

Exploring Memory Level Parallelism For Data Intensive Applications

Abstract—Simulators and performance analysis tools are invaluable for understanding the behaviour of computer architectures and programs. In the ever-evolving landscape of computer science, these tools aid in decision making, optimisation, and the identification of performance bottlenecks. As the proportion of time spent waiting for memory accesses grows, understanding and being able to exploit memory-level parallelism (MLP) has become essential. A recently proposed tool for visualising the MLP of a program is MLP stacks [1]. This tool shows a lot of promise for aiding in program analysis and optimisation, however further testing and development is still needed.

In this paper we review some past developments in the exploitation of MLP and explore different forms of simulation and performance analysis tools. We then develop some microbenchmarks to test the validity of MLP stacks, and discuss the potential causes for the unexpected behaviour these microbenchmarks expose.

I. INTRODUCTION

Over the last few decades there have been many significant developments in computer architecture and design, from runahead execution, to prefetching, to on chip memory caches. While these developments have greatly improved performance, the ever increasing complexity can make it difficult for users and developers to understand what is going on and identify performance bottlenecks. This is where simulators and performance analysis tools come in to play.

Numerous simulators and performance analysis tools exist, each with different purposes in mind. One recently developed tool is MLP stacks [1]. MLP stacks provide a visual representation of the Memory-Level Parallelism (MLP) of a program, including the Cache-Level Parallelism (CLP) of each cache level and their contributions towards Cycles-Per-Instruction (CPI). This tool has significant potential for helping understand program behaviour and guiding optimisations, however it is still in the development stage and requires further testing.

In the following section of this report we discuss a number of papers about recent developments relevant to MLP. Subsequently, we examine simulators and performance analysis tools related to MLP stacks. Finally, we design and test some microbenchmarks to evaluate the validity of MLP stacks and expose potential flaws.

II. WORK RELATING TO MLP

With the gap between CPU and memory speed ever-widening, exploiting MLP has become increasingly important. The following studies discuss optimisations that help exploit MLP.

In their study, Chou et al. explore the impact of microarchitecture on MLP [2]. This study uses the epoch model to simplify calculations and avoid the need for cycle-accurate simulation to understand MLP. An epoch is a period of time of on-chip

computations followed by concurrent off-chip memory accesses. Longer epochs can allow for greater MLP and the authors identify four main reasons for an epoch to end. These are; issue window size and reorder buffer size limitations; serializing instructions; instruction fetch misses; and unresolvable branch miss-predictions. The study looks at how varying L2 cache size, reorder buffer and instruction window sizes, issue policy, and value prediction impact MLP and performance. The different issue policies considered are in order, in order loads, loads wait for earlier stores, in order branches, and runahead execution. Since out-of-order (OOO) processors are so common in high performance computing nowadays, the relevance of the performance of in order processors is limited, however it is helpful in understanding the impact of different optimisations on MLP and to inspire further optimisations. SPECjbb2000, SPECweb99 and a database benchmark are used to study the impact that varying these factors have on MLP and performance, and a stack visualisation is used to represent the contribution of each cause for an epoch to end. These experiments show that more relaxed issue policies are needed to exploit the benefits of large instruction windows and that serialising instructions are a major barrier to MLP. Additionally, the authors find it beneficial to increase the reorder buffer size beyond the instruction window size (which is more costly to enlarge). Furthermore, they find runahead execution to be very effective in increasing MLP as it removes the barriers of issue window size, reorder buffer size and serialising instructions.

Runahead execution is a significant optimisation that is discussed in detail by Mutlu et al. [3]. Runahead execution allows the processor to enter ‘runahead mode’ when a long latency instruction such as a cache miss is the oldest in the instruction window. The current state is checkpointed and the blocking instruction is removed, allowing independent instructions to enter the window and execute, increasing MLP. These executed instructions serve as useful prefetches once the original blocking instruction resolves and the state is returned to where it was checkpointed. Runahead execution requires a moderate amount of hardware support, but was found to improve IPC by 22% compared to an aggressive baseline processor that has a hardware prefetcher, and achieved an IPC similar to that of a much larger instruction window size [3].

Returning to Chou et al. [2], they observe that MLP is dependent on how load misses are clustered. For example, the miss rate of an application could be very low but MLP very high if misses occur close together. This suggests a possibility to improve performance by attempting to cluster independent misses within the same epochs - for example through prefetching. The paper makes some connection to performance, showing that for certain applications increasing MLP results in significant performance gains, however it does

not address multithreading or cache-level parallelism.

The introduction of prefetching was an important development in improving performance through MLP, and is discussed by Chen and Baer [4]. They explain that prefetching involves loading instructions from a lower to higher level in the memory hierarchy before they are needed. When done effectively the latency of prefetched instructions are overlapped with other instructions, using MLP to eliminate latency seen by the CPU. Software prefetching is usually implemented as a compiler optimisation, where predictions are made about when cache misses will occur, and prefetches inserted accordingly. This process often involves loop unrolling, and software prefetchers are unable to predict conflict misses. Since software prefetches are interpreted as regular instructions by the CPU, they incur some more overhead than hardware prefetchers. On the other hand, hardware prefetchers record access patterns and issue prefetches based on this. Since the hardware prefetcher has no knowledge about future instructions there is usually a delay in starting and stopping prefetching, and the ability to identify complex access patterns is limited compared to the software prefetcher. Chen and Baer [4] determine that using both hardware and software prefetching in combination could produce the best results. They propose that software prefetching could have the role of bringing data - such as large user data structures - from memory to the L2 cache, while hardware prefetching could bring this data up to the L1 cache when needed. This strategy makes use of the strengths while minimising the drawbacks of each type of prefetching, resulting in improved performance. The authors also suggest that having larger MSHRQ capacity in the lower level caches and being able to dynamically switch on and off hardware prefetching based on accuracy would greatly improve the effectiveness of prefetchers. The effectiveness of prefetching relies on the ability to have multiple memory accesses in parallel, and can have a big impact on the MLP of a program.

Recovery-free value prediction, proposed by Zhou and Conte [5], is another technique that uses prefetching to increase MLP and improve performance. Value prediction often requires expensive dedicated hardware to recover from mispredictions, but by restricting value prediction to prefetches this recovery mechanism becomes unnecessary. Having value prediction for prefetches allows for more prefetches to be issued, increasing MLP and improving performance for memory intensive workloads. The technique proposed involves executing instructions speculatively, then once the needed values are available, executing them again un-speculatively so that incorrect values are overwritten. The hardware alterations involved also allow for aggressive memory disambiguation, helping overcome load-after-store dependencies. This technique works well with runahead execution [3] as runahead execution allows for the pre-execution of independent instructions, while value prediction allows for the pre-execution of dependent instructions. The authors found that recovery-free value prediction resulted in a speedup of 3.2-24% for memory intensive benchmarks, but was less effective for computation intensive benchmarks. They suggest that dynamically turning on and off value prediction depending on how memory intensive the program is may lead to the best results and most effective exploitation of MLP.

Finally, an aspect that is often overlooked when considering MLP and prefetching is Bank-Level-Parallelism (BLP). Discussed by Lee et al. [6], DRAM chips consist of multiple banks that can be accessed concurrently, however if there are many requests to the same bank in series BLP is unable to be utilised. Additionally, for multicore processors, the order in which requests from different cores are served can destroy the BLP of an individual core. The authors propose two mechanisms to more effectively make use of BLP. These are BLP-Aware prefetch issue - which prioritises prefetches of a core to different banks - and BLP aware multicore request issue - which prioritises issuing prefetches of one core at a time. While it is not directly addressed in much discussion about MLP, BLP is an important and limiting factor in MLP.

III. SIMULATORS AND PERFORMANCE ANALYSIS TOOLS

Simulators and performance analysis tools are essential for computer architects and programmers to be able to make design choices and improve performance. Cycle accurate simulators provide an extreme level of detail regarding the behaviour of a program on a specific architecture. However, these simulators are very time and resource intensive, making them an impractical option when absolute accuracy and detail isn't required, for example when comparing high level architectural tradeoffs [7]. As architectures become more complex and the number of cores increases, this issue is further exacerbated. In response, other methods of simulation/performance analysis have been developed and some will be discussed in the following section.

One of these is interval simulation which was developed by Genbrugge et al. [7]. Interval simulation raises the level of abstraction to achieve much faster simulation while maintaining high accuracy. It does this by accurately modeling the memory hierarchy and branch predictor, but using a mechanistic analytical model for the core. This analytical model essentially estimates core-level performance by analysing the time between miss events (e.g. branch mispredictions, cache misses) as informed by the detailed modeling of the memory hierarchy and branch predictor. In this way a lot of unnecessary detail is saved as the simulator does not need to model behaviour within the core, resulting in the simulator being just one thousand lines of code. Experimentation with the SPEC CPU2000 benchmark, PARSEC benchmark suites and the M5 multi-core simulator on up to 8 cores showed an average error in comparison to a cycle accurate simulator of only 4.6%, but a speedup of 8-9x for multithreaded workloads and 15x for multiprogram workloads.

Genbrugge et al. [7] also mention a few other alternatives to cycle accurate simulation that have been explored. One of these is sampled simulation in which a sample of instructions are modelled using a cycle accurate simulator, and the findings extrapolated to apply to the program as a whole. Another is statistical simulation, which involves developing a statistical profile for a program that is used to create a synthetic clone with similar execution characteristics but far less instructions than the original program. Both of these approaches reduce simulation time by reducing the number of instructions, whereas interval simulation simulates all instructions with less detail.

The Sniper simulator, which is described by Carlson et al. [8] and has been used to implement MLP stacks, is based on the aforementioned interval simulation. This simulator is built onto Graphite parallel simulation infrastructure, adding interval simulation, a shared cache, and some modeling of time taken by the OS. Sniper allows for simulation of multithreading with different synchronisation protocols. In their paper the authors compare this method of simulation to cycle accurate simulation and the 1-IPC model. The 1-IPC model raises the level of abstraction even further than interval simulation, estimating that each instruction takes one cycle to execute except for memory accesses which are assigned a fixed average latency. However, as the authors observe, the 1-IPC model assumes in-order, scalar execution and so does not account for MLP or the ability of OOO-processors to hide latency through the overlapping of instructions. As such, Sniper was found to be on average more than twice as accurate as the 1-IPC model, and only 2-3x slower, with this difference diminishing as the number of cores increases.

In addition to simulators, performance analysis tools that provide specific information about program behaviour, bottlenecks, and possible optimisations are very useful. When considering the value and applications of MLP stacks it is helpful to have an understanding of other performance analysis tools available.

One such tool proposed by Yasin [9] is Top-Down Analysis. In their paper, Yasin identifies that while many performance metrics are available they can be difficult to interpret, and optimising an application's performance for a given architecture can be very challenging. Additionally, they found that previous performance tools used a "naive approach" of stall cycles = #miss events * miss latency which has the drawbacks of not accounting for the overlapping of miss latencies (MLP), as well as being limited to a set of predefined miss events, and assuming constant miss latencies (despite them being workload dependent). The aim of the Top-Down Analysis is to provide a practical method to quickly identify key bottlenecks in OOO-processors. This method involves recursively drilling down into the area identified as a bottleneck to help inform changes that will yield the greatest performance improvement. At the issue point, each pipeline slot is allocated to either Frontend Bound, Backend Bound, Bad Speculation, or Retiring based on if that instruction is allocated and retires/is discarded, or in the case of a stall whether it is caused by the backend or not. Some extra hardware is needed to allow for an additional eight performance events, however this is relatively inexpensive. These performance counters are used to determine the weighting of each category as you continue to drill down to the cause of the performance bottleneck. A few case studies demonstrate how addressing the bottleneck identified by Top-Down Analysis can yield improved performance, however the tool does not provide direction for how to actually address the bottlenecks. Furthermore, this method does not provide a full picture view of what is going on, which could be misleading and limit usefulness.

Alternatively, Eyerman et al. [10] propose an architecture that allows for the creation of accurate CPI stacks. CPI stacks break down the total average cycles per instruction of a program into a base component, as well as the contributions of various miss

events such as cache misses and branch mispredictions. This is useful for both computer architects and software developers to understand the behaviour of a program on a given processor and identify where the greatest latencies lie. CPI stacks are also a key component of MLP stacks. CPI stacks have an advantage over Top-Down Analysis [9] in that they allow for a better overall picture of performance, rather than just identifying the biggest bottleneck. As previously discussed, a naive approach of attributing CPI component as #miss events * avg miss latency is flawed and inaccurate for OOO-processors. In their paper Eyerman et al. [10] use interval analysis to better understand the characteristics of different miss events and how much they really contribute to overall CPI. Based on this they propose an architecture involving a Shared Frontend Miss Event Table that allows for the creation of accurate CPI stacks. They then compare the CPI stacks generated for a number of benchmarks by a simulator, the naive approach, their approach, and for an alternative architecture - the IBM POWER5. They found that their method had an average error of 2.5% compared to the simulator which was much lower than the others.

Both Top-Down Analysis and CPI stacks have the drawbacks of requiring additional architectural support, and not providing specific guidance to the user for how to improve performance. These drawbacks are addressed by Mehta [11] who aimed to create a transferable, useful, and straightforward performance analysis tool. In their paper, Mehta uses Little's Law with measured bandwidth and MSHR occupancy to estimate the MLP of a program, and provide a flowchart based on this to guide software optimisations of a program. A key aspect is that the performance counters needed for this tool are readily available in a range of existing processors, making it easy to implement. Some case studies are provided of how this tool can be used and was shown to be beneficial. However, the relationship between MLP and performance was unclear as some optimisations yielded huge performance increases but only slightly increased MLP, whereas others greatly increased MLP but didn't have much impact on performance. This makes sense as some optimisations may result in less cache misses which will improve performance but reduce MLP.

Similarly to Carlson et al. [8] and Yasin [9], Van de Steen et al. [12] observe that measuring the time spent waiting for memory is difficult due to MLP. Seeking a more efficient solution than simulating an application on various architectures, in their paper Van de Steen et al. [12] create an analytical model that profiles an application once, then can estimate the MLP on any architecture. The statistics are captured on a per-micro-trace basis to reduce profiling time and capture "burstiness" of MLP. The profiling considers the following aspects of the application: load spacing distribution, inter-load dependency distribution, reuse distance distribution, and stride distribution. Additionally, the model uses a prefetch table and stride distribution to model prefetching. From their benchmark testing this model has an average error of 3.3% while it is significantly faster than running simulations of multiple models. However, this paper does not address multicore processors/multithreading.

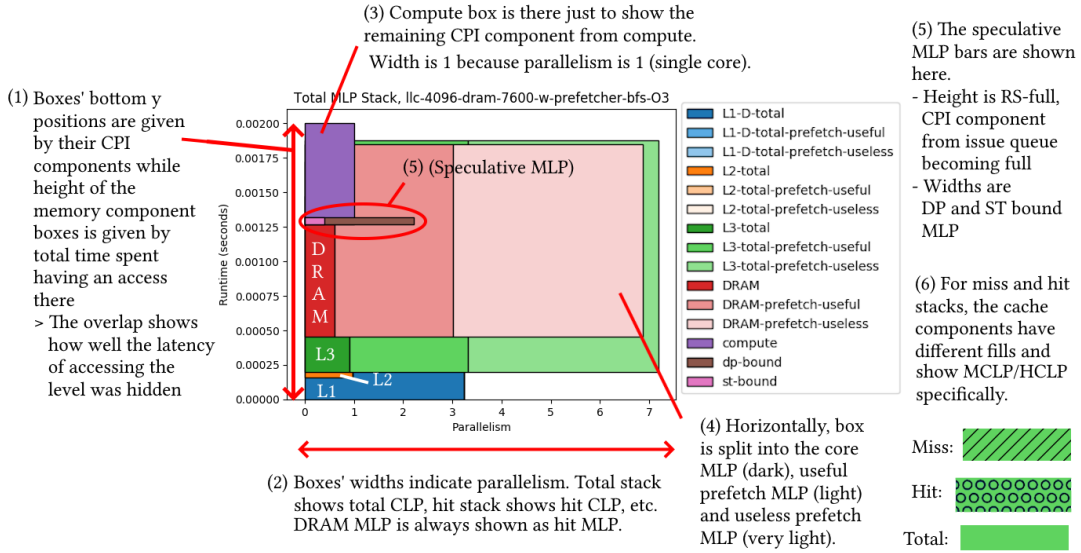


Fig. 1. MLP Stack Guide (sourced from Hasnat [1])

IV. MLP STACKS

MLP stacks are a recently proposed performance analysis tool that visually represent the MLP and CPI of a program [1]. MLP stacks have a lot of similarity to the CPI stacks presented by Eyerman et al. [10], but add an extra dimension of MLP along the x-axis. These stacks include cache-level parallelism, prefetches, and speculative MLP, and allow for separate miss and hit MLP stacks. See Figure 1 for a breakdown of how to interpret an MLP stack. In their paper Hasnat [1] modifies the sniper simulator [8] to include counters for MLP statistics that can be used to create MLP stacks for a range of architectures. Examples are then provided of MLP stacks generated for the SPMV and GAPBS benchmarks.

However, due to the complexity of these benchmarks it is somewhat difficult to determine whether the MLP stacks are accurate. Further testing is needed to validate the accuracy and usefulness of this performance analysis tool. To address this, we create some simple microbenchmarks with certain behaviours that should be observable in their MLP stacks.

V. MICROBENCHMARK DESIGN

In designing these microbenchmarks our aim is to be able to target different cache levels and control MLP. By "targeting cache level k " we mean that many misses will occur at level k that hit at level $k + 1$ - making the level k cache have a significant CPI component. To do this we implement a program that initialises and loops through a linked list of a given number of nodes (elements of the list). If we want to target the L1 cache, we can increase the number of nodes so that the working set is greater than the L1 cache, but less than or equal to the L2 cache size. Additionally, by using a linked list we create true data dependencies between each load (since to get the address of the next node, you must have the previous one), meaning MLP should be restricted to 1.

Initially we implement the linked list with each node pointing to the node sequentially next to it in memory (see

`inorder.c` A). However, to maximise the misses at the level being targeted we want to minimise spatial locality. As such we create a second version (`randbench.c` B) where during initialisation the order of the nodes is randomized. Finally, we want to be able to change the MLP of the program in a controlled manner, so we create a version that allows for multiple independent linked lists but otherwise behaves the same (see `high.c` C). The MLP of `high.c` should be proportional to the number of linked lists, and the behaviour of this benchmark with one linked list should be indistinguishable from `randbench.c`.

We then run these microbenchmarks, targeting each cache level on Sniper using the Gainestown configuration [13] with prefetching disabled, and use the appropriate post-processing scripts to obtain their MLP stacks as described in Hasnat [1].

Note: Although MLP stacks allow us to access information about hit, miss, and total CLP, when we refer to CLP in the following section we are talking about total CLP unless otherwise stated.

VI. RESULTS

Examination of the MLP stacks produced revealed unexpected results. The runs of `randbench.c` all had L1 CLP ≈ 2 , and for `high.c` with one linked list it was around 4 (see figures 2 3). Due to the dependencies between loads, both of these programs should have an L1 CLP = 1. This indicates that either there is something wrong with our microbenchmarks and how we ran them, or there is an underlying issue in the MLP stack accounting.

While we are unable to identify the cause of this unexpected behaviour, we proceed to systematically identify and investigate possible suspects, narrowing down the culprit.

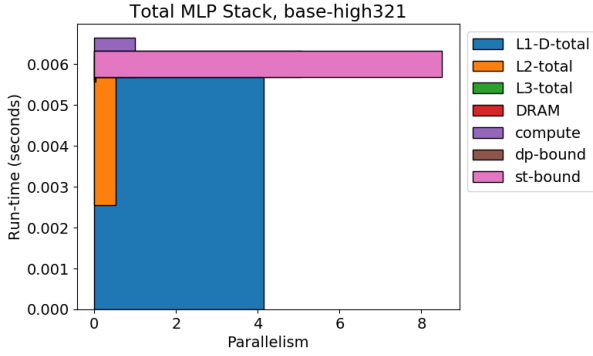


Fig. 2. Total MLP Stack from microbenchmark `high.c` with 32000 nodes and one linked list. Note the high L1 CLP.

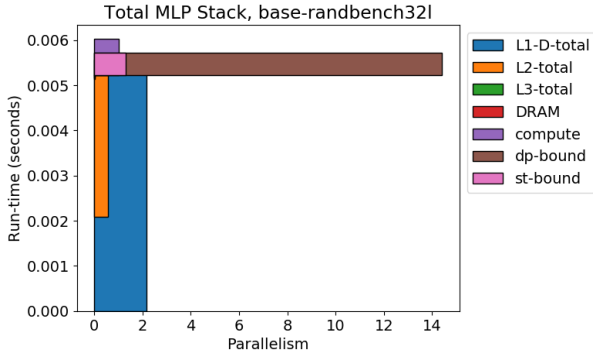


Fig. 3. Total MLP Stack from microbenchmark `randbench.c` with 32000 nodes. Note the high L1 CLP.

A. Microbenchmark Behaviour, Initialisation, and Compiler Optimisations

The first area we look into is reasons for the microbenchmark code to not be behaving as expected and produce high L1 CLP.

One of our first hypotheses is that the initialisation of the linked list is affecting the MLP stack. To address this we use Regions of Interest around the main loop in the code so that only it is simulated in detail and contributes towards the data analysed. While adding Regions of Interest results in some visible change in the MLP stacks (primarily in reducing the CPI contribution of compute), it does not significantly affect L1 CLP.

Secondly, we consider whether compiler optimisations might somehow be interfering. While compilers generally are unable to eliminate the true dependencies involved in linked list traversal, we try compiling the benchmarks without optimisations (using `-O0`) just to be sure. As expected we find little impact on the MLP stacks.

Furthermore, we generate the assembly code for `high.c` and have ChatGPT [14] describe what the program does and the expected L1 CLP it produces. The description aligns with our understanding of the program, and the predicted L1 CLP is around 1 as we'd intended. This indicates that the unexpected L1 CLP is not caused by the benchmark code or compiler optimisations.

To test this further we create a simplified version of the benchmarks called `simple.c` (see D) that still involves

linked list traversal. We try running this version with different numbers of nodes and iterations, including having the same number of nodes and iterations, and a small number of nodes and many iterations. We find that the reported L1 CLP is still greater than 1 although it varies with these different parameters, and generally has more similarity to `randbench.c` than `high.c` benchmark.

B. Choice of Denominator

In the original paper [1] it is noted that the choice of denominator when calculating MLP may not be ideal and could give misleading results. As such we consider whether this may be an explanation for the unexpected behaviour of our microbenchmarks.

As discussed by Hasnat [1], the standard formula used to calculate MLP is as follows:

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

Where $T(DRAM)$ is the total time spent accessing main memory, and $MLP(t)$ is the number of parallel accesses to main memory at time t .

The formula used to calculate CLP for the MLP stacks is the following:

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

Where $XCLP$ is either hit, miss, or total CLP at level Ln , and $T(Hier)$ is the total time spent accessing any level of the memory hierarchy.

Since every memory access will first access the L1 cache (either as a hit or miss), the denominator used is equivalent to the total time spent accessing the L1 cache. Hence in the case of L1 the formula used to calculate total CLP is equivalent to the standard definition of MLP for that level. As such, this choice of denominator presents no reason for calculated CLP to be higher than anticipated.

C. Accounting of CLP

Next we investigate the code responsible for calculating MLP in the ROB timer in Sniper. No issues are immediately apparent, so we include statements within the `doIssue` function to output the start and end time of each load issued. We then use a python script `mlp_calc.py` (generated by ChatGPT [14]) to process these outputs and calculate the total L1 CLP of the program according to the same formula used by Hasnat [1]. Again, since we are interested in L1 CLP, and every load will be an L1 miss or hit, we did not need to worry about where the load would actually hit. This way of accounting MLP is different from how it is done in the ROB timer with the function `countOutstandingMemop` which involves searching the ROB for loads during each execution interval and provides more information about hit and miss CLP at each cache level, as well as the effect of prefetchers and dependency/structurally bound instructions [1].

We find that these two methods of computing L1 CLP produce the same results for the benchmarks we tested. This

suggests that the issue is not caused by the MLP accounting done in the `countOutstandingMemops` function or post-processing done by the `mlpstack.py` script. The most likely remaining possibility is that there is a deeper flaw in the sniper script/`rob_timer.cc` file that also affected our accounting of MLP.

D. Logging entry→issued and entry→done

When creating the logging of load start and end times we first attempted to use entry→issued and entry→done as these values. We tried doing so in various locations within `rob_timer.cc` including inside the `issueInstruction`, `doIssue`, `doCommit`, and `countOutstandingMemops` functions. We were surprised to find that the same values were printed for every load. Most likely there was an issue in our logging statements that caused this, however there remains the concerning possibility that entry→issued and entry→done (which are used in accounting MLP in `countOutstandingMemops`) are not updated correctly.

VII. CONCLUSION

Through creating and running our microbenchmarks we expose unexpected behaviour in the MLP stacks. Specifically, when running two pointer chasing benchmarks that should produce identical observable behaviour and due to true data dependencies have an L1 CLP of 1, the reported L1 CLP was approximately 2 and 4 respectively. In response, we systematically identify and investigate potential causes of this unexpected behaviour. While we are unable to identify the specific cause, we believe there may be a flaw in the Sniper script that was added to track load instructions. Future work to verify this theory would involve running the benchmarks and using an existing tool such as VTune [15] to calculate L1 CLP and see if it agrees with or differs from the sniper results.

Additionally, once this issue has been addressed, future work would include writing microbenchmarks to test the validity of MLP stacks with prefetching enabled. Other related work could include doing case studies of how MLP stacks can be used to guide optimisations, and perhaps producing a flowchart similar to Mehta [11] to help users choose optimisations informed by MLP stacks.

In the ever changing world of computer science, simulators and performance analysis tools are drivers of development and efficiency. As the gap between time spent on-chip and accessing memory grows, it is increasingly essential to utilise MLP in high performance computing. MLP stacks are a recently proposed performance analysis tool that still require some testing and refinement.

Just as we had intended, the microbenchmarks we created may have exposed a flaw in the MLP stack model. However, more work is needed to determine the precise cause of our unexpected findings, and how to amend them. Testing and fine-tuning is an essential aspect of developing performance analysis tools to ensure they are appropriate, useful, and accurate. Being able to systematically identify, investigate, and address potential causes of unexpected behaviour is key to this stage

of development. Eventually, MLP stacks could become a key component of the developer's toolbox, helping users visualise the behaviour of their program and guide optimisations.

REFERENCES

- [1] A. Hasnat, "Analyzing and exploiting memory hierarchy parallelism with mlp stacks," 2024.
- [2] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," *International Symposium on Computer Architecture (ISCA)*, 2004.
- [3] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 129–140.
- [4] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes," in *Proceedings of 21 International Symposium on Computer Architecture*, 1994, pp. 223–232.
- [5] H. Zhou and T. Conte, "Enhancing memory-level parallelism via recovery-free value prediction," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 897–912, 2005.
- [6] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving memory bank-level parallelism in the presence of prefetching," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 327–336.
- [7] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, 2010.
- [8] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.
- [9] 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.
- [10] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 175–184, 2006.
- [11] S. Mehta, "Performance analysis and optimization with Little's law," *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022.
- [12] S. Van den Steen and L. Eeckhout, "Modeling superscalar processor memory-level parallelism," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 9–12, 2018.
- [13] Intel, "Intel® xeon® processor x5550," Online, 2009. [Online]. Available: <http://ark.intel.com/products/37106>
- [14] OpenAI, "Chatgpt (gpt-5) [large language model]," <https://chat.openai.com/>, 2025, accessed: Oct-2025.
- [15] Intel Corporation, "Intel vtune profiler," <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>, 2025, accessed: Oct. 25, 2025.

APPENDIX

A. *inorder.c*

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int nodes = 2048 ;
int iters = 1000000;

typedef struct Node {
    struct Node *next;
} Node;

void low() {
    // start of linked list , malloced space
    Node *array = malloc(nodes*sizeof(Node));
    if (array==NULL) {
        printf("malloc failed\n");
        return;
    }
    // initialising nodes
    for (int i=0;i<nodes-1;i++) {
        array[i].next = &array[i+1];
    }
    array[nodes-1].next = array;

    //iterate through linked list
    Node *p = array;

    for (int i=0; i<iters; i++) {
        p = p->next;
    }

    printf("low done\n");
    return;
}

//First arg is number of nodes*1000
int main(int arg, char *argc[]) {
    if (arg==1) {
        printf("no args\n");
        return 0;
    }
    nodes = 1000 * atoi(argc[1]);
    low();

    return 0;
}

```

B. *randbench.c*

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int nodes = 32000;
int iters = 1000000;

typedef struct Node {
    struct Node *next;
} Node;

void low() {
    // start of linked list , malloced space
    Node *array = malloc(nodes*sizeof(Node));
    if (array==NULL) {
        printf("malloc failed\n");
        return;
    }

    // Random permutation of nodes
    int *perm = malloc(nodes * sizeof(int));
    for (int i = 0; i < nodes; i++) perm[i] = i;
    for (int i = nodes - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        //swaps i and j
        int tmp = perm[i];
        perm[i] = perm[j];
        perm[j] = tmp;
    }

    // initialise nodes
    for (int i = 0; i < nodes - 1; i++) {
        array[perm[i]].next = &array[perm[i + 1]];
    }
    array[perm[nodes - 1]].next = &array[perm[0]];

    //iterate through linked list
    Node *p = array;

    for (int i=0; i<iters; i++) {
        p = p->next;
    }

    printf("low done\n");
    return;
}

int main(int arg, char *argc[]) {
    if (arg==1) {
        printf("no args\n");
        return 0;
    }
    nodes = 1000*atoi(argc[1]);
    low();
    return 0;
}

```

C. high.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int nodes = 32000;
int iters = 1000000;

typedef struct Node {
    struct Node *next;
} Node;

void high(int arrays) {
    Node *array[arrays];

    // creating array of pointers to linked lists
    for (int i=0; i<arrays; i++) {
        array[i] = malloc(nodes*sizeof(Node));
        if (array[i]==NULL) {
            printf("malloc failed %i\n",i);
            return;
        }
    }

    // Random permutation of nodes
    int *perm = malloc(nodes * sizeof(int));
    for (int i = 0; i < nodes; i++) perm[i] = i;
    for (int i = nodes - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        //swaps i and j
        int tmp = perm[i];
        perm[i] = perm[j];
        perm[j] = tmp;
    }

    //initialising nodes
    for (int j=0; j<arrays; j++) {
        for (int i=0; i<nodes-1; i++) {
            array[j][perm[i]].next = &array[j][perm[i+1]];
        }
        array[j][perm[nodes-1]].next = &array[j][perm[0]];
    }

    //looping through linked lists
    for (int i=0; i<iters; i++) {
        for (int j=0; j<arrays; j++) {
            array[j] = array[j]->next;
        }
    }

    printf("high done\n");
    return;
}

int main(int arg, char *argc[]) {
    if (arg==1) {
        printf("no args\n");
    }
}

```

D. simple.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Node {
    struct Node *next;
} Node;

int main() {
    Node *array = malloc(sizeof(Node)*100);

    //initialising loop
    for (int i=0; i<99; i++) {
        array[i].next = &array[i+1];
    }
    array[99].next = array;

    //main loop
    Node *p = array;
    for (int i=0; i<10000; i++) {
        p = p->next;
    }

    printf("done\n");
}
```

E. mlp_calc.py [14]

```
#!/usr/bin/env python3
import sys
import re

def parse_line(line):
    match = re.search(r"Load start (\d+) load done (\d+)", line)
    if not match:
        return None
    start, end = map(int, match.groups())
    return (start, end)

def calculate_mlp(intervals):
    # Build a list of (time, delta) events: +1 at start, -1 at end
    events = []
    for start, end in intervals:
        events.append((start, +1))
        events.append((end, -1))
    events.sort()

    concurrent = 0
    last_time = None
    total_active_time = 0
    sum_concurrent_time = 0

    for time, delta in events:
        if last_time is not None and concurrent > 0:
            duration = time - last_time
            total_active_time += duration
            sum_concurrent_time += concurrent * duration

        concurrent += delta
        last_time = time

    if total_active_time == 0:
        return 0.0
    return sum_concurrent_time / total_active_time

def main():
    if len(sys.argv) != 2:
        print("Usage: python3 mlp_calc.py <input_file>")
        sys.exit(1)

    filename = sys.argv[1]
    intervals = []
    with open(filename, 'r') as f:
        for line in f:
            parsed = parse_line(line)
            if parsed:
                intervals.append(parsed)

    if not intervals:
        print("No valid load entries found.")
        sys.exit(1)

    mlp = calculate_mlp(intervals)
    print(f"Average MLP: {mlp:.4f}")
```

```
if __name__ == "__main__":  
    main()
```