Analyzing and Exploiting Memory Hierarchy Parallelism With MLP Stacks

Adnan Hasnat, Wim Heirman, and Shoaib Akram

***Abstract*—Obtaining high instruction throughput on modern CPUs requires generating a high degree of memory-level parallelism (MLP). MLP is typically reported as a quantitative metric at the DRAM level. However, understanding the reasons that hinder memory parallelism requires more insightful metrics and visualizations. This paper proposes a new taxonomy of MLP metrics, splitting MLP into core and prefetch components and measuring both miss and hit cache level parallelism. Our key contribution is an MLP stack, a visualization that integrates these metrics, and connects then to performance by showing the CPI contribution of each memory level. The stack also shows speculative parallelism from dependency-bound and structural-hazard-bound loads. We implement the MLP stack in a processor simulator and conduct case studies that demonstrate the potential for targeting software optimizations (e.g., software prefetching), and hardware improvements (e.g., instruction window sizing).**

***Index Terms*—Performance analysis, memory-level parallelism, bottleneck identification, cache hierarchy, prefetching.**

I. Introduction

MODERN Cpus exploit memory-level parallelism (MLP) to hide the long latency of memory accesses and improve instruction throughput. Specifically, CPUs use large instruction windows and speculative execution to issue independent load/store instructions to memory. Further, the memory system uses non-blocking caches, multi-banking, and prefetching to resolve multiple requests simultaneously. Application programmers and compiler writers expose MLP through software prefetching, vectorization, and loop unrolling. Thus, quantifying, understanding, and extracting MLP is paramount to high-performance code and hardware development.

To better understand MLP across the memory hierarchy and factors inhibiting it, this work proposes a visualization called MLP stack, similar in spirit to the well-known CPI stack [1]. We construct MLP stacks from a rigorous analysis of parallelism in memory requests across the memory hierarchy and connect it to the program's CPI and run-time.

MLP is the average number of useful accesses to DRAM across cycles with at least one pending access. Simply measuring this metric, however, does not provide an insightful picture. We identify four aspects that motivate our work and that we incorporate in our visualization. ❶ First, an architect or performance engineer must know if improving MLP improves the program's performance (i.e., connecting MLP to performance). Caches and speculative execution hide memory latency, and thus, improving MLP only benefits if the system struggles to hide the latency of cache misses. ❷ Second, it is important to understand the performance impact of cache misses and the resulting cache-level parallelism. Since DRAM accesses result from misses across all cache levels, DRAM MLP is limited by the number of concurrent misses non-blocking caches can support, which is limited by the number of miss status handling registers (MSHRs). Further, shared cache accesses increasingly incur high latency due to a complicated interconnection network [2]. Hence, overlapping shared cache accesses is equally important. ❸ Third, since prefetchers inject extra memory accesses into the system, it helps to understand the MLP impact of core and prefetcher accesses separately. ❹ The final aspect is pinpointing MLP bottlenecks. Attributing a factor to each bottleneck proportional to its significance provides useful insight. Our visualization incorporates these four aspects to enable targeted hardware and software optimizations.

Our proposed MLP stack integrates MLP across the memory hierarchy and the issue queue to provide a better insight into the MLP of a program. Along with DRAM MLP, we quantify and show cache-level parallelism (CLP) split into miss and hit CLP, developing a new MLP taxonomy. Our stack also shows the cycles (or run-time) spent waiting for accesses to each cache level and DRAM, indicating the degree to which out-of-order (OOO) execution hides memory latency. Finally, our proposed stack helps visualize MLP improvements if the OOO CPU could resolve specific hazards inhibiting load execution.

We implement the MLP stack in the Sniper architectural simulator [3] and explore case studies on real programs. These case studies provide insight into MLP across the memory hierarchy and inform computer architects and programmers where to target optimizations and understand their true impact.

II. Related Work

MLP is an established metric, and much prior work aims to exploit it. We discuss the most relevant works. Chou et al. [4] propose a framework for studying MLP called the epoch model, which breaks execution down into windows based on certain microarchitectural events. Their proposed stack shows MLP bottlenecks, but does not connect them to performance. They also do not study cache impacts. Mehta et al. [5] use performance counters and Little's law to quantify MLP on real hardware. They connect peak MLP to MSHR capacity, and target software optimizations based on the gap between observed and peak MLP. We use architecture simulation to provide a detailed view of the processor's execution and avoid their theoretical assumptions. Tang et al. [2] propose a compiler approach to co-optimize cache and memory-level parallelism. Finally, Carlson et al. [6] propose epochs per kilo instructions (EPKI) as a metric to understand the performance impact of MLP. They propose epoch profiles that connect instruction window size and cache capacity to EPKI, whereas we propose an intuitive visualization for understanding MLP.

Received 11 March 2025; accepted 4 April 2025. Date of publication 8 April 2025; date of current version 29 April 2025. *(Corresponding author: Shoaib Akram.)*

Adnan Hasnat and Shoaib Akram are with the School of Computing, Australian National University, Canberra, ACT 2601, Australia (e-mail: adnan.hasnat@anu.edu.au; shoaib.akram@anu.edu.au).

Wim Heirman is with Intel Belgium, 2550 Kontich, Belgium (e-mail: wim.heirman@intel.com).

Digital Object Identifier 10.1109/LCA.2025.3558808

III. Visualizing MLP With MLP Stacks

We first provide a new MLP taxonomy for OOO CPUs to supplement traditional (average DRAM) MLP. We then build our visualization, the MLP stack, that visualizes different types of MLP and connects them to their performance impact.

A. MLP Taxonomy

1) Traditional MLP: At any given time t, one can directly measure the number of concurrent DRAM accesses, giving us the instantaneous MLP, $MLP(t)$. Measuring $MLP(t)$ at all points in time throughout the program's execution provides a complete view of MLP, but it is tedious to collect and analyze so many points. Thus, traditionally, the notion of average MLP is suitable, which we denote MLP, defined as the average number of accesses to DRAM per cycle where there is at least one access to DRAM. If $T(DRAM)$ is the number of cycles spent accessing DRAM, then the formula for MLP is below.

$$MLP = \frac{\sum_t MLP(t)}{T(DRAM)} \tag{1}$$

The above definition deliberately excludes idle cycles (when there is no access to DRAM) in the denominator. Considering idle cycles decreases MLP, which is misleading. DRAM may be idle due to desirable reasons, e.g., high cache locality. The processor waits for memory (i.e., memory bound) when there is at least one access to DRAM, which is when we want to overlap memory accesses. This denominator implies, $MLP \geq 1$.

2) Extending MLP for Prefetches: To isolate the effect of prefetches, we split MLP into parts due to core-generated (MLP_C) and prefetch-generated (MLP_P) accesses. We make the denominators we average over for MLP_C and MLP_P the same, i.e., time spent accessing DRAM, whether a core or prefetch. This choice splits MLP neatly into two summands and is also desirable because we want to overlap core accesses with *both* core and prefetch accesses. Hence, we consider cycles when either core or prefetch access is happening. We note that some prefetched data is never used by the core (useless prefetches). We split prefetch MLP further into useful and useless prefetch MLP.

3) Extending MLP for Caches: We now quantify the parallelism of cache accesses, considering both cache hit and miss parallelism. The former determines the overlapping latency of cache accesses, and the latter indicates the MSHR contention. The total cache access parallelism includes both hits and misses. We define the instantaneous versions of these metrics below.

- $TCLP(Ln, t)$: total (hits and misses) number of accesses at level Ln and time t,
- $MCLP(Ln, t)$: pending misses at level L and time t,
- $HCLP(Ln, t)$: pending hits at level L and time t.

We define the core and prefetch components of the above CLP metrics to isolate accesses attributed to the core and prefetch separately. For example, $HCLP_C(L3, t)$ is the hits that are core accesses at level $L3$ and time t. We use $XCLP$ to denote the specific type of CLP given by an arbitrary variable X. For example, $X = M$ gives $MCLP$, $X = H$ gives $HCLP$.

However, we must choose a suitable denominator when computing an average for these metrics. The most obvious choice is to average each of the $XCLP_i(Ln, t)$s over the number of cycles where it is at least one. However, this would cause the denominators applied to different levels to be different, and we would lose the following properties obeyed by the instantaneous versions of the metrics:

- An access at level Ln must also be pending miss at all previous levels Lk, we have $TCLP_C(Lm, t) \leq TCLP_C(Ln, t)$ and $MCLP_C(Lm, t) \leq MCLP_C(Ln, t)$ for $m > n$. Prefetch CLP does increase at any levels where new prefetches are actively injected but decreases at any levels beyond that, implying $MLP(t) \leq MCLP(Ln, t)$ for all n, since any access that reaches DRAM is pending as a miss at all prior levels,
- $TCLP(Ln, t) = MCLP(Ln, t) + HCLP(Ln, t)$.

Losing these properties makes the metrics less intuitive by obscuring the expected relationships between them. In consequence, building a stack that conveys a cohesive picture becomes hard. Hence, we use the following denominator for all averages: the total time spent accessing any level of the memory hierarchy, which we denote $T(Hier)$. Doing this resolves the issue but requires us to use the same denominator for DRAM MLP, i.e., $T(Hier)$ instead of $T(DRAM)$. Thus, unlike the prior art, the denominator choice in our MLP definition considers the memory hierarchy holistic rather than DRAM alone, enabling us to propose a cohesive visualization of MLP metrics discussed so far. Ultimately, we have:

$$XCLP(Ln) = \frac{\sum_t XCLP(Ln, t)}{T(Hier)}$$

We also redefine MLP at the DRAM level to:

$$MLP = \frac{\sum_t MLP(t)}{T(Hier)}$$

4) Speculative MLP: We consider the MLP of loads unable to issue in any cycle due to dependent address computation (dependency bound or DP-bound) or resource starvation (structural hazard bound or ST-bound). We estimate extra MLP such loads could provide, gaining insight into the type of optimization that can help. We hence define average DP-bound and ST-bound speculative MLP as the average number of loads in the issue queue that are either DP or ST-bound per cycle where there is at least one access to the memory hierarchy.

B. Building MLP Stacks

We now combine different MLP metrics in the form of an MLP stack. We also interpose MLP in our visualization with the CPI stack [1] to see its relationship to performance. A CPI stack attributes portions of the program's CPI to specific bottlenecks, including time spent waiting for access to various memory levels. Combining CPI with MLP provides a first-order indication of MLP's connection to performance, as we can observe the time lost waiting for memory accesses. The MLP stack shows that insufficient MLP is the reason for the inability of the CPU to overlap memory access latencies.

We build separate MLP stacks for hit, miss, and total MLP. Fig. 1 shows a canonical MLP stack. The boxes in each stack represent different components of CPI and their corresponding MLP. For example, `DRAM` shows the contribution of DRAM accesses to CPI on the Y-axis, and DRAM MLP on the X-axis. Similarly, the stack shows different cache components of CPI and their CLP. Total stack shows TCLP, miss stack shows MCLP and hit stack shows HCLP. Speculative MLP represents hidden MLP due to non-issued loads. Finally, unrelated to MLP, compute shows the CPI of non-memory components.

In each graph, the horizontal dimension provides information about MLP. The DRAM and cache components in the horizontal dimension

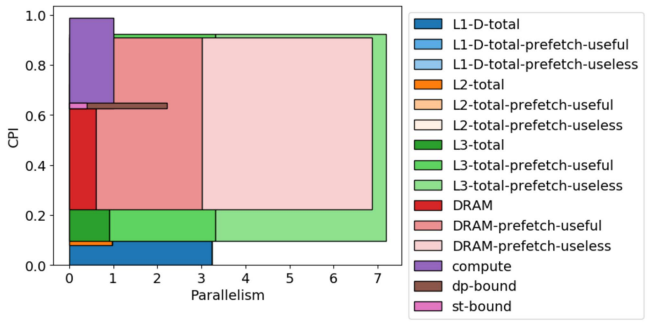

Fig. 1. Canonical MLP stack. The X-axis showing memory-level parallelism, and the Y-axis shows different components of program's CPI.

have a width corresponding to their average parallelism. This parallelism is also split horizontally between the core, useful prefetch, and useless prefetch components using different brightnesses of colors, with the total width indicating the total parallelism (core + prefetch) at that level. In the miss stack, the parallelism shown horizontally is MCLP, which indicates MSHR occupancy, while in the hit stack, it is HCLP, which represents overlapping accesses to that level.

Meanwhile, the vertical dimension provides important information about CPI. The DRAM and cache boxes have heights equal to the total number of cycles with at least one access at that level. Then, the vertical position of each of these boxes corresponds to its CPI component, with each box overlapping with the next and the positions being such that the distance from the bottom of each box to the bottom of the next box, i.e., the exposed vertical distance, is equal to the CPI at that level. For instance, the L1 box begins at vertical position 0 while the L2 box begins at vertical position $CPI[L1]$ (making the exposed height of the L1 box $CPI[L1] - 0 = CPI[L1]$). The L3 box begins at vertical position $CPI[L1] + CPI[L2]$ (so the exposed height of the L2 box is $CPI[L1] + CPI[L2] - CPI[L1] = CPI[L2]$) and so on. The DRAM box is then overlapped by the compute box, whose height represents the remaining non-memory CPI. The hidden part of the height of each box then represents the time when that memory component is active but not seen by the CPU due to its overlap with computation.

For the speculative MLP components, identifying ready but non-issued (ST-bound) and non-ready (DP-bound) loads in a cycle is straightforward. The width of `st-bound` and `dp-bound` in Fig. 1 shows their MLP. To set the height of these components, we measure the time these unresolved loads in the issue queue contribute to the total execution time. In the original CPI stack, this height is part of the compute CPI component (the purple box).

IV. Evaluation

A. Methodology

We implement MLP stacks in Sniper v8.0, modifying its detailed (ROB) model. During each simulation interval, we iterate over entries in the instruction window to identify executing loads, including where they hit in the hierarchy, useful/useless prefetches, and loads unable to issue due to structural hazard or dependence. We find the denominator for calculating average MLP by counting the total number of cycles with access to any memory level. We then use the formulae above to compute the MLP metrics for the three stacks.

We use the v1.3 release of `GAPBS` benchmarks [7]. We use a uniform-random graph with 2^{16} vertices and default node setting. We exclude the graph generation from the simulation. We show results for a memory-intensive and a compute-intensive kernel, namely, Betweenness Centrality (`BC`) and PageRank (`PR`). We compile the benchmarks with GCC 11 using production compiler settings. We also use a sparse matrix-vector multiplication (`SPMV`) kernel with a 256 MB vector size for a case study involving software prefetching.

We simulate the Intel Gainestown CPU used in the 5500 series Xeon cores based on the Nehalem microarchitecture. We use an MSHR capacity of 16 across all cache levels and a linear stride prefetcher at L2. We use a sampling frequency of one sample per 16 ROB simulation intervals, resulting in a less than 1% accuracy loss compared to sampling every interval.

B. Case Studies

The top row in Fig. 2 shows the total, hit, and miss MLP stacks (no prefetching) for `BC`. Looking at the total MLP stack, we observe limited DRAM MLP but high L1 MLP. One key insight the MLP stack provides is that improving the L1 and L2 MLPs brings limited benefits because of their negligible contribution to CPI. An engineer must focus on exploiting the L3 and DRAM MLP more. We also observe that MLP at these levels is below the peak achievable MLP of 16 (i.e., the MSHR capacity). Comparing the miss and hit MLP stacks informs us it may help more to focus on reducing the latency of concurrent cache hits than enabling more miss-level parallelism. We also do not see much MLP among loads that are ready but unable to issue due to structural hazards. On the other hand, MLP is high among data-dependent loads, and microarchitectural optimizations must focus on issuing as many of these dependent loads as possible to exploit more MLP.

Fig. 2(d), (e), and (f) shows the MLP stack of `BC` with `O0`, `O0` with prefetching, and `O3` with prefetching. MLP is exploited much better with `O3` optimization in Fig. 2(a). Enabling hardware prefetching in `O0` brings limited benefits in terms of MLP as there is not much MLP to begin with, showing the interaction of compiler optimizations and hardware prefetching at the memory level. In Fig. 2(f), we observe an increase in MLP at L2, L3, and DRAM levels due to hardware prefetching and wasted MLP due to useless prefetches. These stacks provide insight to the architect about the usefulness of prefetching, and its interaction with the memory hierarchy. Note that run-time (Y-axis) is a better metric for comparing optimizations that change the instruction count. We can trivially change the Y-axis to run-time from CPI if needed. Finally, Fig. 2(g) shows the MLP stack of the compute-bound `PR` with high speculative MLP but negligible DRAM CPI.

Fig. 2(h) and (i) shows the MLP stack for `SPMV` with hardware prefetching (h) and with software prefetching as well (i). We do a software prefetch at a stride of 4 when accessing the vector with non-zero matrix values. We hypothesize that additional software prefetches inject additional DRAM MLP, which should be visible in the graph. We only show the total MLP stack because cache MLPs are obscured entirely by DRAM in the miss and hit stacks. Instead, we observe a dramatic increase in L1 MLP in Fig. 2(i) and a reduction in DRAM MLP. This result is because software prefetching increases the L1 hit rate and reduces the number of accesses to DRAM. If we observe the DRAM MLP as a quantitative metric in isolation, we may conclude that performance and memory behavior degrades due to software prefetching. However, observing L1 MLP in the proposed stack provides more insight into the true impact of software prefetching.

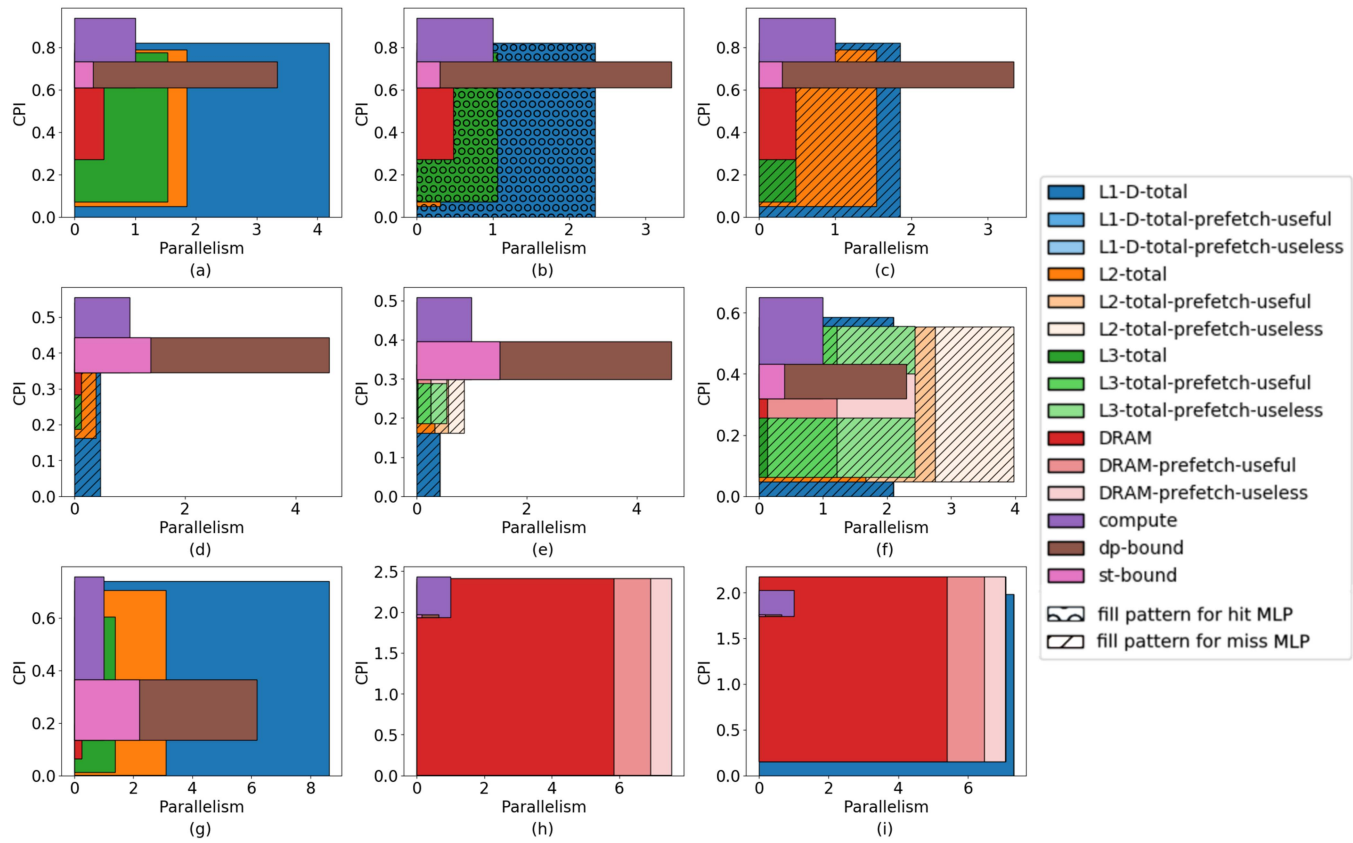

Fig. 2. Top row shows total (a), hit (b), and miss (c) MLP stacks for BC compiled with O0 and no prefetching. The middle row shows the miss MLP stack for BC with O0 (d), BC with O0 and hardware prefetching (e), and BC with O3 and hardware prefetching (f). The bottom row shows the total MLP stack for the compute-bound PR with O3 and no prefetching (g), SPMV with O3 and hardware prefetching (h), and SPMV with hardware and software prefetching (i).

V. Conclusion and Future Work

We have proposed a new visualization to enhance the performance architect's toolbox, namely the MLP stack. MLP stack connects MLP to performance and provides directions for targeting software optimizations and hardware improvements. Our case studies in this paper specifically show that the MLP stack offers greater insight into the changing MLP behavior across the memory hierarchy due to various software optimizations. Future work will use MLP stacks for balanced memory system design, and investigate multithreaded applications. We will also study new ways of exploiting speculative MLP.

References

[1] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2006, pp. 175–184.

[2] X. Tang, M. T. Kandemir, M. Karakoy, and M. Arunachalam, "Co-optimizing memory-level parallelism and cache-level parallelism," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2019, pp. 935–949.

[3] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 1–25, Aug. 2014.

[4] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proc. Int. Symp. Comput. Archit.*, 2004, pp. 76–87.

[5] S. Mehta, "Performance analysis and optimization with littles law," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2022, pp. 12–23.

[6] T. E. Carlson, S. Nilakantan, M. Hempstead, and W. Heirman, "Epoch profiles: Microarchitecture-based application analysis and optimization," *IEEE Comput. Archit. Lett.*, vol. 14, no. 1, pp. 30–33, Jan./Jun. 2015.

[7] S. Beamer et al., "GAPBS benchmark suite (source code)," 2025. [Online]. Available: https://github.com/sbeamer/gapbs