Enabling Spatially Fine-Grained DVFS in Neural Processing Units for Energy-Efficient LLM Serving

Yuqi Xue
yuqixue2@illinois.edu
University of Illinois Urbana-Champaign

Jerry Wu
tw42@illinois.edu
University of Illinois Urbana-Champaign

Corey Yu
kaiyiyu2@illinois.edu
University of Illinois Urbana-Champaign

Jian Huang
jianh@illinois.edu
University of Illinois Urbana-Champaign

Abstract

As neural processing units (NPUs) evolve rapidly to accommodate the ever-increasing compute demand of large language models (LLMs), their power consumption is becoming a limiting factor. Our study shows that using dynamic voltage and frequency scaling (DVFS) to exploit the service-level objective (SLO) slacks is a promising way to improve NPU energy efficiency for LLM services. And as tensor operators in LLMs exhibit diverse bottlenecks across NPU components, it is desirable to configure the frequency separately for each component to maximize their energy efficiency.

In this paper, we develop eNPU that enables hardware and software support for spatially fine-grained, component-level DVFS on NPUs. eNPU refactors the NPU core pipeline to partition components into separate V/f domains. It introduces lightweight cross-domain communication mechanisms to mitigate synchronization overheads across components, and extends the NPU ISA for sub-μs DVFS control. eNPU uses a compiler-driven two-level greedy search to co-optimize instruction scheduling and per-component V/f selection under SLO constraints. We implement eNPU's pipeline design on an open-source NPU core to verify its functionality and evaluate the energy savings with a production-level NPU simulator with various LLMs using production traces. eNPU reduces energy consumption of LLM services by 25.8%–35.2% with 3.45% area overhead on a TPUv4 chip, while preserving strict SLO guarantees.

1 Introduction

As neural processing units (NPUs) are are widely deployed in cloud platforms to accelerate large language model (LLM) inference today [8, 32, 60], they have become a major contributor to power consumption in datacenters [27, 45, 54]. It is critical for cloud providers to maximize the energy efficiency of NPUs while guaranteeing service-level objectives (SLOs) for LLM services.

We first conduct a study to characterize the energy consumption of LLM inference on NPUs (see §2.3). We run production LLM inference traces [55] on NPUs with diverse LLM architectures, including both dense (Llama [39]) and mixture-of-experts (MoE) models (DeepSeek [17]). We quantify the energy consumption and resource utilization of all major components on an NPU (systolic arrays, vector units, SRAM, HBM, and ICI, as described in §2.1).

Our study reveals that dynamic energy accounts for 18%–61% of total energy, and static energy accounts for 27%–55% for serving various LLM models (Figure 2). The energy consumption pattern is determined by the diverse resource utilizations of different components on an NPU chip (Figure 3), such as the systolic arrays, vector units, SRAM, HBM, and ICI (see §2.1). Furthermore, in each LLM inference request, the utilization pattern changes over time (Figure 4), as different tensor operators (e.g., matrix multiplication, softmax) exhibit diverse performance demands on different components. For example, the FFN layers during the prefill phase of a request are often systolic array-bound and underutilize vector units and SRAM bandwidth, while attention layers during the decode phase are often HBM-bound and underutilize systolic arrays.

To save static power, a recent study [65] enabled power gating for NPUs. To save dynamic power, a well-known technique is dynamic voltage and frequency scaling (DVFS), which has been widely employed in CPUs [20] and GPUs [10] by adjusting the voltage and frequency (V/f) of each CPU core or GPU SM (streaming multiprocessor). However, DVFS for NPUs remains coarse-grained, treating the whole NPU core or chip as a single V/f domain [60]. This inevitably misses significant energy-saving opportunities, as it cannot adjust the V/f of each component in an NPU chip independently to precisely match its performance demand. Ideally, we wish to enable *spatially fine-grained DVFS*, allowing each component to operate at its energy-optimal V/f state to maximize energy savings with minimal negative impact on the performance of LLM serving.

However, realizing spatially fine-grained (i.e., component-level) DVFS on NPUs introduces three major challenges:

First, modern NPUs do not provide architectural support for component-level V/f domains. The NPU core pipeline is tightly coupled by design, with all components coordinated through a centralized frontend (including the vector register file and a scoreboard for dependency tracking). Supporting independent V/f domains requires carefully refactoring the pipeline to mitigate frequency mismatches and cross-domain synchronization overhead.

Second, component-level DVFS complicates compiler scheduling. Compilers for NPUs statically schedule instructions, relying on deterministic instruction latencies to perform optimizations. With component-level DVFS, instruction latencies become frequency-dependent, and different V/f choices can change both instruction schedule and the bottleneck of an operator. As a result, the compiler must co-optimize DVFS decisions and instruction scheduling.

Third, component-level DVFS greatly expands the optimization space. Prior DVFS techniques assume a single coarse-grained compute domain and only decide how much to slow down each operator under a performance target. In contrast, component-level DVFS must jointly determine the V/f state of multiple components for every operator, creating a search space that is exponentially larger.

Our solution. We propose eNPU to enable hardware and software support for component-level DVFS on NPUs.

First, we refactor the NPU core pipeline to support component-level V/f domains with low hardware overhead. eNPU partitions the frontend, SA, VU, SRAM, HBM, and ICI into separate V/f domains and introduces lightweight cross-domain communication mechanisms to preserve throughput at varying frequencies. We carefully size asynchronous FIFOs to absorb synchronization delays and introduce a lightweight shadow vector register file to enable local forwarding within the VU domain, preventing severe pipeline stalls caused by back-to-back dependent vector instructions.

eNPU leverages the ML compiler to manage DVFS at the tensor operator granularity. We extend the NPU ISA with a new `vf.set` command (Figure 9) that allows software to set the V/f state of different components at sub-μs scale. This compiler-driven approach exploits the highly predictable execution of ML workloads, allowing the compiler to accurately determine bottlenecking components and estimate the performance of a scheduled VLIW code snippet.

The ML compiler must search a massive optimization space—hundreds to thousands of operators in a request, multiple components per operator, and many V/f states per component—for a solution that does not violate SLO. eNPU decomposes this problem into two levels. It first performs an operator-level search to identify near-Pareto-optimal DVFS plans that capture the energy-delay tradeoff of each operator. Then, it performs a request-level search to select among the Pareto plans to minimize end-to-end energy under available SLO slack (slowdown allowed for a request without violating its SLO)[1]. Since the energy-delay tradeoff of DVFS is convex ($P \propto V^2 f$), eNPU can efficiently search the Pareto plans with a simple gradient-descent search, guided by the ML compiler's instruction-level cost model that captures how DVFS changes instruction latencies and cross-domain synchronization overheads.

To integrate eNPU into the LLM serving stack, we first perform offline analysis to identify the optimized DVFS plans for precompiled graphs for various batch sizes, sequence lengths, and SLO slack settings. At runtime, the request scheduler tracks each request's SLO slack and selects the appropriate DVFS plan. eNPU employs simple safeguards to prevent head-of-line blocking due to V/f throttling and temporarily locks NPU chips to the peak V/f state to prevent SLO violations during load spikes.

We implement eNPU's hardware design with the open-source Coral NPU core [2]. eNPU incurs 6.16% area overhead on Coral NPU, and the projected area overhead is 3.45%/3.61% on TPUv4/TPUv5p (see §3.5). To evaluate eNPU with LLM workloads, we use a production-level NPU simulator that is validated against real TPU chips. We run LLM services with various dense (Llama [39]) and MoE (DeepSeek [17]) models using a production trace [55]. At a 0% SLO slack, eNPU saves 16.9%/18.6% energy for prefill/decode. At a 10% SLO slack, the savings increase to 22.9%/18.6%. Compared to the state-of-the-art NPU DVFS solution, eNPU delivers consistently higher energy savings, improving over it by up to 9.0%/5.7% for prefill/decode. Combining eNPU with power gating [65] further achieves up to 31.5% energy savings. We summarize our contributions as follows:

[1]Our study with real LLM service traces finds that there is abundant SLO slack, offering opportunities for us to exploit fine-grained DVFS on NPUs for energy saving without violating SLO of LLM services (see our detailed study in §2.3).

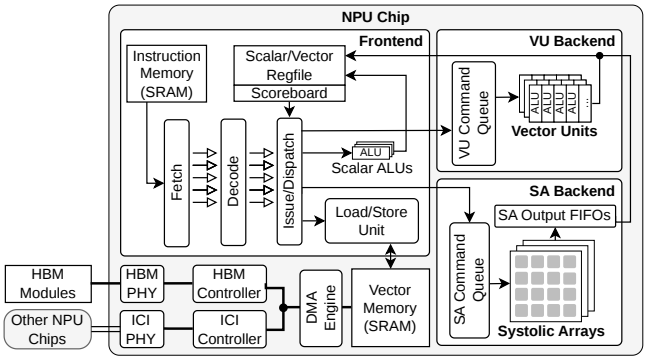

Figure 1: Architecture of an NPU chip (link between instruction memory and DMA engine is omitted for clarity).

- We quantify the DVFS opportunities of LLM serving on NPUs, showing that request-level SLO slack and operator-level component underutilization create substantial room for energy savings.
- We propose eNPU, a hardware-software co-design that enables spatially fine-grained, component-level DVFS on NPUs through low-overhead pipeline support and ISA extensions.
- We develop a compiler-driven DVFS policy that co-optimizes instruction scheduling and per-component V/f settings using an efficient two-level search algorithm.
- We implement and evaluate eNPU with hardware synthesis and a production-level NPU simulator, demonstrating significant energy savings while preserving strict SLO guarantees.

2 Background and Motivation

2.1 NPU Architecture

NPU core pipeline. Neural processing units (NPUs) are specialized machine learning (ML) accelerators. A typical example is Google TPU [2, 32, 41]. Figure 1 shows the NPU core pipeline. The frontend fetches VLIW instructions and dispatches them in-order. Each VLIW instruction consists of multiple *commands*. To prevent data and structural hazards, the architecture employs a lightweight scoreboard using a wait bit per register to stall conflicting commands. Once issued, commands are routed to specialized execution backends: systolic arrays (SAs) handle matrix multiplications and convolutions, while vector units (VUs) execute generic vector computations such as element-wise and activation functions. Commands can retire out-of-order, overlapping execution across components.

The memory hierarchy employs an on-chip SRAM (vector memory) to exploit data reuse and hide the latency of the off-chip High-Bandwidth Memory (HBM). A Direct Memory Access (DMA) engine orchestrates asynchronous data movement between the SRAM and HBM. To scale out, multiple NPU chips can be interconnected via high-speed inter-chip interconnect (ICI) links to form an NPU pod (typically a 2D/3D torus [72]). The DMA engine orchestrates remote DMA (RDMA) to access other chips' HBM or SRAM.

NPU software stack. An ML program is defined as a computation graph of tensor operators [7, 12, 21, 57]. ML compilers apply optimizations such as operator fusion and tiling. For each operator, the compiler generates low-level commands for each component

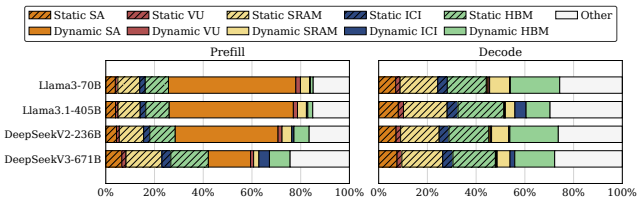

Figure 2: Energy breakdown of an LLM inference request. We use input/output sequence length 4096/512 as an example.

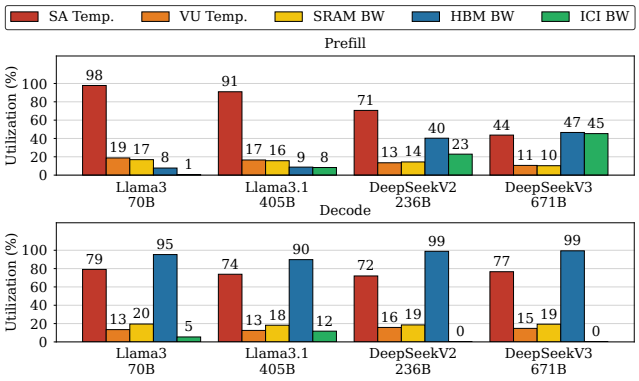

Figure 3: Per-component utilization of an LLM request. "Temp.": temporal utilization. "BW": bandwidth. Numbers smaller than 0.01% are rounded to 0.

(e.g., `push/pop` vectors from/to SAs, vector commands on VUs, `load/store` from/to the SRAM, and DMA commands for HBM/ICI access) and performs static instruction scheduling based on deterministic instruction latencies. The compiler performs register renaming and employs optimizations such as loop unrolling and software pipelining to exploit instruction-level parallelism (ILP).

2.2 Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) is a common technique for managing power and energy consumption. DVFS is enabled by three key primitive components as discussed below.

Integrated voltage regulators (IVRs). IVRs convert a high input voltage to the localized voltage required by a power domain. With recent packaging technologies, IVRs can be integrated within the same package as the NPU die, providing nanosecond-scale voltage transitioning times and high power conversion efficiency [10, 34].

Frequency generation. To enable fast DVFS, IVRs are paired with a clock generator to adjust frequency in nanoseconds [10]. One approach uses a phase-locked loop (PLL) to generate a reference clock and a digital frequency synthesizer (DFS) to quickly select derivative output frequencies. An alternative approach is using voltage-adaptive frequency-locked loops (FLLs/DFLLs), which can adjust output frequency by tracking the supply voltage.

Cross V/f domain synchronization. For voltage domain crossing, *voltage level shifters* [29] are used. They incur negligible wiring delays (typically less than a clock cycle at maximum frequency). For clock domain crossing, control signals rely on multi-stage flip-flops to mitigate metastability [40] and recirculation MUX to ensure data coherency [16]. For high-throughput data buses, asynchronous FIFOs are employed, which consist of a dual-port SRAM to stage data and recirculation MUXes to synchronize control signals [40].

2.3 DVFS Opportunities in NPUs

In this section, we quantify the DVFS opportunities of large language model (LLM) inference services on NPUs. We cover popular LLM architectures and different model sizes, including dense models (Llama3 [39]) and mixture-of-experts (MoE) models (DeepSeek [17]). As LLM prefill and decode phases have distinct resource demands, we employ state-of-the-art prefill/decode (P/D) disaggregation [44, 71] and study the P/D phases separately. We use a production-level NPU simulator (see §3.5) to conduct our study. We study the energy and resource utilization of all major components (SAs, VUs, SRAM, ICI, HBM) when running a single LLM request. As LLM requests have diverse service-level objectives (SLOs) [55], we also analyze cluster-level scheduling to quantify each request's SLO slack (i.e., how much it can slow down without violating its SLO).

Energy consumption breakdown. We run a single request of input/output sequence length 4096/512 (see Table 3 for the P/D node configuration) and break down the component energy in Figure 2. Since prefill is compute-bound, dynamic SA energy generally accounts for over 50% of total energy. An exception is DeepSeekV3-671B, where MoE dispatch/combine incurs high ICI overhead. The decode phase is memory-bound, so most dynamic energy is spent on HBM. Static energy accounts for 27%–55% for prefill and decode.

Diverse component utilization. Figure 3 shows per-component utilization, which correlates with the dynamic energy in Figure 2, except for the SA exhibits over 70% temporal utilization but consumes little dynamic energy in decode phase. This is because decode processes a single token at a time, incurring significant SA padding overhead and underutilizing FLOPS. Underutilized components draw substantial static energy (27%–44% for prefill and 48%–54% for decode). Because they are often not critical to end-to-end performance, we can throttle their voltage and frequency to save energy with minimal performance degradation.

Figure 4 further shows component utilization over time. Both prefill and decode consist of diverse tensor operators, and they exhibit distinct resource demands on each component. Our profiling reveals that the average operator execution time ranges from 1.5 μs to 630.5 μs across the evaluated models (see Table 1). This timescale makes operator-level DVFS feasible with modern IVRs that support nanosecond-scale transition times [10, 20]. We can adjust the V/f separately for each component to match the precise demands of each operator, saving energy with minimal performance penalty.

SLO slacks. LLM requests vary in input/output sequence lengths, which introduce diverse service-level objectives (SLOs) [55]. Moreover, production LLM services often over-provision resources to meet SLOs for large requests and load spikes, allowing performance to be traded for energy savings via DVFS without SLO violation.

To fairly quantify the exploitable SLO slack, we replay a production LLM service trace from AzurePublicDataset [55] (see §4.1). We categorize request sequence lengths into the 33rd/66th/100th

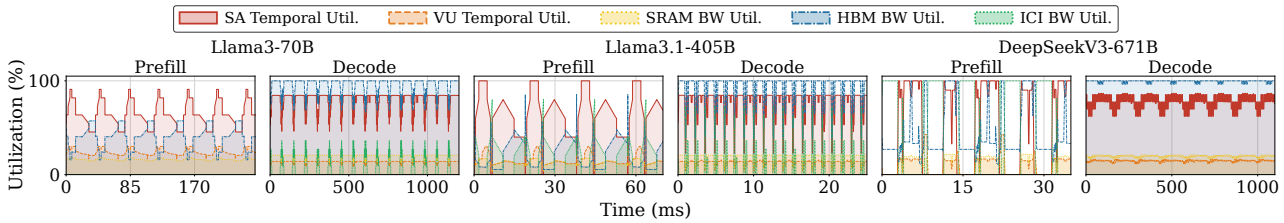

Figure 4: Utilization of each component over time. Since the transformer layers have the same architecture in each model, we show a representative time window as an example in each subplot. We do not show DeepSeekV2-236B due to space limitations.

Table 1: 1st percentile (P1), mean, standard deviation, and 99th percentile (P99) operator execution time in different LLMs. The request input/output sequence length is 4096/512.

Model	Phase	P1 (μs)	Mean (μs)	Std (μs)	P99 (μs)
Llama3-70B	Prefill	3.33	630.45	744.26	2158.85
	Decode	0.50	14.11	13.53	45.57
Llama3.1-405B	Prefill	5.65	305.60	345.37	1002.34
	Decode	0.50	19.76	20.83	49.95
DeepSeekV2-236B	Prefill	0.50	10.51	78.95	173.50
	Decode	0.50	1.45	4.06	9.99
DeepSeekV3-671B	Prefill	0.50	18.43	151.95	308.42
	Decode	0.50	2.03	9.12	49.95

Table 2: SLO settings used in our experiments. Sequence length percentiles: P33 (input: 1,291; total: 1,313), P66 (input: 2,777; total: 2,802), P100 (input: 7,691; total: 9,071). For prefill, we use input lengths. For decode, we use total lengths, as the TPOT is correlated to the total KV cache size.

Model	TTFT (sec)			TPOT (millisec)		
	P33	P66	P100	P33	P66	P100
Llama3-70B	0.725	1.488	4.405	91	93	100
Llama3.1-405B	0.803	1.508	5.660	709	709	710
DeepSeekV2-236B	0.642	1.302	5.158	271	286	339
DeepSeekV3-671B	1.098	2.151	6.113	297	323	412

percentiles and set the SLO target for each request as 5× the single-request TTFT/TPOT on the minimum number of NPU chips [55]. Table 2 shows the SLO settings. We sweep NPU pod sizes, LLM parallelism degrees, and batch sizes to identify the most energy-efficient configuration that satisfies the SLO for all requests. We sweep P/D node counts to determine the minimum cluster provisioning required to achieve 99% SLO satisfaction across the trace [44, 71]. Table 3 shows the P/D node and NPU pod configurations.

Figure 5 reports the SLO slack over time. For most requests, the TTFT/TPOT is shorter than 10%/20% of the SLO target. Ideally, they can be slowed down by up to 10×/5× during prefill/decode without violating SLOs[2]. When the request rate increases (i.e., between 14 and 22 hours), the SLO slack drops as the queuing delay increases.

[2]An alternative way to DVFS is to leverage cluster-level scheduling techniques such as auto-scaling to reduce the number of NPU chips allocated to an LLM service during off-peak hours. They can be employed together with eNPU, as eNPU saves abundant energy during peak hours (see §4.8). Besides, eNPU can also help save energy for a smaller NPU allocation during off-peak hours.

Table 3: The prefill/decode node and cluster configurations used in our experiments. "Config." is shown as data/tensor/pipeline/expert parallelism degrees.

Model	Prefill Node		Decode Node		# of Nodes	
	Chips	Config.	Chips	Config.	P	D
Llama3-70B	4	1/4/1/1	4	1/4/1/1	24	8
Llama3.1-405B	32	1/32/1/1	64	1/64/1/1	24	52
DeepSeekV2-236B	4	1/4/1/1	4	1/4/1/1	24	24
DeepSeekV3-671B	32	1/16/1/2	32	1/16/1/2	36	24

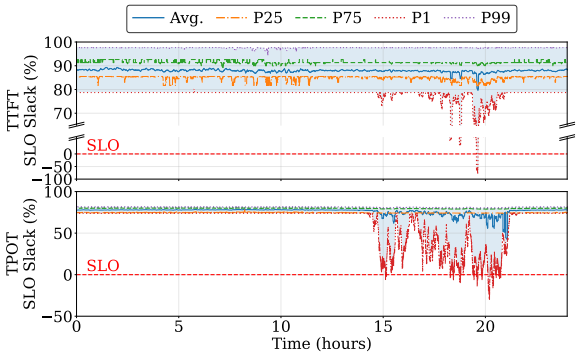

Figure 5: Normalized SLO slack time over time. For each request, the SLO slack is computed as (SLO target − actual latency)/(SLO target). At each timestamp in the graph, the P1, P25, Avg., P75, and P99 values on the curves represent the SLO slack distribution of all requests finished in the past 5-min time window (e.g., 5-min rolling window). We plot Llama3-70B as an example (other models have a similar trend).

However, at least 75% of requests still have more than 80% SLO slack during prefill (similarly for decode). A few requests (less than 1%) violate their SLOs and have negative slacks. This aligns with our goal of 99% SLO satisfaction rate.

2.4 Challenges of Component-Level DVFS

DVFS techniques are widely studied in CPUs/GPUs. They typically employ multiple V/f domains across *homogeneous* cores, along with a separate uncore domain (last-level cache, memory controller, etc.). However, an NPU chip typically features a large core consisting of *heterogeneous* components (SA, VU, etc.). Enabling component-level DVFS on NPU chips introduces unique challenges.

Monolithic NPU pipeline vs. independent V/f domains. Conventional NPU pipelines are tightly coupled: a centralized frontend dispatches commands to components, which share data via the register file. Component-level DVFS requires decoupling them into separate V/f domains and inserting asynchronous FIFOs and voltage level shifters between domains. This requires every instruction to cross V/f domain boundaries twice (dispatch and retire), significantly inflating instruction latency. This can severely degrade performance when dependent instructions are close to each other, such as elementwise or reduction operators on the VU, as the compiler cannot find enough independent work to hide the extra delay. Thus, component-level DVFS requires careful refactoring of the NPU core pipeline beyond simply inserting domain-crossing logic.
Statically scheduled VLIW ISA vs. frequency-dependent instruction latencies. Modern NPUs use a statically scheduled VLIW ISA to reduce hardware complexity. The ML compiler assumes deterministic instruction latencies to perform low-level optimizations such as loop unrolling, software pipelining, and VLIW instruction packing [32, 41]. With component-level DVFS, instruction latencies become frequency-dependent. Changing frequency of one component can alter the optimal instruction schedule (e.g., changing software pipeline depth or loop unrolling factors) or shifting the bottleneck of an operator, such as from compute-bound to memory-bound. This makes prior DVFS approaches ineffective, as they assume a fixed code sequence and only search for the best DVFS setting for that fixed schedule [60]. Instead, the compiler must co-optimize instruction scheduling and DVFS decisions.
Complex DVFS plan search space. Prior compiler-assisted DVFS techniques for NPUs typically assume a single coarse-grained compute domain and decide how much each operator can slow down under a performance target [60]. With component-level DVFS, we must determine the V/f state of all components for every operator. This expands the search space combinatorially. As each candidate V/f plan may change instruction latencies, scheduling decisions, and operator bottlenecks, the compiler cannot rely on simple heuristics over a fixed code sequence. We need a systematic approach to efficiently explore the component-level DVFS search space.

3 Design and Implementation

We propose eNPU, a hardware-software co-design for component-level DVFS on NPUs. eNPU partitions the NPU into separate V/f domains (§3.1). It exposes fine-grained DVFS control by extending the NPU ISA (§3.2), which enables the compiler to control DVFS (§3.3). At runtime, the LLM serving system tracks request-level SLO slack and applies eNPU's DVFS decisions (§3.4).

3.1 Hardware Support for Fine-Grained DVFS

In this section, we present the necessary hardware support for enabling component-level V/f domains on NPUs.
Fine-grained V/f domains. Figure 6 shows the component-level V/f domains in eNPU. Each domain is powered by its own IVR and clock generator. The *frontend* domain coordinates the NPU execution, so its frequency must be higher than all other domains to avoid becoming a bottleneck. The *SA*, *VU*, and *SRAM* domains exchange data via the register file in the frontend.

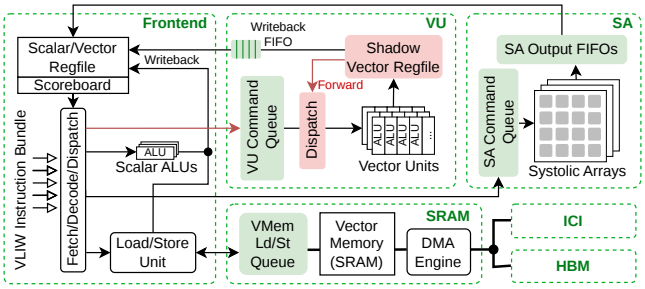

Figure 6: NPU core pipeline divided into different V/f domains. Each V/f domain is marked by a dashed green box. The asynchronous FIFO queues used for inter-domain synchronization are also in green. The queues for HBM and ICI controllers are omitted from the figure for simplicity.

The *HBM* and *ICI* domains include on-chip controllers, I/O links, and DRAM dies. While the controller is standard CMOS logic and supports DVFS by default, the V/f of I/O and DRAM core are defined by strict standards [30]. To abstract this complexity, eNPU interfaces directly with the HBM/ICI controller, allowing software to set a target frequency state (see §3.2). The controller then utilizes an internal lookup table to select the appropriate V/f for the I/O link or DRAM dies. This design ensures eNPU seamlessly supports both current-generation hardware with controller-only DVFS and future standards that support different I/O voltages or DRAM core voltages, such as HBM4 [31]. We evaluate both scenarios in §4.
Cross-domain communications. For the SA, VU, and SRAM, eNPU replaces the command queues, output FIFOs, and load/store queues with async FIFOs and voltage level shifters. Each async FIFO must be sized appropriately to avoid pipeline bubbles at minimal hardware cost. Let T_i and T_o be the clock periods of the input (producer) and output (consumer) domains. To sustain maximum steady-state throughput, the FIFO must absorb the round-trip feedback latency of synchronizing the gray-coded read/write pointers via 3-stage flip-flops. Synchronizing the write pointer to the output domain takes $5T_o$ (1 cycle for phase alignment, 3 cycles for the flip-flops, and 1 cycle for logic). Similarly, synchronizing the read pointer to the input domain takes $5T_i$. The round-trip latency is

$$T_{rt} = 5T_i + 5T_o. \tag{1}$$

Assuming the FIFO input/output rate is 1 entry/cycle (we can use multiple FIFOs to scale the rate), the maximum steady-state pipeline throughput is $1/\max(T_i, T_o)$, and the minimum queue depth is

$$Depth = \left\lceil T_{rt} \cdot \frac{1}{\max(T_i, T_o)} \right\rceil = \left\lceil \frac{5T_i + 5T_o}{\max(T_i, T_o)} \right\rceil. \tag{2}$$

In the worst case, $Depth = 10$ when $T_i = T_o$. Hence, each async FIFO in eNPU has a minimum depth of 10 to prevent pipeline stalling due to cross-domain synchronizations.
VU internal forwarding. Many operators issue back-to-back data-dependent commands, especially elementwise and reduction operators on VUs; profiling shows that in at least 50% of cases, data-dependent VU instructions are separated by at most 3 independent instructions. Figure 8 quantifies the dependency distances (i.e., distance is 1 for consecutive commands, and n+1 for those separated by

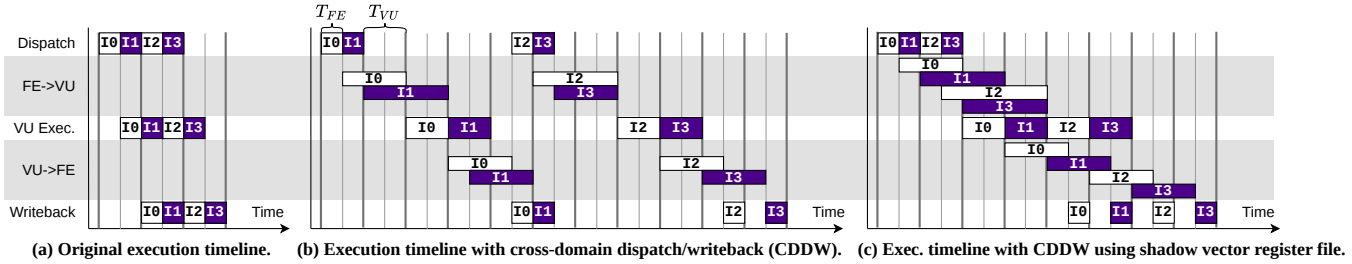

Figure 7: Example of instruction execution timeline with/without VU internal forwarding. VU frequency is set to the frontend frequency in (a), and halved in (b) and (c). Same-color commands have back-to-back data dependencies and must execute in order. FE: frontend. T_{FE}: frontend clock period. T_{VU}: VU clock period. For simplicity, domain crossing takes 1 cycle in each domain (i.e., only clock phase alignment is accounted for).

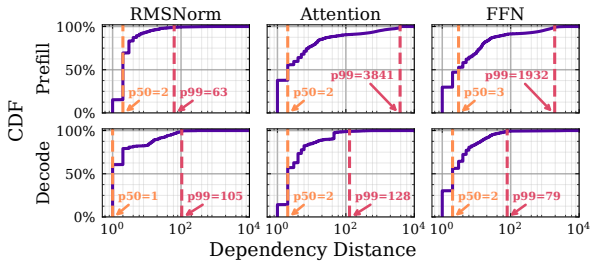

Figure 8: Dependency distances of VU commands in representative Llama3-70B layers, with input/output sequence length 4096/512, for the generated VLIW bundles on TPUv4 chips.

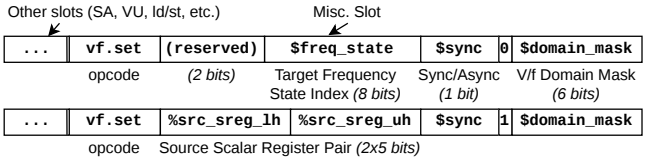

Figure 9: ISA extension for component-level DVFS.

n commands) for all pairs of data-dependent VU commands in representative LLM layers. Only immediate (direct) dependencies are included. With the V/f domain crossing, the frontend scoreboard enforces that a consumer command cannot be dispatched until its producer retires, so each command must wait for the complete round-trip latency of the previous command. This incurs pipeline bubbles that cannot be easily hidden via compiler optimizations.

As an example, Figure 7a shows the execution timeline of two pairs of VU commands (the dependency chain is `I0`→`I2` and `I1`→`I3`) with dependency distance 2 without V/f domain crossing overhead. Assuming VU frequency is halved, with domain crossing overhead (`FE->VU` and `VU->FE` in Figure 7b), the IPC (commands per frontend cycle) drops from 1 to 0.31. The compiler must find at least 6 independent VU commands to achieve the ideal IPC of 0.5.

To address this issue, we employ a lightweight *shadow vector register file* in the VU domain so that register values can be forwarded to dependent VU commands without crossing between VU and frontend. To avoid extra hardware scheduling logic, the n shadow registers map to the first n architectural registers. eNPU leverages the compiler to manage shadow registers. The compiler calculates priority for register allocation based on dependency distance between each producer-consumer command pair. If the dependency distance between two commands is long enough to hide the FE-VU round-trip (T_{rt} in Equation (1) plus VU execution delay), they can directly access the FE regfile without stalling, so shadow registers should not be allocated. Otherwise, the shorter the dependency distance is, the longer the FE would stall without forwarding, so the compiler prioritizes command pairs with shortest dependency distances, assigning the first n architectural registers when possible.

A local scoreboard in the VU domain enforces data dependencies involving shadow registers; the frontend does not stall for VU commands that depend on them. As shown in Figure 7c, this forwarding mechanism achieves ideal IPC without requiring extra independent commands, despite the longer pipeline warmup.

3.2 ISA Extension for µs-scale DVFS on NPUs

As tensor operators execute on a μs-scale (see Table 1), it is desirable to enable software DVFS configuration at sub-μs scale. To do so, we extend the NPU ISA with a new command "`vf.set`", as shown in Figure 9. The command occupies the "misc." slot in a VLIW instruction and can be issued concurrently with SA, VU, and SRAM commands in other slots. It offers two variants for flexibility.

The first variant encodes target frequency as an 8-bit immediate value (`$freq_state`). This can encode up to 256 distinct frequency states, which is typically sufficient for fine-grained DVFS (in §3.5, we use 6 bits to encode 34 states from 0–1.7GHz at every 50MHz step). eNPU uses a lookup table for each component to map each frequency state to its minimum required voltage level. `$domain_mask` specifies which components (frontend, SA, VU, SRAM, HBM, and ICI) to adjust the V/f state. All selected components are scaled to the same target frequency, which is useful for setting all underutilized components in an operator to the lowest V/f level or resetting them to the peak frequency. The `$sync` field specifies whether the `vf.set` is blocking. An asynchronous `vf.set` only blocks the target components, allowing other components to continue execution.

To allow independent V/f adjustments of multiple components in one command, the second variant takes two 32-bit scalar registers

as operands. The two registers are concatenated, and the lower 48 bits are taken as the target V/f states of all domains. Using register operands requires first loading the V/f state values into scalar registers, which typically take an extra 1–3 scalar commands (like `lui` and `addi` in RISC-V). As a VLIW bundle typically contains multiple scalar slots but only one misc. slot, this can save the number of bundles required compared to the first variant. The bit fields for domains that are masked out are ignored by `vf.set`.

3.3 Software-Managed DVFS

Given the SLO slack of an LLM request, eNPU leverages the ML compiler to select an optimized DVFS plan. This requires navigating a combinatorial search space defined by hundreds to thousands of operators, 5 components to configure per operator, and 34 frequency choices – a total of $(34^5)^{1000} \approx 10^{7657}$ DVFS plans.

To systematically navigate this search space, eNPU relies on three key insights. First, we can safely decouple the global search space into tractable *operator-level* and *request-level* search spaces. Second, because the energy-delay trade-off is generally convex ($P \propto V^2 f$), a greedy gradient descent works effectively to find optimized configurations. Third, as we can accurately calculate per-instruction domain crossing delay (see §3.1), we can build a cycle-accurate performance model based on the static instruction schedule within the compiler to rapidly assess a DVFS plan's execution time.

ML compiler workflow. The compiler takes the sequence of operators in an LLM request and its SLO slack as inputs and performs two passes of DVFS plan explorations. Modern ML compilers like XLA [21] and PyTorch [7] typically assume static computational graphs with known tensor shapes and bounded loops. This enables accurate cost models and allows the compiler to perform aggressive compilation-time optimizations, such as tiling, operator fusion, and memory layout selection, based on the tensor shapes.

eNPU performs the DVFS plan search after the compiler has finished all optimizations and generated the final VLIW instruction schedule for each operator. For each operator, eNPU generates DVFS plans that closely estimate the Pareto frontier of energy-delay tradeoffs. Then, it iteratively picks an optimized DVFS plan from the Pareto frontier of each operator until exhausting the SLO slack and generates the NPU program with `vf.set` commands.

Operator-level exploration. To perform static instruction scheduling, NPU compilers typically use an instruction-level cost model, which takes instruction latencies and a sequence of VLIW bundles to evaluate the execution time of an operator. DMA/RDMA operations for HBM/ICI use a well-established link model [3, 5]. eNPU reuses this cost model with new instruction latencies determined by the frequency setting and domain-crossing delay (Equation 1).

For each operator, there are $34^5 \approx 45$M possible DVFS plans. eNPU uses gradient descent to identify the (approximate) Pareto plans that represent the optimal energy-delay tradeoffs for this operator, as shown in Algorithm 1. It starts with the lowest frequencies for all components, which represent the plan with the largest execution time (line 2). In each iteration, it generates the next set of candidate plans by incrementing the frequency of one or more components to the next available frequency point, yielding up to $2^5 - 1 = 31$ plans (line 5). The candidate plans are evaluated with the cost model, and eNPU picks the next plan to be the one that achieves

Algorithm 1: Operator-level DVFS plan exploration.

```
1  Function get_pareto_plans_for_op(op):
       // start with lowest frequencies (50MHz) for all components
2      cur_plan ← (50, 50, 50, 50, 50)
3      explored_plans ← { cur_plan }
4      while any(c < MAX_FREQ for c in cur_plan) do
           // increment freq of ≥ 1 components (up to 31 plans)
5          candidates ← generate_next_candidates(cur_plan)
6          best_cand ← null
7          best_gradient ← ∞
8          for cand in candidates do
               // evaluate candidates using cost model
9              ΔE, ΔT ← evaluate_diff(cur_plan, cand)
               // pick plan with the best energy-delay tradeoff
10             if ΔT < 0 and (ΔE/|ΔT|) < best_gradient then
11                 best_gradient ← ΔE/|ΔT|
12                 best_cand ← cand
           // stop if no candidate achieves better performance
13         if best_cand is null then
14             break
15         cur_plan ← best_cand
16         explored_plans ← explored_plans ∪ { cur_plan }
       // post-processing pass to filter out Pareto plans
17     pareto_plans ← filter_pareto_plans(explored_plans)
18     return pareto_plans
```

the best tradeoff between energy and execution time (i.e., the plan with the smallest $\Delta E/\Delta T$ where ΔE and ΔT are the differences in energy consumption and execution time between the current and candidate plans) (line 8–12). The algorithm stops when no candidate plans achieve better performance or all components reach peak frequency. The selected plan in each iteration is recorded, and a final post-processing pass filters out Pareto plans from the explored plans (line 17). In the worst case, the greedy search increments only one component's frequency per iteration, so the maximum number of evaluated plans per operator is $5 \times 34 \times (2^5 - 1) = 5270$.

Request-level exploration. Given the Pareto plans for all operators, eNPU iteratively searches for the request-level DVFS plan that minimizes energy consumption without violating SLO, using a similar heuristic based on the $\Delta E/\Delta T$ gradient. eNPU starts with the fastest Pareto plan for each operator. In each iteration, it picks the operator whose next plan on the Pareto frontier saves the most energy with the least performance degradation and updates the plan. The algorithm stops if either the SLO slack or candidate plans are exhausted. In the worst case, the request-level search takes $O(NP)$ iterations, where N is the number of operators and P is the number of Pareto plans per operator.

3.4 LLM Serving System Integration

Offline DVFS plan generation. In production LLM serving systems, multiple graphs are compiled ahead-of-time before service deployment for various pre-bucketed batch sizes and sequence lengths. At runtime, the request scheduler [36] selects the most appropriate pre-compiled graph to execute. The batch size and sequence length are padded to the smallest bucket that can accommodate them, avoiding runtime compilation overhead. Background

compilation of extra buckets can be triggered if an unseen batch size or sequence length becomes frequent. eNPU additionally generates the DVFS plan for each batch size, sequence length, and SLO slack. The generated results can be shared by all services using the same backbone LLM. By default, eNPU compiles graphs for all power-of-2 batch sizes from 1 to 16 and multiple-of-16 batch sizes beyond 16, up to 256. For sequence lengths, it uses 128-token increments up to 1024 tokens and doubles the increment at each subsequent power-of-2 boundary, e.g., 256-token increments up to 2048 and 512-token increments up to 4096 [36]. For each graph, eNPU covers an SLO slack of 0%, 2%, 5%, and 10%, which are sufficient for achieving most DVFS benefits (see Figure 11). This yields at most 1728/2112/2304 compilations for up to 128K/512K/1M sequence length. The compilation time is manageable, as this can be done offline just once with massive CPU cores (see §4.3).

DVFS-aware request scheduling. At runtime, the request scheduler picks requests from a queue to form a batch [36, 70]. eNPU preserves the batching policy of the request scheduler, which is typically designed to maintain SLO and improve system throughput. eNPU tracks each request's queuing delay, T_q, and computes its target execution time $T_{target} = SLO - T_q$. After forming a batch, eNPU looks up the best DVFS plan based on the batch size, sequence length, and the smallest T_{target} in the batch. To ensure SLO for all requests in a batch, eNPU selects the DVFS plan assuming all requests have the same sequence length as the longest request.

Slowing down the execution of a batch to save energy may exacerbate head-of-line blocking for subsequent requests. When the system is stressed by load spikes or during peak hours, slowing down the execution causes the entire system throughput to drop proportionally, which can cause significant SLO violations. To avoid this, eNPU enforces two safeguard rules. First, it adjusts the T_{target} of a batch such that it will not directly cause any request in the queue to violate SLO. Second, it monitors the SLO satisfaction rate in the past time window (5 minutes by default) and temporarily locks the NPU chips at the peak frequency if the SLO drops under a specific target (e.g., more than 1% of requests violate SLO).

Support for custom operators. By default, modern ML compilers like XLA [21] and PyTorch [7] assume static computational graphs. For MoE models, dynamic behavior (e.g., varying token counts for MoE experts) is often implemented as a custom operator with a custom cost model; the ML compiler can treat this custom operator similarly to other compiler-generated ones during cost-model-based optimizations. For example, the execution time of an MoE expert depends on the number of tokens assigned to it. By default, the cost model uses an expert capacity factor [25] to estimate the worst-case token count for an expert[3], and eNPU conservatively selects DVFS plans that will not violate SLO, despite sacrificing potential DVFS opportunities. The user can adjust the expert capacity factor for eNPU based on real workload profiling and SLO requirements.

3.5 Implementation

Hardware synthesis. To verify eNPU's pipeline design (§3.1), we implement our design based on the open-source Coral NPU core [2], which is an in-order 4-issue RISC-V core with a RISC-V-compliant vector core (VU), a custom 16×16 matrix core (SA), and SRAM. We implement `vf.set` (§3.2) as a custom RISC-V control instruction and added the async FIFOs and shadow vector register file to the pipeline. We use unified power format (UPF), a standard format to specify power-related settings in EDA tools [1, 13], to define voltage islands for the frontend, SA, VU, and SRAM. We synthesized and placed & routed the pipeline in Synopsys with ASAP7 PDK [15]. For async FIFOs, we use default queue size +10 for each component based on our analysis in §3.1. Table 4 lists the area/power breakdown. eNPU incurs 6.16% area overhead in total: eNPU's microarchitecture and voltage level shifters incur 4.58%/5.45% area/power overhead for the Coral NPU core; the DVFS-enabled IVR incurs 1.58% area overhead compared to a fixed-voltage IVR[4].

Table 4: Area and power overhead breakdown of eNPU for the Coral NPU core prototype [2]. All percentages are normalized to the total area of the original Coral NPU core.

	Area (μm^2)	Area (%)	Power (mW) @ 0.7V, 500MHz
Frontend	11,853.0	4.07%	3.287
Systolic Array (16 × 16)	214,850.4	73.74%	127.074
Vector Unit (128-bit)	46,251.2	15.87%	14.509
SRAM (32KB)	11,007.5	3.78%	1.346
FE ↔ SA Sync FIFO (16 Entries)	2,790.7	0.96%	2.002
FE ↔ VU Sync FIFO (16 Entries)	3,337.4	1.15%	2.394
FE ↔ SRAM Sync FIFO (4 Entries)	1,287.2	0.44%	3.683
Original NPU Core Total	**291,377.4**	**100.0%**	**154.295**
Fixed-Voltage IVR [68]	**1,800**	**0.62%**	-
FE ↔ SA Async FIFO (26 Entries)	4,570.6	1.57%	2.007
FE ↔ VU Async FIFO (26 Entries)	5,465.6	1.88%	2.400
FE ↔ SRAM Async FIFO (14 Entries)	4,540.7	1.56%	3.692
Shadow Vector Regfile (4 Regs)	261.1	0.09%	0.747
Voltage Level Shifters (FE-SA)	1858.5	0.64%	2.40
Voltage Level Shifters (FE-VU)	2222.4	0.76%	2.86
Voltage Level Shifters (FE-SRAM)	1846.3	0.63%	2.38
eNPU Core Total	**304,727.3**	**104.58%**	**162.702**
DVFS-Enabled IVR [10, 34]	**6418.5**	**2.20%**	-

The overhead will be relatively smaller for larger NPU chips whose SAs/VUs and their FIFO sizes are larger. The IVR area scales with the chip's peak power, which is typically manageable for modern datacenter AI chips [10]. As projected in Table 5, for a TPUv4/TPUv5p chip [32, 59], the total area overhead is 3.45%/3.61%, with 0.46%/0.4% due to eNPU's structures, 0.4%/0.37% due to voltage level shifters, and 1.34%/1.54% due to the DVFS-enabled IVR.

Simulator methodology. We build our simulator framework models eNPU at instruction-, request-, and LLM service-level.

First, we build an instruction-level performance model to estimate the impact of eNPU's NPU pipeline changes. It takes the VLIW assemblies for each operator (dumped from the XLA compiler in a Cloud TPUv4 VM [58]), reconstructs the instruction dependencies, and estimates the execution time under different V/f settings by accounting for frequency-dependent instruction latencies, async FIFO delays, and VU internal forwarding (see §3.1). As the TPU compiler is proprietary, we build our own instruction scheduling passes (including loop unrolling and software pipelining) based on the reconstructed instruction dependency graph and implement

[3]The expert capacity factor measures the ratio between the token count for the most loaded expert and the average token count for all experts.

[4]Even without DVFS, a fixed-voltage IVR is necessary for mitigating transient voltage droops and improving off-chip power delivery efficiency.

Table 5: Projected area overhead of eNPU on real TPU chips. The SA/VU sync FIFO is per SA/VU. The SRAM sync FIFO is per chip and scales with the number of read/write (R/W) ports. Each FIFO entry has the same size as a vector register. *TPUv5p die area is not publicly disclosed, so we use NVIDIA A100 GPU's die area [42] as a proxy, as this is one of the biggest 7nm dies given the reticle size limit.

Parameter	TPUv4		TPUv5p	
# of SAs/VUs/SRAM Ports	8/4/4		8/6/6	
Vector Register Size	4 KB		4 KB	
SA/VU/SRAM Sync FIFO Depth	16/16/16		16/16/24	
SA/VU/SRAM Async FIFO Depth	26/26/26		26/26/34	
	(mm^2)	%	(mm^2)	%
Original Total Die Area	**600**	**100%**	***826**	**100%**
FE ↔ SA Sync FIFO	2.405640	0.40%	2.405640	0.29%
FE ↔ VU Sync FIFO	1.202820	0.20%	1.804230	0.22%
FE ↔ SRAM Sync FIFO	0.601410	0.10%	1.353172	0.16%
Fixed-Voltage IVR [68]	3.27	0.54%	5.16	0.62%
FE ↔ SA Async FIFO	3.939825	0.66%	3.939825	0.48%
FE ↔ VU Async FIFO	1.969912	0.33%	2.954869	0.36%
FE ↔ SRAM Async FIFO	0.984956	0.16%	1.932029	0.23%
Shadow Vector Regfile (8 Regs/Chip)	0.075176	0.01%	0.075176	0.01%
Voltage Level Shifters (FE-SA)	1.379705	0.23%	1.379705	0.17%
Voltage Level Shifters (FE-VU)	0.689853	0.11%	1.034779	0.13%
Voltage Level Shifters (FE-SRAM)	0.344926	0.06%	0.676586	0.08%
DVFS-Enabled IVR [10, 34]	11.3	1.88%	17.82	2.16%
Total Area Overhead	**20.68**	**3.45%**	**29.81**	**3.61%**

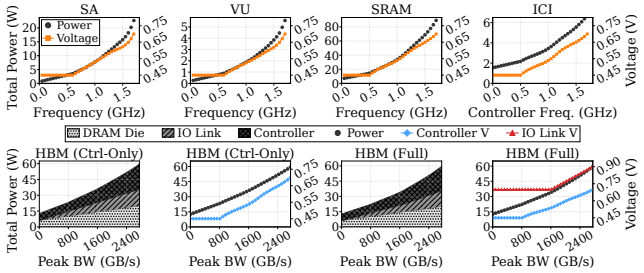

Figure 10: Power usage and minimum required voltage of components at various performance levels (peak frequency or bandwidth). The SA/VU plot shows the power of a single SA/VU. The SRAM plot uses 128MB SRAM with six read/write ports. For HBM, the total power includes on-chip controller, I/O, and HBM die. For ICI, the total power includes controller and I/O. For all plots, we show the power at peak component utilization (i.e., peak achievable FLOPs/sec or bandwidth).

eNPU's DVFS algorithm (§3.3). We validate this performance model on different operator types on a TPUv4 chip.

For request-level simulation, we use a production-level NPU chip simulator that simulates each component (SA, VU, SRAM, ICI, and HBM) at tile level. Taking the model graph as input, it computes each operator's per-component execution time using the tile shape, FLOPs per tile, and component-specific parameters (e.g, SA/VU width, HBM/ICI bandwidth), plus SRAM, HBM, and ICI traffic. The per-component execution time is validated against real TPU chips

Table 6: NPU chip and IVR specifications used in our simulations. The NPU chip data are based on TPUv5p [65]. The IVR data assume a hybrid 22 nm SCVR/DLDO design [10, 34].

Parameter	Value
TFLOPS (bf16)	459
# of SAs/VUs	8 SAs & 6 VUs, 32 architectural vector registers
SA Config.	128×128 (bf16 MAC)
VU Config.	8×128 SIMD ALU, 4 shadow vector registers
SRAM	128 MB, six 128×8×4B read/write ports
HBM	95 GB, 2765 GB/s
ICI	3D torus, 6 links/chip, 100 GB/s/link
DCN	50 GB/s/chip
IVR Transition Delay	4 ns / 54 mV
V_{in} → Min/Max V_{out}	0.75 V → 0.45 V / 0.7 V
Power Conversion Eff.	SCVR: 63%@0.45V; DLDO: 44%@0.45V–87%@0.7V
Clock Frequency	0 (clock-gated)–1.7 GHz, 50 MHz steps

across operator types and tensor shapes. The simulator scales execution time and power of each component based on the selected V/f, models V/f transitioning delay and IVR power conversion loss, and invokes the instruction-level model to quantify the impact of domain-crossing delay and shadow vector regfile size.

We combine the performance simulator with a power model to derive energy consumption. As TPUs lack a publicly available power-measurement API, we build our own power model based on RTL prototypes of SA, VU, and SRAM, plus public datasheets for HBM [6, 19, 31, 48] and Google's datacenter optical network [53]. We use Synopsys to synthesize our components with ASAP7 [15], using the nominal 0.7V supply voltage and a 1.7GHz frequency target (to roughly match TPUv5p). Then, we use Synopsys PrimeTime to estimate static and dynamic power at different V/f corners. Figure 10 shows the power of each component. For HBM, we include two power models: "HBM (Ctrl-Only)" fixes the voltage of I/O link and DRAM dies and only scales the V/f of on-chip memory controller. This resembles current HBM technology that requires fixed I/O and DRAM core voltages for stability [30]. "HBM (Full)" adjusts V/f for both controller and I/O link, reflecting the new HBM4 standard [31] that specifies multiple I/O voltage levels. For ICI, we only apply DVFS to the controller, as the physical link is optical. We validated our power model against published data [32, 56, 65]: the estimated TDP of a TPUv3/TPUv4 chip is within 4%/8%.

To evaluate eNPU at LLM service level, we use the request-level simulator as backend to build a cluster simulator, which replays LLM inference service traces and evaluates latency, end-to-end throughput, and SLO satisfaction rate. It implements the same request routing and continuous batching policies as a production LLM serving engine [36]. It also models the KV cache transfer between prefill/decode instances [44, 71]. We validated the system throughput and TTFT/TPOT of the cluster simulator against vLLM deployment [36] on a Cloud TPUv4-64 instance.

4 Evaluation

We show that: (1) At 0% performance degradation threshold, eNPU saves 16.9%/18.6% energy for LLM prefill/decode. As the threshold increases to 10%, its energy savings rise to 22.9%/18.6%. eNPU achieves 8.4%–9.0% (prefill) and 5.7% (decode) higher energy savings than the state-of-the-art NPU DVFS solution (§4.2) on average. (2)

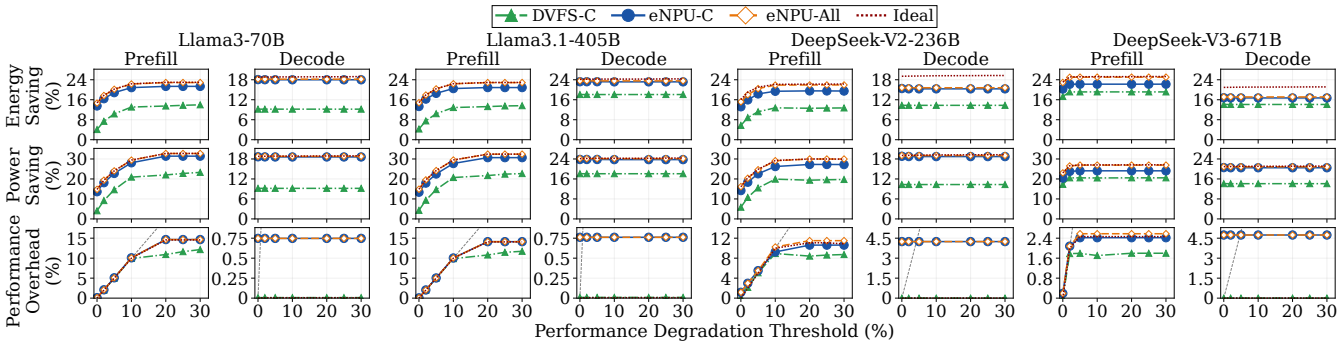

Figure 11: Energy saving (top), power saving (middle), and performance overhead (bottom) of an LLM request, given different SLO slacks (i.e., performance degradation thresholds).

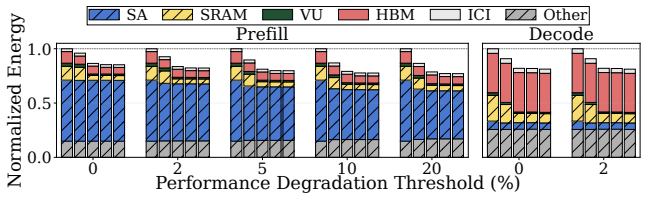

Figure 12: Component-level breakdown of energy savings in an LLM request. The bars from left to right are: *NoDVFS*, *DVFS-C*, eNPU-C, eNPU-All, and *Ideal*. Due to space limitations, we show the breakdown of Llama3-70B as an example. We only show 0% and 2% performance degradation thresholds for decode, as the energy savings already saturate at 2%.

eNPU's compiler-driven DVFS algorithm efficiently navigates the huge search space (§4.3). We also analyze eNPU's overhead (§4.4) and its sensitivity to the shadow vector register file size (§4.5). (3) eNPU's benefits generalize to different temporal DVFS granularities, V/f domain count, sequence lengths, and expert load imbalances in MoE models (§**??**). (4) Combining DVFS and power gating can further achieve up to 30%/31.5% energy savings for prefill/decode (§4.7). (5) For end-to-end LLM services, eNPU achieves 25.8%–35.2% energy savings with strict SLO guarantees (§4.8).

4.1 Experimental Setup

Workloads. We evaluate four LLMs, including dense (Llama) and MoE (DeepSeek) models. We study prefill and decode separately since they exhibit distinct resource demands, following state-of-the-art prefill/decode disaggregation (PDD) [44, 71]. We evaluate both single-request (§4.2–§4.7) and LLM service-level (§4.8) benefits. For single-request evaluation, we use 4096/512 input/output sequence lengths and study different performance degradation thresholds (i.e., SLO slacks). We study other sequence lengths in §4.6. For service-level evaluation, we replay production traces [55] (Table 2 and Table 3 for the SLO targets and cluster setup). As the trace only contains timestamps and input/output sequence lengths per request with no prompt or output content, we simulate the worst-case expert capacity factor [25] for MoE models by default. We study different capacity factors in §4.6.

Simulator setup. We use TPUv5p [59] specifications (see Table 6). We use a modern IVR design [10, 34] that achieves high power conversion efficiency and fast V/f scaling delay.

Baselines. We compare five designs: (1) ***NoDVFS***: the NPU chip always runs at peak V/f. It employs a fixed-voltage IVR with a 89%–93% power conversion efficiency [68]. It does not incur domain crossing overhead. (2) ***DVFS-C*** [60]: the state-of-the-art NPU DVFS solution, which scales the V/f of the compute domain (SA, VU, and SRAM). To ensure fair comparison, we use the ns-scale IVR transition delay in Table 6 and modify the design's genetic algorithm to search for the optimal compute domain frequency per operator, instead of per ms-scale epoch as originally proposed. We study temporal DVFS granularity in §4.6. (3) ***eNPU-C***: our design with DVFS for HBM controller but not I/O and DRAM dies. (4) ***eNPU-All***: our design with full DVFS for HBM controller and I/O link. (5) ***Ideal***: this design uses exhaustive search for operator-level and greedy search for request-level exploration (as sweeping all request-level plans is impossible); it assumes IVR power conversion loss, but no V/f scaling delay or domain crossing overhead.

4.2 Energy and Power Savings

We first analyze energy/power savings for a single LLM request with input/output sequence length 4096/512, as shown in Figure 11. We study various performance degradation thresholds (i.e., SLO slacks). As power-saving mirrors energy, we focus on energy below.

Overall energy savings. With the entire compute core (SAs, VUs, and SRAM) as a single V/f domain, *DVFS-C* achieves 12.8%–19.1% energy savings for prefill and 9.2%–18.1% for decode, compared to *NoDVFS*. *DVFS-C*'s energy saving at low SLO slacks is limited, however, as it must set a uniform frequency for all components, forcing them to follow the bottleneck component's frequency. In contrast, *eNPU-C* precisely throttles each component to its optimal frequency. It achieves 7.2%/5.4% (prefill/decode) higher energy savings than *DVFS-C* at 0 SLO slack and 6.4% (prefill) at large SLO slacks. *eNPU-C* converges to its maximum energy savings at a much lower performance degradation threshold as compared to *DVFS-C*. *eNPU-All* achieves slightly better energy savings than *eNPU-C* by allowing DVFS for HBM I/O voltage, and it closely tracks *Ideal*. The limited extra saving of *eNPU-All* is due to the fixed DRAM core

Table 7: Number of explored DVFS plans with 20% performance degradation threshold. "Op.": Operator-level plans. "Req.": Request-level plans.

		DVFS-C		*eNPU-All*		*Ideal*	
		Op.	Req.	Op.	Req.	Op.	Req.
Llama3-70B	Prefill	578	2.5M	15,345	2881	45M	2881
	Decode	714	2.5M	19,561	1	45M	1
Llama3.1-405B	Prefill	578	2.5M	15,376	4537	45M	4537
	Decode	714	2.5M	19,034	1	45M	1
DeepSeekv2-236B	Prefill	1326	2.5M	36,373	118,861	45M	118,861
	Decode	1326	2.5M	37,045	1	45M	1
DeepSeekv3-671B	Prefill	1394	2.5M	37,944	27,451	45M	27,451
	Decode	1394	2.5M	36,890	1	45M	1

voltage in current HBM generations; supporting DRAM-core DVFS could further improve memory energy efficiency.

Prefill vs. decode. For prefill, most designs reach maximum energy savings at ~10% SLO slack since prefill is compute-bound and offers headroom to trade performance for energy by throttling SA, VU, and SRAM. As their power scales super-linearly with frequency ($P \propto V^2 f$), additional slack enables further energy savings. In contrast, decode reaches maximum energy savings at 0–2% SLO slack since decode is memory-bound and HBM dominates dynamic energy (see Figure 12). The HBM subsystem has a near-linear power-frequency relationship (see Figure 10), so lowering frequency reduces power only near-linearly and extends execution time linearly, yielding minimal net energy reduction while increasing static energy for other components. Thus, most decode energy savings come from throttling non-bottlenecking compute components (SA, VU, and SRAM) to reduce static power.

4.3 Analysis of eNPU's DVFS Plan Search

Operator-level search. Table 7 shows how eNPU tackles the combinatorial search space of component-level DVFS. Because *DVFS-C* restricts its search space to a coarse-grained V/f domain (34 V/f choices per operator), it only evaluates 578–1,394 total operator-level plans across unique operators (LLMs repeat many operators across stacked transformer layers). eNPU's component-level DVFS unlocks $34^5 \approx 45M$ possibilities per operator; its two-level gradient descent explores 15K–38K plans to discover near-Pareto ones.

Request-level search. Table 7 also compares request-level search overhead. We empirically configure *DVFS-C*'s genetic algorithm to use 500 generations and population size 1000 per performance degradation threshold, seeding larger thresholds with results from smaller thresholds to speed up convergence (e.g., results of 2% are used as the seed for 5%). This forces it to explore 2.5M request-level plans total, regardless of the model's operator count. In contrast, eNPU converges to a near-optimal plan within only 2.9K–119K evaluations. Notably, decode has a limited request-level search space. As degrading decode performance often yields no extra energy saving (see §4.2), there is usually only one Pareto plan per operator.

Compilation time. Figure 14 evaluates the DVFS search time of compiling one LLM request with 24 Intel Xeon Gold 6136 CPU cores and 128 GB host memory. We break down the time for operator- and request-level search phases. For each phase, the search time

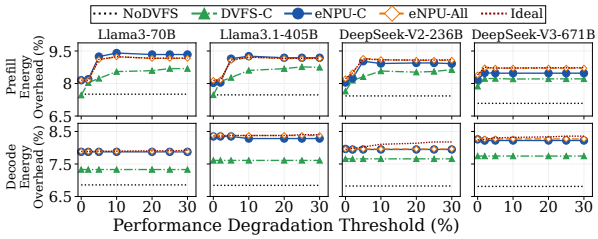

Figure 13: Energy overhead due to IVR power conversion.

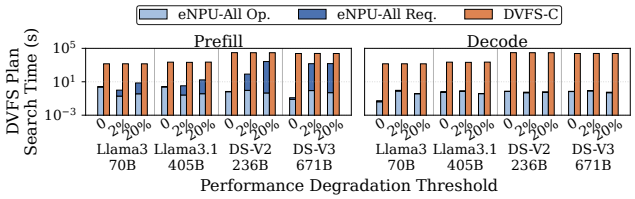

Figure 14: Compilation time overhead of eNPU.

scales with the number of explored plans in Table 7. eNPU's compilation time overhead for individual operators/kernels is small (less than 10 seconds for all operators in a model); it is unlikely to bottleneck kernel development or optimization iterations. Users can also temporarily disable eNPU during development. For each request, eNPU takes at most 40 minutes, while *DVFS-C* can incur more than 8 hours of search time due to inefficient genetic algorithm. With up to 2304 pre-compiled buckets per model (see §3.4), the compilation takes 15360 core-hours (≈5 hours with 3000 CPU cores, which is manageable for production LLM services deployed in datacenters).

4.4 Overhead Breakdown

Performance overhead. As shown in Figure 11, at zero SLO slack, eNPU's performance overhead stays below 0.78%–4.6%. This stems primarily from V/f transition delays and domain-crossing overhead. The domain-crossing penalty would be much higher without eNPU's VU internal forwarding mechanism (§3.1) — we perform a sensitivity study in §4.5. At larger SLO slacks, eNPU's performance degradation always stays below the allowed threshold, preserving strict SLO guarantees in end-to-end serving (see §4.8).

IVR power conversion overhead. Figure 13 quantifies the energy lost to IVR inefficiency. IVR power conversion efficiency degrades at lower output voltage, so conversion overhead grows slightly (e.g., ~1.5% for prefill) as eNPU throttles V/f more at larger SLO slacks. However, eNPU's net energy savings outweigh this overhead.

4.5 Benefits of VU Shadow Vector Register

Figure 15 shows how the VU shadow vector regfile size (§3.1) impacts performance. We focus on 0% performance degradation threshold, as it incurs most overhead. For larger thresholds, component frequencies are lower, making the overhead less obvious.

A shadow regfile size of 4 achieves <4% performance penalty. Decode has a higher sensitivity to the shadow register file size, as many operators are VU-intensive. A larger shadow regfile maps

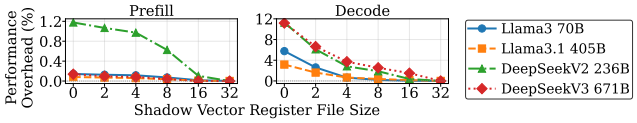

Figure 15: Impact of varying VU shadow vector register file size at performance degradation threshold 0%.

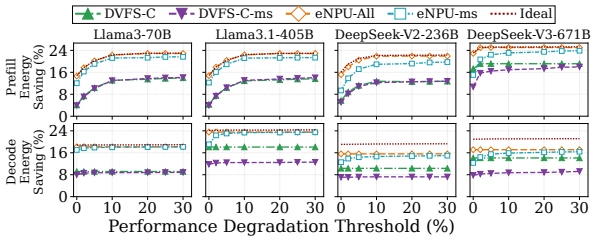

Figure 16: Energy saving of different temporal and spatial DVFS granularities.

more simultaneously live VU dependencies to shadow registers, reducing frontend stalls when the dependency distance is not long enough to hide frontend-VU round-trip latency, despite its larger area and power cost. The performance penalty is near-0 if the shadow regfile is as large as the frontend regfile (32 registers). We empirically pick a default shadow regfile size of 4, a reasonable trade-off between area overhead and performance.

4.6 Sensitivity Analysis

DVFS granularity. Figure 16 isolates the impact of spatially and temporally fine-grained DVFS. In *DVFS-C-ms* and *eNPU-ms*, we merge consecutive sub-millisecond operators into epochs (≥5 milliseconds) following the preprocessing step in [60], and then apply DVFS-C's genetic algorithm or eNPU's DVFS policy for each epoch. *DVFS-C* and *eNPU-All* use operator-level (sub-μs-scale) DVFS.

With more temporally fine-grained DVFS, *eNPU-All* achieves 0.6%–6.3% higher energy saving than *eNPU-ms*. *eNPU-ms* applies a unified V/f setting across an entire time epoch, forcing less critical operators to run at high V/f. In contrast, *eNPU-All* allows a component to change its V/f for different operators. Even with coarse temporal granularity, *eNPU-ms* saves 4.4%–9.6% more energy than *DVFS-C-ms*. *DVFS-C-ms* sometimes saves slightly more energy than *DVFS-C*. This is because after merging operators into fewer epochs, the genetic algorithm is sometimes more effective, as it intrinsically does not scale well with the search space size [11, 69].

V/f domain count. Figure 17 studies eNPU's sensitivity to the number of V/f domains. The 5-domain configuration is *eNPU-All*. The 4-domain version merges the frontend (FE), SA, and VU into one "compute" domain, while keeping the SRAM, HBM, and ICI separate. The 3-domain version forms an "NPU core" domain for FE, SA, VU, and SRAM, plus separate HBM and ICI "uncore" domains.

4-domain saves 0.8%–7.7% more energy than 3-domain, and 5-domain saves 2.3%–3.5% more than 4-domain. The major benefit comes from separating SAs and SRAM into different V/f domains, because in our simulated NPU chip (TPUv5p), SA power dominates

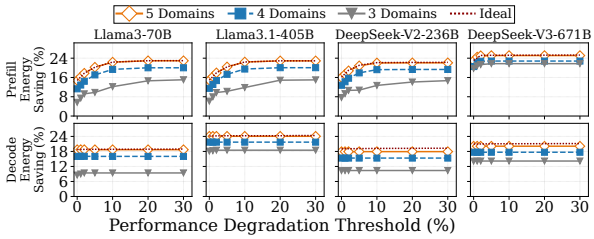

Figure 17: Energy saving of eNPU with different V/f domain configurations.

for prefill, and SRAM dominates for decode (see Figure 12). Hence, the energy saving from separately throttling VUs is relatively modest. Nevertheless, the extra hardware cost of having 5 domains over 4 domains is small (only 0.64% for TPUv5p, including the VU async FIFO, voltage level shifters, and shadow vector regfile in Table 5).

As future NPU chips incorporate more powerful VUs [22] for increasingly VU-intensive LLM operators (e.g., Gated DeltaNet [47, 66] and compressed/sparse-attentions such as HCA and CSA [18]), the VUs will account for a larger portion of NPU core power relative to SAs and SRAM. Similarly, for NPUs with smaller SAs (e.g., the Coral NPU core in Table 4), VU power is relatively larger. In these cases, a separate VU domain would bring more energy savings.

Variable sequence length. Figure 18 shows the energy savings with various input sequence lengths. We keep the output sequence length at 512 tokens, as the decode performance depends on the total context length, which is covered by sweeping the input lengths.

The energy saving depends on the request's bottlenecking component, which depends on the model architecture and the sequence length. For Llama3-70B prefill, the energy saving increases with sequence length. This is because the request becomes more compute-intensive and hence more SA-bound. Hence, other components become more idle, exposing more throttling opportunities. For DeepSeekV3-671B prefill, when we increase sequence length from 256 to 4K, the request is ICI-bound; it cannot saturate the 128×128 SA, but the ICI overhead grows near-linearly. Hence, the SA temporal utilization (similarly for VU and SRAM) drops relatively, exposing more throttling opportunities. As we increase sequence length beyond 4K, the SA, VU, and SRAM busy time increase quadratically due to the attention layer. Hence, the energy savings from SAs, VUs, and SRAM shrink, leading to relatively smaller total energy savings. The energy saving of decode is similar across sequence lengths, as it is always memory-bound.

In Figure 19, we investigate how sequence lengths affect the DVFS plans generated by eNPU. In general, the selected frequency of a component is proportional to its temporal utilization measured at peak frequency (i.e., *NoDVFS*). This is expected, as a highly utilized component has less slack in the operator schedule and hence must run at a higher frequency to meet the performance target. In contrast, a less-utilized component is often off the critical path, so it can be throttled with little impact on end-to-end latency.

MoE expert load imbalance. By default, eNPU selects DVFS plans based on the worst-case expert load imbalance (§3.4), which "overprovisions" energy budget if the real token distribution is more

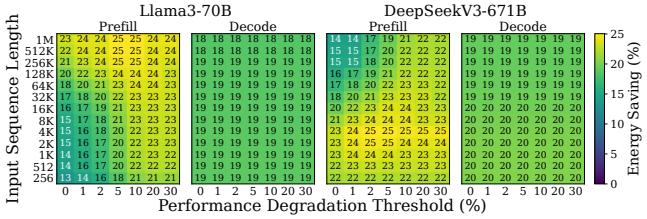

Figure 18: Energy savings of eNPU over *NoDVFS* with various sequence lengths. We show two models due to space limits.

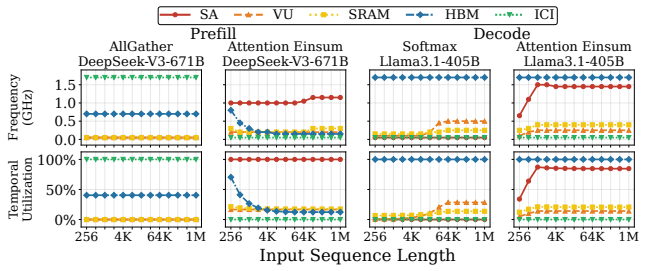

Figure 19: Frequencies selected by *eNPU-All* (top row) and the original component utilization without DVFS (bottom row) for representative operators, w.r.t. various sequence lengths. We plot the results for performance degradation threshold 30%; other thresholds have a similar trend.

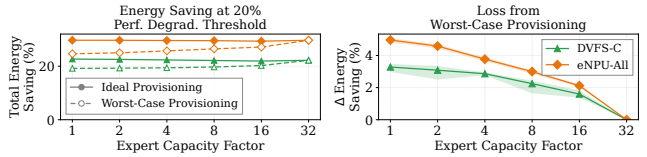

Figure 20: Left: Energy savings over *NoDVFS* at different expert load imbalances. The shaded area covers 0%–20% performance degradation threshold; the curve is the mean. Right: Energy saving difference between ideal and worst-case provisioning. We only show DeepSeekV3-671B prefill due to space limits. The energy saving in decode is not affected by load imbalances since the number of tokens is very small.

balanced. In Figure 20, we study different token distributions by varying the simulated expert capacity factor, and compare eNPU with an ideal provisioning that perfectly predicts the load imbalance and incurs no SLO violations. When the load is perfectly balanced (expert capacity factor = 1), eNPU will save 4.3% less energy, but it still achieves 22% energy saving over *NoDVFS*. Regardless of the load imbalance, eNPU always saves more energy than *DVFS-C*.

4.7 Benefits of DVFS + Power Gating

Figure 21 quantifies the benefits of combining DVFS and power gating (PG). We compare None (no DVFS or PG), PG-Only, DVFS-Only, and PG+DVFS. We apply the state-of-the-art NPU PG solution ReGate [65], which uses the compiler to analyze component idleness and orchestrate PG at instruction level. We first apply eNPU's DVFS

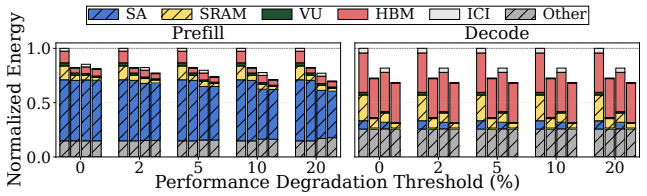

Figure 21: Energy savings with DVFS and power gating (PG). The bars from left to right are None (no DVFS or PG), PG-Only, DVFS-Only, and PG+DVFS.

algorithm to each operator, and then apply ReGate to instrument power gating commands based on the VLIW code sequence after DVFS. When the performance degradation threshold is 0, PG-Only slightly outperforms DVFS-only. This is because PG-Only can exploit component underutilization by throttling the supply voltage to near-zero, yielding more static power reductions than DVFS-Only (which only throttles voltage to 0.45V). However, the unique advantage of DVFS lies in its ability to actively trade excess performance for energy savings. At large SLO slacks for prefill, DVFS-Only outperforms PG-Only. By combining DVFS and PG, we can achieve maximum energy saving (up to 30%/31.5% for prefill/decode).

4.8 Energy Savings for LLM Service

In Figure 22, we evaluate the end-to-end energy savings of eNPU for a 24-hour production LLM serving trace. The trace includes both peak hours (14–20 hours) and off-peak hours. We employ *eNPU-All* and *DVFS-C* using the same DVFS-aware request scheduling mechanism and safeguarding rules (§3.4). We study Llama3-70B as an example. The insights apply to other models as well.

eNPU-All only needs ≈10% SLO slack to achieve maximum energy savings. During off-peak hours, both *DVFS-C* and *eNPU-All* achieve significant energy savings (25.8%/33% and 33%/42% for prefill/decode) with no obvious drop in SLO satisfaction rate. For off-peak hours, cluster-level scheduling techniques can be employed together with eNPU to further reduce energy consumption, such as auto-scaling to shrink the number of allocated NPU chips or putting idle chips into sleep. eNPU can still help save energy for the remaining active NPU chips.

During peak hours, the available SLO slack shrinks, so DVFS is applied less aggressively. In *eNPU-All*, 12%/10% of prefill/decode batches during peak hours use a less aggressive DVFS plan (safeguard rule 1 in §3.4), and only 2.3%/1.1% of prefill/decode batches are locked to the peak clock speed (rule 2). Nevertheless, *eNPU-All* achieves higher energy savings than *DVFS-C* under limited SLO slacks (29.7%/35.2% vs. 23.3%/28.1% for prefill/decode).

5 Discussion

Adapting eNPU techniques for overclocking. eNPU's DVFS policy (§3.3) can be adapted to overclock an NPU chip given a power budget. The operator-level exploration can cover both underclocking and overclocking options for each component. Then, the request-level search can work in the same way, starting from the fastest plan and iteratively throttling the frequency of each component until the DVFS plan satisfies the power budget.

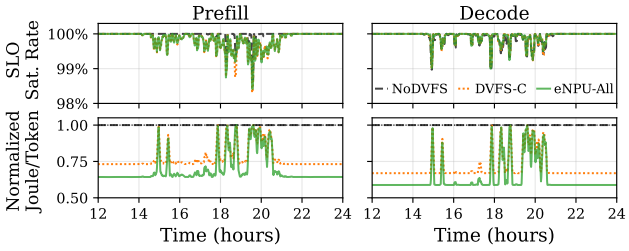

Figure 22: End-to-end energy savings for LLM serving. We show 12–24 hours, as the trend is flat during off-peak hours.

Implications of advanced technology nodes. In our prototype of eNPU, we use ASAP7 PDK as a reproducible case study, which roughly matches the technology nodes of TPUv4/TPUv5p. As long as the $P \propto V^2 f$ law continues to hold, DVFS will remain an effective technique for future nodes. Moreover, as different components scale at different paces (e.g., logic scales faster than SRAM) and have different performance requirements, their V/f and power characteristics will remain diverse. eNPU offers a solution to manage such component diversity and maximize DVFS opportunities.

Prefill/decode disaggregation vs. mixed continuous batching. In our evaluation, we employ prefill/decode (P/D) disaggregation following state-of-the-art LLM serving systems [44, 71], as these two phases have distinct resource demands. eNPU can also be integrated with aggregated P/D serving (i.e., mixed continuous batching [4]) in the same way as described in §3.4, as it picks the DVFS plan for each batch, no matter whether it is a batch of prefill requests or decode requests. The per-batch energy saving trend would be similar to P/D disaggregation, as prefill batches are typically still compute-bound and decode batches are memory-bound.

6 Related Work

DVFS techniques. Prior work explored different DVFS granularities and policies. They often treat an entire chip as one V/f domain, or partition V/f domains only across replicated, homogeneous units such as CPU cores [35] or GPU SM, and core/uncore domains [23, 24, 37]. In contrast, eNPU enables independent DVFS across heterogeneous components within an NPU core. Prior work explores dynamic schemes based on runtime feedback [14, 28, 51, 61–63] and static schemes based on offline profiling or compiler analysis [26, 50, 64]. They assume one coarse-grained V/f domain or a few homogeneous domains, which cannot directly apply when we must jointly optimize V/f choices for multiple components and account for their interaction with VLIW instruction scheduling. eNPU addresses this challenge with a compiler-driven greedy search.

Power management techniques for AI chips. Recent works have explored both static and dynamic power management for AI chips. PCSTALL [10] uses a program counter-based prediction mechanism to configure DVFS on GPUs. The Ascend NPU [60] employs a genetic algorithm to search for the optimal AI core frequency. eNPU enables component-level V/f domains on NPUs and addresses the unique HW and SW challenges of exploiting the new DVFS optimization space. UPTPU [43] power gates MAC units in spatially underutilized systolic arrays. ReGate [65] enables compiler-directed power gating across all major NPU components. eNPU is complementary to these power gating techniques.

Power/energy-aware LLM serving. Power/energy efficiency of LLM serving has become a critical concern. Proposed optimizations include request routing techniques that account for renewable energy [49] or power hotspots across server racks [54]. Many works scale down GPU frequency to exploit SLO slack and save energy, using offline profiling [33, 55], online feedback-based control [46, 67], and ILP solvers [49] to find the optimal frequency. They are designed for whole-chip DVFS on GPUs and cannot directly apply to spatially fine-grained DVFS on NPUs. Prior studies also examine model placement strategies, such as varying the tensor parallelism degree [52] to enable latency-energy trade-offs [9, 33, 38, 46, 55] or grouping operators by their sensitivity to chip frequency [46]. Compared to these system-level optimizations, eNPU addresses the unique HW and SW challenges and provides a holistic solution for exploiting component-level DVFS on NPU chips for LLM serving.

7 Conclusion

We identify the DVFS opportunities for LLM serving on NPUs and propose eNPU, a hardware-software co-design that enables component-level DVFS on NPUs. eNPU improves the energy efficiency of LLM serving while preserving strict SLO guarantees.

References

[1] 2025. IEEE Standard for Design and Verification of Low-Power Energy-Aware Electronic Systems. *IEEE Std 1801-2024 (Revision of IEEE Std 1801-2018)* (2025), 1–614. https://doi.org/10.1109/IEEESTD.2025.10910082

[2] 2026. Coral NPU. https://github.com/google-coral/coralnpu.

[3] Hazem A. Abdelhafez, Christopher Zimmer, Sudharshan S. Vazhkudai, and Matei Ripeanu. 2019. AHEAD: A Tool for Projecting Next-Generation Hardware Enhancements on GPU-Accelerated Systems. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 583–592. https://doi.org/10.1109/IPDPSW.2019.00103

[4] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[5] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Barbara, California, USA) *(SPAA '95)*. Association for Computing Machinery, New York, NY, USA, 95–105. https://doi.org/10.1145/215399.215427

[6] AMD. [n. d.]. AXI High Bandwidth Memory Controller LogiCORE IP Product Guide (PG276). https://docs.amd.com/r/en-US/pg276-axi-hbm

[7] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3620665.3640366

[8] AWS. 2025. Amazon EC2 Trn1/Trn1n Architecture. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/about-neuron/arch/neuron-hardware/trn1-arch.html

[9] Omar Basit, Yunzhao Liu, Z Jonny Kong, and Y Charlie Hu. 2026. BiScale: Energy-Efficient Disaggregated LLM Serving via Phase-Aware Placement and DVFS. *arXiv preprint arXiv:2602.18755* (2026).

[10] Srikant Bharadwaj, Shomit Das, Kaushik Mazumdar, Bradford M. Beckmann, and Stephen Kosonocky. 2024. Predict; Don't React for Enabling Efficient Fine-Grain DVFS in GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 253–267. https://doi.org/10.1145/3623278.3624756

[11] Stephen Chen and Stephen F. Smith. 1999. Improving Genetic Algorithms by Search Space Reductions (with Applications to Flow Shop Scheduling). In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 135–140.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. Carlsbad, CA.

[13] ChipVerify. 2026. Introduction to UPF. https://chipverify.com/unified-power-format/introduction-to-upf. Accessed: June 2026.

[14] Kihwan Choi, R. Soma, and M. Pedram. 2005. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 1 (2005), 18–28. https://doi.org/10.1109/TCAD.2004.839485

[15] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115. https://doi.org/10.1016/j.mejo.2016.04.006

[16] Tejas Dave. 2014. Clock Domain Crossing Techniques & Synchronizers. EDN Network. https://www.edn.com/synchronizer-techniques-for-multi-clock-domain-socs-fpgas/ Accessed: April 3, 2026.

[17] DeepSeek-AI. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[18] DeepSeek-AI. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. Technical report and model card. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro Accessed: Jun 2026.

[19] Atul Dhamba and Anand V Kulkarni. [n. d.]. Design Considerations for High Bandwidth Memory Controller. https://www.design-reuse.com/articles/41186/design-considerations-for-high-bandwidth-memory-controller.html

[20] Stijn Eyerman and Lieven Eeckhout. 2011. Fine-grained DVFS using on-chip regulators. *ACM Trans. Archit. Code Optim.* 8, 1, Article 1 (Feb. 2011), 24 pages. https://doi.org/10.1145/1952998.1952999

[21] Google. 2023. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla

[22] Google Cloud. 2026. TPU 8t and TPU 8i Technical Deep Dive. Google Cloud Blog. https://cloud.google.com/blog/products/compute/tpu-8t-and-tpu-8i-technical-deep-dive Accessed: Jun 2026.

[23] Joao Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomas. 2018. GPGPU Power Modeling for Multi-domain Voltage-Frequency Scaling. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 789–800. https://doi.org/10.1109/HPCA.2018.00072

[24] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. 2015. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW '15)*. IEEE Computer Society, USA, 896–904. https://doi.org/10.1109/IPDPSW.2015.70

[25] Shwai He, Weilin Cai, Jiayi Huang, and Ang Li. 2026. Capacity-Aware Inference: Mitigating the Straggler Effect in Mixture of Experts. arXiv:2503.05066 [cs.LG] https://arxiv.org/abs/2503.05066

[26] Chung-Hsing Hsu and Ulrich Kremer. 2003. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. *SIGPLAN Not.* 38, 5 (May 2003), 38–48. https://doi.org/10.1145/780822.781137

[27] International Energy Agency. 2025. *Energy and AI*. Technical Report. International Energy Agency, Paris. https://www.iea.org/reports/energy-and-ai

[28] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. 2006. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 359–370. https://doi.org/10.1109/MICRO.2006.30

[29] Behdad Jamadi, Meysam Sohani Darban, and Jeffrey S. Walling. 2026. Analysis of Edge Mismatch and Output Power Degradation in Cascoded Class-D Power Amplifiers Using Dual-Range Voltage Level Shifters. arXiv:2602.09820 [eess.SP] https://arxiv.org/abs/2602.09820

[30] JEDEC. 2026. High Bandwidth Memory (HBM) DRAM. https://www.jedec.org/standards-documents/docs/jesd235a

[31] JEDEC Solid State Technology Association. 2025. JEDEC and Industry Leaders Collaborate to Release JESD270-4 HBM4 Standard: Advancing Bandwidth, Efficiency, and Capacity for AI and HPC. https://www.jedec.org/news/pressreleases/jedec%C2%AE-and-industry-leaders-collaborate-release-jesd270-4-hbm4-standard-advancing.

[32] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA'23)*. Orlando, FL, USA. https://doi.org/10.1145/3579371.3589350

[33] Andreas Kosmas Kakolyris, Dimosthenis Masouros, Petros Vavaroutsos, Sotirios Xydis, and Dimitrios Soudris. 2025. throttll'em: Predictive gpu throttling for energy efficient llm inference serving. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1363–1378.

[34] Stephen T. Kim, Yi-Chun Shih, Kaushik Mazumdar, Rinkle Jain, Joseph F. Ryan, Carlos Tokunaga, Charles Augustine, Jaydeep P. Kulkarni, Krishnan Ravichandran, James W. Tschanz, Muhammad M. Khellah, and Vivek De. 2016. Enabling Wide Autonomous DVFS in a 22 nm Graphics Execution Core Using a Digitally Controlled Fully Integrated Voltage Regulator. *IEEE Journal of Solid-State Circuits* 51, 1 (2016), 18–30. https://doi.org/10.1109/JSSC.2015.2457920

[35] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 123–134. https://doi.org/10.1109/hpca.2008.4658633

[36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[37] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 487–498. https://doi.org/10.1145/2485922.2485964

[38] Yueying Li, Zhanqiu Hu, Esha Choukse, Rodrigo Fonseca, G Edward Suh, and Udit Gupta. 2025. Ecoserve: Designing carbon-aware ai inference systems. *arXiv preprint arXiv:2502.05043* (2025).

[39] Llama Team. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[40] Ivan Miro Panades and Alain Greiner. 2007. Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures. In *First International Symposium on Networks-on-Chip (NOCS'07)*. 83–94. https://doi.org/10.1109/NOCS.2007.14

[41] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63. https://doi.org/10.1109/MM.2021.3058217

[42] NVIDIA Corporation. [n. d.]. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/. Accessed: June 2026.

[43] Pramesh Pandey, Noel Daniel Gundi, Koushik Chakraborty, and Sanghamitra Roy. 2021. UPTPU: Improving Energy Efficiency of a Tensor Processing Unit through Underutilization Based Power-Gating. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 325–330. https://doi.org/10.1109/DAC18074.2021.9586224

[44] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[45] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. 2023. POLCA: Power Oversubscription in LLM Cloud Providers. arXiv:2308.12908 [cs.DC] https://arxiv.org/abs/2308.12908

[46] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2024. Power-aware deep learning model serving with {μ-Serve}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 75–93.

[47] Qwen Team. 2025. Qwen3-Next-80B-A3B-Thinking. Hugging Face model card. https://huggingface.co/Qwen/Qwen3-Next-80B-A3B-Thinking Accessed: Jun 2026.

[48] Rambus. [n. d.]. HBM3E / HBM3 Controller IP. https://www.rambus.com/interface-ip/hbm/hbm3-controller/

[49] Tella Rajashekhar Reddy, Rohan Gandhi, Anjaly Parayil, Chaojie Zhang, Mike Shepperd, Liangcheng Yu, Jayashree Mohan, Srinivasan Iyengar, Shivkumar Kalyanaraman, Debopam Bhattacherjee, et al. 2025. AI Greenferencing: Routing AI Inferencing to Green Modular Data Centers with Heron. *arXiv preprint arXiv:2505.09989* (2025).

[50] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. 2002. Energy-conscious compilation based on voltage scaling. In

Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (Berlin, Germany) *(LCTES/SCOPES '02)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/513829.513832

[51] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, and M.L. Scott. 2002. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*. 29–40. https://doi.org/10.1109/HPCA.2002.995696

[52] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[53] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Sigcomm '15*.

[54] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Esha Choukse, Haoran Qiu, Rodrigo Fonseca, Josep Torrellas, and Ricardo Bianchini. 2025. Tapas: Thermal-and power-aware scheduling for LLM inference in cloud platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1266–1281.

[55] Jovan Stojkovic, Chaojie Zhang, Inigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency . In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 1348–1362. https://doi.org/10.1109/HPCA61900.2025.00102

[56] Tianqi Tang, Sheng Li, Lifeng Nai, Norm Jouppi, and Yuan Xie. 2021. NeuroMeter: An Integrated Power, Area, and Timing Modeling Framework for Machine Learning Accelerators Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 841–853. https://doi.org/10.1109/HPCA51647.2021.00075

[57] The JAX Authors. [n. d.]. JAX: High performance array computing. https://docs.jax.dev/en/latest/

[58] Patrick C. Toulme. 2025. From JAX to VLIW: Tracing a Computation Through the TPU Compiler Stack. https://patricktoulme.substack.com/p/from-jax-to-vliw-tracing-a-computation.

[59] Amin Vahdat and Mark Lohmeyer. 2023. Enabling next-generation AI workloads: Announcing TPU v5p and AI Hypercomputer. https://cloud.google.com/blog/products/ai-machine-learning/introducing-cloud-tpu-v5p-and-ai-hypercomputer.

[60] Zibo Wang, Yijia Zhang, Fuchun Wei, Bingqiang Wang, Yanlin Liu, Zhiheng Hu, Jingyi Zhang, Xiaoxin Xu, Jian He, Xiaoliang Wang, Wanchun Dou, Guihai Chen, and Chen Tian. 2025. Using Analytical Performance/Power Model and Fine-Grained DVFS to Enhance AI Accelerator Energy Efficiency. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1118–1132. https://doi.org/10.1145/3669940.3707231

[61] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. 1994. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation* (Monterey, California) *(OSDI '94)*. USENIX Association, USA, 2–es.

[62] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. 2004. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA) *(ASPLOS XI)*. Association for Computing Machinery, New York, NY, USA, 248–259. https://doi.org/10.1145/1024393.1024423

[63] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. 2005. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (Barcelona, Spain) *(MICRO 38)*. IEEE Computer Society, USA, 271–282. https://doi.org/10.1109/MICRO.2005.7

[64] Fen Xie, Margaret Martonosi, and Sharad Malik. 2003. Compile-time dynamic voltage scaling settings: opportunities and limits. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 49–62. https://doi.org/10.1145/781131.781138

[65] Yuqi Xue and Jian Huang. 2025. ReGate: Enabling Power Gating in Neural Processing Units. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 1160–1177. https://doi.org/10.1145/3725843.3756038

[66] Songlin Yang, Jan Kautz, and Ali Hatamizadeh. 2025. Gated Delta Networks: Improving Mamba2 with Delta Rule. In *International Conference on Learning Representations*. https://doi.org/10.48550/arXiv.2412.06464 arXiv:2412.06464 [cs.CL]

[67] Jiahuan Yu, Aryan Taneja, Junfeng Lin, and Minjia Zhang. 2025. Voltanallm: Feedback-driven frequency control and state-space routing for energy-efficient llm serving. *arXiv preprint arXiv:2509.04827* (2025).

[68] Michael Zelikson, Kosta Luria, Lior Gil, Yuval Brown, Vadim Goldenbeg, Dor Kasif, Elias Hlees, and Alex Vinichuk. 2023. 14.3 A Digital Low-Dropout (LDO) Linear Regulator with Adaptive Transfer Function Featuring 125A/mm2 Power Density and Autonomous Bypass Mode. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. 230–232. https://doi.org/10.1109/ISSCC42615.2023.10067368

[69] Yong Zhao and Nobuo Sannomiya. 2001. An Improvement of Genetic Algorithms by Search Space Reductions in Solving Large-scale Flowshop Problems. *IEEJ Transactions on Electronics, Information and Systems* 121, 6 (2001), 1010–1015. https://doi.org/10.1541/ieejeiss1987.121.6_1010

[70] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2000, 27 pages.

[71] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2025. DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 11, 18 pages.

[72] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. 2024. Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 761–774. https://www.usenix.org/conference/nsdi24/presentation/zu