MAPredict: Static Analysis Driven Memory Access Prediction Framework for Modern CPUs

No Author Given

Abstract. Application memory access patterns play a vital role in deciding how much traffic is served by the cache and forwarded to the DRAM. Moreover, prefetchers, compilers, parallel execution, and innovations in manufacturer-specific micro-architectures make prediction more difficult. This research introduces MAPredict, a static analysis-driven framework that addresses these challenges to predict LLC-DRAM traffic. By exploring and analyzing the behavior of modern Intel processors, MAPredict formulates cache-aware analytical models. MAPredict invokes these models to predict LLC-DRAM traffic by combining the application model, the machine model, and user-provided hints for capturing dynamic information. MAPredict successfully predicts LLC-DRAM traffic for different regular access patterns and provides the means to combine static and empirical observations for irregular access patterns. Evaluating 130 workloads from six applications on recent Intel micro-architectures, MAPredict yielded average accuracy of 99% for streaming, 91% for strided, and 92% for stencil patterns. By coupling static and empirical methods, up to 97% average accuracy is obtained on different micro-architectures for random access patterns.

1 Introduction

Innovations in computing in recent times are shaped by the end of Dennard scaling and the need to address the memory wall problem. As a result, multi-core and many-core processors with multi-level memory hierarchies on heterogeneous systems are becoming increasingly common [9]. With increasing hardware complexity, designing analytical models becomes a non-trivial task. With the rise of heterogeneous systems, the importance of such a modeling approach for prediction has increased significantly. Because the execution of an application on an ill-suited processor may lead to non-optimal performance [18], the runtime system needs to make a quick decision on where to execute kernels on the fly. Predicting a kernel's performance and energy consumption can enable runtime systems to make intelligent decisions. For such prediction, floating-point operations (FLOPs) and memory traffic need to be counted. While calculating FLOPs is fairly straightforward, memory traffic prediction is complex and dynamic since the memory access request can be served by the cache or the DRAM.

Statically predicting LLC-DRAM traffic is vital for three reasons. Firstly, a heterogeneous runtime system can make intelligent scheduling decisions if it can statically identify compute and memory-bound kernels based on the Roofline model [30]. A study, Mephesto [18], demonstrated that energy-performance-aware scheduling decisions can be made based on operational intensity (FLOPs/LLC-DRAM Bytes) of kernels. There are tools such as Intel Advisor and NVIDIA

Nsight Compute that generate operational intensity of a kernel. However, a runtime system needs this information *before* executing the kernel to make better placement decisions. While static analysis tools can provide the FLOP count at compile time [12], statically predicting LLC-DRAM traffic needs to be explored. Simulation frameworks can provide LLC-DRAM traffic, but they are not fast enough to be integrated into a runtime system [20]. Secondly, developing a framework to predict the energy and performance of modern CPUs requires predicting LLC-DRAM memory transactions because LLC-DRAM transactions incur a significant amount of energy and time [29]. Finally, a static approach for predicting the LLC-DRAM traffic enables simulation-based design space exploration to determine the best memory configuration. For these reasons, this study aims to build a framework capable of predicting the LLC-DRAM traffic statically. However, a static analysis approach for predicting the LLC-DRAM traffic encounters three major challenges: 1) it must keep up with the continuous innovation in the processors' memory hierarchy,2) it must deal with the complex memory access patterns and different execution models (sequential and parallel), and 3) it does not have access to the dynamic information necessary to obtain high prediction accuracy.

This research presents MAPredict, a framework that predicts the LLC-DRAM traffic for applications in modern CPUs. To the best of our knowledge, this is the first framework that *simultaneously* addresses all of the challenges above. We present systematic experimentation on different Intel micro-architectures to elicit their memory subsystem behavior and build the analytical model for a range of memory access patterns. Through static analysis at compile-time, MAPredict creates Aspen [25] application models from annotated source code, captures the dynamic information, and identifies the memory access patterns. It then couples the application and machine model to accurately predict the LLC-DRAM traffic.

Our study reports the following contributions:

- A systematic unveiling of the behavior of Intel CPUs for read and write strategies, accounting for prefetchers, and compilers, and multi-threaded executions;
- A formulation of a cache- and prefetching-aware analytical model using application, machine, and compiler features;
- A static analysis driven framework named MAPredict to predict LLC-DRAM traffic at compile time by source code analysis, dynamic information, and analytical modeling; and
- An evaluation of the MAPredict using 130 workloads (summation of number_of_functions * input_sizes) from six benchmarks in four micro-architectures of Intel, where we achieve higher prediction accuracy for regular access patterns when compared to the models from literature. MAPredict also provides means to combine static and empirical observation for irregular access.

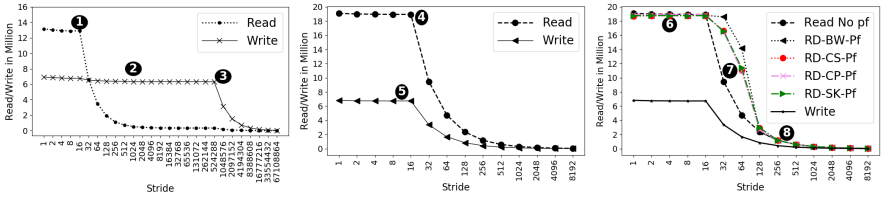
2 Understanding memory reads and writes in Intel processors

This section introduces the hardware, LLC-DRAM traffic measurement strategy, and the factors that trigger an LLC-DRAM transaction. From the application's

Table 1: Machines and micro-architectures.

Name	Year	Processor detail. here L3 = LLC
Broadwell	2016	Xeon E5-2683 v4, 32 cores, L2 - 256 KiB, L3 - 40 MiB
Skylake	2017	Xeon Silver 4114, 20 cores, L2 - 1 MiB, L3 - 14 MiB
Cascade Lake	2019	Xeon Gold 6248, 40 cores, 2 - 1 MiB, L3 - 28 MiB
Cooper Lake	2020	Xeon Gold 6348H, 96 cores, L2 - 1 MiB, L3 - 132 MiB

viewpoint, the memory access pattern plays a vital role. The two most common memory access patterns — sequential streaming access and strided memory access are considered. Cache line size, page size, initialization, and prefetching mechanism are identified as the key factors. This section also explores the effects of the evolution of CPU micro-architectures.



(a) Cache line and page size. (b) Array initialization. (c) Prefetchers.

Fig. 1: LLC-DRAM traffic for different read and write scenarios in Intel processors. LLC-DRAM traffic is shown at Y-axis.

2.1 Description of the hardware

In this study, Intel CPUs are considered because they are the most widely available processors in HPC facilities [1]. Table 1 depicts the four recent micro-architectures that are a part of this study — Broadwell (BW), Skylake (SK), Cascade Lake (CS), and Cooper Lake (CP). The introduction of the non-inclusive victim L3 cache and the larger L2 cache (starting from the SK processors) is the most important change concerning the memory subsystem [2].

2.2 A tool for measuring the LLC-DRAM traffic

All the LLC-DRAM traffic measurements reported in this study (all sections) are gathered through a script-based dynamic analysis tool. The dynamic analysis tool uses TAU [24] and PAPI [26] to measure function-wise LLC-DRAM traffic from uncore counters (imcX::UNC_M_CAS_COUNT) of the integrated memory controllers (IMC). This tool provides LLC-DRAM traffic measurement in the unit of cache line (64 bytes) and this unit is followed throughout this study.

2.3 Different read and write strategies

To investigate the application-cache interplay, a variant of vector multiplication code that exhibits sequential streaming (stride = 1) and strided access pattern (stride > 1) is considered. The code has three arrays (100 million 32-bit floating-points). Since the cache line length of these Intel processors is 64 bytes, in an ideal case, an array size of 100M (M represents million) should generate 6.25M writes and 12.5M reads (two reads and one write per index). However, Fig. 1 tells a different story, which are discussed below.

Impact of cache line size In Fig. 1a, the read-write traffic is shown where the read traffic is close to 12.5M for stride 1. This trend continues until stride 16 (64 bytes/size_of_32bit_float=16), referenced by ❶. Because a cache line is 64 bytes long, while fetching one 32-bit floating-point data, the memory subsystem fetches 15 (60 bytes worth) additional neighboring data.After stride 16 at ❶, the read traffic halves every time the stride is doubled. Write traffic for stride 1 is also close to 6.25M. However, for the write traffic, it appears that the region ❷ stretches up to a stride of 524,288. For a stride of one (100M access) and a stride 524,288 (only 190 access), the same number of cache lines (6.25M) are transferred, which is explained below.

Impact of page size In Fig. 1a, the impact of page zeroing is visible because of uninitialized write array. The write traffic in Fig. 1a is not affected by the cache line size. Instead, it depends on the page size. The default page size on Intel processors is 4 KiB, i.e., a stride of 1024 for 32-bit floating-point. Because Linux supports "transparent huge pages", it allows larger page sizes. Intel processors support large pages of 2 MiB and 1GiB size. Because the data structure size in our study was 100M, a page size of 2MiB was selected. This explains why we see a transition at ❸ on a stride of 524,288.

Impact of Initialization In Fig. 1b traffic is shown when the write array is initialized. The write traffic is close to 6.25M at stride 1, and at this point, the impact of the cache line size is visible at ❺. Specifically, until a stride of 16, no page zeroing takes place. After a stride of 16, the traffic is reduced by half when the stride is doubled. However, the read traffic is close to 18.75M for stride 1, indicating the impact of "allocating store" (the region pointed by ❹).

Impact of hardware prefetching Intel implements aggressive prefetching, but not all the details are openly available to the community. In the experimental results shown in Fig. 1a and Fig. 1b, prefetching is disabled(BW-Pf means Broadwell with prefetching). Three regions in read traffic are shown in Fig. 1c. The regions ❻ (stride 1 to 16) and ❽ (stride 128 and onward) show no visible difference with prefetching enabled. Further investigation by experimenting with a smaller stride confirms that the impact of prefetching vanishes after a stride of 80 pointed (by ❽). The region ❼ (from a stride of 32 to a stride of 128) shows extra cache lines that are fetched. Because of Intel's prefetchers, for a stride of 64, each access could result in three cache lines being fetched. Moreover, the prefetching behavior in region ❼ is not the same for all micro-architectures. Read traffic is 10% higher in BW when compared to others (SK, CS, and CP show the same level of read traffic). This observation can potentially be attributed to the change in the cache subsystem design following the BW micro-architecture.

Impact of compiler and multi-threaded execution The GNU compiler is used to generate Fig. 1. However, using the Intel compiler can provide a different result because of the default "streaming store" or "non-temporal store" for a stride of 1. For streaming store, data is not read from the DRAM for a store miss. Instead, the data is directly written to DRAM (bypassing the cache) through a write-combining buffer. When experimenting with multi-threaded version, we found no difference when one thread and 16 thread executions are compared.

3 Modeling different types of access

A static analysis framework needs analytical models for different types of memory access patterns to predict the LLC-DRAM traffic. For this reason, this section builds on the findings from §2 to formulate analytical models for different access patterns. Three kinds of regular access patterns are discussed in this section. First, the model is formulated for the sequential streaming access pattern to predict LLC-DRAM cacheline transfer. Then, models are prepared for other access patterns by using the model for streaming access patterns. In the end, random access patterns (irregular) are discussed.

3.1 Sequential streaming access pattern

The sequential streaming access pattern (i.e., stride = 1) is one of the most common access patterns found in applications. Prefetching does not impact the amount of traffic transferred between LLC and DRAM for this pattern. However, the impact of the cache line and page size needs to be considered.

Read traffic Because the LLC-DRAM read transaction is done in a unit of cache lines, the amount of read traffic can be expressed using Eq 1. In Eq 1, a data structure size is $\text{Element}_{\text{count}}$ and the size of each element is $\text{Element}_{\text{size}}$ bytes. $\text{Read}_{\text{count}}$ is the number of LLC-DRAM transactions for reading a data structure. Data structure initialization has no impact on $\text{Read}_{\text{count}}$. Because alignment is not certain, the ceiling is considered.

$$\text{Read}_{\text{count}} = \left\lceil \frac{\text{Element}_{\text{count}} * \text{Element}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \right\rceil \tag{1}$$

Write traffic The initialization of the data structures plays an important role for write traffic. At first, the case where the data structure is not initialized but only memory is allocated is discussed. For such a case, the page size becomes the deciding factor because of page zeroing (as shown in §2.3). In Eq 2, $\text{Write}_{\text{not\_init}}$ is the number of cache line transfers when the data structure is not initialized). Because the machines in Table 1 support transparent huge pages by default, the page size picked by the operating system (OS) depends on the data structure size (We made no changes in the OS). The ceiling is considered to capture the extra traffic from the fragmented access on the last page.

When a data structure is initialized, page zeroing does not take place, and the cache line becomes the deciding factor and the write-allocate policy is used. So one write operation also causes one read operation. The write traffic ($\text{Write}_{\text{init}}$) is shown in Eq 3. The extra read traffic ($\text{Read}_{\text{for\_write}}$) generated for the write operation is shown in Eq 4.

$$\text{Write}_{\text{not\_init}} = \left\lceil \frac{\text{Element}_{\text{count}} * \text{Element}_{\text{size}}}{\text{Page}_{\text{size}}} \right\rceil * \frac{\text{Page}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \tag{2}$$

$$\text{Write}_{\text{init}} = \left\lceil \frac{\text{Element}_{\text{count}} * \text{Element}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \right\rceil \tag{3}$$

$$\text{Read}_{\text{for\_write}} = \begin{cases} 0 & \text{if data structure is not initialized} \\ \text{Write}_{\text{init}} & \text{if data structure is initialized} \end{cases} \tag{4}$$

Thus, total read traffic for streaming access, $\text{Read}_{\text{stream}} = \text{Read}_{\text{count}} + \text{Read}_{\text{for\_write}}$ and total write traffic for streaming access, $\text{Write}_{\text{stream}} = \text{Write}_{\text{not\_init}}$ or $\text{Write}_{\text{init}}$ based on data structure initialization. Since streaming store operations do not cause extra read traffic for initialized write data structure, $\text{Read}_{\text{for\_write}}$ is set to zero when Intel compilers are used.

3.2 Strided access pattern

The strided access pattern is another common pattern. Based on the observation in Fig. 1c, there are three regions. Read and write traffic formulation for each region is presented below.

Streaming region When the $(\text{Stride} * \text{Element}_{\text{size}})$ is smaller than the $\text{Cacheline}_{\text{size}}$, both reads and writes are the same as streaming access (region ❻ in Fig. 1c). In this region (stride 1 to 16), read and write traffic are same as streaming access because the whole cache line is transferred. For this reason, total read and write traffic for this region is presented by $\text{Read}_{\text{stream}}$ and $\text{Write}_{\text{stream}}$.

No prefetching region As discussed in §2.3, the impact of prefetching vanishes after stride 80, and hence, this is the starting point of a "no prefetching" region which is pointed by ❽ in Fig. 1c. For this reason, when $(\text{Stride} * \text{Element}_{\text{size}})$ is larger than $(5 * \text{Cacheline}_{\text{size}})$, no prefetching region is considered since $(5 * \text{Cacheline}_{\text{size}})$ = stride 80 for 32-bit floating-point.

At first, write traffic is considered. If the data structure is initialized, the write traffic is decided by the cache line size and stride size. It also causes extra read traffic. This case is expressed in Eq 5.

$$\text{Write}_{\text{init}} \text{ or } \text{Read}_{\text{for\_write}} = \text{Write}_{\text{stream}} / \left(\frac{\text{Stride} * \text{Element}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \right) \tag{5}$$

If the data structure is not initialized, the write traffic is decided by the $\text{Page}_{\text{size}}$. If $(\text{Stride} * \text{Element}_{\text{size}}) > \text{Page}_{\text{size}}$ then Eq 6 expresses write traffic, otherwise write traffic is equal to $\text{Write}_{\text{stream}}$. Read traffic is expressed as Eq 7.

$$\text{Write}_{\text{non\_init}} = \text{Write}_{\text{stream}} / \left(\frac{\text{Stride} * \text{Element}_{\text{size}}}{\text{Page}_{\text{size}}} \right) \tag{6}$$

$$\text{Read}_{\text{count}} = \text{Read}_{\text{stream}} / \left(\frac{\text{Stride} * \text{Element}_{\text{size}}}{\text{Cacheline}_{\text{size}}} \right) \tag{7}$$

Prefetching zone Only when $(\text{Stride} * \text{Element}_{\text{size}})$ is larger than the cache line and smaller than five times of the cache line, the impact of prefetching becomes visible (denoted by region ❼ which starts from stride 16 and ends at stride 80 in Fig. 1c). In this region, if prefetching is disabled, write and read traffic can be expressed as Eq 5, Eq 6, and Eq 7. However, the main difference is observed when prefetching is enabled, and in that case, only read traffic is impacted. Intel prefetching suggests fetching an adjacent cache line and an additional cache line if all MSR bits are set. For this reason, the number of data access is multiplied by three in the prefetching zone. This is expressed in Eq 8. Since prefetching has no impact on write traffic, the write traffic is expressed as the non-prefetching formula given at Eq 5 and Eq 6.

$$\text{Read}_{\text{count}} = 3 * \left(\frac{\text{Element}_{\text{count}}}{\text{Stride}} \right) \tag{8}$$

Moreover, SK, CS, and CP show a 10% read traffic drop when compared to BW (from Fig. 1c), which is considered in the model.

3.3 Stencil access pattern

Stencil access patterns are also common in scientific applications. The write operation in a stencil access pattern usually follows a sequential streaming pattern, and hence, the equations for streaming access are followed. However, read operations need to be considered for different dimensions.

One-dimensional stencil In a one-dimensional stencil pattern, usually consecutive elements are accessed in each operation. Since adjacent elements can be served by cache, the read operations follows a sequential streaming pattern.

Two and three-dimensional stencil Like a one-dimensional stencil, if the elements are adjacent, a streaming access pattern is followed. When the distance is larger than cache line size, individual accesses are counted. However, if the distance between stencil points is high for a large data set, the cache size becomes a limiting factor by causing capacity misses. Old data may need to be brought to the cache more frequently for a large two or three-dimensional when there are multiple iterations.

3.4 Random access pattern and empirical factor

The random access pattern is found in applications with irregular access [20]. Moreover, modern CPUs introduce randomness in data reuse because of their replacement policies and the cache can not retain all data for further use. Therefore, LLC-DRAM traffic prediction for random access must consider the randomness derived from applications and machines. The number of total access in irregular cases is expressed by $\text{Access}_{\text{random}}$. We first discuss randomness in applications, followed by a discussion of randomness derived from machines.

Data structure randomness In data structure randomness, the reuse behavior becomes uncertain because of how the data structures are accessed, e.g., A[B[i]] (A's memory access can be random). In this case, the randomness is one-dimensional since only the location of access is random, and the total number of access, $\text{Access}_{\text{random}}$ is known. In such cases, cache reuse is non-deterministic at compile time because the access depends on another data structure at runtime. Furthermore, prefetchers may fetch some extra cache lines, which adds more uncertainty. So, machine randomness needs to be considered for this case.

Algorithmic randomness The worst case of randomness is algorithmic randomness which has two dimensions, 1) randomness in the number of total access, $\text{Access}_{\text{random}}$ and 2) randomness in which locations are accessed. While the first randomness depends on the data structure size, the second kind may introduce data reuse in the cache. Complex branching usually exists in this kind of randomness. An example of algorithmic randomness is searching algorithms, such as binary search. For such cases, algorithmic complexity analysis provides an upper-bound of memory access on a data structure and is considered to define $\text{Access}_{\text{random}}$. The second dimension is captured through machine randomness.

Machine randomness and empirical factor Machine randomness depends on cache size, replacement policies, and memory access location. In recent Intel processors (Since SK), replacement policies are dynamically selected from a set

of policies at runtime, and the policy is chosen for a given scenario is not disclosed [2]. Moreover, in the cases of algorithmic and data structure randomness, the location of access is random. So, multiple dimensions of randomness from the machine and the application make statically determining the LLC-DRAM traffic a complex problem. Moreover, the undisclosed mapping of dynamic replacement policies from Intel makes it further complicated. To the best of our knowledge, statically determining LLC-DRAM traffic in modern CPUs for irregular cases is an unsolved problem. This study does not claim to solve this problem statically; rather, it combines static analysis and empirical observation. At this point, an empirically obtained $\text{Empirical}_{\text{factor}}$ is introduced to represent machine randomness. The $\text{Empirical}_{\text{factor}}$ is calculated from memory access obtained from the dynamic analysis tool (§2.2) and statically obtained total access ($\text{Access}_{\text{random}}$) where $\text{Empirical}_{\text{factor}}$ = measured_access / statically_obtained_access. This ratio captures the randomness of the application and the underlying machine.

4 MAPredict Framework

This section describes the MAPredict framework. MAPredict statically gathers information from an application and a machine to invoke the appropriate model presented in §3 and generates a prediction for LLC-DRAM traffic. MAPredict depends on OpenARC compiler [13] for static analysis of the code and the COMPASS [12] framework for expressing an application in the Aspen [25] domain-specific modeling language. First, an overview of OpenARC, Aspen, and COMPASS is presented. After, a description of MAPredict framework is provided.

4.1 Aspen, OpenARC, and COMPASS

Aspen (Abstract Scalable Performance Engineering Notation) [25] is a domain-specific language for analytical performance modeling in a structured fashion. Aspen's formal language and methodology provide a way to express applications and machines' characteristics abstractly (e.g., Aspen application model and machine model). Aspen tools can provide various predictions, such as predicting resource counts (e.g., number of loads, stores, FLOPs, etc.). Open Accelerator Research Compiler (OpenARC) [13] is an open-source compiler framework for various directive-based programming research. It provides source-to-source translation, a desired feature to create Aspen application models. COMPASS [12] is an Aspen-based performance modeling and prediction framework, which is built on OpenARC. COMPASS provides a set of Aspen directives (pragma-based) that can be used in source code. MAPredict extends COMPASS by adding new Aspen directives for enabling cache-aware memory access prediction.

4.2 Description of MAPredict Framework

The workflow of the MAPredict framework is shown in Fig. 2. Four phases of MAPredict are described below.

Source code preparation phase The main idea of MAPredict is to prepare a source in such a way that when the preparation is done, MAPredict can statically provide memory access prediction. This one-time effort of source code

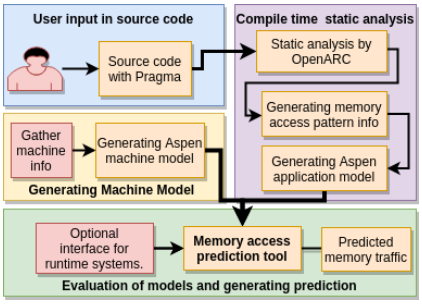


Fig. 2: Workflow of MAPredict framework.

preparation (i.e., phase 1) is necessary to capture the dynamic information unavailable at compile time. First, COMPASS-provided Aspen compiler directives (i.e., pragmas) are used to identify the target model region in the code for capturing information at compile time. MAPredict introduces new traits that need to be included in the directives to specify memory access patterns where necessary. Access pattern traits such as sequential streaming and strided access patterns are automatically generated; however, user input (through pragmas) is needed for stencil and random access patterns. User inputs are also needed for specifying dynamic (e.g., malloc) input sizes of data structures and $\text{Empirical}_{\text{factor}}$ for random access patterns because of their unavailability at compile time.

```
model matmul {
param N = 512
data a [((4*N)*N)]
kernel Matmul_openmp {
  execute [N] "block_Matmul" {
    loads [((1*sizeof_float)*N)] from b as stride(1)
    loads [((1*sizeof_float)*N)] from c as stride(N)
    stores [(1*sizeof_float)*N] to a as stride(1)
}}}
```

Listing 1.1: Application model - Matrix Multiply (partial view)

Compile-time static analysis phase In phase 2, MAPredict gathers application information that is required to execute the model presented in §3. MAPredict invokes OpenARC's compile-time static analysis capability, which generates an intermediate representation of the code and captures variables, variable sizes (i.e.,$\text{Element}_{\text{size}}$), instruction types (load or store), FLOPs, loop information, access pattern information, machine-specific $\text{Empirical}_{\text{factor}}$, etc., from source code. After gathering the needed information, the source-to-source translation feature of OpenARC is invoked to generate the Aspen language's abstract application model by following Aspen's grammar [25]. An application model combines different types of statements in a graph of kernels with one or more execution blocks. An example of an application model is given in Listing 1.1 which shows load and store information of matrix multiplication. Every load and store statement shows the access pattern of that data structure.

Machine model generation phase The machine model is manually prepared by gathering information about the machine, following the Aspen grammar (one

time effort). The machine model contains information unavailable in the application model, and is required to execute the model presented in §3. MAPredict gathers information about the micro-architecture, $\text{Cacheline}_{\text{size}}$, $\text{Page}_{\text{size}}$, prefetching status, compiler, etc., from the machine model.

Prediction generation phase MAPredict's prediction engine is invoked by passing the application and machine model. MAPredict invocation can also be made from a runtime system using the optional runtime invocation feature of COMPASS. When MAPredict is invoked, it traverses the call graph of the Aspen application model in a depth-first manner. In this graph, each node represents an execution block (a part of a function). MAPredict walks through every load and store statement of the application model, collects the access pattern, and evaluates the expression to obtain $\text{Element}_{\text{count}}$, $\text{Element}_{\text{size}}$, Stride, etc. Then, MAPredict uses the machine model information to invoke the appropriate prediction model to generate memory access prediction for that statement. MAPredict does this evaluation for each statement and generates a prediction for the execution block, which is recursively passed to make a kernel/function-wise prediction. When the graph traversal finishes, MAPredict provides a total memory access prediction for the application. MAPredict can provide kernel-wise memory access and execution block-wise memory access. In debug mode, it offers statement-wise detail analysis.

4.3 Identifying randomness and $\boldsymbol{Empirical_{factor}}$

MAPredict combines static and empirical approaches to address randomness. In a large codebase, identifying randomness is challenge because randomness usually exists only in certain functions. MAPredict facilitates identifying randomness in source code. At first, the source is annotated with basic MAPredict traits (without any $\text{Empirical}_{\text{factor}}$). When MAPredict is executed, it provides function-wise memory access prediction. Then the dynamic analysis tool is run on real hardware to get the same function-wise data. Comparing the results from both tools makes it apparent which functions provide low accuracy, indicating a potential source of randomness. However, a function can be large. MAPredict provides execution block-level and statement-wise detailed analysis to pin-point the randomness. After identifying, as described in §3.4, the $\text{Empirical}_{\text{factor}}$ is calculated by comparing the output from the dynamic analysis tool (measured value) and MAPredict (statically obtained value). Then the $\text{Empirical}_{\text{factor}}$ is annotated in the source code for that statement or execution block. When MAPredict is rerun, it uses the $\text{Empirical}_{\text{factor}}$ to generate the prediction.

5 Experimental Setup

The experiment environment is discussed in this section. Processors in Table 1 were used in the experiments. The operating system of these processors is Centos-7, and it supports transparent huge pages by default. The applications, along with their input sizes and access patterns, are listed in Table 2. Forty-four functions from these applications are evaluated for different input sizes, making it a total of 130 workloads. GCC-9.1 and Intel-19.1 compilers are used for experimentation. For parallel execution, the OpenMP programming model is used. In

Table 2: Benchmarks. Here, R = region.

Name	Pattern	Input sizes
STREAM Triad [16]	Sequential streaming access pattern	50M, 100M, 150M
Jacobi [13]	Stencil access pattern without initialization	67M, 268M, 1B
Laplace2D [13]	Stencil access pattern with initialization	16M, 64M, 100M
Vecmul for R - ❼ [13]	Strided pattern in prefetching zone	50M, 100M, 200M
Vecmul for R - ❽ [13]	Strided pattern in no prefetching zone	100M, 200M, 400M
XSBench [27]	Algorithmic randomness	large
Lulesh [10]	Mixed patterns	15M, 27M, 64M

the graphs, BW stands for Broadwell without prefetching, and BW_pf represents Broadwell with prefetch enabled. A similar convention is used for others. All the graphs in the experiment section show accuracy in Y-axis.

5.1 Accuracy calculation

Relative accuracy is considered, where accuracy = [100 - Absolute {(measured-predicted)/measured*100}]. The measured value is generated by the dynamic analysis tool described in §2.2. The predicted values are generated using MAPredict. Both MAPredict and the dynamic analysis tool provide function-wise traffic, making function-wise accuracy calculation possible.

5.2 Comparison with literature

Prediction accuracy of MAPredict is compared with a model from literature [31]. Even though this study [31] investigates data vulnerability, the main contribution is the analytical model for LLC-DRAM traffic prediction by considering application and machine characteristics. Two other studies investigate memory access prediction for static analysis [12, 20]. The main reason they are not considered for comparison is the lack of a detailed analytical model with equations. Moreover, one of these studies depends on cache simulation [20], while another depends on instruction counts without considering machine properties [12].

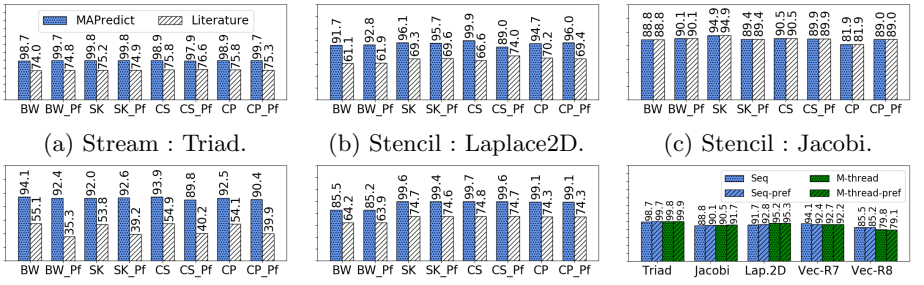
6 Experimental Results

In this section, the accuracy of the MAPredict framework is evaluated in two steps. In the first step, the prediction accuracy of different applications with regular memory access patterns is evaluated. In the second step, irregular access patterns and a large application with mixed access patterns are investigated.

6.1 Regular access patterns

Regular access patterns are investigated for various micro-architectures, input sizes, compilers, and execution models.

Sequential streaming access pattern To evaluate the model for sequential streaming memory access pattern, the triad kernel of STREAM [16] is used. The data structure is initialized, and the size is 50M 64 bit floating-points. The total traffic, which is the summation of read and write traffic, are measured for all the micro-architectures with prefetching disabled and enabled. The prediction accuracy from MAPredict and the model from the literature [31] are compared in Fig. 3a. MAPredict invoked Eq 3 and provided 99.1% and 99.1% average accuracy in all processors when prefetching is disabled and enabled. For the same cases, the model from literature provided 75.0%, and 75.4% average accuracy.



(a) Stream : Triad. (b) Stencil : Laplace2D. (c) Stencil : Jacobi.

(d) Strided : Vecmul_50. (e) Strided : Vecmul_200. (f) Single vs. multi-threaded.

Fig. 3: Accuracy comparison of different regular access patterns. Y-axis is accuracy, and X-axis is micro-architectures with prefetching disabled and enabled. BLUE=MAPredict, WHITE=literature, and GREEN=multi-threaded.

Stencil memory access pattern MAPredict's accuracy for stencil pattern is evaluated using two benchmark kernels, Laplace2d and Jacobi [13]. Both kernels have a 2D stencil access pattern with adjacent points. However, Laplace2D has the write array initialized, and Jacobi has the write array non-initialized. Laplace2D operates on a 4000×4000 matrix of 64-bit floating-points, whereas Jacobi operates on an 8912×8912 matrix of 32-bit floating-points. In Fig. 3b the comparison for Laplace2D is shown. Since the data structure is initialized, allocating-store causes extra read, which the model from literature does not consider. MAPredict provided 95.9% and 92.5% average accuracy when prefetching is disabled and enabled, respectively. However, the model from literature provided 65.7% and 68.5% average accuracy. The prediction accuracy of Jacobi is portrayed in Fig. 3c. Since the write data structure is non-initialized, page zeroing takes place. Even though the model in the literature did not consider page zeroing, the equation remained the same. Thus, the same accuracy is observed.

Strided memory access pattern To evaluate strided access patterns of prefetching and no-prefetching region (pointed by ❼ and ❽ in Fig. 1c), vector multiplication of 100M size is used with stride 50 and 200. The stride size 50 is used with a non-initialized write array to evaluate the page zeroing effect. Initialized write array is considered for stride 200. For the prefetching zone, traffic is significantly different across micro-architectures. Moreover, for stride 50, the whole array is written to the memory instead of one in fifty. Fig. 3d shows that MAPredict captured the prefetching differences between different micro-architectures successfully and provided 93.3% and 91.6% average accuracy when prefetching is disabled and enabled, respectively. However, the model from literature provided 54.6% and 38.2% average accuracy because it does not account for prefetchers and page-zeroing. For stride 200, the initialized data structure causes allocating-store. The comparison is shown in Fig. 3e where MAPredict and the model form literature provided 88.5% and 66.2% average accuracy, respectively.

Multi-threaded execution and impact of compiler Multi-threaded and single-threaded executions are compared in Fig. 3f. Eight threads of BW are used for experimentation, and OpenMP from GCC is used. No significant difference is observed for sequential streaming, stencil, and strided access patterns.

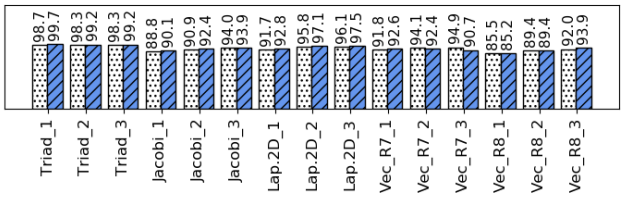


Fig. 4: Accuracy of various input sizes. WHITE = prefetching disabled and BLUE = prefetching enabled.

Moreover, MAPredict is capable of capturing the streaming store operation by Intel compiler and provides better accuracy than the model from literature. Other micro-architectures show a similar trend.

Comparison of different input sizes MAPredict's accuracy is evaluated for different input sizes for each application with regular access pattern given in Table 2. Triad is tested with array sizes of $50M$, $100M$, and $150M$. Matrix sizes for Jacobi are 8192×8192, and 16384×16384, and 32768×32768. Laplace2D is tested with 4000×4000, 8000×800, and 1000×1000 matrix sizes. Strided vector multiplication is tested with vector sizes of $50M$, $100M$, and $200M$ for prefetching region and $100M$, $200M$, and $400M$ for no prefetching region. Prediction accuracy of each data set for prefetching enabled and disabled cases are presented in Fig. 4. The accuracy of different input sizes demonstrates that MAPredict's provides consistent accuracy for varied input sizes. The BW processor is used for this evaluation, and a similar trend is observed for others.

6.2 Irregular access and large application with mixed patterns

To evaluate MAPredict's capability of combining static and empirical data for irregular access and mixed patterns, XSBench and Lulesh, are considered.

Algorithmic randomness XSBench [27] is a proxy application that calculates the macroscopic neutron cross-section by randomly searching for energy and material. The energy search is done by employing a binary search on a unionized energy grid, an example of algorithmic randomness (total access = $\text{Access}_{\text{random}}$ * $\text{Empirical}_{\text{factor}}$). As discussed in §3.4, both the number of access and the location of access are random. Since it follows a binary search, algorithm complexity ($\log n$) is used to measure $\text{Access}_{\text{random}}$. The $\text{Empirical}_{\text{factor}}$ is calculated for BW with prefetching disabled and used for all other processors ($\text{Empirical}_{\text{factor}}$ = the ratio of measured value and $\text{Access}_{\text{random}}$). The predicted value is then compared to the average of five measurements (up to 5% standard deviation) for accuracy calculation. The blue bars in Fig. 5 show that only BW provided high accuracy when prefetching is disabled and hence demonstrating the need for machine-specific $\text{Empirical}_{\text{factor}}$. When individual $\text{Empirical}_{\text{factor}}$ is used, the accuracy of each processor improved (yellow bar). MAPredict provides the option to include multiple machine specific $\text{Empirical}_{\text{factor}}$ in a single pragma; thus, a single source code can be updated for multiple machines. The method presented in [31] does not calculate the total number of random access rather focuses on the access location, which makes the comparison irrelevant. Algorithmic randomness is an extreme case, and it is only present in a certain function. For this reason, a large application with mixed patterns is investigated next for different input sizes.

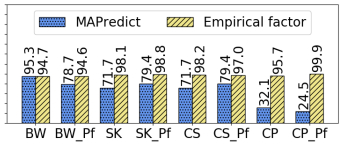


Fig. 5: Accuracy of algorithmic randomness for XSBench

A large application with mixed patterns : Lulesh To demonstrate that MAPredict can work with a large application with different memory access patterns, Lulesh [10] is considered. Lulesh is a well-known app with different memory access patterns for a 3D mesh data structure. It has 38 functions with a complex call graph and 4474 lines of code, making it a large and complex example. Three large data structure sizes ($250 \times 250 \times 250$, $300 \times 300 \times 300$, and $400 \times 400 \times 400$) are used. The SK machine is selected for experimentation because it has the smallest cache and hence, stresses the capability of MAPredict by increasing the probability of machine randomness.

Function categorization of Lulesh Out of 38 functions in Lulesh, 24 functions provide significant memory transactions ($>$ 1 Million LLC-DRAM transactions). The 24 memory intensive functions have different memory access patterns. Most memory intensive functions are shown at Table 3, where column-2 shows access patterns. Here, St=stencil (eight-point non-adjacent 3D stencil), S=stream, DR=data structure randomness, and I=non-initialized arrays, N=nested randomness (DR with branches), and All=all the above patterns.

***Empirical*$_{factor}$** Lulesh has data structure randomness in three functions. The Empirical$_{\text{factor}}$ is calculated by comparing the static and dynamic data to address this randomness (a one-time effort). So, a total of three Empirical$_{\text{factor}}$ are used in three functions (out of 38).

Traffic: number of LLC-DRAM transactions Columns 3 and 4 show the LLC-DRAM transaction (M=Million and B=Billion) obtained for MAPredict and TAU+PAPI (dynamic analysis tool). The last function, which is the parent of all functions, shows a total of 1.7 Billion LLC-DRAM transactions. However, for the largest data size, Lulesh exhibits 3.5 Billion LLC-DRAM transactions.

Scaling and accuracy for Lulesh Scaling in terms of input sizes provides a measure of success for one-time calculation of the Empirical$_{\text{factor}}$. Column 5-10 of Table 3 show the accuracy of different functions for different data sizes for prefetching enabled and disabled cases. Since some functions are parents to other functions and the last function is the parent to all (total traffic), inaccuracy in one function impacts the overall accuracy. MAPredict showed more than 93% accuracy for all data sizes, which demonstrates the model and *Empirical*$_{factor}$ scaled well in terms of input size. However, when multi-threaded experiments are used the overall accuracy dropped but sill provided more than 80% accuracy.

6.3 Discussion

For regular access patterns, MAPredict's static analysis provides higher accuracy than the literature model and can handle different input sizes, micro-architectures, cache sizes, compilers, and execution models. However, MAPredict requires empirical observation for irregular patterns.

Table 3: Analysis of Lulesh (selected functions). d1=data size 1 without prefetching, p-d1=with prefetching.

Function name (Shortened)	Access Pattern	MAP redict	TAU PAPI	Accuracy - 3 data sizes					
				d1	p-d1	d2	p-d2	d3	p-d3
IntegrateStressF.Elm	St,S	81M	83M	99.0	97.4	88.5	88.9	91.8	91.7
CFBHour.ForceF.Elm	St,S	239M	241M	96.9	99.1	97.0	96.5	82.2	82.6
CHourg.Cont.F.Elm	St,I,DR	604M	647M	92.8	93.1	93.0	92.7	76.2	77.9
LagrangeNodal	All	824M	874M	94.2	94.2	95.3	95.1	85.4	86.5
CKinematicsF.Elm	S,St	99M	100M	98.7	99.7	96.0	96.6	98.3	98.7
CLagrangeElements	St,S,I	126M	130M	98.3	97.5	99.7	99.9	98.5	98.3
CMon.QGrad.F.Elm	S,St	99M	105M	95.2	94.4	95.6	95.6	94.6	94.5
CMon.QReg.F.Elm	DR,N,S	141M	150M	94.9	94.6	95.4	93.2	94.3	93.2
CEnergyF.Elm	S	249M	261M	97.7	95.6	94.0	98.6	99.9	93.4
EvalEOSF.Elm	DR,S	429M	451M	99.1	95.1	95.7	97.9	98.4	93.0
UpdateVol.F.Elm	S	10M	10M	99.9	99.9	99.7	99.9	99.9	99.9
LagrangeElements	All	824M	869M	98.4	95.0	99.3	96.7	96.9	93.7
Overall	All	1.6B	1.7B	95.0	93.0	96.6	94.3	95.8	99.2

Overhead of MAPredict One of the objectives of MAPredict is to make it usable from runtime systems for fast decisions. The evaluation of Lulesh takes 28.3 milliseconds (38 functions), averaging to less than a millisecond per function. For source code preparation, 249 lines of Aspen directives (79 MAPredict directives) are used for 4474 lines of code, which is 5.5% source code overhead.

Usability of $Empirical_{factor}$ The calculation of $\text{Empirical}_{\text{factor}}$ is needed for irregular accesses. However, the $\text{Empirical}_{\text{factor}}$ calculation is a one-time effort. Once calculated, it becomes a part of the source code and can provide prediction statically. Moreover, randomness usually occurs only in a small portion of an application (regular access patterns are more commonly found). So, the $\text{Empirical}_{\text{factor}}$ calculation is needed only where randomness exists.

7 Related Works

Related works presented in this section are divided into two categories.

7.1 Memory access prediction

Several studies investigated memory access patterns to make a reasonable prediction. Yu et al. [31] used analytical models of different memory access patterns. In Tuyere [20], Peng et al. used data-centric abstractions in an analytical model to predict memory traffic for different memory technologies. Application models in these aforementioned studies are manually prepared. Moreover, MAPredict goes beyond these wrok by including the impact of page size, prefetchers and compilers in machine model. Moreover, Tuyere framework showed the benefit of analytical models over trace-based or cycle accurate simulator (such as Ramulator [11], DRAMSim [21]) both in terms of time and space. MAPredict further improves upon Tuyere by providing prediction in 1-3 milliseconds per function. Allen et al. [3] investigated the impact of two memory access patterns on GPUs. Some previous works used load and store instruction counts to measure memory access and used that count to predict performance(e.g., COMPASS by Lee at

Table 4: Comparison with other works. A=All, P=Partial.

Studies by	Static analysis	Analytical model	Access patterns	Diff. micro-architecture	Diff. compilers	Multi-threaded	Pref etchers
Peng et al. [20]	✓	✗	A	✗	✗	✓	✗
Yu et al. [31]	✓	✓	A	✗	✗	✗	✗
Monil et al. [19]	✗	✗	P	✓	✗	✗	✓
Lee at al. [12]	✓	✗	P	✓	✗	✓	✗
Marques et al. [15]	✗	✗	P	✗	✗	✓	✗
Alappat et al. [2]	✗	✗	P	✓	✓	✓	✓
Hammond et al. [5]	✗	✗	P	✓	✗	✓	✗
Molka et al. [17]	✗	✗	P	✓	✗	✓	✗
MAPredict	✓	✓	A	✓	✓	✓	✓

al. [12]). Compile-time static analysis tools, such as Cetus [4], OpenARC [13], and Caascade [14] are also used to measure instruction counts at compile time and can provide a prediction. MAPredict does not solely depend on instruction counts; it captures the impact of cache hierarchy through analytical models. In contrast to MAPredict's near-accurate prediction, analytical models such as Roofline Model [30] and Gable [6] provide an upper bound for a system.

7.2 Understanding Intel processors

Some studies delved into Intel processors to understand their performance by using benchmarks. Using the Intel advisor tool, Marques et al. [15] analyzed the performance of benchmark applications to understand and improve cache performance. Alappat et al. [2] investigated Intel BW and CS processors to understand the cache behavior using the likwid tool suite [28]. Hammond et al. investigated the Intel SK processor [5] by running different HPC benchmarks. Hofmann et al. also investigated different Intel processors to analyze core and chip-level features [7, 8]. Park et al. also investigated the performance of different Intel micro-architectures and optimized HPC benchmarks to perform better. Molka et al. [17] used a micro-benchmark framework to analyze the main memory and cache performance of Intel Sandy bridge micro-architecture (also AMD Bulldozer processors). Performance evaluation using benchmarks is also done by Saini et al. for Ivy Bridge, Haswell, and Broadwell micro-architectures [22, 23]. These studies investigated Intel micro-architectures using benchmarks; however, unlike MAPredict, they did not develop strategies for predicting memory traffic.

7.3 Comparing MAPredict with other studies

Table 4 compares MAPredict with other literature where first four rows represent the study of memory access patterns and static analysis. The next four rows represent studies that are focused on understanding of Intel micro-architectures. Table 4 shows that MAPredict addresses the missing parts from both domains to provide a unique framework.

8 Conclusion and Future Work

This research presents the MAPredict framework, which provides a prediction of memory traffic for Intel processors. This study investigates the interplay between an application's memory access pattern and Intel micro-architectures' cache hierarchy. Based on the observation from Intel processors, an analytical model is

derived that takes memory access patterns of an application, properties of a processor, and choice of the compiler into consideration. MAPredict generates an application model for a given application through compile-time analysis. The application is combined with a target machine model to synthesize the appropriate analytical model to predict LLC-DRAM traffic. Through experimentation with benchmarks on processors from Intel Broadwell, Skylake, Cascade Lake, and Cooper Lake micro-architectures, the analytical model's validity is verified by achieving average accuracy of 99% for streaming, 91% for strided, and 92% for stencil patterns. MAPredict also facilitates providing hints in the source code to capture dynamic information and randomness either from the application or machine to obtain better accuracy. By combining static and empirical approaches, MAPredict achieved up to 97% average accuracy on different micro-architectures for random access patterns. Future work will investigate MAPredict on AMD, ARM, and IBM processors.

References

1. Top 500 supercomputers published at sc20. `https://www.top500.org/`.
2. C. Alappat, J. Hofmann, G. Hager, H. Fehske, A. Bishop, and G. Wellein. Understanding hpc benchmark performance on intel broadwell and cascade lake processors. *arXiv preprint arXiv:2002.03344*, 2020.
3. T. Allen and R. Ge. Characterizing power and performance of gpu memory access. In *Intl. Workshop on Energy Efficient Supercomputing (E2SC)*, pages 46–53, 2016.
4. C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, pages 36–42, 2009.
5. S. Hammond, C. Vaughan, and C. Hughes. Evaluating the intel skylake xeon processor for hpc workloads. In *International Conference on High Performance Computing & Simulation (HPCS18)*, pages 342–349, 2018.
6. M. Hill and V. J. Reddi. Gables: A roofline model for mobile socs. In *Intl. Symposium on High Performance Computer Architecture (HPCA)*, pages 317–330, 2019.
7. J. Hofmann, D. Fey, J. Eitzinger, G. Hager, and G. Wellein. Analysis of intel's haswell microarchitecture using the ecm model and microbenchmarks. In *Intl. Conference on Architecture of Computing Systems*, pages 210–222. Springer, 2016.
8. J. Hofmann, G. Hager, G. Wellein, and D. Fey. An analysis of core-and chip-level architectural features in four generations of intel server processors. In *International supercomputing conference*, pages 294–314. Springer, 2017.
9. W. Jalby, D. Kuck, A. Malony, M. Masella, A. Mazouz, and M. Popov. The long and winding road toward efficient high-performance computing. *Proceedings of the IEEE*, 106(11):1985–2003, 2018.
10. I. Karlin. Lulesh programming model and performance ports overview. Technical report, Lawrence Livermore National Lab.(LLNL), CA, United States), 2012.
11. Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2015.
12. S. Lee, J. Meredith, and J. Vetter. Compass: A framework for automated performance modeling and prediction. In *29th International Conference on Supercomputing (ICS15)*, pages 405–414, 2015.
13. S. Lee and J. S. Vetter. OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, Vancouver, 2014. ACM.

14. M. Lopez, O. Hernandez, R. Budiardja, and J. Wells. Caascade: A system for static analysis of hpc software application portfolios. In *Programming and Performance Visualization Tools*, pages 90–104. Springer, 2017.
15. D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. Matveev. Performance analysis with cache-aware roofline model in intel advisor. In *Intl. Conference on High Performance Computing Simulation*, pages 898–907, 2017.
16. J. D. McCalpin. Stream benchmarks, 2002.
17. D. Molka, D. Hackenberg, and R. Schöne. Main memory and cache performance of intel sandy bridge and amd bulldozer. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, pages 1–10, 2014.
18. M.A.H. Monil, M. Belviranli, S. Lee, J. Vetter, and A. Malony. Mephesto: Modeling energy-performancein heterogeneous socs and their trade-offs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020.
19. M.A.H. Monil, S. Lee, J. Vetter, and A. Malony. Understanding the impact of memory access patterns in intel processors. 2020.
20. I. Peng, J. Vetter, S. Moore, and S. Lee. Tuyere: Enabling scalable memory workloads for system exploration. In *International Symposium on High-Performance Parallel and Distributed Computing*, pages 180–191, 2018.
21. P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011.
22. S. Saini and R. Hood. Performance evaluation of intel broadwell nodes based supercomputer using computational fluid dynamics and climate applications. In *2017 IEEE 19th International Conference on High Performance Computing and Communications Workshops (HPCCWS)*, pages 58–65. IEEE, 2017.
23. S. Saini, R. Hood, J. Chang, and J. Baron. Performance evaluation of an intel haswell-and ivy bridge-based supercomputer using scientific and engineering applications. In *2016 IEEE 18th International Conference on High Performance Computing and Communications (HPCC)*, pages 1196–1203. IEEE, 2016.
24. S. Shende and A. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.
25. K. L. Spafford and J. S. Vetter. Aspen: A domain specific language for performance modeling. In *SC12: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Salt Lake City, 2012.
26. D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. 2010.
27. J. Tramm, A. Siegel, T. Islam, and M. Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.
28. J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, 2010.
29. M. Umar, S. V. Moore, J. S. Meredith, J. S. Vetter, and K. W. Cameron. Aspen-based performance and energy modeling frameworks. *Journal of Parallel and Distributed Computing*, 120:222–236, 2018.
30. S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.
31. L. Yu, D. Li, S. Mittal, and J. S. Vetter. Quantitatively modeling application resiliency with the data vulnerability factor. *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2014.