where prefetches are executed targeting the L1 cache as well as prefetches that target the L2 and L3 caches at a distance that is twice that of the one used for prefetching into the L1 cache and which increases in powers of two increments. After these runs are executed, the auto-tuner will then increase the distance value for prefetching into the L1 cache also in powers of two increments and start the same process again.

Since the loops that we prefetch on are vectorised, the initial prefetch distance value is set to be equal to the vector register size for that given processor (i.e., one vector iteration) increasing afterwards in powers of two increments. Therefore, as long as cell-centred and face data structures are aligned to the correct byte boundaries, every prefetch instruction will only bring data to the higher cache hierarchies belonging to a given vector iteration.

6 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 and also for the top four face-based kernels (table 3) 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. We also present the difference in speed-ups when prefetches are only executed for the indirect accesses (i.e., cell-centred data) compared to when prefetches are also executed for the streaming unit-stride accesses used to load the face data structures. Timings have been obtained using the *clock_gettime* function available on Linux. Profiling data such as comparisons between cache miss rates and raw accesses with

```
# find vector register size
k=$(findregsize)
# start software prefetching
export ENABLE_PF=1
# Vary distance for L1 prefethces
for i in $k $((k<<1)) $((k<<2))
do
    # L1
    export L1_INDEX_PF=$i
    unset L2_INDEX_PF
    build
    var[$i,0]=$(run)

    # L2 (and L3 if inclusive) only
    unset L1_INDEX_PF
    export L2_INDEX_PF=$i
    build
    var[0,$i]=$(run)

    # prefetch data into L1 at distance "i"
    # and into L2 at distance "j"
    export L1_INDEX_PF=$i
    for j in $((i<<1)) $((i<<2)) $((i<<3))
    do
        export L2_INDEX_PF=$j
        build
        var[$i,$j]=$(run)
    done
done
```

Listing 5. A simplified example of a shell script for auto-tuning.

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

Table 3. Characteristics of the top four face-based kernels in the application based on their percentage of overall run-time.

software prefetching enabled or disabled were obtained using Intel Vtune Amplifier [Amplifier 2019] and Perf [Perf 2019].

Sandy Bridge. On the Sandy Bridge CPU (figs 7 and 12), software prefetching results in up to 1.43× speed-up in some kernels although shows no benefit in others. This is due to the cache pollution that results from overlapping prefetches at different distances across multiple cache levels and is evidenced by the fact that best results are obtained when we only target the prefetches for either the L1 or the L2 cache rather than both. Perhaps surprisingly, we obtain better results when we execute the application on the whole socket using 8 MPI ranks rather than on a single core. We attribute this to the fact that on Sandy Bridge, the per core memory bandwidth is significantly

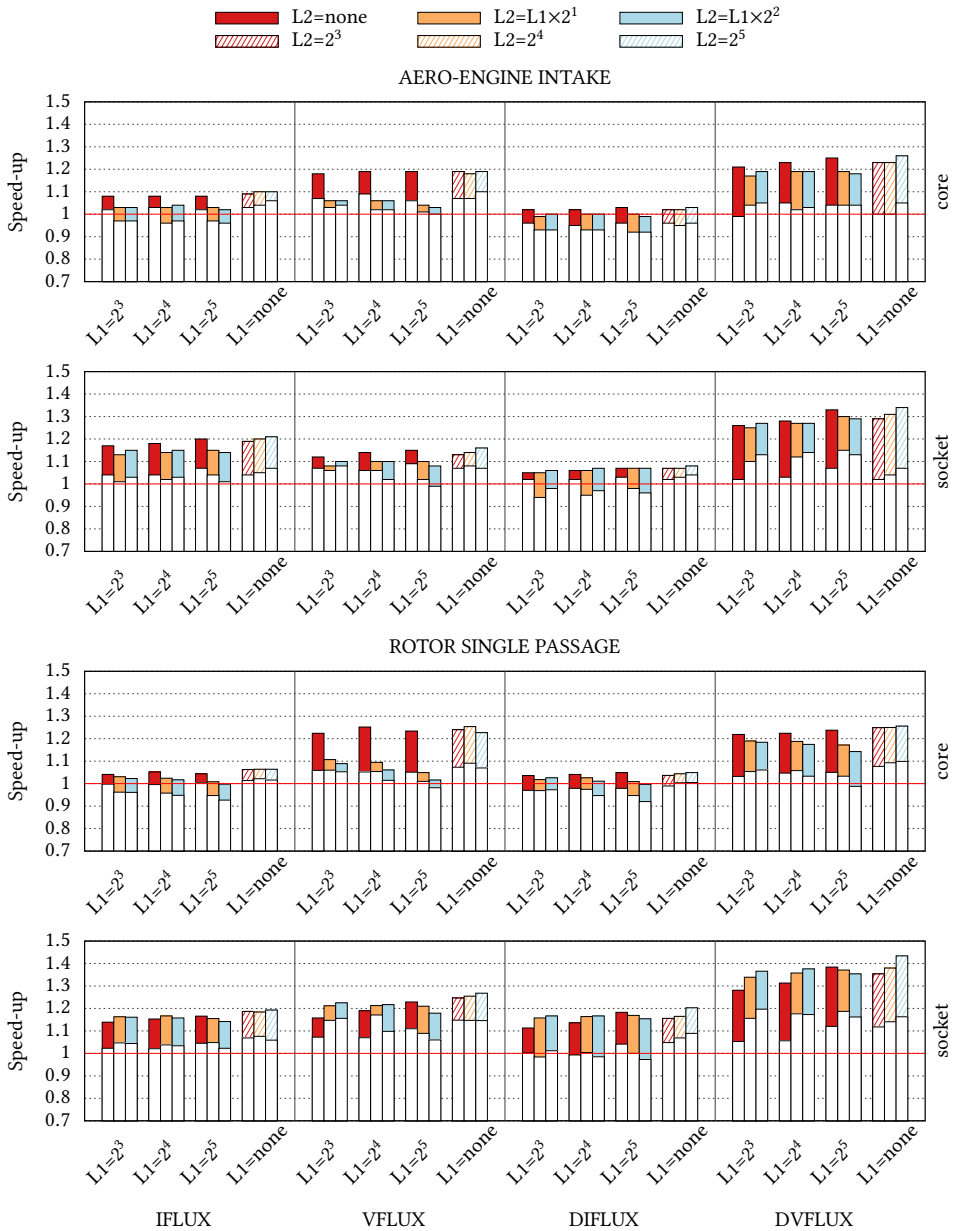

Fig. 7. Results of performing software prefetching in both test cases on the Intel Xeon Sandy Bridge E5-2650 CPU in face-based loops and executed on single core and full socket (8 cores) configurations. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at half the distance of the indices. The white bars represent the speed-ups obtained when software prefetches are executed for irregular accesses only and not for regular stream/stride accesses.

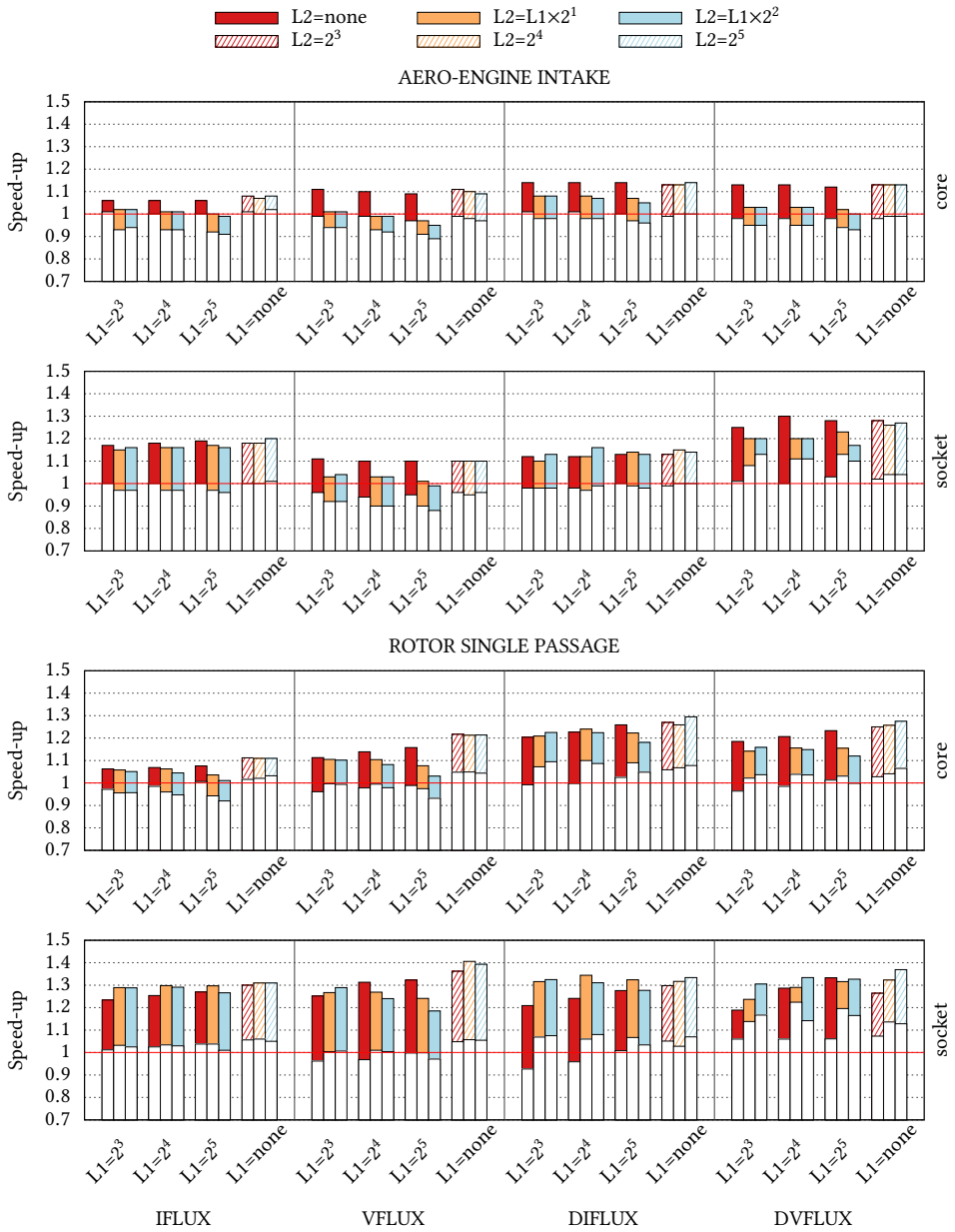

Fig. 8. Results of performing software prefetching in both test cases on the Intel Xeon Broadwell E5-2695 CPU in face-based loops and executed on single core and full socket (18 cores) configurations. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at half the distance of the indices. The white bars represent the speed-ups obtained when software prefetches are executed for irregular accesses only and not for regular stream/stride accesses.

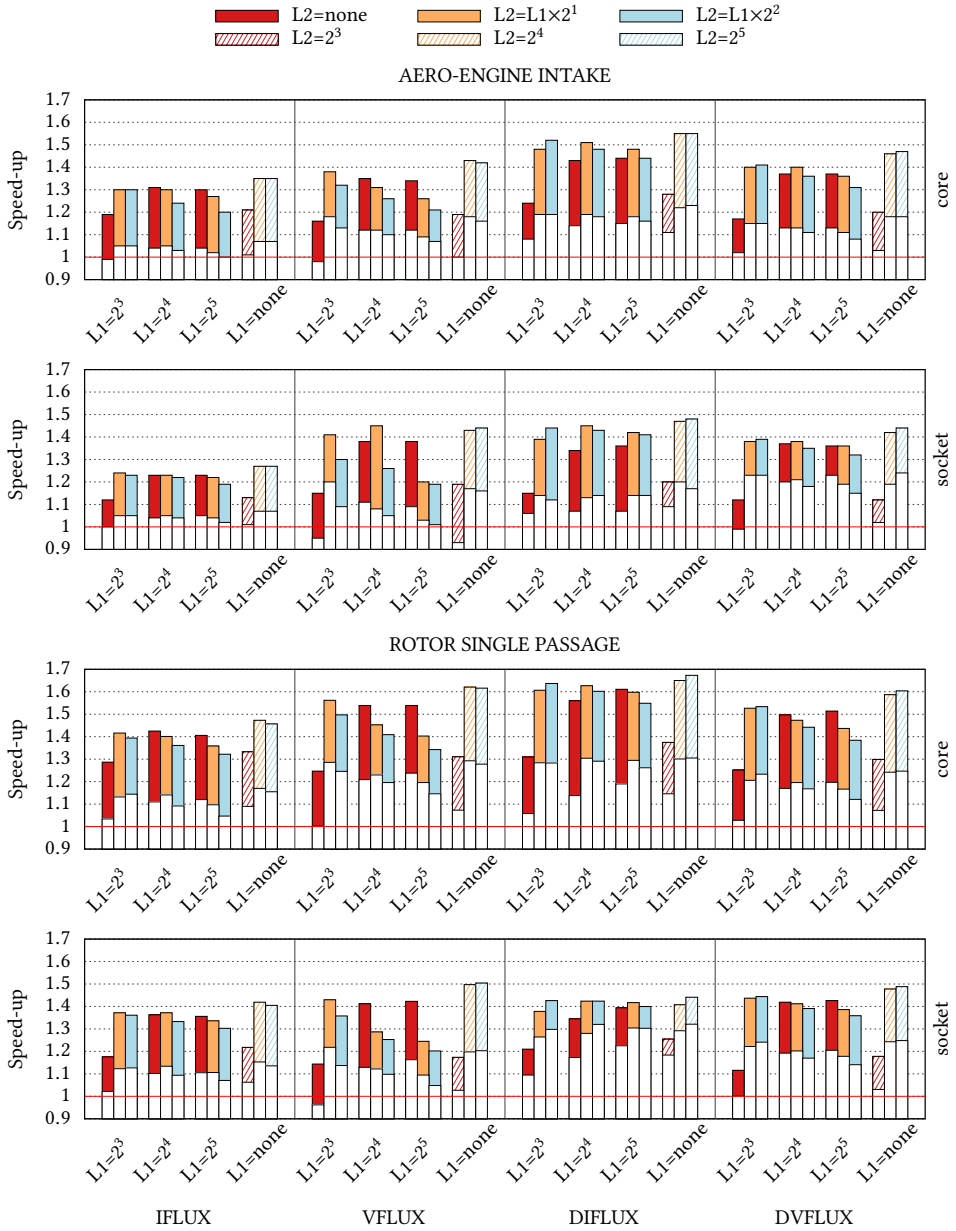

Fig. 9. Results of performing software prefetching in both test cases on the Intel Xeon Skylake Gold 6152 CPU in face-based loops and executed on single core and full socket (22 cores) configurations. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at half the distance of the indices. The white bars represent the speed-ups obtained when software prefetches are executed for irregular accesses only and not for regular stream/stride accesses.

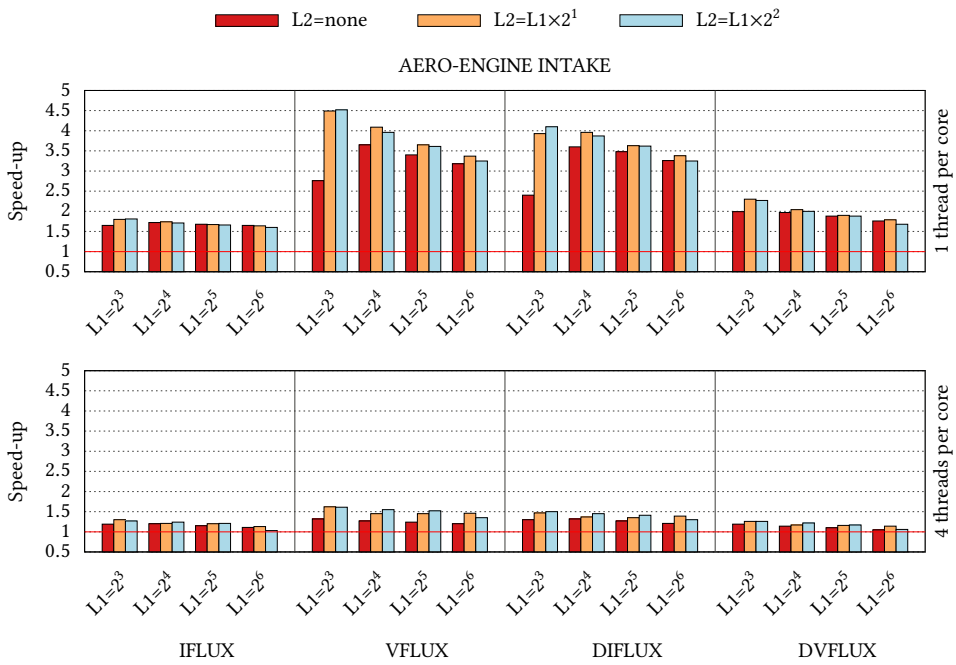

Fig. 10. Results of performing software prefetching in the aero-engine intake test case only on the Intel Xeon Phi Knights Corner 7120P coprocessor. The run was performed on 60 MPI ranks with one thread/rank per physical core (no hyper-threading) and four threads/rank per physical core (with hyper-threading) . The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at half the distance of the indices. Relative to the baseline with no hyperthreading, the version with 4 threads per core is 2.7× faster for iflux, 2.05× for vflux, 2.7× for diflx and 2.0× for dvflux.

smaller since it cannot saturate the entire bandwidth available per socket. With respect to the whole application, software prefetching results in speed-ups of up to 1.09×. The difference in performance improvements between the two test cases is negligible although best results are obtained for the rotor single passage.

On Sandy Bridge, only executing prefetches for the indirectly addressed data and not for the regular streams as well can be the difference between obtaining any benefits from software prefetching or actually slowing down the application. This might be attributed to the fact that our software prefetches interfere with the operation of the hardware prefetchers who would have otherwise operated on the regular streaming accesses [Lee et al. 2012].

Broadwell. The results on the Broadwell system are similar to those on Sandy Bridge and are presented in figs 8 and 12. Both Sandy Bridge and Broadwell CPUs have smaller L2 caches compared to the other processors used in this work. As a result, the risk of capacity misses is higher in these architectures due to the large volume of data that is accessed per vector iteration in the face-based kernels. Furthermore, if we only prefetch the indirect loads on Broadwell, software prefetching results in almost no speed-up and can in fact be detrimental to performance. The best improvements are obtained on the single passage rotor test case that is based on a block-structured mesh and therefore exhibits more regular strides than the unstructured intake mesh. Among the face-based kernels, the highest speed-up is obtained in the routine computing the viscous fluxes (1.39×) when

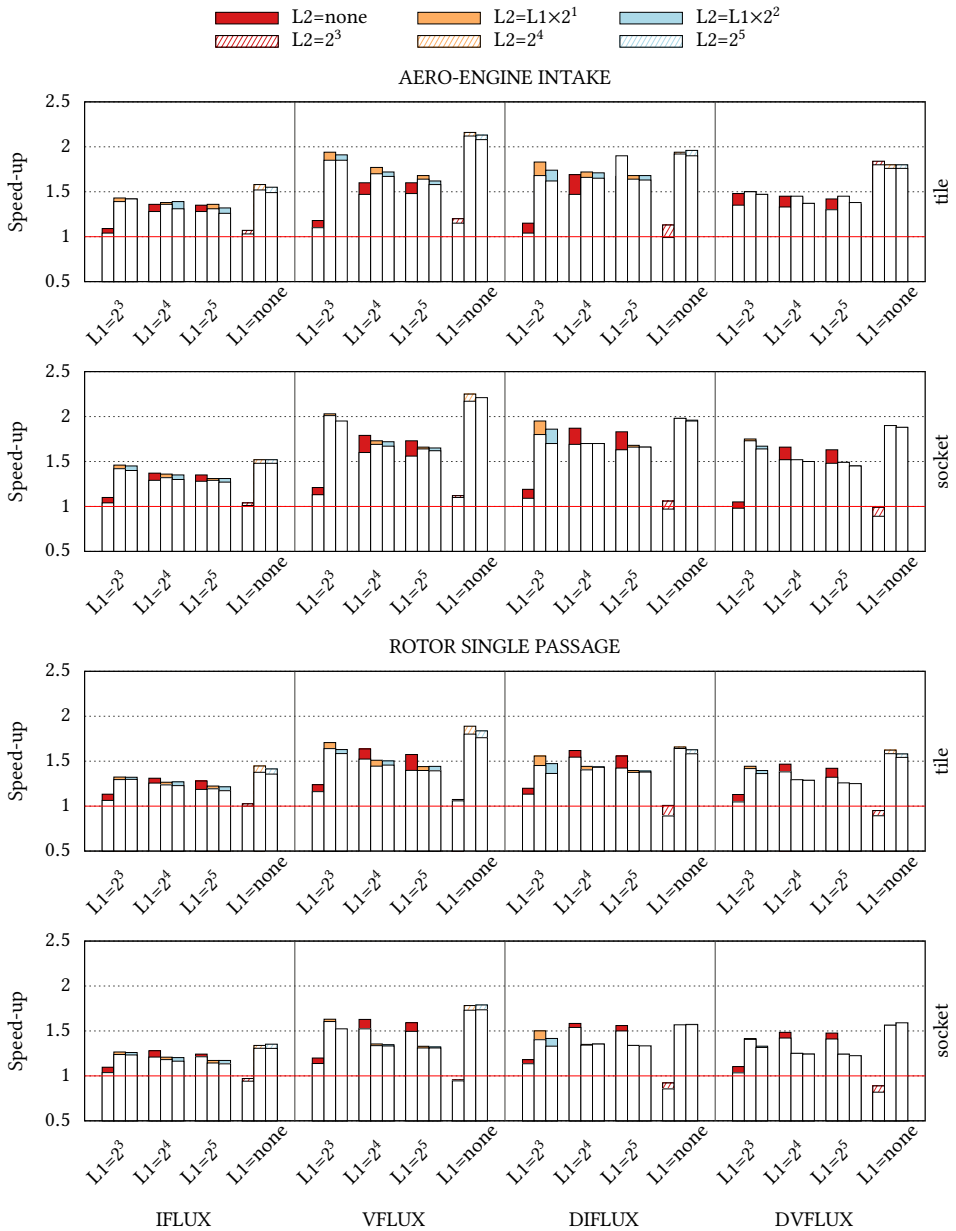

Fig. 11. Results of performing software prefetching in both test cases on the Intel Xeon Phi Knights Landing 7210 CPU. The run was performed on 64 MPI ranks with one thread/rank per physical core (no hyper-threading) and on 2 MPI ranks pinned to one tile sharing an L2 cache. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at half the distance of the indices. The white bars represent the speed-ups obtained when software prefetches are executed for irregular accesses only and not for regular stream/stride accesses.

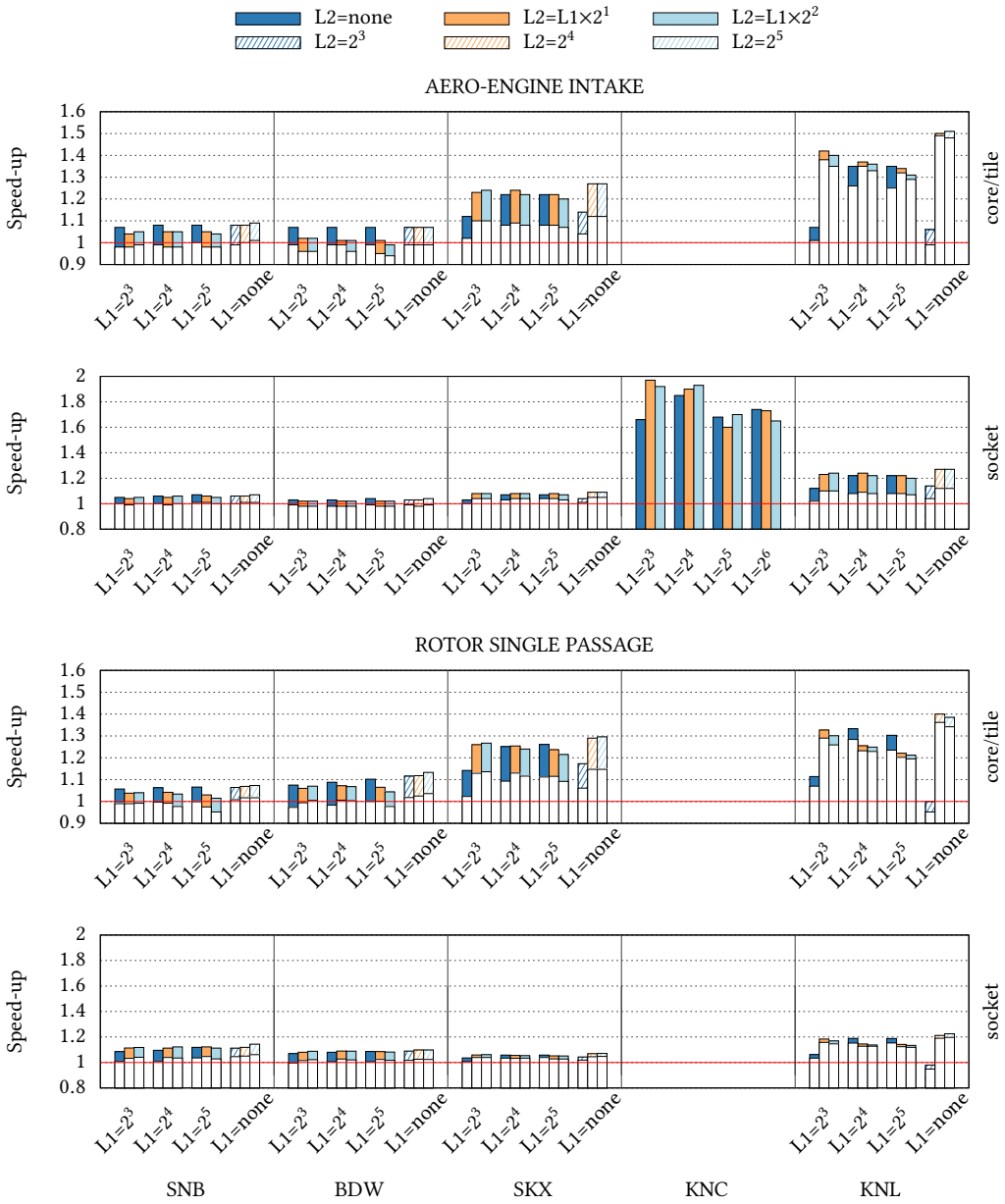

Fig. 12. Results of performing software prefetching in both test cases on the whole application and across all processors. The distance values presented for the L1 and L2 cache hierarchies are for prefetching the indices. Prefetches for the data are executed at half the distance of the indices. The white bars represent the speed-ups obtained when software prefetches are executed for irregular accesses only and not for regular stream/stride accesses.

executing prefetches targeting the L2/L3 caches only and at distances value of 16 for the indices and 8 for the data which translates to prefetching two vector iterations in advance. For the entire application, software prefetching results in a maximum speed-up of 1.11×.

Similar to Sandy Bridge, good performance can also be obtained on Broadwell if prefetches are only executed for the L1 cache although at a shorter distance of 1 vector iteration compared to 2 vector iterations for prefetches in the L2 only. However, this approach is not as efficient as prefetching in L2/L3 only due to the inclusive nature of the cache hierarchy on both Broadwell and Sandy Bridge architectures. In essence, any prefetches into the L1 cache will also bring data into the L2 and L3 levels however prefetches into L2/L3 only will not bring data into the L1 as well. We refer readers to table 2 for further information on prefetch instructions and their respective destination.

Skylake. The impact of software prefetching on Skylake is presented in figs 9 and 12. Compared to the Sandy Bridge and Broadwell systems, software prefetching results in up to 1.29× full application speed-up on the Skylake system in a single core and 1.09× at full socket concurrency. The speed-ups obtained in the face-based loops range between 1.18× and 1.61× across both single core and full socket runs and test cases. We attribute these 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 (non-inclusive). This is further evidenced by the fact that best results are obtained when prefetches are only executed for the larger L2 cache (prefetch1). The significant difference in performance for the whole application on Skylake between single core and full socket runs is attributed to the large per socket core count as well as the fact that the L3 victim cache is shared among all 22 cores which reduces the per core allocation compared to single core runs. Furthermore, in our analysis, we also found memory bandwidth to be a particular issue on Skylake as this quickly gets saturated once we run on more than 14 cores. As a result, a large proportion of benefits derived from software prefetching disappears once we run on the full socket (22 cores) rather than on a single core.

In contrast to Sandy Bridge and Broadwell, prefetching only the indirect accesses in face-based loops does lead to significant improvements in performance although inserting prefetches for the regular accesses as well is by far the best approach.

Knights Corner. Software prefetching exhibits substantial speed-ups on the Knights Corner co-processor as presented in figs 10 and 12. 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 L1 hardware prefetcher. As a result, our implementation leads to 1.99× speed-up for the full application (fig. 12) and between 1.81× and 4.10× in the four face-based kernels (fig. 10). When also running with four hyperthreads per core, the speed-ups are not as large since both software prefetching and multithreading achieve similar goals on the KNC architecture, that of circumventing pipeline stalls as a result of a cache miss. Nevertheless, absolute best performance is obtained when both hyperthreading and software prefetching are implemented. Relative to the baseline with software prefetching and no hyper-threading, running with 4 threads per core and with software prefetching is 2.7× faster for iflux, 2.05× for vflux, 2.7× for diflx and 2.0× for dvflux. However, it is perhaps important to remark that implementing software prefetching in a large application is significantly easier than exploiting parallelism at another granularity.

Another aspect worth highlighting is that on KNC, best performance is obtained by overlapping prefetches in the L1 with prefetches in the L2 whereas this is not the case on all the other processors where best results are obtained by prefetching into L2/L3 only. This is to be expected due to the

aforementioned lack of an L1 hardware prefetcher on KNC which means that prefetching into L2 only will still lead to cache misses in L1 if data is not prefetched from L2.

Finally, it is important to mention that results on KNC were only obtained using the intake test case and running in a single core (1 MPI rank) configuration due to memory size limitations.

Knights Landing. For Knights Landing, software prefetching results in a 1.25× speed-up for the full application when running on all 64 cores and more than 1.5× on a tile (2 cores sharing an L2 cache) as shown in fig. 12. In face-based loops, software prefetching obtains speed-ups between 1.54× and 2.2× (fig. 11). 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 that the L1 prefetches hold critical hardware resources such as Line Fill Buffers until the cache line fill completes whereas L2 prefetchers do not [Jeffers et al. 2016]. An important aspect to take into account on KNL is that not prefetching the regular accesses in face-based loops is not as detrimental to performance compared to all other architectures. This might be because the stream/stride hardware prefetchers are not affected by the software prefetches as on the other architectures.

7 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 and using two realistic test cases popular in the aerospace community.

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 targeting specific cache levels (i.e., L1 and/or L2 only). 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. Moreover, we have demonstrated that in loops with mixed access patterns (i.e., irregular as well as regular), it is imperative that prefetches target both, as otherwise the impact of prefetching the irregular accesses only is reduced significantly on the majority of processors. Finally, we showed significant full application speed-ups across a number of processors in both single core/tile and full socket configurations, such as the Intel Xeon Sandy Bridge (1.14×), Broadwell (1.09×) and Skylake CPUs (1.29×) as well as on the in-order Intel Xeon Phi Knights Corner architecture (1.99×) and the out-of-order Knights Landing many-core processor (1.51×).

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 Cirrus UK National Tier-2 HPC Service at EPCC (http://www.cirrus.ac.uk) funded by the University of Edinburgh and EPSRC (EP/P020267/1) and the ARCHER KNL Testing and Development Platform part of the UK National Supercomputing Service (http://wwww.archer.ac.uk). The authors are indebted to Adrian Jackson of the University of Edinburgh for providing access to the Cirrus and KNL systems.

REFERENCES

2016. OpenMP 4.0 Specifications. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf. Accessed: 15-05-2016.

2017. FUN3D. https://fun3d.larc.nasa.gov/. Accessed: 31-08-2017.

Sam Ainsworth and Timothy M. Jones. 2017. Software Prefetching for Indirect Memory Accesses. In *CGO '17*. IEEE Press, Piscataway, NJ, USA, 305–317.

Intel Vtune Amplifier. 2019. Intel Vtune Amplifier. https://software.intel.com/en-us/vtune.

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software Prefetching. In *ASPLOS*. ACM, New York, NY, USA, 40–52.

Mauro Carnevale, Jeffrey S Green, and Luca Di Mare. 2014. Numerical studies into intake flow for fan forcing assessment. In *Proceedings of ASME Turbo Expo 2014: Turbine Technical Conference and Exposition*. 16–20.

Mauro Carnevale, Feng Wang, and Luca di Mare. 2017. Low Frequency Distortion in Civil Aero-engine Intake. *Journal of Engineering for Gas Turbines and Power* 139, 4 (2017), 041203.

Luca Di Mare, Davendu Y Kulkarni, Feng Wang, Artyom Romanov, Pandia R Ramar, and Zacharias I Zachariadis. 2011. Virtual gas turbines: Geometry and conceptual description. *Proceedings of ASME TurboExpo, Vancouver, Canada* (2011).

M. A. Al Farhan and D. Keyes. 2018. Optimizations of Unstructured Aerodynamics Computations for Many-core Architectures. *IEEE Transactions on Parallel and Distributed Systems* (2018), 1–1. https://doi.org/10.1109/TPDS.2018.2826533

William D Gropp, Dinesh K Kaushik, David E Keyes, and Barry F Smith. 2001. High-performance parallel implicit CFD. *Parallel Comput.* 27, 4 (2001), 337 – 362. https://doi.org/10.1016/S0167-8191(00)00075-2 Parallel computing in aerospace.

Ioan Hadade. 2018. Bitbucket. https://bitbucket.org/ioanhadade/au3x-ia3-reproduce/.

Ioan Hadade, Timothy M. Jones, Feng Wang, and Luca di Mare. 2018a. Software prefetching for unstructured mesh applications. In *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 11–19.

Ioan Hadade, Feng Wang, Mauro Carnevale, and Luca di Mare. 2018b. Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures. *Computer Physics Communications* (2018).

Charles Hirsch. 1990. *Numerical Computation of Internal and External Flows*. John Wiley and Sons, Chichester, West Sussex, UK.

Intel Corporation. 2017. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-037.

Jim Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming*. Morgan Kaufmann.

Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9, 1, Article 2 (March 2012), 29 pages. https://doi.org/10.1145/2133382.2133384

Rainald Löhner. 2010. Cache-efficient renumbering for vectorization. *International Journal for Numerical Methods in Biomedical Engineering* 26, 5 (2010), 628–636. https://doi.org/10.1002/cnm.1160

D.J. Mavriplis. 2003. *Revisiting the Least-squares Procedure for Gradient Reconstruction on Unstructured Meshes*. Technical Report NASA/CR-2003-212683. National Aeronautics and Space Administration.

Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. *SIGPLAN Not.* 27, 9 (Sept. 1992), 62–73. https://doi.org/10.1145/143371.143488

D. Mudigere, S. Sridharan, A. Deshpande, J. Park, A. Heinecke, M. Smelyanskiy, B. Kaul, P. Dubey, D. Kaushik, and D. Keyes. 2015. Exploring Shared-Memory Optimizations for an Unstructured Mesh CFD Application on Modern Parallel Systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 723–732. https://doi.org/10.1109/IPDPS.2015.114

J. P. Murphy and D. G. MacManus. 2011. Ground vortex aerodynamics under crosswind conditions. *Experiments in Fluids* 50, 1 (01 Jan 2011), 109–124. https://doi.org/10.1007/s00348-010-0902-4

Perf. 2019. Perk Wiki. https://perf.wiki.kernel.org/index.php/Main_Page.

L. Reid and D. Moore. 1978. *Design and overall performance of four highly loaded, high-speed inlet stages for an advanced high-pressure-ratio core compressor*. Technical Report TP-1337. National Aeronautics and Space Administration.

Philip L Roe. 1981. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics* 43, 2 (1981), 357–372.

Bram van Leer. 1979. Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *J. Comput. Phys.* 32, 1 (1979), 101 – 136. https://doi.org/10.1016/0021-9991(79)90145-1

Steven P. Vanderwiel and David J. Lilja. 2000. Data Prefetch Mechanisms. *ACM Comput. Surv.* 32, 2 (June 2000), 174–199.

Feng Wang, Mauro Carnevale, Gan Lu, Luca di Mare, and Davendu Kulkarni. 2016. Virtual Gas Turbine: Pre-Processing and Numerical Simulations. In *ASME Turbo Expo 2016*. American Society of Mechanical Engineers.

D. C. Wilcox. 1988. Reassessment of the scale-determining equation for advanced turbulence models. *AIAA Journal* 26 (Nov. 1988), 1299–1310. https://doi.org/10.2514/3.10041

Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588