RESEARCH-ARTICLE

Efficient Server Consolidation through a balanced mix of Transformer-based and Conventional Applications

PABLO ABAD, University of Cantabria, Santander, Cantabria, Spain

PABLO PRIETO, University of Cantabria, Santander, Cantabria, Spain

VALENTIN PUENTE, University of Cantabria, Santander, Cantabria, Spain

JOSE ANGEL GREGORIO, University of Cantabria, Santander, Cantabria, Spain

Open Access Support provided by:

University of Cantabria

Published: 22 August 2025

Citation in BibTeX format

ICS '25: 2025 International Conference on Supercomputing
June 8 - 11, 2025
Salt Lake City, USA

Conference Sponsors:
SIGARCH

Efficient Server Consolidation through a balanced mix of Transformer-based and Conventional Applications

Pablo Abad
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
abadp@unican.es

Pablo Prieto
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
prietop@unican.es

Valentin Puente
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
vpuente@unican.es

Jose Angel Gregorio
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
monaster@unican.es

Abstract

Although not optimal for Large Language Model (LLM) inference, general-purpose processors remain a practical choice due to their ubiquity and accessibility. This work explores strategies to maximize server utilization when incorporating such applications into the workload mix. To do that, we conduct an exhaustive profiling process to analyze hardware-software interactions, leading to two key observations. First, we have validated and quantified the common intuitions regarding LLM execution, with a specific focus on the microarchitecture's backend. Second, we observe the relatively low contention of LLMs and conventional applications (e.g., SPEC CPU17) when running together. Inspired by these findings, we explore whether combining applications of each type on a common server, if properly balanced, could lead to a better system utilization. Using both state-of-the-art server configurations and slightly older systems, we demonstrate that executing LLMs on general-purpose processors is feasible with minimal impact on co-located applications and lead to a better overall server performance.

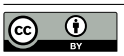
CCS Concepts

• **Computer systems organization**; • **Architectures**; • **Parallel architectures**; • **Single instruction, multiple data**; • **Computing methodologies**; • **Artificial intelligence**; • **Natural language processing**; • **Multicore architectures**;

Keywords

Large Language Model, Transformer, CPU, SIMD, Memory Hierarchy

ACM Reference Format:
Pablo Abad, Pablo Prieto, Valentin Puente, and Jose Angel Gregorio. 2025. Efficient Server Consolidation through a balanced mix of Transformer-based and Conventional Applications. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3721145.3725751



ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/2025/06
https://doi.org/10.1145/3721145.3725751

1 Introduction

The current interest in generative AI stems mainly from the exceptional performance of transformer-based models [1] on tasks such as text generation [2] or translation [3]. This success implies that their adoption rate is substantial, from on-the-edge devices to cloud computing. The common assumption is that specialized hardware such as GPUs or even offload accelerators are the preferred execution platform. Although this seems to be the case for model training in the latter or inference in the former, the availability of general-purpose resources in the cloud might lead us to question whether there is a way around such a requirement. Given the massive cost and limited availability of current specialized hardware [4], such a question is appealing. This is especially important if such workloads can run with minimal disruption to traditional workloads sharing the underlying hardware.

Many server-oriented CPUs available in the cloud already implement hardware features targeted at AI applications. For example, recent generations of Intel®Xeon®Scalable processors include AVX-512 ISA extensions [5] that implement both deep learning inference (Fused Multiply-Add instruction for 8-bit multiplies and 32-bit accumulates) and training-oriented instructions (several instructions to operate on the bfloat16 data type [6]).

Efficiency (hardware utilization) and quality of service (QoS) are two key issues in a cloud environment. This relevance is confirmed by the amount of research done on CMP resource sharing over the last 20 years [7][8]. Some of these works have focused on the analysis of application benchmarks to group them based on their features (memory, CPU, disk) so that they can share resources minimizing the number of conflicts to meet efficiency and QoS requirements [9][10]. Guided by the relevance acquired by machine learning applications, this type of exploration work has been extended to AI workloads running on general-purpose processors [11] and the impact of features such as simultaneous multithreading or core scalability [10]. To date, most research has focused on non-generative models for tasks such as recommendation or computer vision [12], which have distinct characteristics from transformer-based architectures.

Recently, driven by the growing importance of transformer-based workloads, research interest has shifted to this area. Currently, most efficiency proposals target GPUs [13][14], which are a priori the most common execution substrate. However, thanks to the efforts of processor vendors to integrate AI acceleration mechanisms into general-purpose CPUs, recent work shows that the latest CPU models can compete with GPUs in inference tasks [15][16].

Through exhaustive profiling, we precisely identify how the LLMs fail to utilize certain resources available in a general-purpose processor. As a result, these workloads are unable to take full advantage of ILP due to their specific behavior at the backend of the core. In addition, we have also observed that due to their distinct nature, LLM applications may experience reduced contention when run alongside more conventional workloads, such as a statistically relevant mix of applications from the SPEC CPU benchmarks [17] [18]. We apply these findings to a real-world scenario and show that an appropriate combination of workloads can improve processor utilization, a critical factor for efficiency in cloud environments. While the optimal mix depends on the server configuration, the trend remains consistent. For example, our experimental evidence suggests that even on slightly older systems where the core lacks AI-focused features or memory bandwidth is constrained, combining both workloads can still improve server utilization.

The main contributions of the paper are as follows:

- Precise identification of microarchitectural bottlenecks for LLM inference using general-purpose CPUs.
- Extensive characterization of LLM memory behavior, highlighting the absolute lack of locality and the poor utilization of cache hierarchy.
- Comprehensive evaluation of collision sources when LLM inference runs alongside conventional applications, demonstrating a non-intrusive behavior.
- Discussion of the potential benefits of allowing LLM inference to share execution resources, maximizing sever utilization while considering the heterogeneous nature of cloud resources.

2 LLM Profiling

LLMs are a class of applications characterized by distinctive features, including the massive size of the working dataset (with models significantly exceeding the typical assumptions underlying memory hierarchy design) and a strong reliance on a narrow set of operations. As illustrated by the stack traces in Figure 1, profiling the execution of some models reveals that the most frequently executed code paths using some commonly used CPU frameworks, (llama.cpp [19] and neuralspeed [16]) are highly concentrated. Each colored box represents a code function, and its width is proportional to the execution time spent on it. Functions in a particular row are called by those in the row below, building the call hierarchy from bottom to top. The majority of the execution time is spent within a single or a few functions comprising a small number of lines of code. As a specific example, llama.cpp and neuralspeed spend 98% of its time in a single function (ggml_vec_dot_32 and ne_vec_dot_f32 respectively) for fp32-weight models and nearly 90% for quantized ones. The dominant function has a simple structure, consisting of

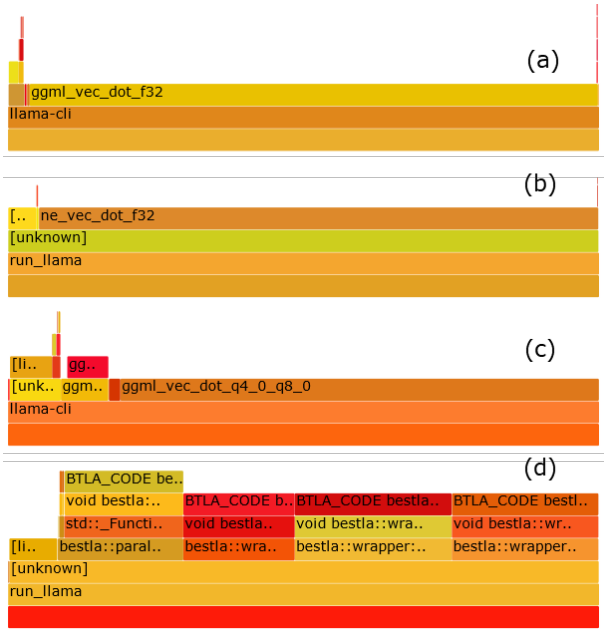


Figure 1: Execution flamegraph with fp32 model: (a) llama.cpp, (b) nspeed and Q4 model: (c) llama.cpp, (d) nspeed.

an iterative loop with two loads and a few SIMD operations (FMA + dequantization if necessary).

Given this particular structure, conducting a detailed profiling process is an essential first step in gaining a comprehensive understanding of the interaction between the workload and the CPU. With the rapid evolution of modern processors and the growing number of hardware features tailored for AI applications, it is easy to overlook the inherent limitations imposed by design decisions rooted in the behavior of conventional applications. In this section, we aim to address this issue by quantifying the impact of certain performance issues that are often vaguely referenced in the literature, highlighting their significance. We will conduct a detailed analysis using a robust methodology to examine how the software under test interacts with the conventional components of the CPU microarchitecture.

2.1 Hardware & Software Stack

To draw valid conclusions, the set of LLM applications must be representative, covering a wide range of use case scenarios. To achieve this, we run several LLM models with different architectures, sizes, and data types on two different frameworks targeting CPUs.

The first framework used, llama.cpp [19], is one of the most widely adopted in the open-source community. Built on top of the ggml tensor library [20], it supports a large number of models and quantization techniques. Written in C/C++, it also includes support for SIMD extensions on X86 and other architectures. We use a second framework, neural speed [16], which inspired by llama.cpp but highly optimized for quantized models running on Intel CPUs with latest SIMD ISA extensions. In both cases, the software has

Table 1: Summary of LLM models evaluated.

Model	Size (gguf weight file)			
	Fp32	Fp16	Q8	Q4
TinyLlama [21]	4.1G	2.1G	1.1G	608M
Starcoder [22]	12G	5.7G	3.1G	1.7G
Llama2 [23]	26G	13G	6.7G	3.6G
Mistral [24]	27G	14G	7.2G	3.9G
Llama3 [25]	30G	15G	8G	4.4G
Gptneox [26]	77G	39G	21G	11G

been compiled to take advantage of all AVX512 family extensions [5] available in the processor used for profiling.

A summary of the models used is given in Table 1., ranging from 1 (TinyLlama) to 20 (gptneox) billion weights. File model size information is provided for the different data types used. Most of the relevant open-weight models are represented, such as the latest Llama architectures, the mistral model, as well as starcoder or gptneox. In terms of the data type used to store the model weights, we use the two most common floating-point formats, single-precision (fp32) and half-precision (fp16), as well as 8-bit and 4-bit integer quantization formats. SIMD units can operate natively on floating-point weights, but in the processor used, quantized weights, require a dequantization step prior to operation.

All profiling experiments are run on a 5th generation Xeon Scalable Processor (4514 family), with a server configuration described in Table III (test system 1). This processor, released in the third quarter of 2024, includes Intel Deep Learning Boost technology [5] that targets IA applications. In addition, the core provides useful features for profiling, such as many hardware event counters and parametrization of some microarchitecture features. MSRs can be used to enable/disable SMT and hardware prefetching features, and Resource Director technology [27] provides granular control over L2, LLC and memory bandwidth allocated to each core (or group of cores).

2.2 CPU Bottlenecks (ILP Limitation)

The starting point of our analysis must be the measurement of the overall performance of LLM applications, with a special focus on microarchitectural efficiency. To do so, we combine a raw performance value, such as the tokens per second generated by each model, with an Instruction Per Cycle (IPC) metric.

Figure 2 illustrates the prediction times for each of the LLM model evaluated in the test system. As anticipated, both model size and data type play a critical role in inference latencies. In terms of model size, smaller models require fewer operations per predicted token, while quantization reduces the memory footprint, enhancing overall performance. When restricted to a single processor, 8 billion parameters models appear to be the upper limit for generating tokens at human reading speed (~5-10 tokens per second [28]) on the test system. Running larger models requires multi-node execution to better exploit data parallelism or transitioning to quantized models.

In terms of efficiency, Figure 3 shows the IPC obtained, with the y-axis set to the maximum achievable on the test system, 6. As can

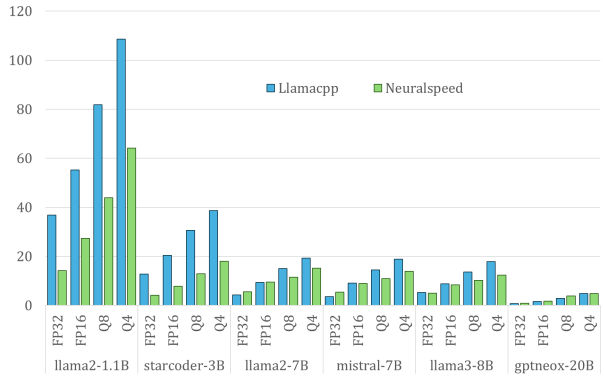


Figure 2: Inference performance for different models. Tokens per second.

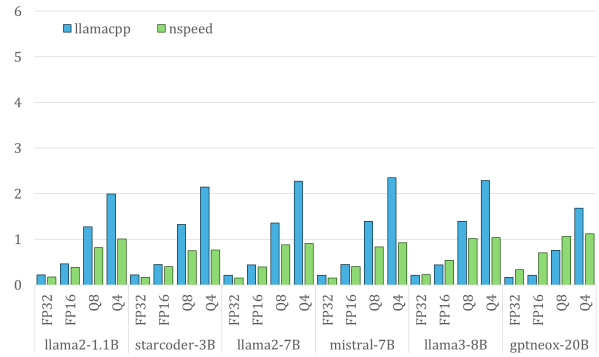


Figure 3: IPC achieved for inference process.

be seen, the token generation speed is achieved by under-utilizing hardware resources, reaching only two instructions per cycle in the best cases. In contrast to the performance metric, it is noteworthy that these values are independent of model size, suggesting that both large and small models appear to face similar limitations. In this case, only data type seems to be relevant for the results. In addition, it should be noted that the IPC improvement of quantized models is not proportional to their performance gains. Therefore, beyond synchronization instructions, IPC is not a reliable proxy metric for performance. For example, llama2-1.1B improves its IPC by 20X when moving from fp32 to q4 weights, but this increment turns only in less than 3X raw performance improvement (Figure 3 vs. Figure 2). The dequantization process introduces additional instructions that are efficiently handled by the backend, leading to a significant increase in the IPC but a relatively smaller impact on actual performance. On average, quantized models require three times as many instructions to perform the same effective work.

The significant gap between the observed and maximum IPC warrants further analysis to determine which parts of the microarchitecture are causing this behavior. Profiling methodologies, such as top-down analysis [29], are helpful in this process. They organize hardware event counters in a hierarchical manner, helping to

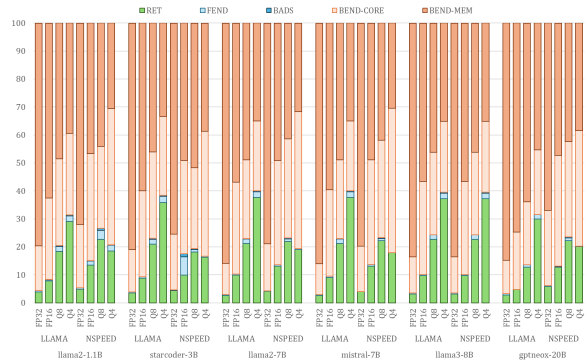


Figure 4: Level 1 top-down profiling for LLM models.

map the processor pipeline and identify the contribution of each component to the performance loss. In this work we make use of pmu-tools [30], a collection of CLI tools and libraries that implement this methodology on top of Linux perf [31]. At its most basic level, this hierarchy divides performance into five main categories, described as follows:

- Frontend Bound (FEND): fraction of the pipeline stalls originated during the process of instruction fetching and decoding. Mainly related to cache instruction misses, or decoding contention.
- Bad Speculation (BADS): performance stalls caused by wrong speculation decisions, that require a processor state restoration process. Branch miss-predictions are the main contributions.
- Backend Bound (BEND): The stalls in this category are related to the inability to complete an operation, caused by the lack of any operand (Memory Bound) or the lack of available functional units (Core Bound).
- Retiring (RET): this category represents retired instructions. It does not represent stalls, but real performance, corresponding to the IPC performance metric.

As the number of levels of the analysis increases, each part of the pipeline is profiled in more detail, splitting the microarchitecture into more specific components, as we will see in the Memory Hierarchy evaluation. Figure 4 shows the LLMs results for the top-down categories described in the previous paragraph. As can be seen, the nature of these applications results in a significant backend penalty. While the frontend components (including fetch, decode and branch prediction) appear to work properly, being the processor backend the culprit for the poor performance observed. In particular, the memory hierarchy is the main contributor to execution stalls for floating-point datatypes, regardless of model size and datatype width. As the weight size decreases (quantized data types), the impact of functional units on performance becomes more relevant. However, the poor memory hierarchy behavior remains the dominant bottleneck in most cases. Only for the models running on the NSpeed framework, with the smallest weight size, both core and memory have a similar impact.

Table 2: Functional unit organization at core backend.

	Integer	FP/Vec	Load/Store
P0	ALU, LEA, Shift, JMP	FMA, ALU, Shift, fpDIV	
P1	ALU, LEA, Mul, iDIV	FMA, ALU, Shift, Shuffle, FADD	
P2			AGU, Load
P3			AGU, Load
P4			Store Data
P5	ALU, LEA, MulHi	FMA512, ALU, AMX, Shuffle, FADD	
P6	ALU, LEA, Shift, JMP		
P7			AGU, STA
P8			AGU, STA
P9			Store Data
P10			AGU, Load
P11	ALU, LEA		

The constraints imposed by the available functional units become apparent when we combine the code profiling of Section I with the port usage analysis in Figure 5. In this graph, the y-axis represents the fraction of instructions issued to each execution port (the evaluated core implements a large number of functional units with a port-based organization for scheduling, shown in TABLE II). As can be seen, 90% of the instructions are scheduled to SIMD (P0, P1 and P5) and load ports (P2, P3 and P10). As mentioned in the code profiling at the beginning of Section II, this instruction mix corresponds to the FMA operation loop on large vectors, so in a general-purpose architecture, core limitations in this part of the backend are going to be highly dependent on the number of concurrent FMA operations supported by the processor.

Focusing on the memory hierarchy, spanning from the L1 cache to the memory controller, as the main bottleneck, the next step is to refine the top-down analysis for more detail on these components. Figure 6 shows the contribution of each level of the hierarchy to the performance degradation. Each data series in the graph is described in Intel's documentation as the fraction of cycles that the CPU was stalled at each level. These results indicate that the primary bottleneck lies in the distant levels of the hierarchy (both the last-level cache and the memory controller). For LLM models with weights stored in memory using a floating-point format, main memory accesses are the predominant factor contributing to performance degradation. For quantized models, the number of weights loaded per memory access increases, slightly alleviating the pressure on the memory controller. At this stage, the performance bottleneck

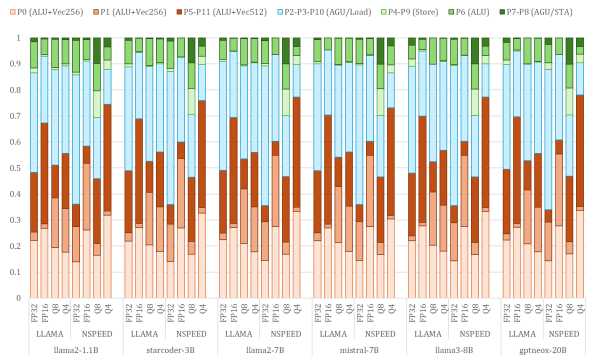


Figure 5: Port utilization (fraction of total instructions).

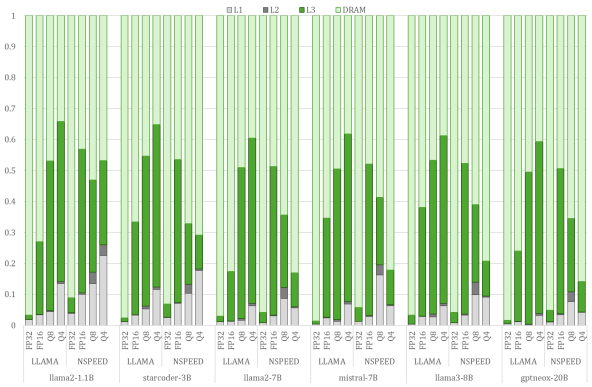


Figure 6: Memory Bound. Fraction of requests responded at each cache level.

partially shifts to the cache. However, it should be noted that the behavior is highly dependent on the execution framework. For instance, in models executed with NSpeed, memory controller issues remain dominant in all cases.

The increasing proportion of load stalls resolved in the last-level cache can give the misleading impression that a portion of the data working-set fits into this cache. Actually, a lot of these hits are not due to data reuse, but to the implementation of aggressive hardware prefetching mechanisms in the CPU of the system under test. In applications with simple access patterns, such as FMA operations on large vectors, the prefetcher works effectively, resulting in a large number of hits on the LLC. From a top-down methodology standpoint, these loads are resolved in LLC, but from a performance perspective, the requests generated by the prefetcher are the critical ones, and these are resolved at the rate supported by the memory controller.

The critical role of the memory controller in LLM inference, highlighted by the top-down methodology, is further supported by additional experiments. One of the features available in the Resource Director technology of Intel processors, called MBA [32][33] (Memory Bandwidth Allocation), allows approximate control over

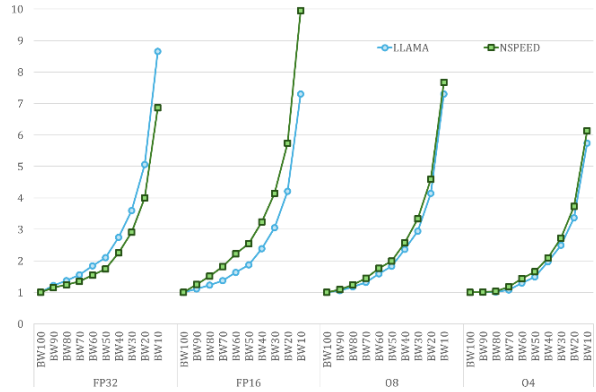


Figure 7: Performance degradation as bandwidth decreases from 100 to 10%.

the memory bandwidth available to the workload. Using this feature, Figure 7 illustrates how raw performance (tokens per second) evolves as the bandwidth available for the inference process decreases from 100 to 10%. The results on the y-axis have been normalized to those obtained with full bandwidth availability. Given the uniformity observed across models, we present the results for a single model (llama3-8B). As observed, this experiment enables us to quantify the impact of reduced bandwidth on performance loss in this specific CPU. Halving the bandwidth roughly doubles the inference time, while limiting it to 10% can increase the token prediction time up to a factor of 10. It should be noted that for other processors that have a lower peak bandwidth per CPU, the impact on performance will be even more substantial. As shown, the results are consistent across data types (despite the differences observed in Figure 6), confirming that the size of L3 data series in Figure 6 has a minimal impact on performance.

2.3 LLMs and Cache Hierarchy

The fact that most CPU memory requests are resolved at the memory controller indicates that the cache hierarchy is not working as expected. The design principles of the cache hierarchy are based on memory access locality, which is common in most conventional applications. However, this does not appear to be the case for LLM inference workloads. The platform's support for tuning certain configuration parameters of the cache hierarchy online, allows us to extend the profiling process and evaluate the spatial and temporal locality of these workloads.

Mechanisms like simple next-block prefetching are designed to exploit spatial locality. For linear memory accesses (high spatial locality), prefetching the next contiguous blocks can provide a significant performance benefit. Using Model-Specific Registers (MSRs), we can manipulate the hardware prefetching process to assess the occurrence of this type of locality. In our particular case, according to Intel's documentation [34], the five least significant bits of MSR 0x1A4 are responsible for enabling and disabling prefetching. Figure 8 shows the token prediction slowdown for each model evaluated when hardware prefetching is disabled. The y-axis values are normalized to the prediction time with prefetching. We can see two different results. On the one hand, for data types fp16,

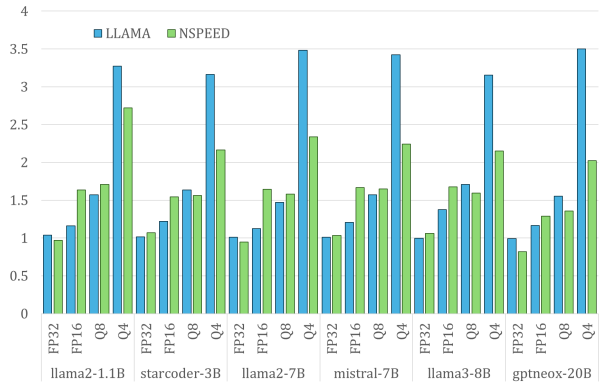


Figure 8: Prefetcher effect on performance.

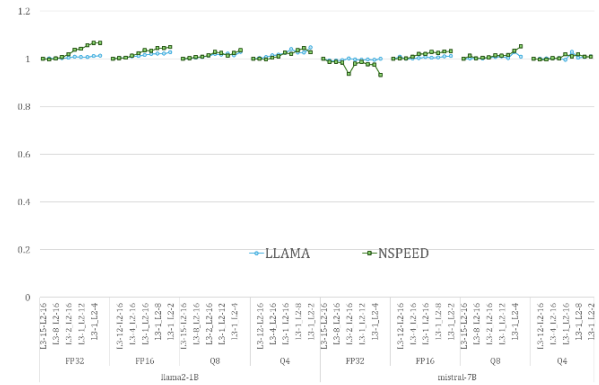


Figure 9: LLC and L2 capacity reduction, performance effect.

q8 and q4, the performance benefit is clear, confirming the spatial locality associated with the predominant FMA operation on large vectors. On the other hand, models with fp32 weights do not seem to benefit from prefetching. The same locality should be present regardless of the data type, so this behavior must be attributed to some other underlying factor. Using the Linux perf profiler, we can gather information about hardware prefetching activity. In the case of the fp32 models we have observed that no prefetching activity is detected (even though the prefetcher is enabled). This is likely due to the high bandwidth consumption that the fp32 data type imposes on the memory controller.

Another locality type exploited by the cache hierarchy is temporal locality. In this case, cache capacity plays a key role in exploiting this property, since a larger cache reduces the chance of collision (and replacement), allowing the same data to be reused multiple times. To evaluate the presence of this locality, we again use a Resource Director feature, called Cache Allocation Technology [35][36]. This allows control over the amount of cache allocated to each core-set by limiting the cache ways where they can write data. In Figure 9, we have progressively limited the size of LLC and L2 to the LLM execution, first reducing the number of LLC ways from 15 to 1 and then reducing the number of L2 ways from 16 to 2. The y-axis represents the slowdown in token prediction time, normalized to the time with the entire cache hierarchy available. Given the uniformity across models, we limit the results to two models, llama2-1B and mistral-7B. As can be seen, reducing the cache capacity by almost 16 has a negligible impact on performance, suggesting that there is no reuse at these levels, due to the massive working set of the LLM models.

In Sections 2.2 and 2.3 we have described in detail the microarchitectural bottlenecks resulting from the LLM code structure and a general-purpose processor backend. The dominance of a specific SIMD operation limits the utilization of available execution ports and functional units. In addition, a very specific access pattern and large data structures favor spatial locality but almost prevent cache reuse, rendering a significant portion of the memory hierarchy useless. These two factors (the memory hierarchy to a greater extent) are behind the limited performance observed in Figure 2. Future generations of processors will need to progressively increase memory bandwidth and vector processing capacity to further improve the per-core performance for these types of applications.

2.4 Thread Scaling (TLP Limitation)

LLMs are highly data parallel (that's why they work so well on GPUs), which eases DLP exploitation through multi-threaded execution. However, thread scaling can be strongly limited by one of the bottlenecks observed in previous section, BW availability.

To evaluate this, we will gradually increase the number of threads from one to 32 (upper limit, with one thread per logical core while SMT is enabled). In executions with 16 threads or less, each thread is mapped to a different physical core to avoid unnecessary contention.

Figure 10 displays how the inference performance of the quantized models (q4 and q8) increases almost linearly with the number of threads until it reaches the number of physical cores (16). Beyond this point, the performance continues to increase, but at a slower rate due to core resource sharing through SMT. In contrast, models with fp weights do not scale as well as quantized models. Looking at the fp16 data series, the performance improvement almost levels off when 16 threads are reached. This behavior appears to be caused not only by SMT utilization, but also by exceeding the bandwidth availability. We have observed that each thread reduces its bandwidth utilization by half when moving from 16 to 32 threads. The scaling limitations are even more apparent with fp32, where the performance improvement plateaus at 8-10 threads. This is because the bandwidth requirements are twice as large as fp16 due to the difference in data size.

This behavior is consistent across model sizes, as shown when comparing Llama3-8B model with Llama2-1B model. Bandwidth requirements appear to be independent of the model and primarily depend on the data type. The number of threads that reach peak performance is mainly limited by the bandwidth available in the entire system.

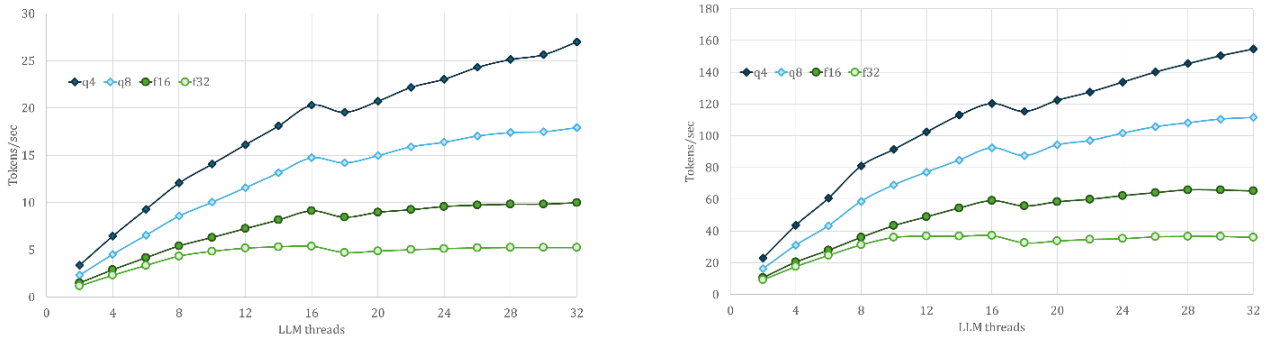


Figure 10: LLM performance, thread scaling. Llama3-8B model (left) and Llama2-1B model (right).

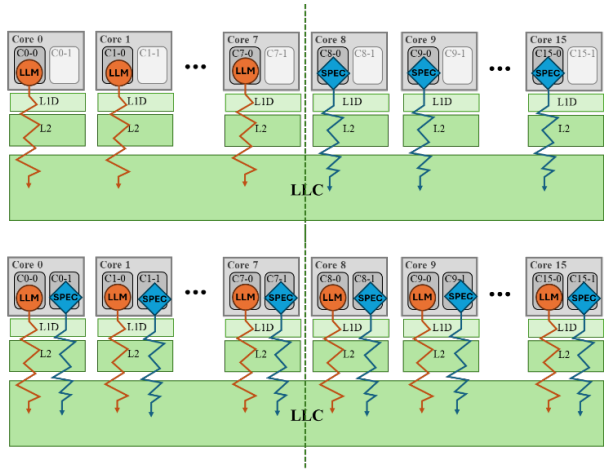


Figure 11: Thread mapping to evaluate collision in last-level cache (up) and in each physical core when shared through SMT (down).

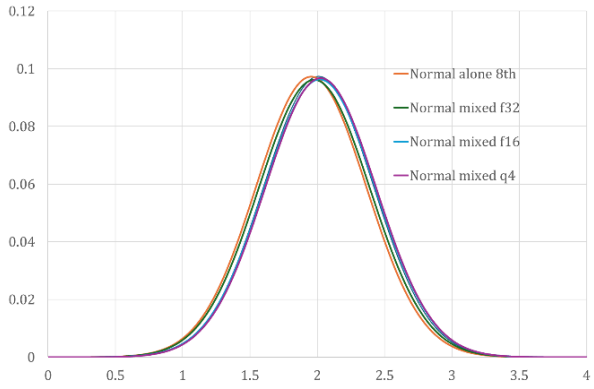


Figure 12: IPC evolution of SPEC applications when sharing LLC with LLMs.

3 LLM-Spec resource sharing

The previous section demonstrated that, even when LLMs extract the maximum performance from the processor, they make inefficient use of a significant portion of the hardware resources, as both the cache hierarchy and part of the functional units remain largely underutilized. The underlying cause of this behavior is the unique structure of these applications, which differs significantly from conventional applications in general-purpose environments. In fact, applications from benchmarks such as SPEC CPU 2017 [17] have much less available data parallelism available and are generally more cache-friendly [37].

Since each type of application appears to use different elements of the microarchitecture, the question arises as to whether running both applications concurrently could enhance the processor utilization. To explore this possibility, the first step is to analyze in detail the interaction of the shared microarchitectural resources. To this end, the experiments are divided into two categories. First, we investigate collisions at the last level cache by mapping LLM and SPEC threads to different cores, as shown in Figure 11(top). Next, we evaluate the degree of collision when SPEC and LLM applications share physical cores via SMT. In this case, in addition to the LLC, the processor backend and L1/L2 caches are shared between two different threads. The following subsections describe these experiments and the results obtained.

3.1 Global Resources (LLC)

Despite the lack of temporal locality, the large working-set of LLMs could pollute shared levels of the cache hierarchy, negatively affecting the performance of collocated threads. To analyze the degree of interference, we divide the processor in half, one for each type of application, as shown in Figure 11(top), keeping both applications in different physical cores. In the SPEC half of the CPU, we will make use of the Benchcast tool [18] to generate a sufficiently large population of randomized conventional workloads to ensure statistically meaningful results. First, we will obtain a normal distribution of IPC results when such workloads are executed alone. Next, we will compare these results to those obtained when SPEC is run alongside an LLM model. In the LLM portion, we will run each model with a number of threads equal to the number of cores, ensuring that the inference process runs continuously throughout the evaluation.

Figure 12 shows the performance results for SPEC applications under both scenarios, running alone (Normal alone data series)

Figure 13: Performance degradation of LLM when physical core is shared through SMT.

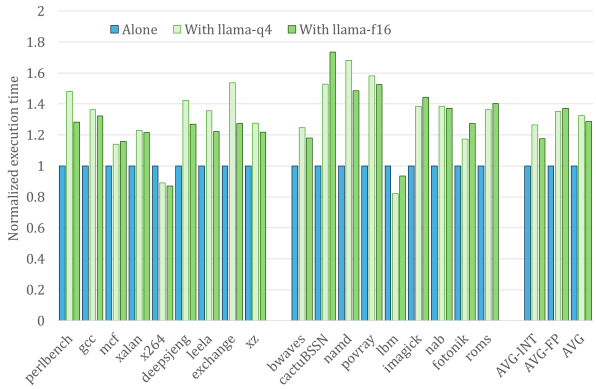


Figure 14: Performance degradation of SPEC when physical core is shared through SMT.

and with LLM. As discussed in previous experiments, we limit the evaluation to a single LLM model (llama3-8B) for clarity. We will evaluate the degree of collision in three data types used in the weights. Two are floating-point (Normal mixed f32 and f16) and one is quantized (Normal mixed q4). The graph shows a frequency histogram of the IPC distribution, obtained from over 300 random combinations of SPEC applications running for at least 40 seconds in their region of interest. Since all four data series show a very similar IPC distribution, we can conclude that the presence of an LLM application on the same processor has, on average, a minimal impact on the performance achieved by SPEC applications. This lack of contention seems to indicate that there is no pollution of last-level cache with data blocks from LLM execution. Since the lack of temporal locality minimizes the number of references to LLM blocks, they appear to be quickly selected for eviction by the replacement policy. On the other side of the processor, in terms of LLM thread performance, we know from Figure 9 that reducing the LLC size (through Cache Allocation Technology available in Resource Director) has a negligible effect on inference performance. This supports the hypothesis that due to the effectiveness of the replacement policy, collisions between both types of workloads are minimal in this microarchitecture component.

Based on the results obtained so far, we can conclude that in cases where LLM is constrained by memory bandwidth limitations, introducing conventional applications in the remaining cores can be a viable strategy to improve overall processor utilization. In the specific case of the test system, with 8 threads of Llama3-8B and fp32 data type, the memory bandwidth is almost exhausted and it is not possible to obtain higher performance by increasing the number of threads (Figure 10). In this situation, it is possible to add 8 SPEC applications at almost no cost (Figure 10 and Figure 12). Due to the efficiency of the on-chip memory hierarchy, most SPEC-based workloads do not seem to suffer from bandwidth exhaustion. The LLM model will run close to peak performance on this system, and the SPECs will make the best possible use of the CPU without the interference from the LLM model.

3.2 SMT-Shared Resources (L1-L2, FUs)

To further explore the collision effects, we next organize thread execution by mapping SPEC and LLM threads to each logical core (SMT enabled) of the same physical core (Figure 11, bottom). Again, due to the consistency of the results, we limit the exploration to a single model for the LLM (Llama3-8B). We chose the q4 and fp16 data types because their performance scales with the number of threads until they occupy all physical cores, and because they represent different ways of using core resources. To better understand the exploration, instead of using BenchCast to generate combinations, we first evaluate the collision of each SPEC application individually. First, Figure 13 shows the performance degradation (normalized prediction time, tokens per second) experienced by the LLM execution when sharing the physical core with each SPEC application. As can be seen, the LLM data type has a negligible impact on the results, and each combination moves in a narrow range around 30% performance degradation. Looking more closely at the individual results, we observe some peculiar behavior. First, the x264 combination suffers a much smaller degradation. The reason for this is the large amount of IO operations of this SPEC, which reduces its CPU utilization and favors LLM execution. The other notable result is the difference observed between INT and FP SPEC applications. As can be seen, when the core is shared with FP applications (cactuBSSN to roms), there is an increase in the loss of performance. The processor's port organization, which groups FP and SIMD units for scheduling, may increase contention in the core's backend, further exacerbating the performance degradation when both types of applications run concurrently.

From the opposite perspective, Figure 14 displays the same results for SPEC applications showing how their performance is affected when sharing the core with LLM applications. The y-axis represents the normalized execution time, and the columns show the performance degradation of each SPEC running next to the selected LLM model with two different data types. These results show a similar trend, with performance degradation approaching 30%. In addition, the distinct behavior of x264 and the differences between INT and FP applications are even more noteworthy in this case.

Table 3: System under test.

Characteristic	Test system 1	Test system 2
Model name	INTEL®XEON®SILVER 4514Y	AMD EPYC 7713P
Num cores	16 (32 SMT)	64 (128 SMT)
L1 ICache	32KB/core	32KB/core
L1 DCache	48KB/core	32KB/core
L2 Cache	2MB/core	512KB/core
LLC	30MB	256MB
Memory Channels	8	8
RAM	256 GB (8x32GB) DDR5	256GB (8x32GB) DDR4
Max. Memory Bandwidth	275 GB/s	205 GB/s
AVX FMA Units	2xFMA-256b 1xFMA-512b	2xFMA-256b
Operating System	Debian 12.0	Debian 12.0
Kernel	Linux 5.10.0-28-amd64	Linux 5.10.0-28-amd64

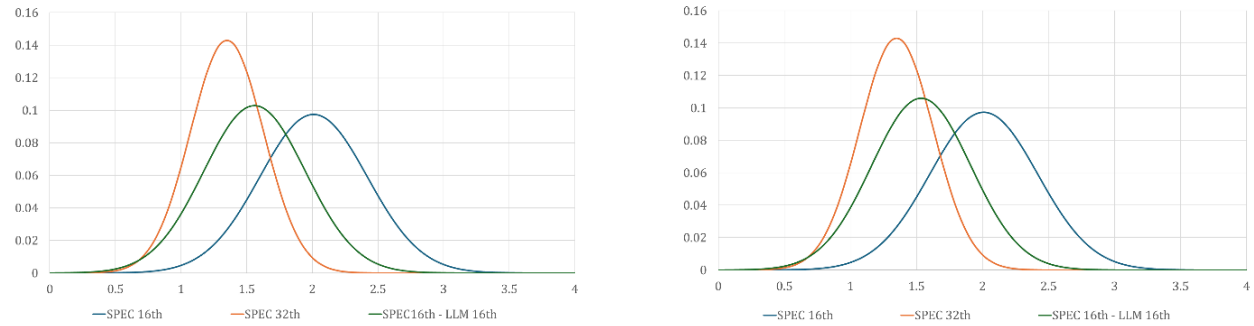


Figure 15: IPC distribution of SPEC applications using SMT alone or along with Llama3-8B fp16 (left) and Q4 (right).

By sharing both last-level cache resources or core and private cache levels through SMT, the performance degradation experienced by each type of application seems to be limited (minimal in the case of last-level cache). This fact, coupled with the inability of LLM models to fully utilize hardware resources, raises the key question that drives this work. If circumstances make a general-purpose CPU the only available/appropriate environment to run an LLM model, can we efficiently combine the inference with other applications to improve the aggregate utilization of the underlying hardware?

4 Server optimization

Building on the findings so far, which emphasized the low utilization of the cache hierarchy by LLMs and the minimal collision between the inference workload and regular applications, we now focus on strategies to maximize overall server utilization by running both applications with SMT enabled.

We have already assessed the behavior of LLM models when running alone in the SMT (Figure 10). Next, we evaluate the impact of the SMT on conventional workloads in the absence of the LLM inference, and then assess the overall system performance when these workloads run concurrently with an LLM model. As discussed in the previous sections, we limit our evaluation to a single model (Llama3-8B) and fp16 and q4 datatypes. To obtain statistically meaningful results, we use the Benchcast tool in test system 1 (Table III) and run more than 300 random combinations of SPECs for each curve. Figure 15 shows a frequency histogram of the IPC distribution of SPECs running on all threads of the processor (32th, using SMT) and SPECs together with LLM, each using half of the processor (16 SPEC threads and 16 LLM threads), as seen in Figure 11 bottom, all compared to the execution of SPECs not using SMT (16^{th}).

As shown in Figure 15, SPEC applications tend to interfere with each other more than SPEC and LLM applications combined. Conventional applications running alone using SMT reach an average of 1.3 instructions per cycle (lower than the 2.1 IPC average obtained without SMT), which means that using SMT the overall utilization of the server has been increased by ~25% (i.e., from 2.1 to 2.6). Although enabling SMT is beneficial, when the same class of workloads are paired with the LLM inference workloads, they are able to achieve up to 1.6 instructions per second on average. This suggests that, from the SPEC application's point of view, sharing a core with an LLM model is more favorable than sharing it with another regular application. This result builds on previous observations of limited interference between LLM models and conventional applications at distant levels of the memory hierarchy. It also suggests that the conflict that arises in cores is less significant

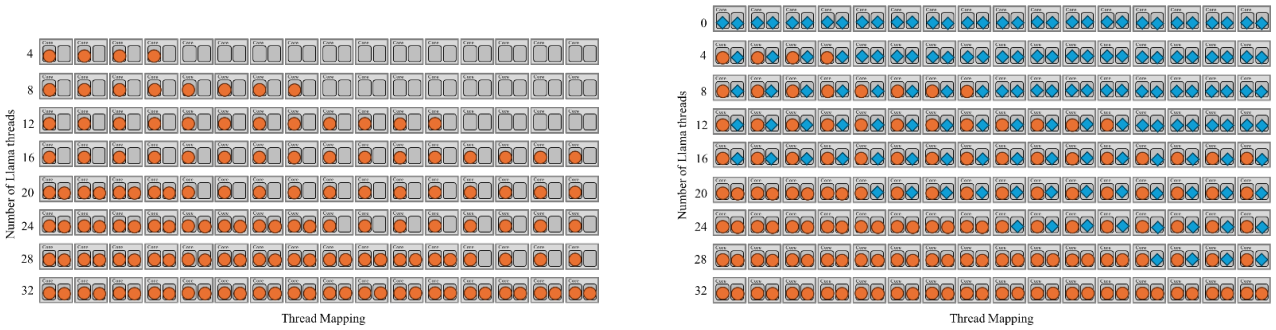


Figure 16: Thread distribution among logical cores. LLM alone (left) and LLM (red) and SPEC (blue) mixed (right).

than that observed when conventional applications compete with each other.

Building on this result, we extend our analysis to determine where the optimal configuration that maximizes server utilization is by varying the number of hardware threads allocated to the LLM model and conventional applications. To achieve this, we will evaluate the performance of the test system 1 when fully dedicated to LLM execution and its performance when fully dedicated to conventional applications. We will then compare these results to the performance achieved when both are run in combination, as shown in Figure 16.

To quantify the performance of LLM on its own, we gradually increase the number of threads used for inference. Threads are pinned to logical cores, first filling the physical cores (to avoid collisions), then adding more and more colliding hardware contexts (Figure 16 left). As a measure of performance, we use the average "eval time" tokens/sec for each configuration. As noted earlier, unlike in SPEC-based workloads, IPC is not an appropriate performance metric for LLM due to the synchronization requirements and the significant number of instructions required in the dequantization processes.

Conventional workload evaluation is performed by launching more than 300 random combinations of SPEC applications per measure, gradually increasing the number of running threads. First the physical cores are filled, then interfering hardware contexts are allocated (as on Figure 16 up). As a performance metric, we use the cumulative IPC provided by Benchcast, which is obtained by multiplying the average IPC by the number of running applications. This yields the system performance for each thread configuration.

Finally, we measure the performance of both workloads (LLM Llama3-8B and the conventional applications) running simultaneously, gradually increasing the number of LLM threads while reducing the number of SPEC threads through Benchcast (Figure 16 right).

Figure 17 shows the performance results when running SPEC-based workloads alongside with Llama3-8b FP16. All performance results are normalized to the maximum performance obtained when each workload is run alone (LLM and SPEC, respectively). A 95% confidence interval was used for all results. The horizontal axis represents the number of threads dedicated to the LLM model, while the number of SPEC threads is the remainder (32 - number of LLM threads). The vertical axis is the normalized performance.

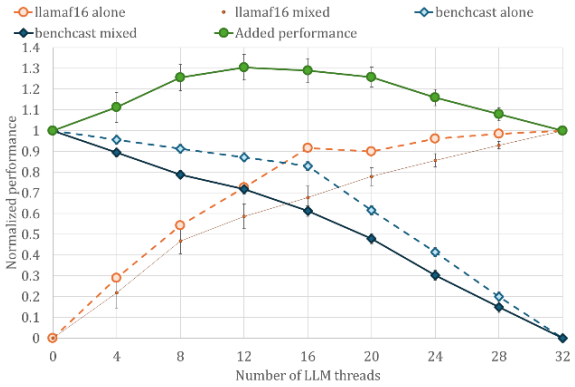


Figure 17: Normalized performance of Llama3-8B fp16 alone, SPEC alone and both applications running together in test system 1.

As shown, the behavior of both LLM and SPEC workloads running alone is similar (note that the number of SPEC threads grows inversely with the number of LLM allocated threads). Both applications exhibit relatively linear throughput growth with the number of threads up to the number of physical cores. Beyond that point, adding more threads leads to diminishing returns as they begin to interfere in the core and private sections of the cache hierarchy. In the case of the LLM model, enabling SMT has a slight negative impact, as there is an imbalance in the work rate of the different threads and the bandwidth limit is almost reached. Bandwidth saturation also explains why performance remains almost identical when moving from 16 to 32 threads (less than 10% difference).

When both workloads run concurrently (solid lines), there is an expected performance loss when looking at them individually, because all cores use SMT. The observed loss is similar to that discussed in section III.B. However, the performance loss from sharing core and cache resources is less than when they have to share them with themselves. When evaluating the normalized added performance of the two workloads, we can see that any combination of them improves the performance obtained when running either of them alone. The best case being when applications of the same type do not collide, and the processor is shared close to 50% (12-16 LLM threads). In this scenario, where both applications

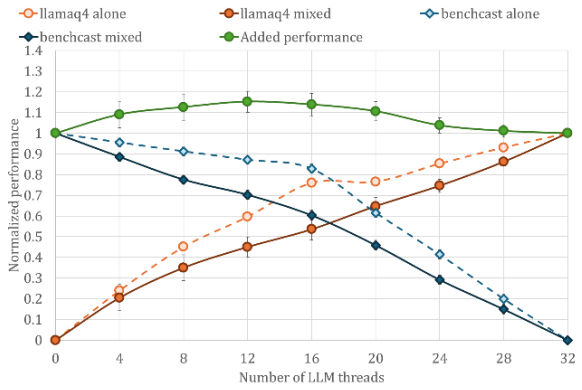


Figure 18: Normalized performance of Llama3-8B Q4 alone, SPEC alone and both applications running together in test system 1.

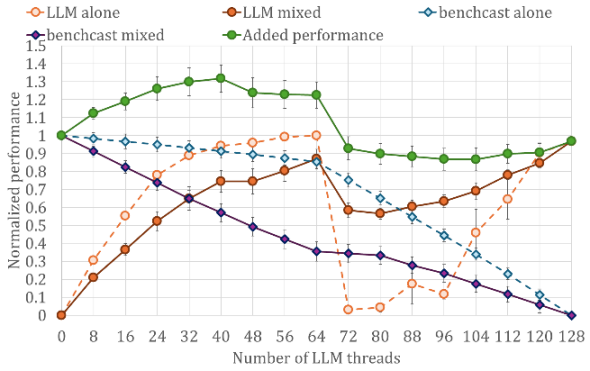


Figure 19: Normalized performance of Llama3-8B fp16 alone, SPEC alone and both applications running together in test system 2.

achieve their maximum performance per thread, an improvement of nearly 30% in overall system utilization is observed.

If we repeat the same analysis with Llama3-8B Q4 (Figure 18), we can see that although the result is similar, the total improvement observed with the combined execution is slightly lower, with maximum values close to a 20%. This is consistent with what was observed in Section III.B, where we saw that both SPEC applications and LLM had slightly higher interference when sharing the core due to the dequantization. Furthermore, as previously noted, Llama3-8B is able to further increase its performance in the presence of SMT using Q4, primarily due to the memory bandwidth utilization, which limits any performance gains at lower thread counts. Nevertheless, combined execution still yields performance improvements up to 24 LLM threads, peaking at 12-16 threads, similar to the fp16 configuration.

Finally, we conduct a similar experiment on a slightly older system with different characteristics. Test system 2 represents a very different use case than the previous one but is still a representative system of what can be found in a cloud infrastructure. While test system 1 had a generous available bandwidth per core, the available bandwidth per core in test system 2 is lower because it has a different memory standard (DDR4 vs. DDR5) and 4 times the number of cores than test system 1 has. The number of memory channels used is limited to half of the available hardware capacity to intentionally exacerbate the memory constraints faced by the LLM (i.e., a form of worst-case scenario). In addition, the processor architecture is different in terms of cache hierarchy and core organization, and does not support AVX 512 (i.e., approximately half of compute bandwidth of test system 1).

Figure 19 presents the performance results on test system 2, where SPEC and LLM are run in a similar fashion to test system 1. As shown, Llama3-8B FP 16 reaches memory bandwidth saturation at 32 threads when running alone on this system, meaning that performance hardly improves beyond that point. In addition, performance drops dramatically in this system when LLM uses SMT, probably due to the high degree of collision in core and memory (at saturated bandwidth). Synchronization between threads, which must run at the pace of the slowest one, could exacerbate this effect, coupled with thread migrations observed during the execution (not observed in test system 1, despite of the use of the same software stack in both systems). In contrast, conventional applications get similar results when running alone, as observed in test system 1.

When both applications are combined, the conventional applications suffer a higher degree of interference than in test system 1, with some impact from collision in the LLC. This is most likely because the insertion-replacement policy in test system 1 might effectively prevent LLC pollution from the LLM, while it is less able to do so in test system 2. In any case, the near-peak performance achieved by the 32-40-thread LLM model allows the normalized combined performance of the two applications to improve by up to 30% over the performance when either is run alone, since there is almost no benefit to adding additional LLM threads and conventional applications get to run with little impact on the LLM model performance. Thus, even in this worst-case scenario, the observations from the more favorable scenario still hold.

5 Conclusions

The results presented in this paper demonstrate the low hardware resource utilization of the LLM models on a general-purpose processor. Their limited use of the cache hierarchy, coupled with the simplicity of their code, allows them to run alongside other conventional applications such as SPEC with reduced impact.

Based on the findings presented in this paper, running both SPEC applications and LLM models in parallel, without leveraging SMT, shows minimal impact on their behavior. Since the main limitation on thread scaling in LLMs is memory bandwidth, it would be possible to launch as many LLM threads as possible until reaching a point close to memory bandwidth saturation and then fill the remaining CPU cores with conventional applications such as SPECs to achieve maximum performance results at reduced cost.

In addition, the limited in-core collision that LLM models have with conventional applications such as SPECs allows for the combined execution of both using the SMT. This results in a cumulative weighted performance that is higher than running either of them independently. This sweet spot depends on the system, but it has been shown to be close to 30% improvement in a real system.

Acknowledgments

This work was supported by the Spanish Government (Ministerio de Ciencia, Innovación y Universidades / Agencia Estatal de Investigación) under grant PID2022-139664NB-I00.

References

[1] [1] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 6000–6010.
[2] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," May 2020, doi: 10.48550/arxiv.2005.14165.
[3] D. Amodei *et al.*, "Deep speech 2: End-to-end speech recognition in English and Mandarin," in *33rd International Conference on Machine Learning, ICML 2016*, 2016, vol. 1.
[4] J. Li *et al.*, "Large Language Model Inference Acceleration: A Comprehensive Hardware Perspective." 2024.
[5] A. Rodriguez, E. Segal, E. Meiri, E. Fomenko, Y. J. Kim, and H. Shen, "Lower Numerical Precision Deep Learning Inference and Training," *Intel White Paper*, 2018.
[6] G. Henry, P. T. P. Tang, and A. Heinecke, "Leveraging the bfloat16 Artificial Intelligence Datatype for Higher-Precision Computations," in *Proceedings - Symposium on Computer Arithmetic*, 2019, pp. 69–76. doi: 10.1109/ARITH.2019.00019.
[7] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 450–462. doi: 10.1145/2749469.2749475.
[8] S. Chen, C. Delimitrou, and J. F. Martinez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2019. doi: 10.1145/3297858.3304005.
[9] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2014. doi: 10.1145/2541940.2541941.
[10] L. Pons, M. Navarro, S. Petit, J. Pons, M. E. Gómez, and J. Sahuquillo, "SMT efficiency in supervised ML methods: a throughput and interference analysis," *Journal of Big Data*, vol. 11, no. 1, p. 152, 2024, doi: 10.1186/s40537-024-01013-5.
[11] S. Mittal, P. Rajput, and S. Subramoney, "A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, 2021, doi: 10.1109/TNNLS.2021.3071762.
[12] J. Park *et al.*, "Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications." 2018.
[13] Q. Hu *et al.*, "Characterization of large language model development in the datacenter," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 709–729.
[14] M. Wang *et al.*, "Characterizing Deep Learning Training Workloads on Alibaba-PAI," in *Proceedings of the 2019 IEEE International Symposium on Workload Characterization, IISWC 2019*, 2019. doi: 10.1109/IISWC47752.2019.9042047.
[15] S. Na, G. Jeong, B. H. Ahn, J. Young, T. Krishna, and H. Kim, " Understanding Performance Implications of LLM Inference on CPUs ," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, 2024, pp. 169–180. doi: 10.1109/IISWC63097.2024.00024.
[16] H. Shen, H. Chang, B. Dong, Y. Luo, and H. Meng, "Efficient LLM Inference on CPUs," *arXiv*, vol. abs/2311.0, 2023.
[17] "SPEC CPU 2017," 2017. https://www.spec.org/
[18] P. Prieto, P. Abad, J. A. Gregorio, and V. Puente, "Fast, Accurate Processor Evaluation through Heterogeneous, Sample-based Benchmarking," *IEEE Transactions on Parallel and Distributed Systems*, 2021, doi: 10.1109/TPDS.2021.3080702.
[19] G. Gerganov, "Llama.cpp: Inference in pure C/C++."
[20] G. Gerganov, "ggml: tensor library for machine learning."
[21] P. Zhang, G. Zeng, T. Wang, and W. Lu, "TinyLlama: An Open-Source Small Language Model," *arXiv*, vol. 2401.02385, 2024.
[22] R. Li *et al.*, "StarCoder: may the source be with you!," *arXiv*, vol. abs/2305.0, 2023.
[23] H. Touvron *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models." 2023.
[24] A. Q. Jiang *et al.*, "Mistral 7B." 2023.
[25] A. Grattafiori *et al.*, "The Llama 3 Herd of Models." 2024.
[26] S. Black *et al.*, "GPT-NeoX-20B: An Open-Source Autoregressive Language Model." 2022.
[27] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger, "A Closer Look at Intel Resource Director Technology (RDT)," in *ACM International Conference Proceeding Series*, 2022. doi: 10.1145/3534879.3534882.
[28] M. Brysbaert, "How many words do we read per minute? A review and meta-analysis of reading rate," *Journal of Memory and Language*, vol. 109, 2019, doi: 10.1016/j.jml.2019.104047.
[29] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2014. doi: 10.1109/ISPASS.2014.6844459.
[30] "Intel PMU Profiling Tools Source Code and Documentation." https://github.com/andikleen/pmu-tools
[31] A. C. de Melo, "The New Linux 'perf' Tools," *17 International Linux System Technology Conference*. Nuremberg, 2010.
[32] Y. Xiang, C. Ye, X. Wang, Y. Luo, and Z. Wang, "EMBA: Efficient memory bandwidth allocation to improve performance on intel commodity processor," in *ACM International Conference Proceeding Series*, 2019. doi: 10.1145/3337821.3337863.
[33] J. Park, S. Park, and W. Baek, "CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proceedings of the 14th EuroSys Conference 2019*, 2019. doi: 10.1145/3302424.3303963.
[34] I. Corporation, "Intel®64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," vol. 3, no. 253665, pp. 1–1386, 2013.
[35] M. Xu, L. T. X. Phan, H. Y. Choi, and I. Lee, "VCAT: Dynamic cache management using CAT virtualization," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, 2017. doi: 10.1109/RTAS.2017.15.
[36] M. Xu, R. Gifford, and L. T. X. Phan, "Holistic multi-resource allocation for multicore real-time virtualization," in *Proceedings - Design Automation Conference*, 2019. doi: 10.1145/3316781.3317840.
[37] A. Limaye and T. Adegbija, "A Workload Characterization of the SPEC CPU2017 Benchmark Suite," in *Proceedings - 2018 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2018*, 2018, pp. 149–158. doi: 10.1109/ISPASS.2018.00028.