Performance Analysis and Optimization with Little's Law

Sanyam Mehta
Hardware Software Co-design
HPE, Bloomington, MN, USA
sanyam.mehta@gmail.com

Abstract—**Performance tools are the bridge between processor architecture and a user. However, with the increasingly complex processor architectures, it is becoming increasingly difficult for the users to comprehend the information generated by the performance tools to help diagnose and fix the performance bottlenecks. In addition, the performance tools are themselves limited in many cases. Finally, there is wide variability in the kind of performance counters provided by the different processor vendors, making performance tools unportable across emerging architectures. In this work, we propose to solve these problems by accurately computing a portable and easily comprehensible performance metric - the (Memory-Level Parallelism) MLP of an application. The observed MLP when seen as a fraction of peak theoretical MLP supported by the host processor provides important guidance on the applicability of various popular program optimizations. Six case studies on three different processors each with a different memory technology show that our metric is both effective in program analysis and provides useful guidance on program optimization.**

Keywords - Performance Analysis; Memory-level Parallelism; Compiler Optimizations;

I. Introduction and Motivation

The world of modern processor architecture with multiple instruction issue and out-of-order execution is ever-increasingly complex with so many potential causes of performance bottleneck. These causes include processor stalls due to filled up fetch queue, scheduler, reorder buffer (ROB), load queue, store queue, execution ports, fill buffers (or Miss Status Handling Registers or MSHRs) and memory controller queue, and also cycles wasted at branch mispredictions. With so many causes of performance bottleneck, it is not viable for the user to always know which exact cause ends up being the performance bottleneck for the host application. In such cases of user's limited vision, performance tools could serve as a 'pair of correcting glasses' by which to visualize the processor architecture. In other words, the performance tools could expose the right set of performance counters that reveal the performance bottlenecks to the user. However, we find that existing performance tools fall considerably short due to a disconnect from both the processor architecture and the user.

Disconnect from the processor architecture. Out-of-order execution in modern processors is achieved at the expense of a complex interplay between various structures in the core that complicate the interpretation of performance counters. We explain these complications that prevail even in the Top-Down Microarchitectural Analysis (TMA) Method [1], which we believe represents the state-of-the-art in performance analysis. The TMA method is used in Intel VTune [2] Profiler's popular microarchitecture exploration analysis to guide users towards finding performance bottlenecks. While sophisticated, our analysis reveals the following shortcomings.

- The bottlenecks are broken down into four categories: Retiring, Frontend, Bad Speculation and Backend. These bottlenecks happen in different pipeline stages of an out-of-order superscalar processor and thus there is inherent imprecision due to overcounting or undercounting stalls among these categories. For example, while instruction issue may be stalled due to a fully filled scheduler or ROB, the processor may still be optimally utilizing its execution units by allowing previously issued instructions to enter execution. In another case, a long latency load may similarly stall instruction issue, but there may be many parallel loads in flight that fully consume the memory bandwidth. In another example, fetch related front-end stalls may overlap with issue-related backend stalls leading to double counting. Furthermore, the multiple issue causes ambiguities - in an n-wide issue processor, whether or not a cycle with less than n issued instructions counts towards a backend stall.
- The most useful of these categories is the 'backend bound' category, which is further divided into 'core bound' and 'memory bound' categories. The percentage distribution between these two sub-categories is a function of cycles during which either no instruction can enter execution due to outstanding memory requests (indicating memory bound) and cycles featuring poor (determined heauristically) port utilization. We find this distribution to be not useful in practice since a long latency memory access usually leads to poor port utilization which may be misrepresented by this distribution. In such a case, a useful solution is to perform memory optimizations; poor port utilization in such a case is usually a result of dependencies among instructions, which often cannot be fixed without a code rewrite.
- Another interesting feature of TMA is that it subdivides 'memory bound' category into 'bandwidth bound' and 'latency bound', which can be very useful. However, this distribution is guided by the percentage occupancy of the memory controller - if the occupancy exceeds its self-

defined threshold, the cycles are attributed to 'bandwidth bound' and 'latency bound' otherwise. We tested this strategy on a real application, SNAP, on Intel Skylake Gold 6130 processor. When using the full socket, we found that TMA reports SNAP to be 27% bandwidth bound and 23% latency bound. Also, the memory-access analysis suggested by the preliminary microarchitecture analysis reports an average memory latency of just 9 cycles, suggesting a non-significant latency problem. Amid this unclear guidance, we do find however, that latency reducing software prefetching optimization does indeed improve SNAP performance by 8%. Another drawback is that TMA reports this distribution *only* for the whole program and not for individual routines, which compromises the ability to focus memory problems to a particular routine as different routines can behave very differently in a real application. In SNAP, for instance, we observe the above performance benefit when applying prefetching to a single routine, dim3_sweep, that actually was latency bound.

- Finally, there is sometimes errors in these performance counters. For instance, the FLOP counting performance counters were reported to be inaccurate in some Intel processors [3].

In summary, these metrics on stalls do not provide a clear picture of an application's performance on the processor. These problems are compounded by the fact that not all processors even expose the required performance counters to trace the performance bottleneck and that the performance counters offered by various processor vendors (such as Intel, AMD and now ARM). For example, the TMA method is specific to Intel and performance counters in the latest Intel processors are in fact implemented to support the TMA method in VTune. Furthermore, performance counters among processors by even the same vendor (such as Intel's Xeon and Xeon Phi) sometimes differ significantly. Table I summarizes the situation.

Processor	Breakdown of stalls	L1-MSHRQ-full stalls	L2-MSHRQ-full stalls	Memory Latency
Intel	Limited	Yes	No	Limited
AMD	Limited	Yes	No	Limited
Cavium	Very limited	No	No	No
Fujitsu	Limited	No	No	No

TABLE I
EXTENT OF VISIBILITY INTO SPECIFIC EVENTS ACROSS PROCESSOR VENDORS (IN GENERAL)

Disconnect from the user. Assume two kinds of users - an expert and a non-expert. For the non-expert that does not understand the microarchitectural details, the performance counters reporting various kinds of stalls is not meaningful. For the expert that understands the processor's microarchitecture, these counters are still not helpful because they do not point to concrete actionable steps such as which optimizations to try to improve performance. Again considering TMA or Intel VTune as an example, we find that while VTune does an excellent job of explaining the different bottlenecks themselves, it often does not offer useful guidance or a concrete recipe in terms of actionable steps to the user. For example, even in the important memory bound category, VTune offers limited guidance such as 'Consider improving data locality in NUMA multi-socket systems' when bandwidth bound and 'Consider optimizing data layout or using Software Prefetches (through the compiler)' when latency bound. In summary, the user is faced not only with a lack of precise metrics to determine exact performance bottleneck but also with the task of choosing the right optimizations from the many options.

This work proposes to fill these gaps in existing performance tools. We calculate the MLP of an application using a minimum number of commonly available performance counters. That is, we obtain memory bandwidth from the performance counters (either through L3 misses as in x86 processors or directly from memory reads/writes as in ARM-based processors), which is then used to calculate observed latency (at that bandwidth from the bandwidth-latency plot for the processor calculated once only by customizing an open-source memory characterization tool [4]). Using little's law, we obtain the MLP from memory bandwidth and latency. This MLP has three interesting properties that make it an 'appropriate prescription glasses' between the user and the processor. (1) It is portable and can be accurately calculated with our approach across a variety of processors. (2) It abstracts away the details of out-of-order execution from the user. In particular, the MLP can be directly correlated to the Miss Status Handling Registers (MSHRs) at the cache, and thus the user contends with a single comprehensible structure in the core. (3) It has useful correlation with various important compiler optimizations, and thus provides natural pointers to the user towards specific optimizations.

In this paper, we make the following contributions:

1) We propose to use MLP as a metric to both analyze performance bottlenecks and optimize the application performance.
2) We propose a novel way to calculate observed MLP accurately. We use observed bandwidth to calculate observed latency, both of which combined yield MLP (Little's law). Since performance counters are merely used to obtain memory bandwidth, which is supported on all popular processors (both x86 and ARM), our technique is portable across different platforms.
3) Our experiments on six applications across three different hardware platforms - Intel Skylake (SKL), Intel Knights Landing (KNL), and Fujitsu's A64FX with high bandwidth memory (HBM2) based on the ARM ISA) reveal that using MLP is not only entirely portable, but is also very effective in pointing out performance bottlenecks and suggesting appropriate compiler optimizations. We thus present a general platform for performance analysis and optimization for all existing and upcoming processors to our knowledge.

II. Related Work and Discussion

We categorize our discussion of related work into three areas.

Performance tools. There are many popular performance tools such as Intel VTune Profiler [2], Barcelona Supercomputing Centre's Paraver [5], University of Oregon's TAU [6], Rice University's HPCTooklit [7], [8], Cray's CrayPat [9]. All of these tools rely on host processor's performance counters to measure data during the course of program execution and report either the counts for the chosen events themselves or derived counters. Intel VTune Profiler uses the top-down microarchitectural analysis (TMA) [1]. TMA breaks down the pipeline slots spent in various structures in the core (such as front-end, back-end, etc.). Apart from overlapped counting of these complex structures, this analysis requires significant user expertise to unravel and still does not point the user towards specific program optimizations. Also, TMA is Intel-specific and uses custom performance counters [10].

The importance of memory latency for understanding application behavior has previously been recognized in performance tools. For example, TMA reports average memory latency and bandwidth of a routine as derived metrics; Paraver and HPCToolkit can correlate high memory latency accesses with data objects in the source program using Intel's Precise Event-Based Sampling (PEBS) [2] (or AMD's Instruction Based Sampling (IBS) [11]). For measuring memory latency, these tools use an Intel-specific performance counter that reports the number of memory requests whose latency is above a particular threshold (4, 8, 16, 32, 64, 128, 256 and 512). We analyzed this performance counter in detail. This has the following shortcomings when measuring memory latency. (a) This counter measures cycles from first load dispatch to completion. The load may have to dispatch again in the event of a load mis-speculation, which will attribute cycles to this counter. A load may encounter TLB miss and even a page table walk, all of which will be attributed to this counter. In other words, this counter is not precisely related with memory latency and may be considerably greater than actual observed memory latency. In fact, Intel notes, 'Reported latency may be longer than just the memory latency' in the description for the counter [12]. Our experiment with ISx (that involves random accesses and therefore TLB misses and page table walks) reveals that this counter reports 75% of the memory loads to have a latency of greater than 512, whereas we understand from observed bandwidth and latency profile for our host processor that the observed memory latency is a fairly constant 180ns or 378 cycles. (b) Furthermore, using this performance counter cannot help us obtain any reasonably accurate latency number to calculate MLP given the large bins. (c) We ran hpcg on a fully loaded Skylake node with VTune. hpcg achieves peak bandwidth utilization on Skylake, given its bandwidth intensive nature. While the 'memory-access' analysis in VTune showed that hpcg spends more than 85% of time utilizing 'high' bandwidth, it reports an average latency of just 32 cycles. This is because most loads are effectively prefetched given the streaming nature of memory accesses in hpcg. However, the actual memory latency at full load in Skylake is roughly 180ns or 378 cycles, and thus this counter proved to be again misleading in such a scenario. Using this counter while disabling hardware prefetching is also inadequate since that drastically changes application behavior. Also, as noted in Table I, the ARM based processors do not support counters to help calculate memory latency, and some old AMD processors offered support for calculating average L2 latency seen by L2 load misses, which had some of the afore-mentioned limitations and was later withdrawn. Our work, on the other hand, relies on fairly accurate calculation of memory latency and thus MLP, which is then used to guide program optimizations.

The Roofline Model and extensions. Williams et al. proposed the original roofline model [13]. The roofline model provides insight into the compute and memory behavior of an application. In particular, it tells if an application is closer to being bound by peak performance or peak bandwidth of the processor. The roofline model serves as a good starting point (even with our approach) in analyzing an application since cache/memory bandwidth and peak FLOP rate of the processor are the most fundamental metrics that bound application performance. Given the popularity of the original roofline model, it has been used to study various applications [14]–[16]. However, we find that most applications are neither bound by FLOPs or bandwidth and it is important to know the way to approach program optimization in those cases.

Architectural simulators. As discussed above, use of performance counters or high-level microarchitecture models suffer from the drawback of accuracy. In this case, the architectural simulators [17], [18] could make up for this limitation. However, the significant overhead is usually unacceptable, and not all application developers have access to or are used to using such simulators.

III. Our Approach

The key to our approach is the correlation that exists between observed MLP and the maximum MLP supported by the core. This section provides the necessary background towards this end.

A. Background

We first discuss the interaction between demand requests, prefetching (software and hardware) and MSHRs.

Role of MSHRs: Out-of-order execution relies on executing multiple operations and memory requests in parallel. At any time, all unique memory requests that have missed the cache are tracked (at cache line granularity) using MSHRs at that cache (L1 or L2). This tracking thus avoids duplicate memory requests. When the hardware prefetcher is triggered, it issues prefetch requests at L2 cache (although there is usually a hardware prefetcher at L1 too, L2 hardware prefetcher is much more aggressive and therefore most useful), the addresses of which are also buffered in the MSHR queue at the L2 cache.

Similarly, software prefetches too utilize these MSHRs for memory requests.

MSHRs and MLP: The occupancy of MSHR queue in the processor is the correct indicator of all outstanding memory requests at any given time and therefore the MLP of an application. However, since there are two MSHR queues (at L1 and L2), it is important to understand which of the two queues would become the bottleneck for any given application. This depends on two factors: (a) size of each queue, and (b) nature of the application. The size of L1 MSHR queue is kept as small as possible to meet L1 cache access timing constraints since each arriving request must simultaneously search all entries in the MSHR queue. The size of the MSHR queue at the L2 cache is usually considerably larger than the size of L1 MSHR queue. Also, if an application does not trigger the L2 hardware prefetcher (such as in case of random accesses), then the occupancy of the L2 MSHR queue is not more than the L1 MSHR queue. Thus, we conclude that for applications that have random accesses, L1 MSHR queue is the potential cause of limited MLP, whereas for applications that have streaming accesses benefit from L2 hardware prefetcher, the L2 MSHR queue is the potential cause of performance limitation.

B. Little's Law: Correlating MLP, Memory Latency and Memory Bandwidth

Let n_{avg} be the average number of MSHRs occupied or the average MSHRQ occupancy or average observed MLP of an application, lat_{avg} be the average memory latency observed, R be the total number of memory requests (including hardware prefetch requests) and T be the total time taken by the program/subroutine. Applying Little's law [19][1] from the queueing theory, we can say that the long-term average number of outstanding requests (n_{avg}) is given by the long-term average memory requests arrival rate ($\frac{R}{T}$) multiplied by the average memory latency (lat_{avg}).

$$n_{avg} = \frac{lat_{avg} \times R}{T} \tag{1}$$

Since memory request arrival rate is related to memory bandwidth, we introduce achieved bandwidth BW into the equation. BW can be calculated as $\frac{R * cls}{T}$, where cls is the cache line size at the cache level being considered. We thus obtain the following relation:

$$n_{avg} = \frac{lat_{avg} \times BW}{cls} \tag{2}$$

It is important to note lat_{avg} in the above formulation is the observed memory latency in the processor at a particular bandwidth utilization (BW), and not the idle latency (no other load) as commonly reported by processor vendors. The observed latency increases as bandwidth utilization increases and can be 2x or more than the idle latency at peak bandwidth utilization. To our knowledge, there are no existing performance tools that report this average latency (as discussed earlier) and thus Little's law has remained unused for performance analysis.

[1]Little's law states that the long-term average number of customers in a stationary system is equal to the long-term average effective arrival rate multiplied by the average time that a customer spends in the system. It is important to note that Little's law assumes a stationary system and we thus propose to apply it on individual subroutines or individual long loops and not to the entire program as discussed more later

C. Average MSHR occupancy and Program Optimizations

We find the MLP or MSHRQ occupancy obtained above to be a useful guide in determining the utility of various program optimizations that are described next.

Vectorization. A well known optimization where a single operation is applied on multiple operands. Vectorization offers another level of parallelism in addition to thread-level parallelism, and is therefore very effective in increasing MLP. This is essential to obtaining good performance on emerging processors with HBM. Also, the degree of parallelism (vector width) and coverage (with gather/scatter, predication, etc.) through vectorization is increasing in recent processors, making it more widely applicable than before. Since vectorization improves MLP, vectorization also increases the average MSHRQ occupancy. Thus, if an application's average MSHRQ occupancy is close to MSHRQ size, then it is unlikely for the application to see additional benefit from vectorization. We shall see examples of this scenario in Section IV.

Software prefetching. In this optimization, the user or the compiler inserts software prefetch instructions in the source code and can prefetch data to a specific level of cache [20]. This is especially useful for certain irregular access patterns since hardware prefetchers either miss those patterns or are not timely enough. Each software prefetch request occupies an MSHR thus denying another demand load request (or the hardware prefetcher [21], [22]) from acquiring an MSHR. We therefore do not recommend this optimization when the MSHRQ occupancy is high. Interestingly however, processor vendors offer the opportunity to prefetch data to selective levels of cache, which makes this optimization promising for random accesses because it can lead to use of L2 MSHRs which are otherwise unused when the hardware prefetcher is ineffective [23]. We shall cover this scenario in the case of ISx in Section IV. We also discuss the benefit of software prefetching on SNAP.

Loop tiling. Loop tiling partitions a loop's iteration space into smaller chunks or blocks so that data accessed in those smaller blocks stays in the cache until reused [24]. Loop tiling can be done targeting reuse in different levels of the memory hierarchy [25]. Loop tiling is an excellent choice if the application sees high occupancy (where most other optimizations fail to increase performance) since tiling reduces the number of memory requests and therefore the MSHRQ occupancy. We cover an example, *Minighost*, in Section IV that has high MSHRQ occupancy and benefits significantly from tiling.

Unroll and jam. Also called register tiling, this is similar to loop tiling except that this targets data reuse in registers. This could be done in addition to loop tiling as in *dgemm*. This optimization is usually beneficial when memory accesses already see a small latency due to few memory accesses (i.e. most data fits in the higher levels of cache). Interestingly, this situation can be inferred from a low MSHRQ occupancy.

Loop fusion [26]. Loop fusion fuses bodies of different loops or loop-nests and as a result can significantly reduce the reuse distance of certain accesses. Like loop tiling, loop fusion is particularly useful in reducing the MSHRQ occupancy as it promotes data reuse. In some rare cases, however, loop fusion could hurt performance as it increases the number of data streams in the loop [21].

Loop distribution. Loop distribution is the exact opposite of loop fusion. It is a supporting optimization for loop fusion or vectorization like loop interchange. When used by itself, it is expected to benefit only when distributing loops can reduce the number of active streams or the memory bandwidth contention. It is for these reasons that loop distribution is unlikely to benefit applications that have low MLP or low MSHRQ occupancy.

Simultaneous multithreading (SMT) or hyperthreading (HT). Not exactly a program optimization, but a different way of running the application. This involves using the simultaneous multithreading or hyperthreading capability in the host processor. We envision SMT to be widely useful like vectorization on emerging processors with HBM since SMT can significantly increase MLP. We also demonstrate this in Section IV. Threads on a core (that participate in SMT) share most of core's resources including MSHRs, and MSHRQ occupancy is directly useful in understanding the benefit from SMT. A close to full MSHRQ implies insufficient resources for more threads in a core. We thus recommend all applications except those with high (close to max for the level of cache in concern) MSHRQ occupancy to benefit from SMT except in special cases such as cache residency contention among threads.

D. A Recipe for Performance Analysis and Program Optimization

In this section, we lay out our proposed recipe that shall serve as a guide for program optimization based on observed MSHRQ occupancy. While collecting data and deriving metrics, two conditions must be satisfied for best results. (a) The data must be collected in a loaded run, i.e. all (or most or as many as in the actual run) cores on a node must be used if the goal is to maximize throughput (otherwise as many cores as in real run must be used). We do not recommend using SMT for this initial profile since SMT can hurt performance. The reason for using all cores is that this will generate maximum MLP possible and will expose the potential bottlenecks. (b) We strongly recommend obtaining the above profile on a per-routine basis or even per-loop basis for routines with multiple loops as explained before. This is because averaging counter data from multiple routines that often behave differently usually provides misleading guidance.

Figure 1 depicts our recipe. To begin with, the values of the achieved bandwidth can be obtained using appropriate performance counters or directly from the application. The value of average latency can be deduced from observed bandwidth based on observed loaded latency numbers for the processor[2] (our approach for obtaining average latency relies on the X-Mem [4] tool discussed in Section IV). With these values, average MSHRQ occupancy or MLP is obtained using Equation 2. As explained before, it is important to note that the contention found with MSHRQ could be associated with the L1 cache or the L2 cache. This is a function of the routine in question - if the routine is dominated by random memory accesses (i.e. the prefetcher is largely ineffective), then L1 MSHRQ is of our concern, otherwise L2 MSHRQ is the source of potential bottleneck. This could also be determined by observing the fraction of memory requests that are generated from hardware prefetcher versus demand loads - this data is also often exposed through performance counters or one may determine it by disabling the hardware prefetcher [33]. It is important to note here that in the event of a mix of sequential and random memory accesses such as in sparse matrix-vector multiplication operation, the data structure generating random memory accesses usually easily dominates memory traffic since each reference is usually to a different cache line as opposed to a different word on the same cache line.

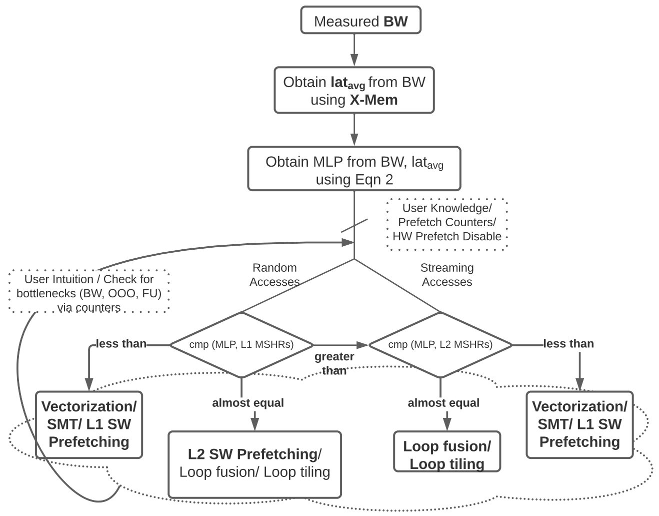
Measured BW
Obtain lat_avg from BW using X-Mem
Obtain MLP from BW, lat_avg using Eqn 2
User Knowledge/ Prefetch Counters/ HW Prefetch Disable
User Intuition / Check for bottlenecks (BW, OOO, FU) via counters
Random Accesses
Streaming Accesses
less than
cmp (MLP, L1 MSHRs)
greater than
cmp (MLP, L2 MSHRs)
less than
Vectorization/ SMT/ L1 SW Prefetching
almost equal
L2 SW Prefetching/ Loop fusion/ Loop tiling
almost equal
Loop fusion/ Loop tiling
Vectorization/ SMT/ L1 SW Prefetching

Fig. 1. Flowchart depicting our recipe

After we know the MSHRQ occupancy and the MSHRQ that will first become a potential bottleneck, we use this information to answer the two important questions. (1) If MSHRQ occupancy is almost the size of the MSHRQ, then we should either stop or only apply those optimizations that can help reduce MSHRQ occupancy and not increase it. (2) If MSHRQ occupancy is less than the size of the MSHRQ, all optimizations including those that increase MSHRQ occupancy or MLP could be applied. The process may be repeated to consider another optimization depending upon changes in

[2]Please note that obtaining the latency profile is independent of the application and is only computed once for a processor

Application	Description	Problem size	Routine chosen
ISx [27]	Scalable Integer Sort	Keys per PE = 25165824	count_local_keys
HPCG [28]	Sparse matrix-vector multiplication	40^3	ComputeSPMV_ref
PENNANT [29]	Unstructured mesh physics miniapp	meshparams = 960, 1080, 1.0, 1.125	setCornerDiv
CoMD [30]	Classical molecular dynamics	x=y=z=24, T=4000	eamForce
MiniGhost [31]	Difference stencil miniapp	nx=504, ny=126, nz=768, num_vars=40	mg_stencil_3d27pt
SNAP [32]	Discrete ordinates neutral particle transport	nx=64, ny=16, nz=24, nang=48, ng=54, cor_swp=1	dim3_sweep

TABLE II
APPLICATIONS USED IN EXPERIMENTS

MSHRQ occupancy and observed performance as noted in Figure 1.

After a particular optimization has been applied, there may be cases where there is no performance speedup obtained. This stage may come after one or more optimizations have already yielded performance benefit but the MLP is not yet equal to L1 or L2 MSHRs (indicating more scope for improvement). In this case, user intuition guided by other performance counters becomes important. While MLP does serve as a useful and quantitative way to reason about program optimizations, there are other structures in the core (such as floating point units in case of GEMM like applications, divide unit for division heavy applications, load queue for certain gather/scatter heavy applications, etc.) or memory bandwidth of the socket/processor that may become performance bottlenecks limiting any further performance improvement. It is important to note that the above recipe will still help performance improvement for these applications before hitting the bottleneck (for example, GEMM becomes FLOP bound after prefetching, cache and register tiling are applied, and vectorization still benefits division heavy and gather/scatter heavy applications, etc.), but the recipe may not always help to maximize MLP and user intuition in terms of understanding the specific bottleneck and applying custom optimizations is still useful. We discuss a scenario in Section IV-F.

IV. Results and Discussion

This section serves to test our proposed recipe. For this purpose, we test six HPC applications. For each application, we have chosen the most important routine (in terms of execution time) for the purpose of analysis. Tables II and III list the applications and platforms used for experiments, respectively. For all our tests, we fixed the core frequency at 2.1GHz and 1.4GHz for SKL and KNL, respectively, whereas A64FX's frequency is fixed at 1.8GHz by default. This was done to easily measure the benefit from optimizations such as vectorization that can significantly alter core frequency. Also, we use KNL in flat mode and allocate all structures in MCDRAM to extract maximum bandwidth. All applications were compiled using the Cray C, C++, and Fortran compilers (version 10.0). It is important to note that compiler-based software prefetching is on by default with the Cray Compiler. So, when we mention using software prefetching in ISx and SNAP below, we refer to user-directed software prefetching using prefetch instructions/directives. For the purpose of parallelization, we use MPI programming model in case of ISx, HPCG, PENNANT and SNAP, whereas we used OpenMP for parallelizing CoMD and MiniGhost.

Platform	# Cores @ Rate	Theoretical Peak BW	L1 MSHRs per core	L2 MSHRs per core
Xeon Platinum 8160 (SKL)	24 @ 2.1GHz	128 GB/s	10 [34]	16 [34]
Xeon Phi 7250 (KNL)	68[3] @ 1.4GHz	400 GB/s	12 [35]	32 [36]
Fujitsu A64FX	48 @ 1.8GHz	1024 GB/s	12 [23]	∼20 [23]

TABLE III
DESCRIPTION OF PLATFORMS USED IN EXPERIMENTS

From Equation 2, we gather that for the purpose of calculating average MSHRQ occupancy, we need observed bandwidth and average latency. We measure the observed bandwidth for an application using Cray's profiler, CrayPat, which reports this number in its default output using readily available counters for all three processors chosen for our experiments. CrayPat uses the following counters for obtaining bandwidth: 'OFFCORE_RESPONSE_0:ANY_REQUEST:L3_MISS_LOCAL'[4] (we limited our runs to a single socket or NUMA node for simplicity; if more than one socket is used, L3 misses going to remote DRAM should also be included) in SKL, 'OFFCORE_RESPONSE_1:ANY_REQUEST:MC-DRAM + OFFCORE_RESPONSE_1:ANY_REQUEST:DDR' in KNL, and 'BUS_WRITE_TOTAL_MEM + BUS_READ_-TOTAL_MEM' for A64FX. For SKL and KNL, these counters do not include writebacks from L3 cache (since those are not L3 misses), so we use heuristics to estimate the bandwidth consumed by writebacks based on information from other counters. Note that we obtain observed bandwidth on a per-routine basis. For the purpose of measuring average latency, however, performance counters prove insufficient as noted in Section I. It must also be noted that idle memory latency cannot be used for this purpose since the loaded memory latency can be much higher. For the purpose of measuring loaded latency, although Intel's Memory Latency Checker (MLC) tool [37] is useful for Intel (or x86 in general) processors, we also need this data for ARM based A64FX processor. As a result, we use an open source tool called X-Mem [4] that is a cross-platform and extensible memory characterization tool. Although X-Mem contains support to obtain the latency profile (i.e. observed latency at a particular

[3]We use 64 cores in our runs since it is not always possible to partition the problem among 68 cores (as in CoMD, SNAP) and also to allocate some resources for the OS

[4]This also includes memory traffic due to page table walks from memory, and thus contribution of the most expensive TLB misses towards bandwidth utilization (and therefore latency) is accounted for in this way

observed bandwidth) of both x86 and ARM processors, we extended it to A64FX (that has large cache lines) to obtain the latency profile at high bandwidth usage. In summary, we obtain the latency profile for a processor using X-Mem, which lists the observed memory latency at many values of bandwidth utilization (configured using user-specified load on system through inserted delays or through thread-level parallelism - this does not require root privileges). With this profile and a measured value of bandwidth for a routine of concern, we can obtain the observed memory latency, and thus average MSHRQ occupancy.

We next discuss each application in detail. Tables IV through IX summarize results from performance analysis and program optimization efforts for each application. Each of these tables should be read as follows. Each row of the table summarizes a single experiment. The column, *Source*, denotes the version of the code used for the experiment - in some cases, this is the base or unoptimized version of the application source and in other cases, partially optimized version containing the optimization applied in a previous step of the optimization process. The following three columns report the observed bandwidth (and % of peak bandwidth), average latency (obtained from X-Mem), and calculated average MSHRQ occupancy, respectively, for the run involving the *Source*. Based on this calculated value of average MSHRQ occupancy and the guidance we provide on optimizations in Section III-C, we apply specific optimizations. The last and the most important column of each table then reports the optimizations applied over the *Source* and the corresponding improvement in performance (i.e. execution time). The last column serves to validate our approach by demonstrating that whenever there is headroom in increasing MLP (i.e. average MSHRQ occupancy is less than maximum and observed bandwidth is less than maximum achievable bandwidth), proposed optimizations result in improved performance and vice-versa. In a few cases, if proposed optimizations are unhelpful even when directed by the MSHRQ occupancy metric, the reason is discussed - for example, when using SMT, the contention for shared resources among threads can shadow the possible performance improvement even though there is headroom to increase MLP indicated by our calculations.

A. ISx

ISx is an extension of the Integer Sort benchmark of the NAS Parallel Benchmark Suite [38]. ISx is designed to aid co-design in the exascale era. ISx belongs to the class of bucket sort algorithms and uses a uniform random key distribution. As a result, the most time consuming routine in ISx (when run on an individual node), called count_local_keys, is characterized by random accesses to memory. ISx is therefore classified as memory bound (at least when run at small scale). Table IV analyses ISx on SKL, KNL and A64FX. From the table, we make following important observations for the different processors:

SKL: On SKL, unoptimized (i.e. base or original version without any code modifications) ISx achieves a bandwidth utilization of 106.9 GB/s. Using X-Mem, we find that the loaded latency observed at this bandwidth is 145ns. Using these values, we obtain an average MSHRQ occupancy of 10.1 using Equation 2. As discussed above, ISx involves random memory accesses and thus we understand that hardware prefetching is largely ineffective (also observed from performance counters). As a result, L1 MSHRQ will be the potential cause of bottleneck. Since SKL has 10 MSHRs at the L1 cache, MSHRQ occupancy of 10.1[5] implies that there is no headroom to push MLP any further through optimizations such as vectorization and hyperthreading as indicated by our recipe. *Our experiments confirm that this is indeed the case as vectorization and hyperthreading yield no benefit at all on SKL as shown in Table IV.*

Proc	Source	BW_{obs} (GB/s)	lat_{avg} (ns)	n_{avg}	Opt: Performance
SKL	base	106.9 (84%)	145	10.1	Vect: 1x
	+ vect	107.1 (84%)	145	10.1	2-way HT: 1x
KNL	base	233 (58%)	180	10.23	Vect: 1.02x
	+ vect	240 (60%)	182	10.66	2-way HT: 1.04x
	+ vect, 2-ht	253 (63%)	187	11.6	4-way HT: 0.98x
	+ vect, 2-ht	253 (63%)	187	11.6	**L2 Pref: 1.4x**
	+ vect, 2-ht, l2-pref	**344 (86%)**	238	**20**	-
A64FX	base	649 (63%)	188	9.92	**L2 Pref: 1.3x**
	+ l2-pref	**788 (77%)**	280	**17.95**	-

TABLE IV
SUMMARY OF PROGRAM OPTIMIZATIONS IN ISx

KNL: On KNL, the average occupancy of unoptimized ISx is calculated as 10.23. We know that KNL has 12 L1 MSHRs, implying a tiny headroom to increase MLP based on our recipe. *ISx does see a tiny performance benefit of 1.02x from vectorization.* The average MSHRQ occupancy of vectorized ISx is 10.66, again implying potential for some performance gain. *Using 2-way hyperthreading, we do see a 1.04x performance gain.* At this point, the calculated average occupancy is 11.6, almost equal to the number of L1 MSHRs. Based on our recipe, we would expect little to no performance benefit from 4-way hyperthreading. *The same was confirmed in our experiments - the performance actually degrades with 4-way hyperthreading*. We thus stick to 2-way hyperthreading for further experiments with ISx on KNL. Finally, the average MSHRQ occupancy reveals that there are unused MSHRs at L2 cache (since KNL has 32 MSHRs at L2 cache) and also more available bandwidth. Thus, as suggested earlier in Section III-C, L2 software prefetching could be promising in this case. *We indeed find that L2 software prefetching increases the average MSHRQ occupancy to 20 and the bandwidth utilization to 344 GB/s.* While we show MSHRQ occupancy to be particularly helpful in this case, we use this example to also discuss the limitation of the roofline model in identifying the precise bottleneck in this case and how MSHRQ occupancy could help draw another relevant roofline to improve the utility of the roofline model.

Figure 2 shows the roofline (Performance vs Intensity) plot [13] of ISx on KNL. The straight line with y-intercept of 12.48

[5]The MSHRQ occupancy is slightly higher than 10 since there is an additional reference in the routine of concern that issues contiguous memory accesses thereby utilizing L2 MSHRs although the amount of data touched is very small as compared to that touched by random accesses.

represents the peak bandwidth (400 GB/s) of KNL, whereas the peak performance is the horizontal line at the top (2867 GFlop/sec). A second (dotted) straight line with y-intercept of 8 (representing bandwidth of 256 GB/s) denotes another roofline (that we draw) that corresponds to the maximum bandwidth achievable with just L1 MSHRs. The two points on the plot, O and O1, represent baseline code (n_{avg} = 10.23) and the most optimized code (n_{avg} = 20), respectively. Since O is considerably below the solid line, the traditional roofline model would imply a considerable performance benefit from SMT whereas we observe a degradation from 4-way SMT on KNL. However, if the second dotted line is considered as an additional ceiling as in our recipe, we find that the performance is already hitting the roof. This also guides the user to shift the bottleneck to L2 MSHRs to utilize the surplus bandwidth using L2 software prefetching. This indeed provides considerable performance improvement and clearly breaks through the ceiling imposed by L1 MSHRs.

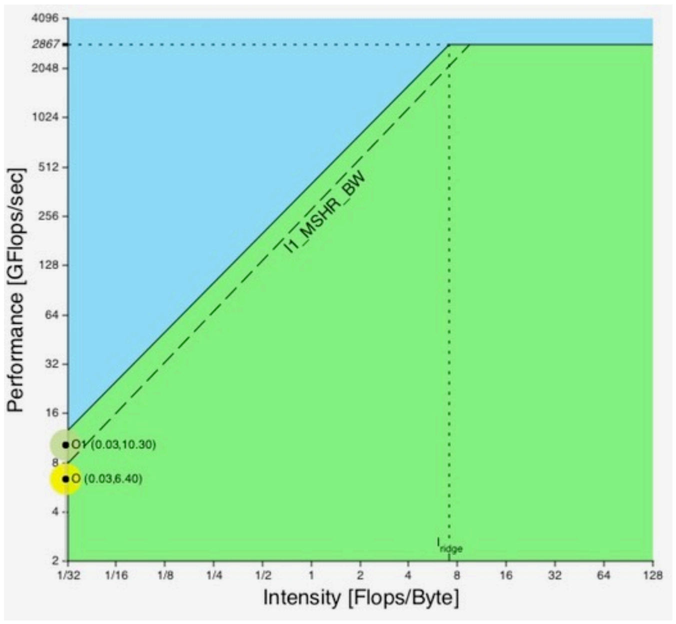

Fig. 2. Roofline model showing an additional ceiling from L1 MSHRs in ISx featuring random accesses

A64FX: On A64FX, the average MSHRQ occupancy of unoptimized ISx is 9.92. Given that A64FX has 12 L1 MSHRs, we observe that there is only a small headroom for an MLP increase from vectorization as in SKL and KNL (note that A64FX does not support SMT). We thus apply L2 software prefetching as in KNL above to utilize the extra MSHRs available at the L2 cache (around 20 per core in A64FX). *We again find that this optimization proves very useful and the average MSHRQ occupancy is increased to 17.95 much greater than the number of L1 MSHRs.* We also verify separately using Cray/HPE's proprietary cycle-level simulator that the original code leads to significant L1 MSHRQ full stalls, whereas the bottleneck is transferred to L2 MSHRQ after software prefetching, which reduces the time for which an MSHR in the L1 cache is occupied given that the data is now in the L2 cache instead of the memory.

B. HPCG

HPCG, a popular HPC application containing the sparse matrix vector computation. It has considerable spatial locality such that HPCG can benefit from hardware prefetching and vectorization. It is also important to note that SKL, KNL and A64FX all offer efficient support for gather/scatter operations through the AVX-512 or SVE instruction set extensions, and thus all processors are similarly capable of efficient vectorization of sparse matrix vector computation. Table V analyzes HPCG on SKL, KNL and A64FX. Following observations can be made.

SKL: On SKL, unoptimized HPCG achieves a bandwidth utilization of 109.9 GB/s. Using this observed bandwidth and latency deduced using X-Mem, we find that the average MSHRQ occupancy is 12.6. As discussed above, HPCG involves streaming accesses (although contains many streams), and thus hardware prefetching is very effective. This is confirmed by explicitly disabling the hardware prefetcher which results in more than 3x performance degradation - in other words, hardware prefetching alone improves performance by more than 3x. Since hardware prefetching is aggressively active, many more memory requests than L1 MSHRQ size could be in flight at the same time. The number of outstanding requests in this case in bound by the size of the L2 MSHRQ. We know that SKL has 16 MSHRs at the L2 cache, implying some headroom. However, the performance is also bounded by peak achievable (streams) bandwidth. In this case, since we are already almost at peak achievable (streams) bandwidth, we do not see any performance improvement from MLP increasing optimizations such as vectorization and hyperthreading. *The same can be inferred from our recipe in Section III-D, which suggests looking at peak achievable bandwidth whenever a preferred optimization provides no benefit.*

Proc	Source	BW_{obs} (GB/s)	lat_{avg} (ns)	n_{avg}	Opt: Performance
SKL	base	109.9 (86%)	171	12.6	Vect: 1x
	+ vect	108 (84%)	171	12.6	2-way HT: 0.98x
KNL	base	205 (51%)	179	8.95	**Vect: 1.15x**
	+ vect	**235 (59%)**	181	**10.38**	**2-way HT: 1.26x**
	+ vect, 2-ht	**296 (74%)**	209	**15.1**	4-way HT: 1.03x
A64FX	base	271 (26%)	156	3.44	**Vect: 1.7x**
	+ vect	**418 (41%)**	165	**5.62**	-

TABLE V
SUMMARY OF PROGRAM OPTIMIZATIONS IN HPCG

KNL: On KNL, the average occupancy of unoptimized HPCG is calculated as 8.95. We know that KNL has up to 32 L2 MSHRs, implying significant headroom to increase MLP as per our recipe. *Experiments confirm that vectorization provides a significant 1.15x gain in performance.* On vectorized HPCG, the calculated average occupancy is 10.38, still implying more headroom based on our approach. *As a result, 2-way hyperthreading yields another improvement in performance of 1.26x.* At this point, the average occupancy of optimized HPCG is 15.1, still implying opportunity for considerable performance improvement due to unused L2

MSHRs. However, with 4-way hyperthreading, we achieve a smaller than expected performance gain of 1.03x. This is because the L2 hardware prefetcher can track only 16 prefetch streams [39], and since each thread introduces 8-10 prefetch streams, the hardware prefetcher cannot keep up with 4-way hyperthreading.

A64FX: On A64FX, the average MSHRQ occupancy is 3.44 in unoptimized HPCG. This implies that there is significant headroom for increasing MLP. *We find that vectorization significantly improves performance on A64FX as in KNL.* The performance improvement on A64FX is, however, larger than KNL due to a lower MSHRQ utilization (and lower observed latency).

C. PENNANT

PENNANT, which is dominated by irregular memory accesses, represents unstructured mesh physics. We analyze the setCornerDiv routine since it is the most time consuming user routine in PENNANT. This routine contains a single long loop that has bulk of the computation. This routine contains several arrays that are involved in irregular memory accesses. Since the routine uses pointers, the compiler assumes aliasing and cannot reason the validity of vectorization. The routine is therefore not vectorized by the compiler. However, we know that it is safe to vectorize the long loop in the routine since no two pointers map to the same memory and so we can safely vectorize (vectorization generates correct result) the loop with the use of either a pragma to ignore vector dependences or by qualifying the pointers with the restrict keyword or using specific command-line options. Note that the loop of concern here would involve gather/scatter operations once vectorized and also predication support since it contains some conditional code. Again, such support in AVX-512/SVE deems it profitable to vectorize the loop. Table VI summarizes the analysis report using our proposed metric. We observe the following:

Proc	Source	BW$_{obs}$ (GB/s)	lat$_{avg}$ (ns)	n$_{avg}$	Opt: Performance
SKL	base	37.9 (30%)	93	2.29	**Vect: 2x**
	+ vect	**46.8 (37%)**	95	**2.89**	**2-way HT: 1.4x**
	+ vect, 2-ht	**58.5 (46%)**	98	**3.73**	-
KNL	base	78.2 (19%)	183	3.49	**Vect: 5.76x**
	+ vect	**130.6 (33%)**	187	**5.96**	**2-way HT: 1.17x**
	+ vect, 2-ht	**233.6 (58%)**	199	**11.34**	4-way HT: 1x
A64FX	base	69.3 (7%)	144	0.81	**Vect: 3.83x**
	+ vect	**102 (10%)**	146	**1.21**	-

TABLE VI
SUMMARY OF PROGRAM OPTIMIZATIONS IN PENNANT

SKL: On SKL, the unoptimized version of PENNANT has an average MSHRQ occupancy of 2.29. Clearly, the nature of code in PENNANT is the limiting factor to MLP generation. We therefore force vectorization by ignoring vector dependences as mentioned above. *This gives a performance improvement of 2x - such a gain was expected following our recipe since the core can support such an increase in MLP.* The vectorized code reports an average MSHR occupancy of 2.89, implying still more potential for performance improvement. *We thus use 2-way hyperthreading and as expected see a performance benefit.* The improvement is 1.4x.

KNL: On KNL, the base version reports an MSHRQ occupancy of 3.49. *We force vectorization and in this case, observe a performance improvement of 5.76x as predicted by our recipe.* The improvement is very significant and more pronounced than SKL - this is because the memory latency on KNL is twice that of SKL and thus being able to keep more memory requests in flight (through vectorization) has a more significant effect. Also, since KNL has less out-of-order capability, more MLP implies more improvements than SKL. The vectorized code has an average MSHR occupancy of 5.96, implying that there is still potential for performance improvement. *We therefore use 2-way hyperthreading and see a performance improvement of 1.17x.* At this point, the calculated average MSHRQ occupancy is 11.34. It is important to note that since PENNANT involves irregular memory accesses, it is limited by the L1 MSHRQ, which has 12 entries. We thus see PENNANT hitting a bottleneck in the core. *As expected, using 4-way hyperthreading yields no further performance benefit. This is another example where PENNANT only utilized 58% of peak memory bandwidth, but still could not be optimized further (at least with the help of optimizations that would further increase MLP).*

A64FX: On A64FX, unoptimized PENNANT reports a very low MSHRQ occupancy of 0.81. *We again force vectorization using pragmas and observe a bug jump in performance like KNL.* Also, the improvement is much larger than SKL given the less out-of-order capability than SKL.

D. CoMD

CoMD is a compute dominated application with only a small fraction of accesses going to memory. Almost all of the user time is spent in a single user function, eamForce(), which is not auto-vectorized by the compiler. This is because there is loop carried dependence preventing vectorization on the innermost loop, which is the compiler's favorite loop for auto-vectorization. Also, even if an outer loop was vectorized, there would be introduction of gather/scatter operations. To complicate compiler's task of auto-vectorization, there is conditional code in the loop body. However, the occurrence of gather/scatter or conditional code does not inhibit processors like Skylake, Knights Landing and A64FX from vectorization due to the presence of efficient gather/scatter and support for predication. Upon careful analysis, we find that the next to innermost loop is in fact safe to vectorize and adding pragmas allows to successfully vectorize those loops. Table VII presents a summary of our performance analysis and optimizations. We make the following observations from the table.

SKL: On SKL, CoMD reports a low average MSHRQ occupancy of 0.17. *Vectorization therefore achieves a significant performance improvement of 1.4x.* Vectorized code still shows considerable promise for improvement with an MSHRQ occupancy of 0.29. *Thus, 2-way hyperthreading achieves a further performance benefit of 1.22x.*

KNL: On KNL, the base version reports a similarly low MSHRQ occupancy of 1.17. *As expected, vectorization yields a performance benefit (1.35x).* However, the benefit from vectorization in this case is not as dramatic as in case of PENNANT. This is because CoMD unlike PENNANT has very few memory accesses (very low MLP), and so vectorization only leads to a small increase in MLP as compared to the increase observed in PENNANT. Low MSHRQ occupancy of 1.55 in the vectorized code offer promise for further performance improvement through MLP increasing optimizations. *Using 2-way hyperthreading does indeed provide a 1.52x performance improvement.* The MSHRQ occupancy sees a jump from 2-way hyperthreading, but at 3.76, is still far from causing the core to become the performance bottleneck. *Thus, using 4-way hyperthreading adds another 1.25x in performance.*

Proc	Source	BW$_{obs}$ (GB/s)	lat$_{avg}$ (ns)	n$_{avg}$	Opt: Performance
SKL	base	3.19 (3%)	82	0.17	**Vect: 1.4x**
	+ vect	**4.56 (4%)**	82	**0.29**	**2-way HT: 1.22x**
	+ vect, 2-ht	**7.8 (6%)**	82	**0.41**	-
KNL	base	26.88 (7%)	179	1.17	**Vect: 1.35x**
	+ vect	**35.39 (9%)**	180	**1.55**	**2-way HT: 1.52x**
	+ vect, 2-ht	**82.82 (20%)**	186	**3.76**	**4-way HT: 1.25x**
	+ vect, 4-ht	**141 (35%)**	190	**6.54**	-
A64FX	base	10.75 (1%)	142	0.12	**Vect: 1.24x**
	+ vect	**13.44 (1%)**	142	**0.16**	-

TABLE VII
SUMMARY OF PROGRAM OPTIMIZATIONS IN CoMD

A64FX: A64FX behaves very similarly to KNL here - unoptimized CoMD reports low MSHRQ occupancy indicating profitability of vectorization according to our recipe. *Upon vectorizing, we observe a considerable performance benefit of 1.24x.*

E. MiniGhost

MiniGhost contains a 27-point stencil computation. The routine contains a 3D loop nest, and the compiler automatically vectorizes the innermost loop. The loop body contains multiple access streams with unit-stride accesses, and MiniGhost therefore benefits significantly from the hardware prefetcher. Table VIII summarizes our finding based on our proposed metric. The following observations may be made:

SKL: On SKL, MiniGhost reports an observed bandwidth of 92.93 GB/s and an average MSHRQ occupancy of 7.07. Since the bandwidth utilization is already very high, our recipe deems it beneficial to perform loop tiling and blocking especially since stencils are known to benefit from loop tiling or blocking. *Tiling improves performance by 1.14x on Skylake.* However, we see that the bandwidth utilization goes up instead of going down. This is because although tiling does considerably reduce the total number of memory accesses (as we observe in the performance report from CrayPat), the rate of generating memory accesses goes up. After tiling, the MSHRQ occupancy stands at 10.32 and the bandwidth utilization is at 107.14 GB/s. Since MiniGhost benefits significantly from the hardware prefetcher, the limiting factor is L2 MSHRs, which is 16 in SKL. Even though the core can in this case support more MLP, MiniGhost seems to become limited by memory bandwidth being already very close to peak achievable bandwidth on SKL. We therefore expect limited gains from hyperthreading due to limitation from memory bandwidth. *This is indeed observed as hyperthreading provides a mere 1.02x performance gain.*

KNL: On KNL, the base version drives 232.96 GB/s of memory bandwidth. This corresponds to an average MSHRQ occupancy of 11.26. *As with SKL, tiling improves performance by 1.47x.* The improvement is higher in case of KNL because of the higher memory latency that is avoided for some of the accesses due to tiling. Post tiling, the MSHRQ occupancy stands at 12.79. Since KNL has more than twice as many MSHRs at L2 than at L1, the average MSHRQ occupancy does not reveal limitation by MSHRQ size or in other words, the core. We however, do not see a noticeable performance benefit from 2-way hyperthreading even though the MLP increases (to 13.74 from 2-way hyperthreading). This is because we observe a noticeable increase in the memory accesses due to contention between hyperthreads for L2/LLC cache occupancy. Finally, with 4-way hyperthreading too, we do not observe a performance improvement. This is again the effect of LLC cache contention or thrashing.

Proc	Source	BW$_{obs}$ (GB/s)	lat$_{avg}$ (ns)	n$_{avg}$	Opt: Performance
Skylake	base	92.93 (73%)	117	7.07	**Tiling: 1.14x**
	+ tiling	**107.14 (84%)**	148	**10.32**	2-way HT: 1.02x
KNL	base	232.96 (58%)	198	11.26	**Tiling: 1.47x**
	+ tiling	**260.8 (65%)**	201	**12.79**	2-way HT: 1x
	+ tiling, 2-ht	274.56 (69%)	205	13.74	4-way HT: 1x
A64FX	base	575 (56%)	179	8.38	**Tiling: 1.51x**
	+ tiling	**554 (54%)**	174	**7.85**	-

TABLE VIII
SUMMARY OF PROGRAM OPTIMIZATIONS IN MINIGHOST

A64FX: On A64FX, unoptimized MiniGhost utilizes 575 GB/s of memory bandwidth, corresponding to an MSHRQ occupancy of 8.38. A64FX benefits in the same way as KNL (and much more than SKL) given the higher memory latency as compared to SKL. As discussed earlier in Section III-C, tiling can potentially reduce MSHRQ occupancy while improving performance - the same is observed in this case.

F. SNAP

SNAP is a proxy application that is used to model the performance of a modern discrete ordinates neutral particle transport application. Among user functions, the execution time is largely dominated by a single user routine, *dim3_sweep*. This routine contains a multi-level loop nest that contains several innermost loops that have small trip counts. The compiler automatically vectorizes most of these small innermost loops. However, since the innermost loops are small, it is difficult for the hardware prefetcher to prefetch all the data and in a timely fashion. This creates an opportunity of software prefetching. Table IX summarizes our approach to performance analysis and optimization in SNAP. We make the following observations.

SKL: On SKL, the base version sees a bandwidth utilization of 58.2 GB/s and an average occupancy of 3.79. SNAP is not memory bound - this is because there is considerable computation interleaving memory accesses in the routine dim3_sweep and there is also significant cache reuse related to many temporary variables. As discussed, software prefetching could be helpful. On SKL though, we see only a 1% performance gain from software prefetching (although other platforms show more gains). This is due to additional overhead of prefetch instructions and also an aggressive hardware prefetcher being good enough. We observe that after prefetching, the average occupancy (3.87) is still considerably lower than the max occupancy. *Using 2-way hyperthreading on SKL gives another 1.03x gain.* We find that this smaller gain from hyperthreading can be attributed to considerably more cache miss rates due to hyperthreading.

Proc	Source	BW_{obs} (GB/s)	lat_{avg} (ns)	n_{avg}	Opt: Performance
Skylake	base	58.2 (45%)	100.1	3.79	Pref: 1.01x
	+ pref	59 (46%)	101	3.87	2-way HT: 1.03x
KNL	base	122.9 (31%)	167	5	**Pref: 1.08x**
	+ pref	**126.4 (32%)**	168	**5.2**	**2-way HT: 1.14x**
	+ pref, 2-ht	**166.4 (42%)**	172	**6.98**	4-way HT: 1.02x
A64FX	base	93.88 (9%)	145	1.1	**Pref: 1.07x**
	+ pref	**97.3 (10%)**	145	**1.2**	-

TABLE IX
SUMMARY OF PROGRAM OPTIMIZATIONS IN SNAP

KNL: On KNL, the base version achieves a bandwidth of 122.9 GB/s. This is much more than SKL due to more cores. The average MSHRQ occupancy is 5, so sufficient room for optimizations. *Software prefetching yields a considerable performance improvement (1.08x) in KNL*, also increasing the average MSHRQ occupancy to 5.2. *2-way hyperthreading followed by prefetching yields another substantial performance gain of 1.14x as expected from the observed value of our metric.* This gain is despite increase in cache misses. Since the average occupancy after 2-way hyperthreading is still 6.98, our metric encourages further improvement. So, we use 4-way hyperthreading. *This again yields another 1.02x improvement.* Again, the gain is reduced by increased cache misses.

A64FX: Given the low MSHRQ occupancy, software prefetching is deemed profitable on A64FX. *The performance improvement seen is similar to that in KNL, indicating correct guidance from our proposed metric of MSHRQ occupancy.* As seen from Table IX, the MSHRQ occupancy is still very low which indicates a lot of scope for performance optimization. However, Figure 1 does not indicate any specific optimizations to apply in this case. At this point, we apply our own intuition guided by performance counters. In SNAP, we observed a particular loop that consumed most of the cycles in dim3_sweep. Upon analysis, we found that this loop performs particularly poorly due to automatic loop fusion by the compiler, which introduces store-to-load forwarding that is not well supported by A64FX. In this case, adding directives to disallow loop fusion improved the loop of concern by 4x and the overall application by 20%. This is an example to demonstrate while a recipe is a helpful guide, user intuition is never unimportant.

G. MSHRQ Occupancy and Compute Bound Applications

We emphasized the need for user intuition in Section III-D to identify bottlenecks when our recipe does not result in maximizing MLP. We discussed that optimizations suggested in such cases will still improve performance, but not enough to achieve peak MLP. These scenarios are especially common in compute-bound applications that have limited memory accesses. It may be argued then that in such compute-bounded applications that usually have very low MLP, almost all optimizations will yield performance benefit and thus the recipe is neither particularly valuable in identifying optimization and nor in identifying bottlenecks to achieve peak MLP. For instance, it is evident that vectorization will help a compute-bound application like CoMD. But, we make an important argument here. In such cases, we determine an application to be compute bound in the first place if it utilizes less than peak bandwidth *and its MSHRQ is not full*. In upcoming processors with HBM2e/3, L2 MSHRQ becomes full prior to achieving peak bandwidth even for streaming applications. So, in summary, MSHRQ utilization is full proof in judging whether an application is indeed compute bound and thus the utility of vectorization and other optimizations.

H. MSHRQ Occupancy and GPUs

While this work is focused on CPUs, GPUs too rely on MSHRs in the same way as CPUs. In contrast to CPUs, GPUs rely heavily on many concurrent threads/warps for latency hiding. It is this feature in GPUs that allows to generate significant MLP (generally more than CPUs). As a result, analyzing the occupancy of the MSHRQ, which tracks all the outstanding memory requests from all the concurrent threads, could be directly useful in understanding performance bottlenecks and guiding optimizations. For example, apart from applying traditional optimizations as discussed in this work, a low MSHRQ occupancy could imply increasing number of concurrent threads/blocks, which could be achieved by reducing register usage per thread or amount of shared memory used per thread block. Alternatively, a high MSHRQ occupancy could imply (increased) use of shared memory to improve performance. We deem this as an interesting area for future investigation.

V. CONCLUSIONS

Given the increased microarchitectural complexity and diversity, the performance tools need to simplify the task of performance analysis and optimization for the average user. We thus identify MSHRQ occupancy as a single comprehensible metric for performance analysis and program optimization. This metric is directly related to memory-level parallelism seen by an application on a processor, and importantly, can be easily calculated (using a minimal set of program counters easily found on all contemporary processors) based on Little's law. We further show that the average MSHRQ occupancy metric is particularly useful in assessing the utility of any program optimization that the user may consider, and we also provide a recipe to that end. To show the effectiveness of our recipe, we discuss six case studies on three different processors and find that the guidance given in terms of when an optimization will be performant or not is indeed very appropriate. We believe this recipe is first of its kind and hope to see it evolve as a useful tool in the hands of future users and researchers attempting performance optimization.

References

[1] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[2] Intel, "Intel®vtune™profiler," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[3] T. Gruber, "Measuring flop/s on intel haswell platforms." [Online]. Available: https://github.com/RRZE-HPC/likwid/wiki/FlopsHaswell

[4] M. Gottscho, S. Govindan, B. Sharma, M. Shoaib, and P. Gupta, "X-Mem: A Cross-Platform and Extensible Memory Characterization Tool for the Cloud," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 263–273.

[5] H. Servat, J. Labarta, H. Hoppe, J. Gimenez, and A. J. Pela, "Integrating memory perspective into the bsc performance tools," in *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, 2017, pp. 231–232.

[6] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, p. 287?311, May 2006. [Online]. Available: https://doi.org/10.1177/1094342006064482

[7] X. Liu and J. Mellor-Crummey, "A data-centric profiler for parallel programs," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[8] ——, "Pinpointing data locality bottlenecks with low overhead," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 183–193.

[9] S. Kaufmann and B. Homer, "Craypat - cray x1 performance analysis tool," *Cray User Group*, May 2003.

[10] A. Yasin, "How tma addresses challenges in modern servers and enhancements coming in icelake," in *Scalable Tools Workshop*, 2018.

[11] P. Drongowski, "Instruction-based sampling: A new performance analysis technique for amd family 10h processors," 2007.

[12] Intel, "Intel®microarchitecture code named skylake-x events." [Online]. Available: https://download.01.org/perfmon/index/skylake_server.html

[13] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[14] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, and H. Vincenti, "Applying the roofline performance model to the intel xeon phi knights landing processor," in *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds. Cham: Springer International Publishing, 2016, pp. 339–353.

[15] K.-H. Kim, K. Kim, and Q.-H. Park, "Performance analysis and optimization of three-dimensional fdtd on gpu using roofline model," *Computer Physics Communications*, vol. 182, no. 6, pp. 1201 – 1207, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465511000452

[16] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline model toolkit: A practical tool for architectural and program analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, 2015, pp. 129–148.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002. [Online]. Available: https://doi.org/10.1109/2.982916

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[19] J. L. Gustafson, *Little's Law*. Boston, MA: Springer US, 2011, pp. 1038–1041.

[20] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 40–52. [Online]. Available: http://doi.acm.org/10.1145/106972.106979

[21] S. Mehta, Z. Fang, A. Zhai, and P.-C. Yew, "Multi-stage coordinated prefetching for present-day processors," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 73–82.

[22] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang, "Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, jan 2015. [Online]. Available: https://doi.org/10.1145/2687356

[23] S. Mehta, G. Elsesser, and T. Greyzck, "Software pre-execution for irregular memory accesses in the hbm era," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 231–242. [Online]. Available: https://doi.org/10.1145/3497776.3517783

[24] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91. New York, NY, USA: ACM, 1991, pp. 30–44. [Online]. Available: http://doi.acm.org/10.1145/113445.113449

[25] S. Mehta, G. Beeraka, and P.-C. Yew, "Tile size selection revisited," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 35:1–35:27, Dec. 2013.

[26] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *LCPC*, 1993.

[27] U. Hanebutte and J. Hemstad, "Isx: A scalable integer sort for co-design in the exascale era," in *2015 9th International Conference on Partitioned Global Address Space Programming Models*, 2015, pp. 102–104.

[28] M. Heroux and J. Dongarra, "Toward a new metric for ranking high performance computing systems," 2013.

[29] C. Ferenbaugh, "The pennant mini-app," 2016. [Online]. Available: https://github.com/lanl/PENNANT

[30] "Comd: Classical molecular dynamics proxy application," 2013. [Online]. Available: https://github.com/ECP-copa/CoMD

[31] "minighost 3d halo-exchange mini-application," 2016. [Online]. Available: https://github.com/Mantevo/miniGhost

[32] "Snap: Sn (discrete ordinates) application proxy," 2020. [Online]. Available: https://github.com/lanl/SNAP

[33] V. Weaver, "uarch-configure," 2020. [Online]. Available: https://github.com/deater/uarch-configure

[34] Z. Fang, B. Zheng, and C. Weng, "Interleaved multi-vectorizing," *Proc. VLDB Endow.*, vol. 13, no. 3, p. 226?238, Nov. 2019. [Online]. Available: https://doi.org/10.14778/3368289.3368290

[35] S. Eyerman, W. Heirman, K. D. Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[36] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243181

[37] V. Viswanathan, K. Kumar, T. Willhalm, P. Lu, B. Filipiak, and S. Sakthivelu, "Intel®memory latency checker v3.8," 2020.

[38] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The nas parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, p. 63–73, sep 1991. [Online]. Available: https://doi.org/10.1177/109434209100500306

[39] Intel, "Intel® xeon phi™coprocessor system software developers guide," Mar. 2014.