Modeling Stencil Computations on Modern HPC Architectures

Raúl de la Cruz[1] and Mauricio Araya-Polo[2]

[1] CASE Department, Barcelona Supercomputing Center, Barcelona, Spain
delacruz@bsc.es
[2] Shell International Exploration & Production Inc., Houston, Texas, USA
mauricio.araya@shell.com

Abstract. Stencil computations are widely used for solving Partial Differential Equations (PDEs) explicitly by Finite Difference schemes. The stencil solver alone -depending on the governing equation- can represent up to 90% of the overall elapsed time, of which moving data back and forth from memory to CPU is a major concern. Therefore, the development and analysis of source code modifications that can effectively use the memory hierarchy of modern architectures is crucial. Performance models help expose bottlenecks and predict suitable tuning parameters in order to boost stencil performance on any given platform. To achieve that, the following two considerations need to be accurately modeled: first, modern architectures, such as Intel Xeon Phi, sport multi- or many-core processors with shared multi-level caches featuring one or several prefetching engines. Second, algorithmic optimizations, such as spatial blocking or Semi-stencil, have complex behaviors that follow the intricacy of the above described modern architectures. In this work, a previously published performance model is extended to effectively capture these architectural and algorithmic characteristics. The extended model results show an accuracy error ranging from 5-15%.

Keywords: stencil computation, FD, modeling, HPC, prefetching, spatial blocking, semi-stencil, multi-core, Intel Xeon Phi

1 Introduction

Stencil computations are the core of many Scientific Computing applications. Geophysics [1], astrophysics [2], nuclear physics [20] or oceanography [8, 13] are scientific fields where large computer simulations are frequently carried out. Their governing PDEs are usually solved by the Finite-Difference (FD) method, using stencil computations to explicitly calculate the differential operators which represent a large fraction of the total execution time.

In a stencil computation, each point of the computational domain accumulates the weighted contribution of certain neighboring points through every axis, thus solving the spatial differential operator. The more neighboring points are used for this operation, the higher accuracy is obtained. Two inherent problems can be identified from the structure of the stencil computation [6]. First

is the noncontiguous memory access pattern while accessing neighbors in the least-stride dimensions. Second is the low *Operational Intensity* (OI) of stencil computations, which leads to a poor data reuse of the values fetched to the CPU through the memory hierarchy. Therefore, optimizing stencil computations is crucial in order to reduce the application execution time.

The manual trial-and-error approach turns the process of optimizing codes lengthy and tedious. The large number of stencil optimization combinations, which might consume days of computing time, makes the process lengthy. Furthermore, the process is tedious due to the slightly different versions of code that must be implemented and assessed. To alleviate the cumbersome optimization process from user supervision, several auto-tuning frameworks [10, 3] have been developed to automatize the search by using heuristics to guide the parameter subspace. As an alternative, models that predict performance can be built without the requirement of any actual stencil computation execution. These models can be used in auto-tuning frameworks for compile- and run-time optimizations; making guided decisions about the best algorithmic parameters, thread execution configuration or even suggesting code modifications.

We propose a model that is highly time-cost effective compared to other approaches based on regression analysis. In regression-based analysis, users are required to conduct extensive and costly experiments in order to obtain the input data for regression. A wide range of hardware performance counters are gathered and machine learning algorithms used to determine correlations between architectural events and compiler optimizations. The more complex the model is, the more data is required to estimate the correlation coefficients. Furthermore, regression models lack of cache miss predictors and neither provide hints about algorithmic parameter candidates (e.g. spatial blocking). Albeit, regression analysis can be partially useful whether it is intended to give indications of possible performance bottlenecks and is combined with knowledge-based systems.

The performance characterization of a kernel code is not trivial and relies heavily on the ability to capture the algorithm's behavior in an accurate fashion, independently of the platform and the execution environment. In order to do so, the estimation of memory latencies is critical in memory-bound kernels. This is why predicting 3C (*compulsory*, *conflict* and *capacity*) misses accurately play an important role to effectively characterize the kernel performance.

In this paper, we extend our multi-level cache model for 3D stencil computations [5] by consolidating HPC support. Previous works have already proposed cache misses and execution time models for specific stencil optimizations. However, most of them have been designed for simplified architectures or low order stencil sizes (7-point), leaving aside many considerations of modern HPC architectures. Nowadays, multi- and many-core architectures with multi-level cache hierarchies, prefetching engines and SMT capabilities are common on HPC platforms but also disregarded or barely covered by previous works. The challenge is to cover all these features to effectively model stencil computation performance.

We have used a leading hardware architecture in our experiments, the popular Intel Xeon Phi 5100 series (SE10X model), also known as MIC. This architecture

shows outstanding appeal for this work due to its support for all of the new features that our extended model intend to cover.

The remaining paper is organized as follows. Section 2 overviews briefly the related work. In section 3, we elaborate on the basic fundamentals of our performance model, including some phenomena such as cache interference. Section 4 details the considerations to extrapolate the model to multi- and many-core architectures. Section 5 explains how the prefetching effect can be modeled. In section 6, two stencil optimizations are discussed and added to the model. Section 7 presents the experimental results and evaluates their accuracy. Finally, section 8 summarizes the findings of this work and concludes the paper.

2 Related Work

The modeling topic on stencil computations has been fairly studied in the recent years. A straightforward model was initially published by Kamil *et al.* [12], where they proposed cost models to capture the performance of 7-point stencils by taking into account three types of memory accesses (*first*, *intermediate* and *stream*) in a flat memory hierarchy. Then, a simple approach was devised by setting a lower bound ($2C_{stencil}$) with only compulsory misses and an upper bound ($4C_{stencil}$) with no cache reuse at all. Spatial blocking support was also added by modifing the number of cache-lines fetched using the three types of memory accesses due to the disruption of the prefetching effect. Regression analysis has also shown some appeal for modeling stencil computations [19]. They developed a set of formulas via regression analysis to model the overall performance on 7 and 27-point Jacobi and Gauss-Seidel computations. Their intent was not to predict absolute execution time but to extract meaningful insights that might help developers to effectively improve their codes. The time-skewing technique has been also modelized by Strzodka *et al.* [22]. They proposed a performance model for their cache accurate time skewing (CATS) algorithm, where the system and the cache bandwidths were estimated using regression analysis. The CATS performance model considered only two levels of memory hierarchy, and therefore it could be inaccurate on HPC architectures. Their aim was to find out which hardware improvements were required in single-core architectures to match the performance of future multi-core systems.

Likewise, performance modeling has been successfully deployed on numerical areas such as sparse matrix vector multiplications [18] and generic performance models for bandwidth-limited loop kernels [9, 24].

3 Stencil Model

In this work, we use the model initially published at [5] as starting point. This performance model considers stencil computations as memory-bound, where the cost of computing the floating-point operations is assumed negligible due to the overlap with considerable memory transfers. This assumption is especially true for large domain problems where, apart from compulsory misses, capacity and

conflict misses arise commonly leading to a low OI [6]. Some concepts of the initial model are improved and extended to fulfill the coverage of the current work. For a better understanding of the remaining sections, the main concepts and assumptions of the base model are briefly reviewed in the next section.

Algorithm 1 The classical stencil algorithm pseudo-code. II, JJ, KK are the dimensions of the data set including ghost points. ℓ denotes the neighbors used for the central point contribution. $C_{Z1...Z\ell}$, $C_{X1...X\ell}$, $C_{Y1...Y\ell}$ are the spatial discretization coefficients for each direction and C_0 for the self-contribution. Notice that the coefficients are considered symmetric and constant for each axis.

1: **for** $t = 0$ to *timesteps* **do** ▷ Iterate in time
2: **for** $k = \ell$ to $KK - \ell$ **do** ▷ Y axis
3: **for** $j = \ell$ to $JJ - \ell$ **do** ▷ X axis
4: **for** $i = \ell$ to $II - \ell$ **do** ▷ Z axis
5:

$$\begin{aligned}\mathcal{X}^t_{i,j,k} = {} & C_0 * \mathcal{X}^{t-1}_{i,j,k} \\ & + C_{Z1} * (\mathcal{X}^{t-1}_{i-1,j,k} + \mathcal{X}^{t-1}_{i+1,j,k}) + \ldots + C_{Z\ell} * (\mathcal{X}^{t-1}_{i-\ell,j,k} + \mathcal{X}^{t-1}_{i+\ell,j,k}) \\ & + C_{X1} * (\mathcal{X}^{t-1}_{i,j-1,k} + \mathcal{X}^{t-1}_{i,j+1,k}) + \ldots + C_{X\ell} * (\mathcal{X}^{t-1}_{i,j-\ell,k} + \mathcal{X}^{t-1}_{i,j+\ell,k}) \\ & + C_{Y1} * (\mathcal{X}^{t-1}_{i,j,k-1} + \mathcal{X}^{t-1}_{i,j,k+1}) + \ldots + C_{Y\ell} * (\mathcal{X}^{t-1}_{i,j,k-\ell} + \mathcal{X}^{t-1}_{i,j,k+\ell})\end{aligned}$$

3.1 Base Model

Considering a problem size of $I \times J \times K$ points of order ℓ, where I is the unit-stride (Z axis) and J and K the least-stride dimensions (X and Y axes), an amount of P_{read} $(2 \times \ell + 1)$ and P_{write} (1) Z-X planes of $\mathcal{X}^{t-1}$ is required to compute a single $\mathcal{X}^t$ plane (see Algorithm 1). Thus, the total data to be held is $S_{total} = P_{read} \times S_{read} + P_{write} \times S_{write}$, being $S_{read} = II \times JJ$ and $S_{write} = I \times J$ their size in words. Note that II and JJ include ghost points.

Likewise, the whole execution time (T_{total}) on an architecture with n levels of cache is estimated based on the aggregated cost of transferring data on three memory hierarchy groups: *first* (T_{L1}), *intermediate* (T_{L2} to T_{Ln}) and *last* (T_{Memory}). Each transferring cost depends on their hits and misses and is computed differently. In general, the transferring cost ($T_{Li} = Hits^{data}_{Li} \times T^{data}_{Li}$) is based on the latency of bringing as much data (word or cacheline) as required ($Hits^{data}_{Li} = Misses^{data}_{Li-1} - Misses^{data}_{Li}$) from the cache level to the CPU ($T^{data}_{Li} = data/Bw^{read}_{Li}$) in order to compute the stencil. Finally, the amount of misses issued at each cache level is estimated as

$$Misses_{Li} = \lceil II/W \rceil \times JJ \times KK \times nplanes_{Li} \ , \tag{1}$$

where $W = cacheline/word$ is the number of words per cacheline and $nplanes_{Li}$ is the number of $II \times JJ$ planes read from the next cache level $(Li + 1)$ for each k iteration due to possible compulsory, conflict or capacity misses. The cache miss calculations are described in the following section.

3.2 Cache Miss Cases and Rules

The correct estimation of $nplanes_{Li}$ is crucial for the model accuracy. To do so, four miss cases (C_1, C_2, C_3 and C_4, ordered from lower to higher penalty) and four rules (R_1, R_2, R_3 and R_4) are devised. Each of these rules triggers the transition from one miss case scenario to the next one. In this model, the rules are linked and therefore triggered in sequential order, thus exposing different levels of miss penalty.

Rule 1 *(R_1):* The best possible scenario (lower bound) is likely to happen when all the required Z-X planes (S_{total}) to compute one k iteration fit loosely (R_{col} factor) into the cache level ($size_{Li}$). This yields to only compulsory misses and to the following rule, $R_1 : ((size_{Li}/w) \times R_{col} \geq S_{total})$.

Rule 2 *(R_2):* Conversely to R_1, when all the required planes do not fit loosely in cache except the k-central plane with a higher temporal reuse (less chance to be evicted from cache), conflict misses are produced among planes. This scenario is likely to happen when the following rule is true, $R_2 : ((size_{Li}/w) > S_{total})$.

Rule 3 *(R_3):* On a third possible scenario, it is assumed that despite the whole data set does not fit in cache (S_{total}), the k-central plane does not overwhelm a significant part of the cache (R_{col} factor). Therefore, the possibility of temporal reuse is reduced compared to R_2 but not canceled completely. This scenario can occur when, $R_3 : ((size_{Li}/w) \times R_{col} > S_{read})$.

Rule 4 *(R_4):* The worst scenario (upper bound) appears when neither the planes nor the columns of the k-central plane fit loosely in the cache level. Then, capacity and conflict misses arise frequently, resulting as well in fetching the k-central plane at each j iteration of the loop. This scenario gives the following rule, $R_4 : ((size_{Li}/w) \times R_{col} < P_{read} \times II)$.

w is the word size (in single or double precision), and R_{col} is a factor proportional to the required data by the k-central plane with respect to the whole dataset ($P_{read}/2P_{read}-1$). Putting all the ingredients together, the computation of $nplanes_{Li}$ is yielded by the following conditional equations:

$$nplanes_{Li}(II, JJ) = \begin{cases} C_1 : 1, & \text{if } R_1 \\ C_1 \sqcup C_2 : (1, P_{read} - 1], & \text{if } \neg R_1 \wedge R_2 \\ C_2 \sqcup C_3 : (P_{read} - 1, P_{read}], & \text{if } \neg R_2 \wedge R_3 \\ C_3 \sqcup C_4 : (P_{read}, 2P_{read} - 1], & \text{if } \neg R_3 \wedge \neg R_4 \\ C_4 : 2P_{read} - 1, & \text{if } R_4 \ , \end{cases} \tag{2}$$

which only depends on II and JJ parameters for a given architecture and a stencil order (ℓ). Figure 1 shows an example of how $nplanes_{Li}$ evolves with respect to $II \times JJ$ parameter.

Large discontinuities can appear in Equation 2 when transitioning from one case to the next case ($C_1 \sqcup C_2$, $C_2 \sqcup C_3$ and $C_3 \sqcup C_4$). This effect can be partially

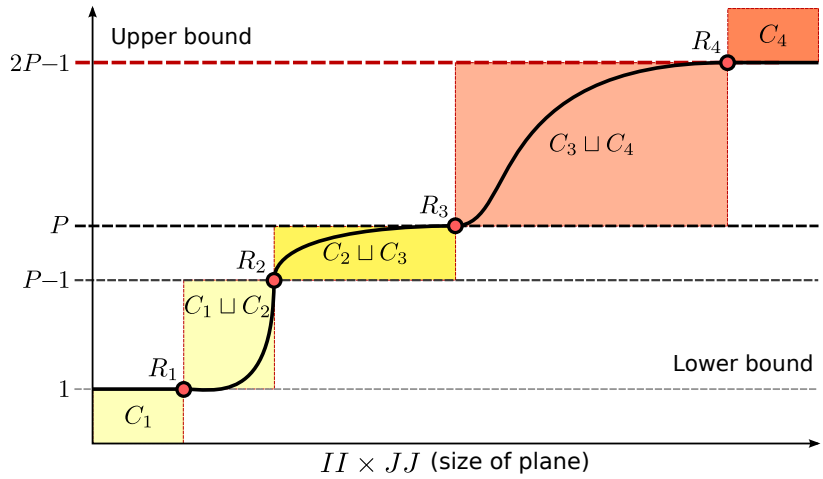

Fig. 1: The different rules (R_1, R_2, R_3 and R_4) bound the size of the problem (abscissa: $II \times JJ$) with the miss case penalties (ordinate: 1, $P_{read} - 1$, P_{read} and $2P_{read} - 1$).

smoothed by using interpolation methods. Apart from the discrete transitioning, three types of interpolations have been added in our model: linear, exponential and logarithmic. An interpolation function ($f(x, x_0, x_1, y_0, y_1)$) requires five input parameters, the X-axis bounds (x_0 and x_1), the Y-axis bounds (y_0 and y_1) and the point in the X-axis (x) to be mapped into the Y-axis (y). In our problem domain, the X-axis represents the $II \times JJ$ parameters whereas the Y-axis is the unknown $nplanes_{Li}$. For instance, for $C_1 \sqcup C_2$ transition, isolating II from R_1 and R_2 rules, II_{min} (x_0) and II_{max} (x_1) are respectively obtained, bounding the interpolation. By using their respective rules and isolating the required variable for X-axis, the same procedure is also applied to the remaining transitions of Equation 2. In this way, an easy methodology is presented to avoid unrealistic discontinuities for the model.

3.3 Cache Interference Phenomena: II × JJ effect

As stated before, three types of cache misses (3C) can be distinguished: compulsory (*cold-start*), capacity and conflict (*interference*) misses. Compulsory and capacity misses are relatively easily predicted and estimated [23]. Contrarily, conflict misses are hard to evaluate because it must be known where data are mapped in cache and when it will be referenced. In addition, conflict misses disrupt data reuse, spatial or temporal. For instance, a high frequency of cache interferences can lead to the rare *ping-pong* phenomena, where two or more memory references fall into the same cache location, therefore competing for cache-lines. Cache associativity can alleviate this issue to a certain extent by increasing the cache locations for the same address.

The cache miss model presented in Subsection 3.2 sets the upper bound for each of the four cases in terms of number of planes read for each plane writ-

ten ($nplanes_{Li}$), thus establishing a discrete model. Nevertheless, this discrete scenario is unlikely to happen for cases C_2, C_3 and mainly C_4, due to their dependency on capacity and especially on conflict misses. There are two factors that clearly affect conflict misses: the reuse distance for a given datum [23] and the intersection of two data sets [9], giving consequently a continuum scenario. The former depends on temporal locality; the more data is loaded, the higher the probability that a given datum may be flushed from cache before its reuse. On the other hand, the latter depends on two parameters: the array base address and its leading dimensions.

In stencil computations the Z-X plane ($II \times JJ$ size) and the order of the stencil ($P_{read} = 2 \times \ell + 1$) are the critical parameters that exacerbate conflict misses. The conflict misses to estimate are related with the probability of interference, $P(i)$, and the column reuse of the central k-plane. $P(i)$ is proportional to the size in words of the columns to be reused ($II \times (P_{read} - 1)$) after reading the first central column with respect to the whole size of the central k-plane to be held in cache ($II \times JJ$),

$$P(i) = \frac{II \times JJ - II \times (P_{read} - 1)}{II \times JJ} = 1 - \frac{P_{read} - 1}{JJ} \in [0, 1] \ , \tag{3}$$

which yields to a logarithmic function depending on P_{read}, II and JJ parameters. A zero value means no conflict misses at all, whereas a probability of one means disruption of temporal reuse (high ratio of interferences) for columns of the central k-plane. Therefore, the $P(i)$ probability can be added as

$$nplanes_{Li}\prime = nplanes_{Li} \times P(i) \ , \tag{4}$$

tailoring the read misses case boundary to their right value depending on the conflict misses issued. Thus, the larger the data to be used to compute one output plane ($I \times J$), the higher the probability of having capacity and conflict misses. Figure 2 shows the accuracy difference between the model with and without cache interference effect.

3.4 Additional Time Overheads

During the execution of HPC stencil codes, some additional overheads may arise. In this subsection, we explain how these overheads are weighed when modeling the stencil computation performance. The overheads are categorized into three groups: parallelism, memory interferences and computational bottlenecks.

- Intra-node parallelism (OpenMP and Posix threads): small overheads may appear due to the thread initialization and synchronization tasks whether data is disjoint among threads. This overhead usually has a clear impact only on small dataset problems. In order to characterize its effect on the stencil model, a small (order of milliseconds) and constant ϵ (T_{OMP}) is included.
- Memory contention: TLB misses, ECC memories (error checking & corruption) and cache coherence policies between cores (e.g. MESI protocol) affect noticeably the memory performance. Nevertheless, all these effects

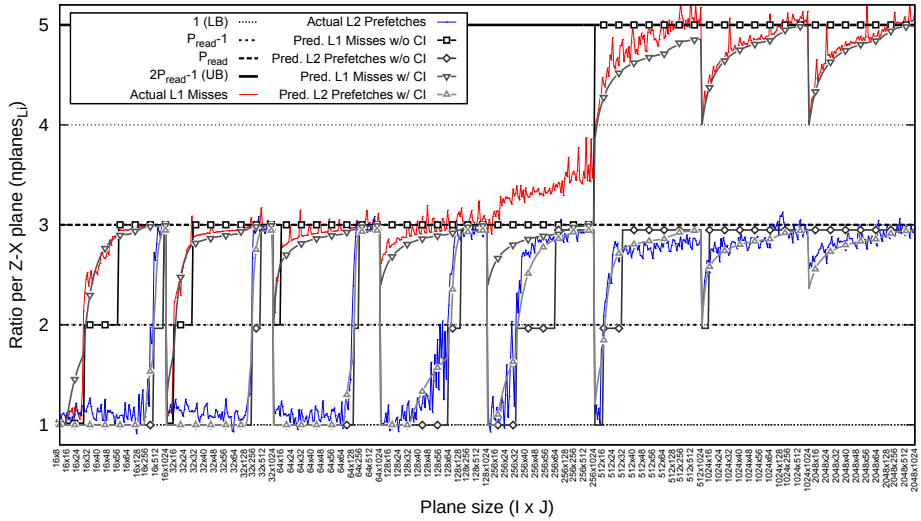

Fig. 2: Cache interference effect as a function of problem size. Whilst equation 4 is not applied, a discrete model is obtained (straight lines with squares and diamonds). Conversely, its use leads to a continuum model (inverted and non-inverted triangles).

are already taken into account in the memory characterization through our STREAM2 tool (see Section 4 for further details).

– Computational bottlenecks: stencil computations are mainly considered memory bound instead of compute bound (the OI is low) [25, 6]. Therefore, for the sake of simplicity, the tampering effect of floating-point operations is expected to be negligible, and thus not considered.

4 From Single-core to Multi-core and Many-core

Current HPC platforms are suboptimal for scientific codes unless they take fully advantage of simultaneous threads running on multi- and many-cores chips. Some clear examples of such architectures are Intel Xeon family, IBM POWER7 or GPGPUs. All of them with tens of cores and their ability to run in SMT mode. So, the parallel nature of the current stencil computation deployments leads us to extend our model accordingly. To that end, the parallel memory management is a main concern, and this section is fully devoted to sort it out.

In order to characterize the memory management of multi-core architectures, the bandwidth measurement is critical. The bandwidth metrics are captured for different configurations using a bandwidth profiler such as STREAM2 benchmark [15]. Our STREAM2 version [5] has been significantly extended by adding new features such as vectorization (SSE, AVX and Xeon Phi ISAs), aligned and unaligned memory access, non-temporal writes (through Intel pragmas), prefetching and non-prefetching bandwidths, thread-level execution (OpenMP) and hardware counters instrumentation (PAPI) in order to validate results.

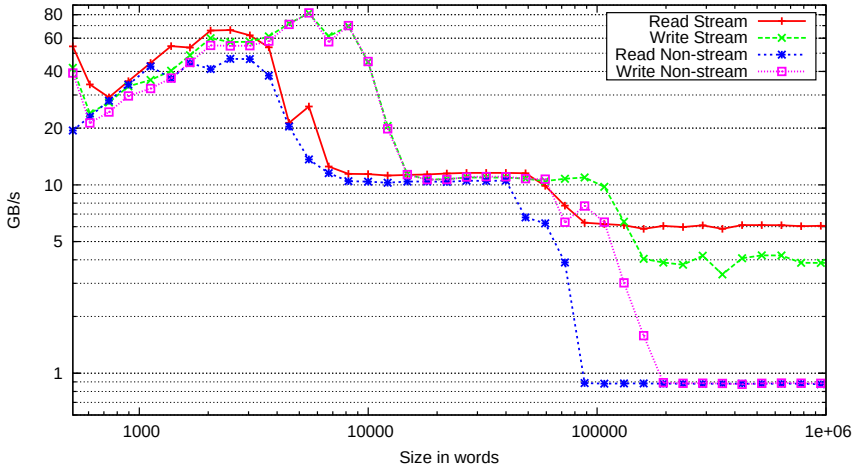

Fig. 3: STREAM2 results for Intel Xeon Phi architecture (4 threads, 2 per core). Each plateaux represents the sustainable bandwidth of a cache level.

The process to obtain bandwidth measurements is straightforward. First, the thread number is set through the OMP_NUM_THREADS environment variable. Then, each thread is pinned to a specific core of the platform (e.g. using *numactl* or KMP_AFFINITY variable in Xeon Phi architecture). Finally, the results obtained for DOT (16 bytes/read) and FILL (8 bytes/write) kernels are respectively used as read and write bandwidths for the different cache hierarchies of the model. Figure 3 shows an example of the bandwidths used for a particular case in the Intel Xeon Phi platform. The importance of mimicking the environment conditions is crucial, in particular the execution time accuracy of the model is very sensitive to the real execution conditions. This means that the characterization of the memory bandwidth must be similarly performed in terms of: number of threads, threads per core, memory access alignment, temporal or non-temporal writes and SISD or SIMD instruction set.

Additionally, there are some memory resources that might be shared among different threads running in the same core or die. In order to model the behaviour in such cases, the memory resources are equally split among all threads. This is, if we have a cache size ($size_{Li}$ in rules $R_{1,2,3,4}$) of N KBytes, then each thread would turn out to have a cache size of $size_{Li} = N/nthreads_{core}$.

5 Modeling Prefetching

5.1 Hardware Prefetching

Modern computer architectures incorporate prefetching engines in their cache hierarchy. Its aim is to reduce the memory latency by eagerly fetching data that is expected to be required in the near future.

The prefetching mechanism modeled in our previous work [5] lacked accuracy when several threads were triggering the prefetching engine concurrently. As stated in [5], the modeling of the prefetching mechanism is not straightforward. In that work, a simple approach was devised. The miss model was divided into two groups, prefetched and non-prefetched misses, depending on the concurrent streams that the prefetching engine supported. Next, two different bandwidths were used for each cache miss group in order to compute their time penalty.

Recent works [14, 16] have characterized the impact of prefetching mechanism on scientific application performance. They establish a new metric called *prefetching effectiveness*, which computes the fraction of data accesses to the next memory level that are initiated by the hardware prefetcher. Therefore, for a given data cache level (DC), its prefetching effectiveness is computed as

$$DC_{effectiveness} = DC\_Req\_PF/DC\_Req\_All \in [0,1] \ , \tag{5}$$

where DC_Req_PF refers to the number of cache-lines requests initiated by the prefetching engine, and DC_Req_All represents the total number of cache-lines requests initiated at the DC level (including demanding and non-demanding loads). This approach has been adopted in our model as the way to accurately capture the prefetching behaviour.

In order to be able to characterize the prefetching effectiveness in our testbed platform, a new micro-benchmark was developed from scratch. This benchmark traverses a chunk of memory simultaneously by different threads and changes the number of stream accesses in a round-robin fashion. Then, to compute their effectiveness, a set of hardware performance counters were gathered through PAPI. For instance, on Intel Xeon Phi architecture, two native events were instrumented to compute the *prefetching effectiveness*: HWP_L2MISS and L2_DATA_READ_MISS_MEM_FILL. Figure 4 shows the results obtained for this platform over the L2 hardware prefetcher.

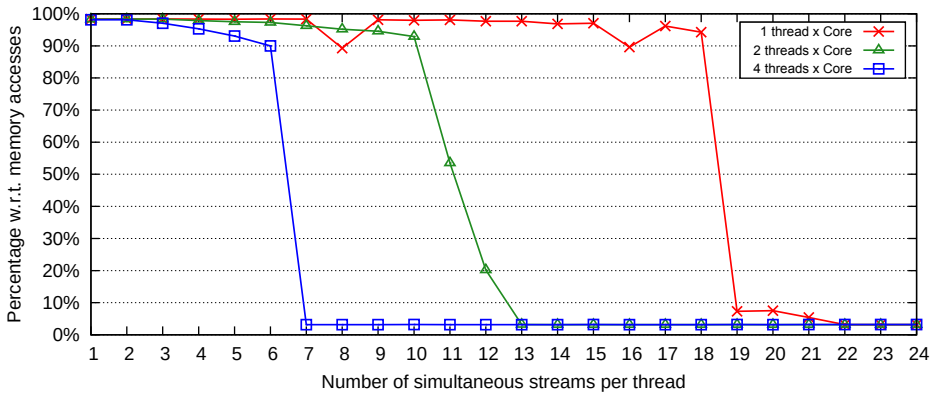

Fig. 4: L2 prefetching efficiency for Intel Xeon Phi architecture. The efficiency has been computed using one core and varying the SMT configuration from 1 to 4 threads.

The *prefetching effectiveness* ($DC_{effectiveness}$) is then used to compute the total number of cache-line misses that are fetched using streaming bandwidths ($nplanes^{S}_{Li}$) and those that are fetched using a regular bandwidth ($nplanes^{NS}_{Li}$):

$$\begin{aligned} nplanes^{S}_{Li} &= nplanes_{Li} \times DC_{effectiveness}, \\ nplanes^{NS}_{Li} &= nplanes_{Li} \times (1 - DC_{effectiveness}) \ . \end{aligned} \tag{6}$$

Similarly to the memory resources, prefetching engines might be shared among threads running on the same core. In such scenarios, the *prefetching effectiveness* is computed with our prefetching tool varying the number of threads per core (for instance, 2 and 4 threads results can be observed in Figure 4). In fact, these results are insightful and help to understand when the core performance might be degraded due to excessive simultaneous streams, thus adversely affecting the parallel scaling of stencil computations.

5.2 Software Prefetching

Software prefetching is a technique where compilers, and also programmers, explicitly insert prefetching operations similar to load instructions into the code. Predicting the performance of software prefetching is challenging. Compilers use proprietary heuristics in order to decide where (code location), which (data array) and how much in advance (look-ahead in bytes) start prefetching data. Furthermore, programmers can even harden this task by adding special hints in the code to help the compiler make some of these decisions [17]. As software prefeching produces regular loads on the cache hierarchy, it also prevents hardware prefetcher to be triggered when it performs properly [7]. Thus, the failure or success of software prefetching affects collaterally the hardware prefeching behaviour.

Due to all above commented issues, software prefetching has not been taken into account in the present work. The software prefetching can be disabled in Intel compilers by using the `-opt-prefetch=0` flag during the compilation.

6 Stencil Optimizations

The state-of-the-art in stencil computation is constantly being extended with the publication of several optimization techniques in recent years. Under specific circumstances, some of those techniques improve the execution performance. For instance, space blocking is a tiling strategy widely used in multi-level cache hierarchy architectures. It promotes data reuse by traversing the entire domain into small blocks of size $TI \times TJ$ which must fit into the cache [21, 12]. Therefore, space blocking is especially useful when the dataset structure does not fit into the memory hierarchy. This traversal order reduces capacity and conflict misses in least-stride dimensions increasing data locality and overall performance. Note that a search of the best block size parameter ($TI \times TJ$) must be performed for each problem size and architecture.

A second example of stencil optimization is the Semi-stencil algorithm [6]. This algorithm changes the way in which the spatial operator is calculated and how data is accessed in the most inner loop. Actually, the inner loop involves two phases called *forward* and *backward* where several grid points are updated simultaneously. By doing so, the dataset requirements of the internal loop is reduced, while keeping the same number of floating-point operations. Thereby, increasing data reuse and thus the OI. Conversely to read operations, the number of writes are slightly increased because the additional point updates. Due to this issue, this algorithm only improves performance on medium-large stencil orders ($\ell > 2$).

These two stencil optimizations have been included into our model. The motivation of modeling them is two-fold. First, to reveal insights of where and why an algorithm may perform inadequately for a given architecture and environment. Second, to analytically guide the search for good algorithmic parameter candidates without the necessity of obtaining them empirically (brute force).

6.1 Spatial Blocking

Space blocking is implemented in our model by including similar general ideas as [4], but adapting them in order to suit the advantages of our cost model. Basically, the problem domain is traversed in $TI \times TJ \times TK$ blocks. Then, first the blocks on each direction are computed as $NBI = I/TI$, $NBJ = J/TJ$, and $NBK = K/TK$. Therefore, the total number of tiling iterations to perform are $NB = NBI \times NBJ \times NBK$. Blocking may be performed in the unit-stride dimension as well. Given that data is brought to cache in multiples of the cache-line, additional transfer overhead may arise when TI size is not multiple of cache-line. This is considered into the model by reassigning I, J, K and their extended dimensions as follows:

$$\begin{array}{lll} I = \lceil TI/W \rceil \times W, & J = TJ, & K = TK, \\ II = \lceil (TI + 2 \times \ell)/W \rceil \times W, & JJ = TJ + 2 \times \ell, & KK = TK + 2 \times \ell \ . \end{array} \tag{7}$$

The new II and JJ parameters are then used for rules $R_{1,2,3,4}$ to estimate $nplanes_{Li}$ based on the blocking size. Finally, Equation 1 shall be rewritten as

$$Misses_{Li}^{[S,NS]} = \lceil II/W \rceil \times JJ \times KK \times nplanes_{Li}^{[S,NS]} \times NB \ , \tag{8}$$

where NB factor is considered to adjust streamed ($^{S}_{Li}$) and non-streamed ($^{NS}_{Li}$) misses depending on the total number of blocking iterations.

Architectures with prefetching features may present performance degradation when $TI \neq I$ [11]. Blocking on the unit-stride dimension may tamper streaming performance due to the interference caused to the memory access pattern detection of the prefetching engine. The triggering of the prefetching engine involves a *warm-up* phase, where a number of cache-lines must be previously read (TP). Additionally, prefetching engines keep a look-ahead distance (LAP) of how many

cache-lines in advance to prefetch. Disrupting a regular memory access will produce LAP additional fetches to the next cache level if the prefetching engine was triggered. Considering all these penalties, the cache misses are updated with:

$$\begin{aligned} Misses_{Li}^{NS} &\stackrel{+}{=} TP \times JJ \times KK \times nplanes_{Li}^{NS} \times NB, \quad \text{if } II/W \geq TP, \\ Misses_{Li}^{S} &\stackrel{+}{=} LAP \times JJ \times KK \times nplanes_{Li}^{S} \times NB, \quad \text{if } II/W \geq TP\ . \end{aligned} \tag{9}$$

TP and LAP parameters can be obtained from processor manufacturer's manuals or empirically through our prefetching benchmark. To deduce such parameters, the prefetching benchmark was modifed to traverse arrays in a blocked fashion whilst TI parameter was slowly increased along different executions. Then, the prefetching hardware counter was monitored in order to flag at what precise point ($TP = \lceil TI/W \rceil$) the prefetching metric soared significantly. Likewise, LAP parameter was estimated by counting the extra prefetching loads (apart from the TP) that were issued.

6.2 Semi-stencil Algorithm

Adapt the model for the Semi-stencil algorithm is equally straightforward. Indeed, this can be achieved by setting P_{read} and P_{write} parameters correctly. By default, in a partial Semi-stencil implementation (*forward* and *backward* phases on X and Y axes), $\ell + 1$ Z-X planes from $\mathcal{X}^t$ and one $\mathcal{X}^{t+1}$ plane (k-central plane update) are read for each k iteration. As output, two planes are written back as partial ($\mathcal{X}^{t+1}_{i,j,k+\ell}$) and final ($\mathcal{X}^{t+1}_{i,j,k}$) results. However, these values can slightly increase when no room is left for the k-central columns; thus yielding

$$\begin{aligned} P_{read} &= \ell + 2, \quad P_{write} = 2, \quad \text{if } \neg R_4 \\ P_{read} &= \ell + 3, \quad P_{write} = 3, \quad \text{if } R_4 \end{aligned} \tag{10}$$

as the new data requirements to compute one output plane. This adaptability reveals the model resilience, where an absolutely different stencil algorithm can be modeled by simply tuning a couple of parameters.

7 Experimental Results

This section estimates through experimental results how accurate the model is when exposed to: prefetching, thread parallelism and code optimizations techniques. All experimental results in this section were validated using the StencilProbe [12], a synthetic benchmark that we have extended. The new StencilProbe features [6] include: different stencil orders (ℓ), thread support (OpenMP), SIMD code, instrumentation and new optimization techniques (e.g. spatial blocking and Semi-stencil). This benchmark implements the stencil scheme shown in Algorithm 1, where star-like stencils with symetric and constant coefficients are computed using 1^{st} order in time and different orders in space (see Table 1).

A large number of different problem sizes were explored in order to validate the model accuracy for a wide parametrical space. Recall that the two first

dimensions (on Z and X axes) are the critical parameters that increase the cache miss ratio ($nplanes_{Li}$) for a given stencil order (ℓ) and architecture. Therefore, the last dimension K was set to a fixed number, and the I and J dimensions were widely varied covering a large spectrum of grid sizes. All the experiments were conducted using double-precision, and the domain decomposition across threads was conducted by cutting in the least-stride dimension (Y axis) with static scheduling. Table 1 summarizes the different parameters used.

Parameters	Range of values
Naive sizes ($I \times J \times K$)	$8 \times 8 \times 128 \ldots 2048 \times 1024 \times 128$
Rivera sizes ($I \times J \times K$)	$512 \times 2048 \times 128$
Stencil sizes (ℓ)	1, 2, 4 and 7 (7, 13, 25 and 43-point respectively)
Algorithms	{Naive, Rivera} $\times$ {Classical, *Semi-stencil*}
Block sizes (TI and TJ)	$\{8, 16, 24, 32, 64, 128, 256, 512, 1024, 1536, 2048\}$

Table 1: List of parameters used for the model and the StencilProbe benchmark.

The testbed platform for all experiments is based on Intel Xeon Phi. The 22 nm Xeon Phi processor include 61 cores with 4-way SMT capabilities running at 1.1GHz. Each core is in-order and contains a 512-bit vector unit (VPU). Additionally, each core has a 32KB L1D cache, a 32KB L1I cache and a private 512KB L2 cache. This cache includes a hardware prefetcher able to prefetch 16 forward or backward sequential streams into 4KB page-size boundaries. All cores are connected together via a bi-directional ring with the standard MESI coherency protocol for maintaining the shared state among cores.

Hardware counters were gathered for all experiments in order to validate the model results against actual executions. Table 2 shows the hardware performance counters instrumented. The stencil code generated by StencilProbe is vectorized, and therefore only vector reads were fetched (VPU_DATA_READ) during executions. Additionally, the L2 prefetcher in Xeon Phi can also prefetch reads for a miss in a write-back operation (L2_WRITE_HIT) when it has the opportunity. Then, in order to fairly compare the prefetched read misses of the model with actual metrics, the L2 prefetches (HWP_L2MISS) were normalized. This normalization was performed by subtracting reads due to a miss in a write operation scaled by the *prefetching efficiency*. Likewise, some writes were considered prefetched (L2_WRITE_HIT $\times DC_{effectiveness}$) and others not (L2_WRITE_HIT $\times(1 - DC_{effectiveness})$) due to contention of the L2 prefetching engine. Finally, the remaining miss counters (VPU_DATA_READ_MISS and L2_DATA_READ_MISS_MEM_FILL) only consider demanding reads, initiated by explicit reads, and therefore were directly used as non-prefetched read misses. It is important to mention that, in our previous model [5], several complex formulas were derived to estimate the number of reads issued to the first level cache ($Hits_{L1}^{word}$). This estimation was not straightforward and lacked accuracy.

However, we realized that this parameter kept constant per loop iteration and could be precisely estimated by performing static analysis of the inner stencil loop only once (counting the numbers of reads in the object file).

Description	Intel Xeon Phi Events	Time Cost Formulas
Cycles	CPU_CLK_UNHALTED	T_{L1} = (L1 Hits - L1 Misses) $\times Bw_{L1}^{cline}$
L1 Hits	VPU_DATA_READ	$T_{L2} = Bw_{L2}^{cline}\times$ (L1 Misses - L2 Misses -
L1 Misses	VPU_DATA_READ_MISS	(L2 Prefetches - L2 Writes $\times$ Pref Eff))
L2 Misses	L2_DATA_READ_MISS_	T_{Mem} = L2 Misses $\times Bw_{Mem}^{NS} + Bw_{Mem}^{S}\times$
	MEM_FILL	(L2 Prefetches - L2 Writes $\times$ Pref Eff)
L2 Prefetches	HWP_L2MISS	T_{Writes} = L2 Writes $\times$ Pref Eff $\times Bw_{Write}^{S}$
L2 Writes	L2_WRITE_HIT	+ L2 Writes $\times$ (1 - Pref Eff) $\times Bw_{Write}^{NS}$

Table 2: Hardware counters and the formulas used to compute the projected time.

An aim of this research is to prove that stencil computations can be accurately modeled on SMT architectures. Therefore, all possible SMT combinations for a single core were sampled. Our tests were conducted using 4 threads varying their pinning to cores. KMP_AFFINITY environment variable was accordingly set to bind threads to the desired cores. The SMT configurations tried for each test were: 1 core in full usage (4 threads per core), 2 cores in half usage (2 threads per core) and 4 cores in fourth usage (1 thread per core).

Due to the sheer number of combinations sampled, only the most representative and interesting results are shown. Results have been categorized as a function of core occupancy (1, 2 and 4 threads per core) in order to explicitly visualize the effect of resource contention on the actual metrics and test the predicted results.

Figure 5 shows the actual and the predicted misses with our model (prefetched and non-prefetched for L2) on all three SMT configurations using a Naive stencil order of $\ell = 4$. In this case 680 different problem sizes (X axis in figures) were tested per configuration. Recall that software prefetching was disabled and therefore L1 or L2 cache levels do not exhibit collateral effects due to compiler-assisted prefetch. This figure is very insightful because the empirical results clearly corroborate our thoughts regarding the different bounds applied in the stencil model. Indeed, in a $\ell = 4$ stencil the read miss bounds for the model are: 1, 8 ($P_{read} - 1$), 9 (P_{read}) and 17 ($2P_{read} - 1$) per each $I \times J$ plane computed. Actual L1 and L2 misses tend to these bounds when a specific problem size is reached, never reaching beyond the upper bound ($2P_{read} - 1$) which is showed as a solid coarse horizontal line in all plots. Cache levels with prefetched and non-prefetched misses are a special case due to their direct relation with $DC_{effectiveness}$ ratio, and therefore they might be under the lower bound (1). Additionally, as the threads per core are increased, the inflection points (transitions) between bounds ($C_1 \sqcup C_2$, $C_2 \sqcup C_3$ and $C_3 \sqcup C_4$) are triggered earlier in terms of plane size ($I \times J$). The larger the number of threads running concur-

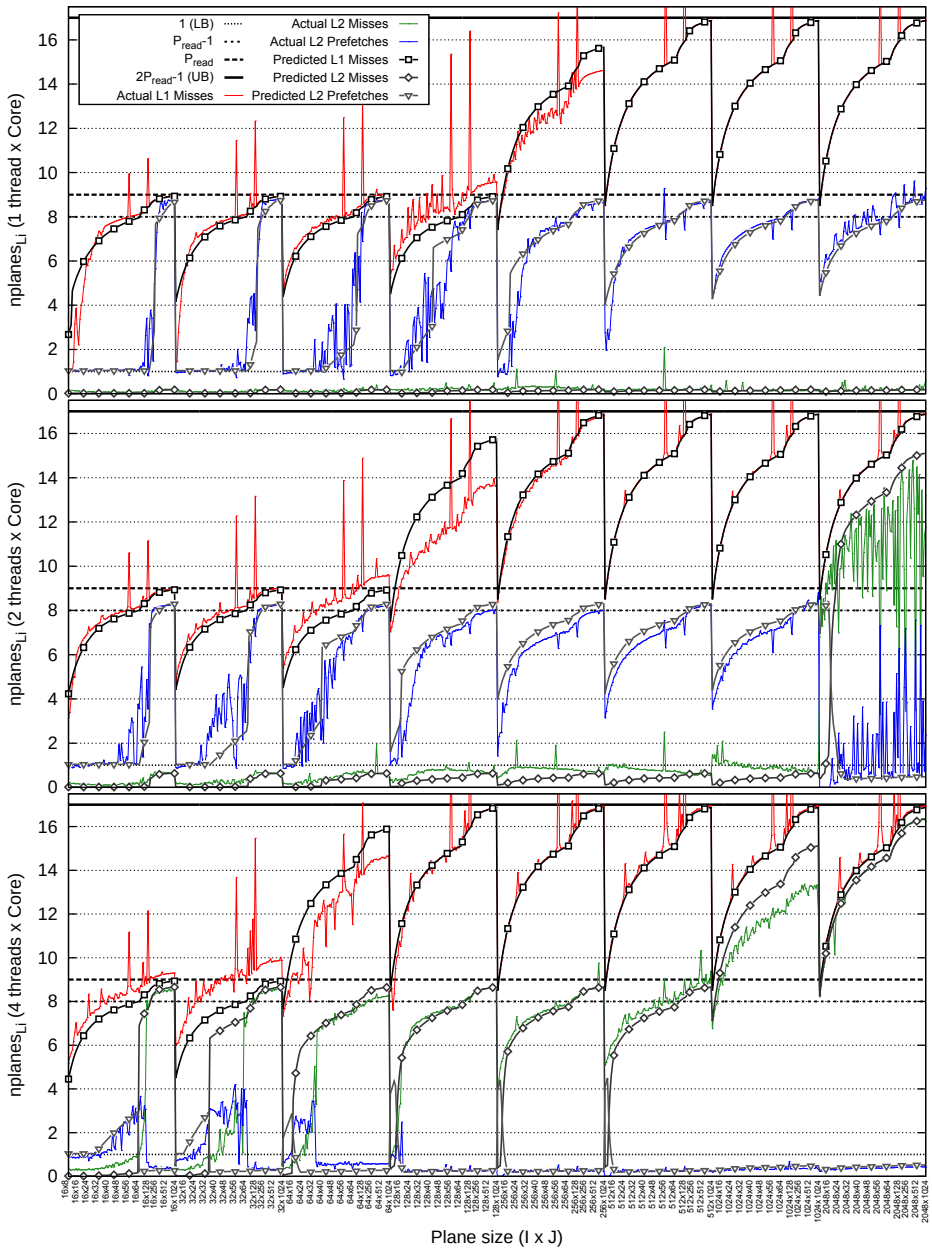

Fig. 5: Actual and predicted prefetched (inverted triangles) and non-prefetched (squares and diamonds) cache-lines for the three SMT configurations. These results are for the Naive implementation of a medium-high order ($\ell = 4$) stencil.

rently on the same core, the more contention and struggle for shared resources occurs. Likewise, some spikes appear on account of *ping-pong* effect, where different planes and columns addresses fall in the same cache set. This effect is also exacerbated as more threads are pinned to the same core. However, this effect is not captured by our model because it would require a multi-level set-associative cache model, which is not covered yet in our model.

Comparing the empirical (hardware counters) versus the analytical results (model), it can be observed that the model accurately predicts the number of misses on both levels of cache hierarchies, including those reads that are prefetched. However, some slight mispredictions appear on specific sizes when the transition between miss cases is triggered. Deciding a discrete point ($I \times J$) for transitions is difficult, and it might depend on other parameters apart from those considered in this work. Nevertheless, we think that our rules ($R_{1,2,3,4}$) have approximated these transitions fairly well. It is also important to mention the prediction of the L2 prefetching engine, especially in the late executions for 2 threads and in the early ones for 4 threads per core configurations. As hardware metrics show, on these cases the prefetching effect starts disrupting the results due to contention. Nonetheless, the predicted results follow the trend of both type of misses properly as a result of the $DC_{effectiveness}$ parameter.

The model accuracy is verified in Figure 6, which shows a summary of three types of execution times: actual, projected and predicted. The actual times were obtained using the CPU clock cycles metric (CPU_CLK_UNHALTED). On the other hand, the projected times were computed with the aggregated time of T_{L1}, T_{L2}, T_{Mem} and T_{Write} by using actual hardware counters of reads, writes and misses with their respective bandwidth parameters (STREAM2 characterization). Finally, the predicted times follow the same idea than the projected but using the estimations of our model instead of the instrumented ones. The purpose of the projected time is that it verifies the aggregated equation and calibrates the bandwidth parameters at each cache level. Therefore, it plays an important role ensuring that predicted times are a faithful representation of an actual execution.

Comparing the execution times shown in Figure 6, we observe that the predicted relative error (right axis) is very low on most of the cases. However, as the results reveal, some predictions have a high error (2 threads per core). Reviewing the cache miss predictions (not shown here), this is due to a late deactivation of the L2 prefetching engine, misleading the aggregated predicted time. Once the prefetching efficiency is again correctly predicted, the relative error drops considerably under 10%. Equally, some actual executions also present peaks due to the *ping-pong* effect. Projected times clearly follow this instabilities because their mirroring on cache misses. On the contrary, our model can not mimic such situations, and therefore the relative error increases considerably on those cases.

Results considering stencil optimizations such as Semi-stencil and spatial blocking are shown in Figure 7. In this test, 88 different tiling sizes were compared. The TP and LAP parameters used for the model were set to 3 and 5 cache-lines respectively. These values were obtained empirically using the

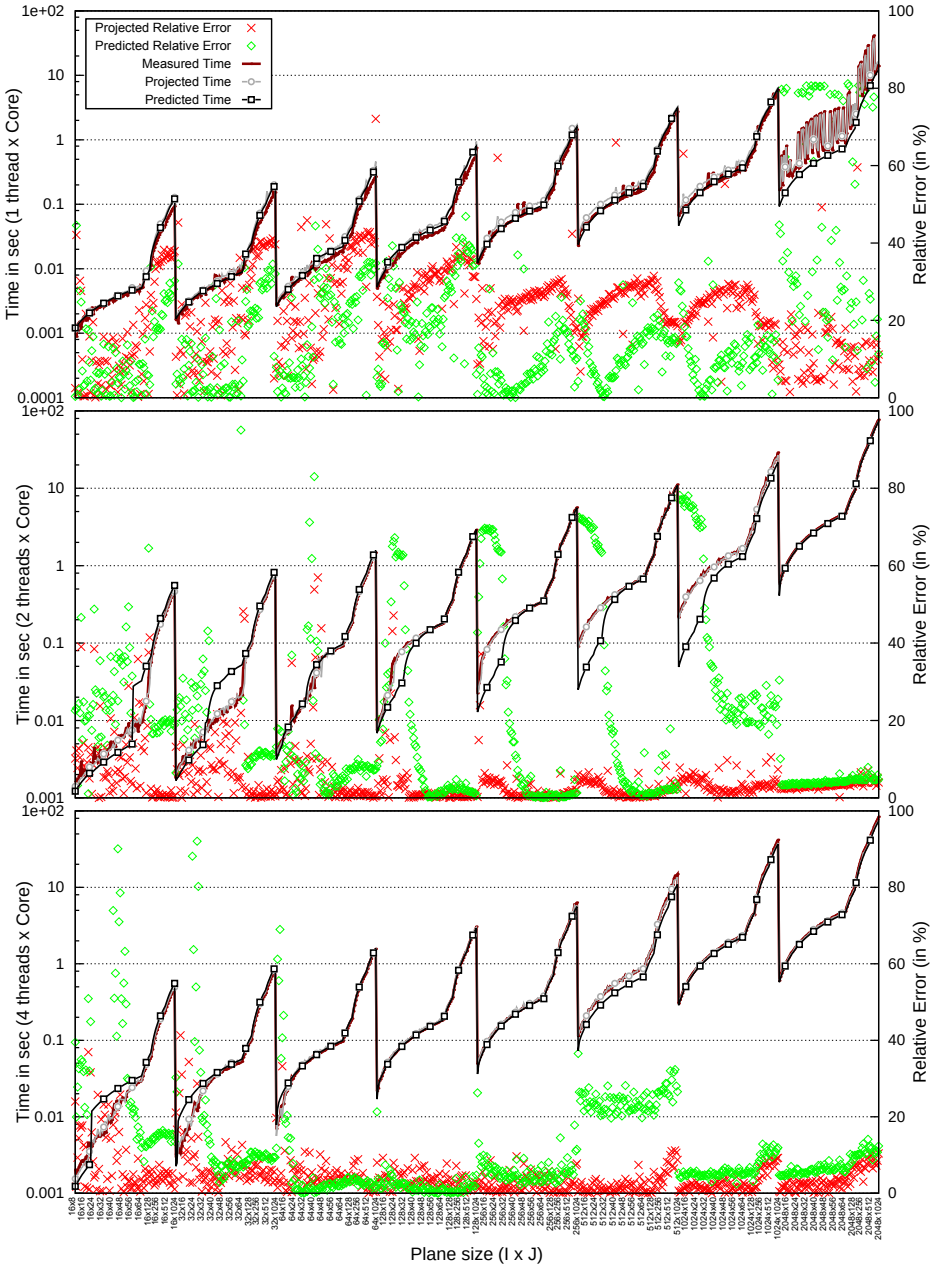

Fig. 6: Left axis: actual (solid line), projected (circles) and predicted (squares) execution times for the three SMT configurations. Right axis: relative errors compared with actual times. These results are for a high order ($\ell = 7$) Naive stencil.

prefetching benchmark as explained in Section 6. As shown in Figure 7, the model clearly estimates the different valleys (local minima) that appear when searching for the best tiling parameters due to the disruption of prefetched data and the increase of cache-line misses. The model is even able to suggest some good parameter candidates. For instance, taking a look to the *Naive+Blocking* results, the model successfully predicts the best tiling parameter for 1 and 2 threads per core configurations (512×16 and 512×8 respectively). This is not the case when running 4 threads per core. However, in this latter case, the actual best parameter is given as third candidate (512×8). On the other hand, reviewing the *Semi+Blocking* results, despite of some mispredictions especially for 4 threads per core, most of the local minima areas are well predicted.

Additionally, the model can reveal other insightful hints regarding the efficiency in SMT executions. It can help to decide the best SMT configuration to be conducted in terms of core efficiency. Let τ^{SMT_i} be the execution time for a SMT_i configuration of n different combinations, we define the *core efficiency* as

$$Core^{SMT_i}_{efficiency} = \frac{\min(\tau^{SMT_1}, \ldots, \tau^{SMT_n})}{\tau^{SMT_i}} \in [0, 1] \ , \tag{11}$$

where a *core efficiency* of 1 represents the best performance-wise SMT configuration for a set of specific stencil parameters (ℓ, $I \times J$ plane size, spatial blocking, Semi-stencil, etc.) and a given architecture. Therefore, the desirable decision would be to run the stencil code using the SMT_i configuration that maximizes the *core efficiency*. Normalizing our experiments for all three SMT combinations on a Naive stencil ($\ell = 4$) the Figure 8 is obtained. Note that depending on the problem size, the best SMT configuration ranges from 4 threads for small sizes to 2 threads for medium sizes and just only 1 thread per core for very large problems. The factor leading to this behavior is the contention of shared resources, especially the prefetching engine.

8 Conclusions and Analysis

This paper presents a thorough methodology to evaluate and predict stencil codes performance on complex HPC architectures. We have included several new features in our model such as: multi- many-core support, better hardware prefetching modeling, cache interference due to conflict and capacity misses and other optimization techniques such as spatial blocking and Semi-stencil. The aim of this work was to develop a performance model with minimal architectural parameter dependency (flexible) and at the same time reporting accurate results (reliable). In this regard, we have obtained fairly good prediction results, where the average error for most relevant cases floats between 5-15%. All these results factored in cache's associativities, TLB page size or complex prefetching engine specifications, but are not explicitly modeled.

Our proposed methodology also helps to unveil insights about how stencil codes might be built or executed in order to leverage prefetching efficiency. The

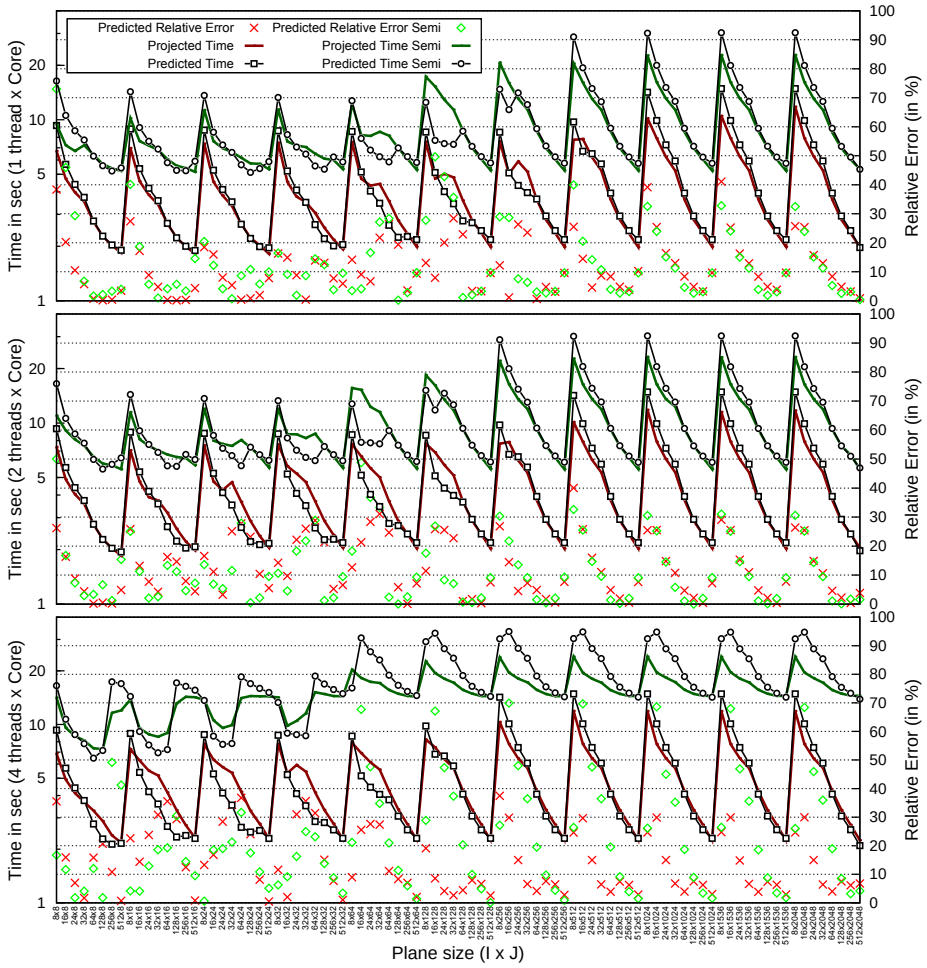

Fig. 7: Left axis: projected (solid line) and predicted (squares and circles) execution times for spatial blocking results. Right axis: relative errors compared with projected times. Results shown are for Naive ($\ell = 1$) and for Semi-stencil ($\ell = 4$).

prefetching modeling is not straightforward, especially when too many arrays are accessed concurrently, which overwhelm the hardware prefetching system and hamper the bandwidth performance. Furthermore, an aggressive prefetching intervention may also cause eviction of data that could have been reused later (temporal reuse), polluting the cache and affecting adversely the bandwidth performance. Loop fission and data layout transformations can occasionally improve the performance in these cases. Nevertheless, they must be applied carefully because some side effects may appear. In order to effectively capture the stream engine behavior in all above mentioned cases, the *prefetching effectiveness* approach has been adopted. As shown in the experiments, this approach can be

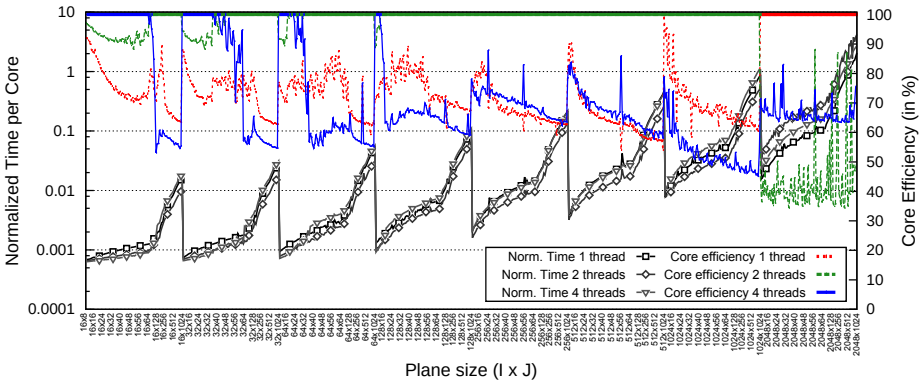

Fig. 8: Core efficiency for all three SMT combinations using a Naive stencil ($\ell = 4$).

successfully used in SMT context, where the prefetching efficiency is substantially reduced due to contention of the shared resources.

The proposed model could be included as static analysis in auto-tuning frameworks to guide making decisions about algorithmic parameters for stencil codes. Likewise, our model might be useful in expert systems, not only for compilers or auto-tuning tools, but also in run-time optimizations for dynamic analysis. For instance, the model might decide the SMT configuration and the number of threads to spawn per processor that outperforms the remaining combinations based on the prefetching engines, the problem size ($I \times J$) and the stencil order (ℓ).

To our knowledge, this is the first stencil model that takes into account two important phenomena: the cache interference (due to $II \times JJ$ and P_{read} parameters) and the prefetching effectiveness when concurrent threads are running in the same core. Despite the current work has been only conducted for 1$^{\text{st}}$ order in time and constant coefficient stencils, the model could be adapted to higher orders in time and variable coefficients (anisotropic medium) by adjusting the cost of cache miss cases ($C_{1,2,3,4}$) and their rules ($R_{1,2,3,4}$) through $P_{read,write}$ and $S_{read,write}$ variables.

Future work will include temporal blocking as optimization method, and different thread domain decomposition strategies apart from the static scheduling. Nonetheless, addition of software prefetching behavior into the model is unattainable since it depends on the internal compiler heuristics and the pragmas inserted by the user.

References

1. M. Araya-Polo, F. Rubio, M. Hanzich, R. de la Cruz, J. M. Cela, and D. P. Scarpazza. 3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors. *Scientific Programming: Special Issue on the Cell Processor*, 17:185–198, January 2008.

2. Axel Brandenburg. *Computational aspects of astrophysical MHD and turbulence*, volume 9. London: Taylor & Francis, 2003.
3. Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.
4. Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 51(1):129–159, 2009.
5. Raúl de la Cruz and Mauricio Araya-Polo. Towards a multi-level cache performance model for 3D stencil computation. In *Proceedings of the International Conference on Computational Science, ICCS 2011, Singapore*, volume 4 of *Procedia Computer Science*, pages 2146–2155. Elsevier, 2011.
6. Raúl de la Cruz and Mauricio Araya-Polo. Algorithm 942: Semi-stencil. *ACM Transactions on Mathematical Software*, 40(3):23:1–23:39, April 2014.
7. Jianbin Fang, Ana Lucia Varbanescu, Henk J. Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. An empirical study of intel xeon phi. *CoRR*, abs/1310.5842, 2013.
8. C. De Groot-Hedlin. A finite difference solution to the Helmholtz equation in a radially symmetric waveguide: Application to near-source scattering in ocean acoustics. *Journal of Computational Acoustics*, 16:447–464, 2008.
9. John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Efficient analytical modelling of multi-level set-associative caches. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, HPCN Europe '99, pages 473–482, London, UK, UK, 1999. Springer-Verlag.
10. Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12, April 2010.
11. Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory System Performance and Correctness*, pages 51–60, New York, NY, USA, 2006. ACM.
12. Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP '05: Proceedings of the 2005 workshop on Memory System Performance*, pages 36–43, New York, NY, USA, 2005. ACM Press.
13. Jean Kormann, Pedro Cobo, and Andres Prieto. Perfectly matched layers for modelling seismic oceanography experiments. *Journal of Sound and Vibration*, 317(1-2):354 – 365, 2008.
14. Gabriel Marin, Collin McCurdy, and Jeffrey S. Vetter. Diagnosis and optimization of application prefetching performance. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 303–312, New York, NY, USA, 2013. ACM.
15. John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.
16. Collin McCurdy, Gabriel Marin, and Jeffrey S Vetter. Characterizing the impact of prefetching on scientific application performance. In *International Workshop on*

Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS13), Denver, CO, 2013.
17. Sanyam Mehta, Zhenman Fang, Antonia Zhai, and Pen-Chung Yew. Multi-stage coordinated prefetching for present-day processors. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 73–82, New York, NY, USA, 2014. ACM.
18. Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Technical Report UCB/CSD-04-1335, EECS Department, University of California, Berkeley, 2004.
19. Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 30:1–30:10, New York, NY, USA, 2011. ACM.
20. Aditi Ray, G. Kondayya, and S. V. G. Menon. Developing a finite difference time domain parallel code for nuclear electromagnetic field simulation. *IEEE Transaction on Antennas and Propagation*, 54:1192–1199, April 2006.
21. Gabriel Rivera and Chau Wen Tseng. Tiling optimizations for 3D scientific computations. In *Proc. ACM/IEEE Supercomputing Conference (SC 2000)*, page 32, Washington, DC, USA, November 2000. IEEE Computer Society.
22. Robert Strzodka, Mohammed Shaheen, and Dawid Pajak. Impact of system and cache bandwidth on stencil computation across multiple processor generations. In *Proc. Workshop on Applications for Multi- and Many-Core Processors (A4MMC) at ISCA 2011*, June 2011.
23. O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 261–271, New York, NY, USA, 1994. ACM.
24. Jan Treibig and Georg Hager. Introducing a performance model for bandwidth-limited loop kernels. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics*, volume 6067 of *PPAM'09*, pages 615–624. Springer-Verlag, 2009.
25. Samuel Webb Williams, Andrew Waterman, and David A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008.