POLYUFC: Polyhedral Compilation Meets Roofline Analysis for Uncore Frequency Capping

Nilesh Rajendra Shah
Department of CSE
IIT Hyderabad, India
cs22resch12001@iith.ac.in

M V V S Manoj Kumar
Department of CSE
IIT Hyderabad, India
mvvsmanojkumar.official@gmail.com

Dhairya Baxi
Department of CSE
IIT Hyderabad, India
baxidhairya2312@gmail.com

Ramakrishna Upadrasta
Department of CSE
IIT Hyderabad, India
ramakrishna@cse.iith.ac.in

***Abstract*—We present POLYUFC, an MLIR based compilation flow for uncore frequency capping that combines (performance and power) roofline analyses and polyhedral compilation-based static analysis for characterization of affine programs. We introduce a parametric mathematical model that links operational intensity and uncore frequency to derive frequency caps, validated through empirical evaluation on real hardware. By embedding these caps into Pluto optimized code generated by Polygeist, we achieve improvements in Energy Delay Product (EDP) up to 42% on compute-bound, and up to 54% on bandwidth-bound programs—carefully selected from ML-models from vision/NLP domains and POLYBENCH—over Intel UFS driver. Our framework is retargetable across multiple micro-architectures; and can handle multiple optimization goals like performance, energy and EDP, and is applicable across inter/intra dialects.**

***Index Terms*—static analysis, cache model, performance analysis, power analysis, capping, roofline model, mlir**

I. INTRODUCTION AND MOTIVATION

Motivation A sudden power crunch has recently been observed for data centre operators [33, 106, 75, 100] having fixed power budget due to increased demand for running ML models for inference and scientific codes on the existing infrastructure, which primarily contains CPUs. Managing power consumption in these platforms, where maximizing performance and energy efficiency is paramount, remains a complex and persistent challenge [62, 84]. Accurate prediction and control of peak power usage are essential not only for preventing hardware damage but also for enabling compilers and system software to optimize energy profiles for domain-specific applications and hardware.

(Uncore) Frequency Capping: Frequency capping is a practical and widely applicable *"knob"* for power management that imposes explicit limits on frequency of a component to enforce energy budgets to eliminate frequency over-provisioning. Modern processors provide mechanisms for frequency capping [54, 40, 4] in two primary components: the computational cores (*core*), and the non-core[1] components (*uncore*). Although both frequency managements are important, the uncore subsystem has emerged as a significant contributor to overall processor power [45, 37, 88, 36, 4, 107]- it can account for average 30% of total package power [37, 18, 34, 4]. For uncore power management, frequency capping emerges as an effective technique to satisfy memory bandwidth requirements of programs.

[1]Includes last-level cache (LLC), memory controllers, and interconnects.

Compiler-Driven Uncore Frequency Capping Polyhedral compilers like Pluto [13, 14] are well established to optimize affine programs for performance. However, the analytical modeling based cost functions in these compilers do not consider alternate energy-efficiency based cost models that are retargetable to different platforms. This motivates our first challenge about necessity of an ***[C1] automatic (compiler-driven)*** *selection of* ***frequency capping*** *for (performance+) energy efficiency adapted to the target hardware platform.* Such a frequency capping based technique is naturally implemented in versatile modern compiler infrastructures like MLIR [53, 64], that allow analyses at multiple dialect representations, and application of frequency caps at multiple program-points and dialect-levels.

Characterization Necessity For any (static-analysis) compiler driven performance optimization, the key is an accurate *characterization* of programs into the so-called compute-bound or memory (bandwidth)-bound (CB/BB) categories [26, 27, 69, 70, 2]. For CB programs, the performance remains stable across different uncore frequencies. But energy consumption increases at higher frequencies due to unnecessary power allocated to uncore. In contrast, BB programs require more uncore power and benefit from higher uncore frequencies for better performance. In Fig. 1, we show execution time, energy, and EDP for key affine programs with varying uncore frequency caps on Pluto-tiled codes. We mark the maximum improvement possible on various metrics.

Roofline-Aware Mathematical Model–Driven Characterization: Architecture-aware performance [101] and energy [20] *"bound and bottleneck"* roofline models have been proposed in the literature, and these (per-micro-architecture) roofline models encode key information that can be exploited for performance improvements. There is a necessity for performance and power characterization that *utilize* this information for compiler optimizations. The next challenge naturally arises: ***[C2] mathematical modeling*** *based estimation of runtime performance, bandwidth and power consumption for roofline based* ***characterization*** *of programs.* Such an estimate should be parametrized by uncore frequencies.

Characterization-Aware Performance Optimizations CB programs (with high operational intensity), generally require

979-8-3315-9288-2

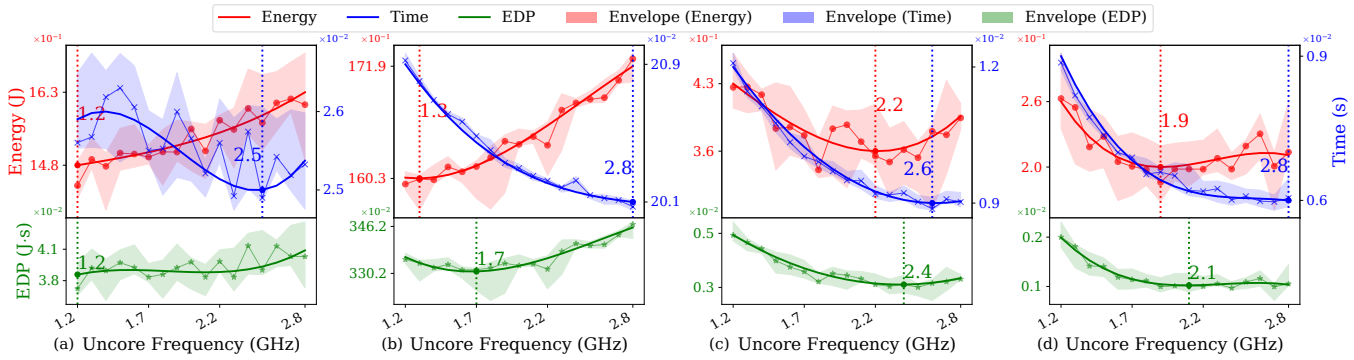


Fig. 1. Exec. time, Energy, and EDP across varying uncore frequency caps for representative kernels: (a) `conv2d`, (b) `2mm`, (c) `gemver`, and (d) `mvt`; all compiled with Pluto [13], and polynomial curve fitting applied to the medians. For CB workloads like `conv2d` and `2mm`, the minimum EDP is observed near lower uncore frequencies of 1.2 GHz and 1.8 GHz, respectively, which are much less than the peak 2.8 GHz. For BB workloads like `gemver` and `mvt`, optimal EDP (Energy) occurs at frequencies of 2.2 GHz (1.9 GHz) and 2.3 GHz (2.1 GHz), respectively. The max improvement possible is shown by dotted vertical bars.

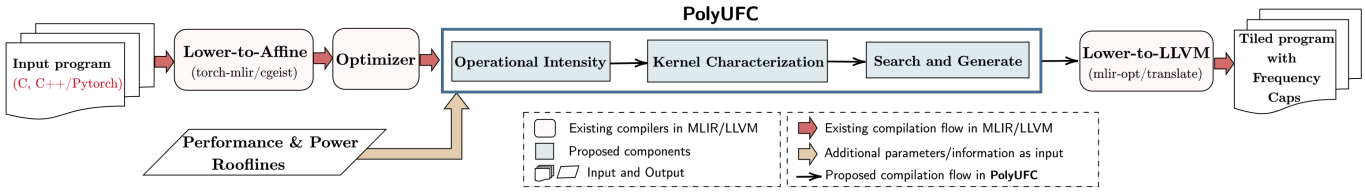


Fig. 2. **Simplified flow of** POLYUFC: Input programs (`C/C++/PyTorch`) are compiled to MLIR, lowered to MLIR's `affine` dialect, and optimized (for time/performance) using `polygeist` optimizer. POLYUFC calculates `OI`, followed by a program characterization, and searches for uncore frequency caps. The output of POLYUFC is optimized for both performance and energy efficiency. It is lowered to MLIR's `scf` dialect, translated to `LLVM-IR`, and compiled into binaries with frequency caps.

less memory bandwidth and can be transformed to operate efficiently at lower uncore frequencies, resulting in substantial power savings. In contrast, BB programs (with low operational intensity), demand higher bandwidth and thus can be transformed to benefit from higher uncore frequencies to avoid performance bottlenecks.

`OI` Necessity A compiler based characterization of programs relies on Operational Intensity (`OI`), for positioning the program vis-a-vis the rooflines, and measuring the effect of the transformations. Several works [7, 8, 38, 85, 76] calculate cache misses; however, they do not explicitly model `OI` as a means for performance and power roofline-characterizations. This brings us to our third key challenge for compiler researchers in ***[C3]*** *developing accurate estimations of Operational Intensity using* ***static analysis (and polyhedral compilation)***. Such an estimate is essential for enabling a compile-time characterization, and also removes the limitations[2] of performance counters [91, 65], or simulators [25, 43], both of which are prohibitively expensive.

We introduce POLYUFC, the *first compilation flow* in MLIR for uncore frequency capping, unifying roofline-aware performance and power characterization with polyhedral compilation based cache analysis. Our main contributions are:

- A static analysis based calculation of Operational Intensity for affine programs, driven by POLYUFC-CM, a polyhedral compilation based approximate cache miss analysis for set-associative caches. (Sec. IV)
- A mathematical model based on polyhedral compilation techniques combined with roofline analyses[3] to obtain performance and power characterization (CB or BB) for affine programs on modern CPUs. Applying the above for estimating performance, bandwidth and power with uncore frequency (f) + Operational Intensity ($\mathcal{I}$) as parameters. (Sec. V)
- ML-POLYUFC, a framework that enables multi-level MLIR-dialect-aware compilation-flow that encapsulates analysis and application of POLYUFC techniques; and a simple algorithm that enables to apply POLYUFC for different metrics, like performance-only, performance and power, EDP. (Sec. VI)
- An experimental evaluation of POLYUFC for uncore frequency capping on programs from vision and NLP domains, like `conv2d`, `matmul` from ALEXNET, LLAMA2 respectively, and POLYBENCH. In particular, we compare against the Intel Uncore Scaling driver, on which we obtain: (i) minimal performance loss of $\approx$ 7% on CB workloads, and up to 42% EDP improvement. (ii) performance improvements of upto $\approx$ 30%, and EDP

[2]Overheads in runtime, and dependence on SW/HW support from vendors.

[3]Performance rooflines [72] are usually made available by hardware vendors. In this work, we need *both* performance and power rooflines. So, we rely on our own *one-time micro benchmarking*.

improvements of up to $\approx 54\%$ on BB. (Sec. VII)

In Fig. 2, we show a simplified flow of our proposed POLYUFC.

II. BACKGROUND

In this section, we introduce the terms used in this work related to performance and power modeling, along with the mathematical terms used in formulation.

A. Affine Programs

We model affine control programs, including affine loop bounds and branches, i.e., all the constraints are conjunctions and disjunctions of the form $a_1 \times x_1 + \cdots + a_n \times x_n \geq c$, where the branches are independent of the data accesses and all the memory accesses involve indexes which are affine functions of the loop iterators and symbolic constants. This class of programs includes image processing, scientific programs, and neural network applications from vision or NLP (like ALEXNET, BERT, LLAMA2). We refer affine programs as affine kernels, benchmarks in different sections of the paper.

B. Polyhedral Compilation

Polyhedral compilation [30, 31, 77, 78, 13, 3, 64] is a powerful technique for analyzing and optimizing affine programs involving nested loops and arrays. It represents program elements as geometric objects called integer polyhedra, enabling advanced transformations. At the core of many polyhedral compilation frameworks is the Integer Set Library (`isl`) [96], which provides efficient manipulation of sets and relations of integer points bounded by linear constraints. `isl` supports operations such as intersection, union, and projection on these sets and relations, making it an essential tool for program analysis and transformation. For instance, a simple loop with bounds `for(i=0;i<n;i++)` can be described by the affine set: $[n] \rightarrow \{[i] : 0 \leq i \leq n\}, i \in \mathbb{Z}$. Integer sets support various operations like intersection, union, difference, projection, and cardinality. Relations between pairs of integer tuples that satisfy affine constraints are defined as integer maps: $[n, m] \rightarrow \{[i, j] \rightarrow [i] : 0 \leq i < n, 0 \leq j < m\}$. In addition to various set operations, these maps support inversion, composition, and domain intersection.

C. Set-Associative Caches and Cache Misses

For LLC with k-way set associative cache, we have k cache lines in a single set. Cache misses are categorized [44] into three types: *compulsory/cold misses*, that occur the first time a program accesses a cache line; *capacity misses*, that occur when the cache is too small to hold the working set of blocks, causing blocks to be evicted and later reloaded; and *conflict misses*, that occur when multiple blocks map to the same cache set, leading to unnecessary evictions.

D. Operational Intensity

The Operational Intensity (`OI` or $\mathcal{I}$) of a program is defined by the number of floating point operations executed per byte transfer by considering the data movement between the Last Level Cache (LLC) and DRAM. $\mathcal{I}$ is typically calculated in FLOP-per-Byte (FpB).

E. Roofline Models for Performance and Power

The *"Original" Roofline Model* [101] is a widely adopted analytical framework for identifying computational bottlenecks in software-hardware systems. It correlates the Operational Intensity (measured in FpB) of an application vis-a-vis the theoretical peak performance of the target architecture, providing a graphical representation with two fundamental ceilings; a horizontal *compute roof*, determined by the system's FLOP/s and a diagonal *memory roof*, constrained by the system's memory bandwidth. By comparing the achieved performance of a program against these ceilings, the model reveals whether it is *compute-bound* (bottleneck is FLOPs per sec) or *bandwidth-bound* (bottleneck is bandwidth, Bytes per sec). This insight guides optimizations for both software efficiency and architectural design.

Extending this, Choi et al. [20, 19] introduced the *Energy Roofline Model*, for energy bottleneck analysis. Its smooth *"arch curve"* captures the effects of static and dynamic power, enabling direct tradeoff analysis between performance and energy efficiency. In Tab. I, we show roofline constants defined for modeling execution time and power/energy.

TABLE I
PERFORMANCE/POWER ROOFLINE CONSTANTS AND THEIR DESCRIPTIONS

Constants	Description
t_{FPU}/ t_{byte}	Time taken per floating-point-operation/byte-transfer
$\mathcal{B}^t_{DRAM}$/$\mathcal{B}^e_{DRAM}$	Time/Energy balance with respect to $DRAM$
e_{FPU}/ $\widehat{p}_{FPU}$	Energy/Peak power per flop
e_{byte}/ $\widehat{p}_{byte}$	Energy/Peak power per byte transfer
p_{con}	Constant power

F. Frequency Capping

Power consumption scales linearly with frequency and quadratically with voltage ($P \propto f \cdot V^2$). Frequency capping [54, 73, 4] is a technique proposed to limit the usage of dynamic power. It is applicable across different *"energy zones"* like core/uncore to reduce over-provisioning of peak power consumption. It comes under the broad category of *"power capping"* technologies [108, 83, 62, 81, 74]. Power capping has advantages like "interval based power limiting", and "execution time window", while frequency capping does not. More crucially, power capping *cannot* be applied to *only uncore*, while frequency capping can be.

Frequency scaling [103, 102, 7, 41] is a more established technology that dynamically adjusts the frequency f_s based on workload demands to balance performance and power consumption. However, such scaling techniques often do an overprovisioning: setting scaling frequency f_s to a value significantly higher than the saturating frequency f_c, beyond which the performance gains are negligible; i.e., $f_s \gg f_c$. To mitigate this inefficiency, frequency capping *constraints* the maximum allowable frequency $f_{\max}$ such that $f_{\max} \leq f_c$, thereby avoiding unnecessary energy expenditure without sacrificing performance. Currently available frequency

drivers [56, 57, 55, 68] from various vendors allow scaling techniques by the end-user.

III. Overview of PolyUFC

In Fig. 3, we show the detailed overview of the PolyUFC compilation flow. Input programs in C/C++ are compiled to MLIR modules using `cgeist`, the frontend of Polygeist [64]; while PyTorch-programs are compiled using `torch-mlir` [61] frontend. PolyUFC lowers the input MLIR code from the `linalg` [60] dialect to the `affine` dialect [59], enabling polyhedral optimizations using `polygeist` [64] compiler. Affine loops are tiled using the Pluto [13] compiler and the code is converted to OpenSCoP [9] and PET [97] representations for analyses. We characterize the code using our proposed cache model PolyUFC-CM and derive a mathematical model for estimating the performance, bandwidth and power consumption of the input program with (uncore) frequency as a parameter. Finally, we search and embed the obtained frequency caps in the `affine` IR.

IV. Characterization of Affine Programs

In Sec. IV-A, we discuss PolyUFC-CM, an approach to model set-associative cache behavior to calculate the data movement cost of an affine program. In Sec. IV-B, we discuss the counting of various cache misses. In Sec. IV-C, we compute the Operational Intensity ($\mathcal{I}$) of programs using PolyUFC-CM. In Sec. IV-D, we discuss about how an affine program can be *(bottleneck) characterized* as CB/BB bound.

A. An Approximate Set-Associative Cache Model

We model inclusive, set-associative caches that use LRU replacement policy, with write-allocate and write-through policies. To model set-associative caches, we use integer sets and relations from `isl` [96], and symbolic counting using `barvinok` [98] of integer sets and relations. First, an access map is used that maps the statement instances of a statement (s_0) to the memory accesses of an array. Then, we model associativity for multi-level caches with additional dimensions for the index of the line and cache set in the access map A_{c_i} with c_i being the cache level, $1 \leq i \leq N$ where N is the number of cache levels. For Code. 4(a), we have the access map (A_{c_i}), for statement s_0 parametric in d input dimension, $A_{c_i} = \{s_0(d) \rightarrow B(d, line = \lfloor (d \cdot e)/\ell \rfloor, set = line \,\%\, \mathcal{N}_{c_i});\}$ and schedule map (S) for statements s_0, s_1 as follows:

$$S = \{s_0(d) \rightarrow (0, d); s_1(d) \rightarrow (1, d)\} \cap \mathcal{D}$$

where, ℓ is the cache line size, e is the element size in bytes and $\mathcal{N}_{c_i}$ is the number of cache sets in cache level c_i. Both maps are constrained by the iteration domain $\mathcal{D}$. Our model explicitly accounts for all three types of cache misses in set-associative caches, enabling us to accurately estimate cache miss rates under these assumptions. To model capacity/conflict misses, we show the formulation[4] for obtaining the reuse pairs of each cache set to calculate reuse distances [11, 12].

[4]We make some simplifying assumptions for modeling data-caches: no hardware prefetching, empty initial cache, and homogeneous cache associativity

Forward and Backward Reuse Distance Maps The forward map (F_{c_i}) links each schedule point to all lexicographically larger or equal points ($L_{\preceq}$) that access the same array element within the same set (Fig. 4). Similarly, the backward map (B_{c_i}) connects each point to all lexicographically smaller points, also restricted to the same set:

$$F_{c_i} = lexmin((S_{c_i} \circ S_{c_i}^{-1}) \cap L_{\prec})^{-1} \circ L_{\preceq}$$
$$B_{c_i} = S \circ \{(j) \rightarrow (i) \mid j \succeq i\} \circ S^{-1}$$

Where, $\circ$ is integer relation composition and same-set successor map is used to obtain accesses to the same set, defined as $S_{t,c_i} = \{(S_{c_i} \circ S_{c_i}^{-1}) \cap L_{\prec}\}$ with S_{c_i} maps schedule values to the memory accesses computed as $\{S^{-1} \circ A_{c_i}\}$.

We compute the intersection of access maps to capture all iterations between two memory accesses. Fig. 4 shows reuse pairs in the same set ($F_{c_i} \cap B_{c_i}$). The reuse distance RD_{c_i} is derived by composing access and reuse maps, identifying all intervening statements: $RD_{c_i} = (F_{c_i} \cap B_{c_i}) \circ A_{c_i}$. After obtaining the reuse pairs for the target associativity, we calculate the new set of capacity/conflict misses.

B. Counting Cache Misses

We discuss the computation of compulsory misses, and then of the capacity and conflict misses.

Counting Compulsory/Cold Misses We use A_{c_i} to obtain all the first schedule values to each memory element i.e., accesses with lexicographically minimal schedule value defined as $COLDMISS = lexmin({A_{c_i}}^{-1} \circ S) \circ S^{-1}$. Cardinality of $COLDMISS$ counts the total compulsory misses.

Counting Capacity and Conflict Misses Using RD_{c_i}, we model the conflict misses defined as all the accesses that are mapped to the same set. We count the reuse distances larger than $k_{c_i} \cdot \frac{\ell}{e}$, where k_{c_i} is associativity of cache level c_i as follows: $M_{c_i} = \{RD_{c_i} > k_{c_i} \cdot \frac{\ell}{e}\}$.

The cardinality of M_{c_i}, $|M_{c_i}|$ counts the total of capacity and conflict misses.[5] When the M_{c_i} is non-empty, and all the cache-sets are occupied in the cache, then the misses are categorized as capacity misses. Otherwise, they are categorized as conflict misses. With write-allocate and write-through, all misses at level c_i result in read accesses to c_{i+1}, and all writes are also forwarded to c_{i+1}.

Further, total cache misses at cache level c_i are calculated as $|COLDMISS| + |M_{c_i}|$. For example, LLC data cache misses can be calculated as $Miss_{LLC} = |COLDMISS| + |M_{LLC}|$.

An (Approximate) Extension for Loop-Level Parallel Programs: To model sharing of working set sizes across threads, the total cache misses (of any category) are approximated by dividing the sequential miss-count by the number of OpenMP [22, 71] program threads. This simple heuristic provides a first-order approximation, though it could miss inter-thread conflict and coherence misses.

C. Calculating Operational Intensity ($\mathcal{I}$)

The flop count is given by: $\Omega = \sum_{s \in \mathcal{S}} \omega_s \times |\mathcal{D}_s|$, where $s \in \mathcal{S}$ is set of statements in the affine program; for a statement

[5]This could be a non-affine (polynomial) set that is parametric (in problem-size); but, there exist efficient techniques [38, 85] to count these sets.

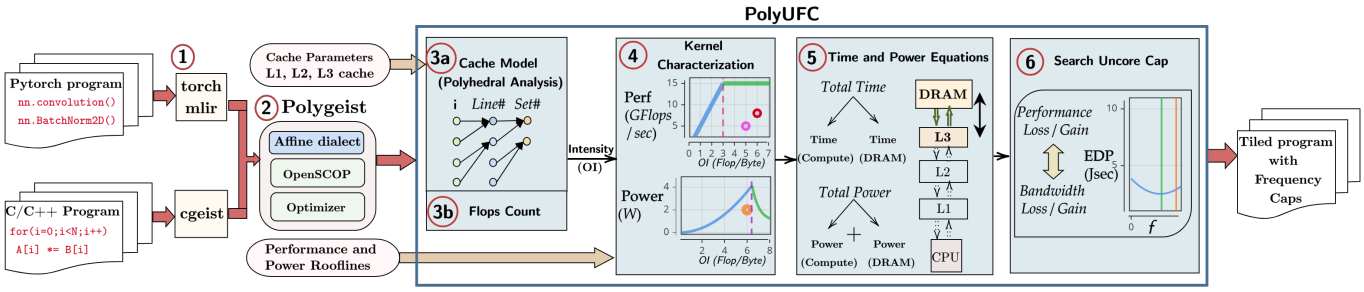
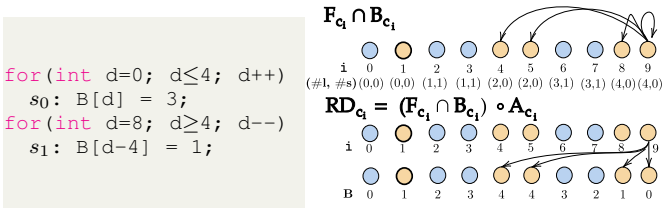


Fig. 3. **Detailed flow of** POLYUFC**:** (1) Input programs (`C/C++/PyTorch`) are compiled to MLIR modules, and lowered to MLIR in `affine` and `memref` dialects. (2) Lowered programs (in MLIR) are converted to OpenScop using `polygeist-opt` compiler, that also applies Pluto optimizations (tiling and parallelization). POLYUFC then analyzes these `affine` MLIR programs, with our proposed pass-pipeline. (3a) POLYUFC-CM calculates the total cache miss count for the target platform cache parameters (cache&line-size). (3b) flop counting algorithm calculates Operational Intensity (`OI`). (4) Use performance/power rooflines to do (bottleneck) characterization. (5) Estimate execution time and average/peak power using mathematical model with uncore frequency (f_c) and `OI` as parameters. (6) Search for best uncore frequency cap based on the (bottleneck) characterization.



(a) (b)

Fig. 4. Illustration of (a) an example affine program and, its corresponding (b) forward/backward distance maps.

s, the cardinality of iteration domain is $|\mathcal{D}_s|$, and the number of arithmetic operations is ω_s. The total data movement between LLC and $DRAM$ in bytes is $\mathrm{Q}_{DRAM} = Miss_{LLC} \cdot \ell$. We obtain $\mathcal{I}$ (measured in FpB) as follows:

$$\mathcal{I} = \frac{\Omega}{\mathrm{Q}_{DRAM}} \tag{1}$$

D. Kernel Characterization Using Rooflines

We evaluate the value of Operational Intensity ($\mathcal{I}$) against both performance and power roofline boundaries to determine the *bound and bottleneck characteristics* of the program. So, the $\mathcal{I}$ metric provides crucial insights into the compute and bandwidth characteristics of the program. Based on the relationship between $\mathcal{I}$ and the time machine balance ($\mathcal{B}^t_{DRAM}$), programs are classified as follows:

- **Compute-Bound** (CB): $\mathcal{I} \geq \mathcal{B}^t_{DRAM}$: These kernels are limited by computational capacity. Their performance scales primarily with processor computational power rather than memory bandwidth.
- **Bandwidth-Bound** (BB): $\mathcal{I} < \mathcal{B}^t_{DRAM}$: These kernels are limited by memory bandwidth. Their performance scales primarily with memory bandwidth rather than computational resources.

V. PARAMETRIC PERFORMANCE/POWER ESTIMATION

In this section, we propose a mathematical model that estimates performance, bandwidth and power, with uncore frequency cap (f_c), and Operational Intensity (`OI` or $\mathcal{I}$) as parameters under the given performance and power roofs. In Sec. V-A, we discuss estimation of performance and bandwidth of an affine kernel with f_c and $\mathcal{I}$ as parameters. In Sec. V-B, we discuss estimation of uncore power and energy.

A. Performance and Bandwidth

We estimate the performance and bandwidth for affine programs on a target architecture by decomposing total execution time ($T_{f_c,\mathcal{I}}$) into computation time ($T^{\Omega}_{\mathcal{I}}$) and memory time ($T^{Q}_{f_c,\mathcal{I}}$); each are parameterized by uncore frequency cap (f_c) and Operational Intensity ($\mathcal{I}$). Computation time $T^{\Omega}_{\mathcal{I}}$ is based on total FLOPs and FPU throughput at a fixed core frequency. And, time taken by data movement $T^{Q}_{f_c,\mathcal{I}}$ is derived from cache analysis (Sec. IV-A); this accounts for both cache hits and misses to accurately estimate data movement costs between the processor core and DRAM.

$$T_{f_c,\mathcal{I}} = T^{\Omega}_{\mathcal{I}} + T^{Q}_{f_c,\mathcal{I}} \quad (2) \qquad T^{\Omega}_{\mathcal{I}} = \Omega \cdot t_{FPU} \quad (3)$$

$$T^{Q}_{f_c,\mathcal{I}} = \sum_{i=1}^{N} \left(\prod_{j=1}^{i-1} \rho^m_{c_j} \cdot \rho^h_{c_i} \right) \cdot Q_{c_i} \cdot \mathcal{H}_{c_i} + \left(\prod_{j=1}^{N} \rho^m_{c_j} \right) \cdot Q_{DRAM} \cdot \mathcal{M}^t_{f_c,LLC} \tag{4}$$

The total time (Eqn. 2) combines computational latency from floating-point operations (Eqn. 3) and data movement average latency (Eqn. 4). We compute the level-wise cache miss ratios ($\rho^m_{c_i}$) and cache hit ratios ($\rho^h_{c_i}$), each for level i where, $1 \leq i \leq N$ and N is number of cache levels using our POLYUFC-CM, while deriving DRAM miss penalty ($\mathcal{M}^t_{f_c,LLC}$) with f_c as a parameter as shown below: $\mathcal{M}^t_{f_c,LLC} \propto \frac{1}{f_c} = \frac{a}{f_c} + b$ where, a and b are constants of the curve.

Hit latency ($\mathcal{H}_{c_i}$) is determined using the performance roofline model's t_{byte} at (maximum non-turbo) base core frequency. Note: all uncore time and power variables are explicitly parametric on f_c to enable a systematic analysis/exploration of uncore capping effects. Further, using Eqn. 2, we calculate the performance ($Perf_{f_c,\mathcal{I}}$) and bandwidth ($BW_{f_c,\mathcal{I}}$) of the input program as shown below:

$$Perf_{f_c,\mathcal{I}} = \frac{\Omega}{T_{f_c,\mathcal{I}}} \quad (5) \qquad BW_{f_c,\mathcal{I}} = \frac{Q_{DRAM}}{T_{f_c,\mathcal{I}}} \quad (6)$$

B. Uncore Power and Energy

We estimate uncore power consumption using established roofline constants [20] for DRAM ($\widehat{P}_{DRAM}$ and $\mathcal{B}^t_{DRAM}$) combined with our computed OI ($\mathcal{I}$). For a multi-core processor, total power consumption consists of three components:

$$P_{f_c,\mathcal{I}} = p_{con} + P^{core}_{\mathcal{I}} + P^{uncore}_{f_c,\mathcal{I}} \tag{7}$$

Where p_{con} represents static power [16], while $P^{core}_{\mathcal{I}}$ and $P^{uncore}_{f_c,\mathcal{I}}$ represent the dynamic power consumption of core and uncore components, respectively. The total peak power (ceiling) estimation is specialized based on the characterization of the kernel (as CB or BB):

$$\widehat{P}_{f_s,\mathcal{I}} = p_{con} + \begin{cases} \widehat{P}_{f_s,DRAM} \cdot \frac{\mathcal{B}^t_{DRAM}}{\mathcal{I}} + \widehat{p}_{FPU} & \rightsquigarrow \text{CB} \\ \widehat{P}_{f_s,DRAM} + \widehat{p}_{FPU} \cdot \frac{\mathcal{I}}{\mathcal{B}^t_{DRAM}} & \rightsquigarrow \text{BB} \end{cases}$$

As $\mathcal{I}$ increases beyond $\mathcal{B}^t_{DRAM}$, peak-power for CB approaches flop-only peak power $\widehat{p}_{FPU}$, and for BB it approaches bandwidth-bound peak power $\widehat{P}_{f_s,DRAM}$. To account for different uncore frequencies, we approximate the peak power per byte of roofline variable $\widehat{P}_{f_s,DRAM}$ as a linear function of f_s, where $\alpha_{\hat{P}}$ and $\gamma_{\hat{P}}$ are constants derived from its linear curve fitting. This leads to:

$$\widehat{P}_{f_s,\mathcal{I}} = p_{con} + \begin{cases} (\alpha_{\hat{P}} \cdot f_s + \gamma_{\hat{P}}) \cdot \frac{\mathcal{B}^t_{DRAM}}{\mathcal{I}} + \widehat{p}_{FPU} \rightsquigarrow \text{CB} \\ (\alpha_{\hat{P}} \cdot f_s + \gamma_{\hat{P}}) + \widehat{p}_{FPU} \cdot \frac{\mathcal{I}}{\mathcal{B}^t_{DRAM}} \rightsquigarrow \text{BB} \end{cases} \tag{8}$$

Further, to estimate the total power consumption of an affine program, we estimate the average total power as follows[6]:

$$P_{f_c,\mathcal{I}} = p_{con} + \begin{cases} Q_{DRAM} \cdot \mathcal{M}^p_{f_c,\text{LLC}} \cdot \frac{\mathcal{B}^t_{DRAM}}{\mathcal{I}} + \widehat{p}_{FPU} & \rightsquigarrow \text{CB} \\ Q_{DRAM} \cdot \mathcal{M}^p_{f_c,\text{LLC}} + \widehat{p}_{FPU} \cdot \frac{\mathcal{I}}{\mathcal{B}^t_{DRAM}} & \rightsquigarrow \text{BB} \end{cases} \tag{9}$$

Let $\mathcal{M}^p_{f_c,\text{LLC}}$ denotes the power consumed to serve a miss penalty in the LLC at frequency f_c. Utilizing the miss penalty power from microbenchmarks (such as pointer chasing) ensures the power estimates are an upper bound for average power. Using curve fitting, we model this power as a function of f_c. To use the above equation for uncore frequency capping[7] we require estimating the upper bound for dynamic power of the *uncore* ($P^{uncore}_{f_c,\mathcal{I}}$) with $\mathcal{I}, f_c$ as parameters; while the dynamic power of the core ($P^{core}_{\mathcal{I}}$) is obtained with a fixed base core frequency as follows:

$$P_{f_c,\mathcal{I}} = p_{\text{con}} + \begin{cases} Q_{DRAM} \cdot (\alpha_P \cdot f_c + \gamma_P) \cdot \frac{\mathcal{B}^t_{\text{DRAM}}}{\mathcal{I}} + \widehat{p}_{\text{FPU}} \rightsquigarrow \text{CB} \\ Q_{DRAM} \cdot (\alpha_P \cdot f_c + \gamma_P) + \widehat{p}_{\text{FPU}} \cdot \frac{\mathcal{I}}{\mathcal{B}^t_{\text{DRAM}}} \rightsquigarrow \text{BB} \end{cases} \tag{10}$$

The constants α_P and γ_P are obtained from linear curve fitting[8] of $\mathcal{M}^p_{f_c,\text{LLC}}$, similar to curve fitting of $\widehat{P}_{f_s,DRAM}$. Note that quadratic curve fitting provides a more accurate estimate by minimizing the error in power prediction.

[6]Note that Eqn. 9 is similar to the classic (energy) roofline equation [20] except this is architecture and application specific, tuned for capping.

[7]This is in contrast to works [48, 7], that proposed *core* frequency scaling.

[8]Linear-Fitting is used for average-power $P_{f_c,\mathcal{I}}$ as a reasonable approximation for modeling miss-penalty-power-estimation due to its monotonic-nature.

Total energy Using f_c and $\mathcal{I}$ as parameters, we estimate the total energy of the kernel using the proposed execution time ($T_{f_c,\mathcal{I}}$) and total power ($P_{f_c,\mathcal{I}}$).

$$E_{f_c,\mathcal{I}} = E^{\Omega}_{\mathcal{I}} + E^{Q}_{f_c,\mathcal{I}} = \Omega \cdot e_{FPU} + T^{Q}_{f_c,\mathcal{I}} \cdot P_{f_c,\mathcal{I}} \tag{11}$$

VI. ML-PolyUFC: Multi-Level Application of Uncore Frequency Caps

In this section, we present how our PolyUFC compiler can be extended as a ML-PolyUFC[9] framework, that can be used to apply the frequency capping at multiple levels of granularities of MLIR dialects, at multiple transition points (phase-changes) in the program, and with multiple metrics (performance-only, energy and EDP). In Sec. VI-A, using the MLIR characterization pass (Sec. IV), we do a study of granularity on some real-world affine programs, both across/inter dialects. Then, in Sec. VI-B, we explore different methods and trade-offs for applying frequency caps across MLIR dialects, and show that `linalg` emerges as the natural choice for phase change analysis. Then, in Sec. VI-C we sketch a simple search algorithm that guides the selection of best uncore frequency caps, balancing performance and energy efficiency.

A. A Study of Multi-Level Granularity: Across Inter/Intra Dialects

We study the phase change problem across MLIR dialects; then, the changes that occur within the `linalg` dialect.

Phase Changes Across Inter Dialects: As shown in Fig. 5, high-level PyTorch [5] operations like `sdpa` (a key computational Op in `bert` [24] with CB/BB regions) decompose into multiple `linalg` operations (e.g., element-wise and matrix multiplications), which are further lower (`linalg`→`affine`) to multiple perfectly-nested `affine` loop-nests.

Phase Changes in Intra (`linalg`) Dialect: A study on untiled implementation of `sdpa` from BERT reveals the following pattern[10]: CB → BB* → CB, where the middle BB* section spans 7 `linalg` Ops in length. The initial and last CB are `matmul`. While characterization at the torch-IR level reveals the `sdpa` operation as BB due to low OI from `matmul` layers, the fine-grained `linalg`-IR analysis reveals CB phases in the structured Ops.

From the above real-world examples, it can be seen that the nested IR levels of MLIR exposes distinct multi-level computational phases, resulting in different kinds of phase transitions in the `torch`, `linalg`, and `affine` dialects. This creates a complex landscape for frequency cap optimization. Therefore, these phase changes highlight the need for a *granular, dialect-aware frequency capping strategy*, which avoids both over-provisioning or under-utilization of the bandwidth.

[9]Here, ML itself has *multi-level* meanings, and the similarity of this naming to the popular MLIR compiler [53] is apparent.

[10]We use regular expression notation using Kleene star [50].

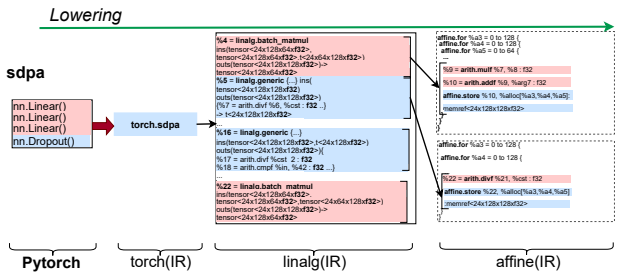


Fig. 5. Illustration of phase changes of characterization for scaled dot product attention (`sdpa`) from BERT across `torch`, `linalg`, and `affine` dialect IRs. (CB and BB)

B. Dialect-Aware Frequency Cap Strategies: for Analysis and Application

Analyzing and applying frequency caps is challenging due to phase changes that occur across multiple IR levels and within individual dialects.

Granularity for Analysis Given the affine structure of our input programs, and our static analyses that rely on polyhedral analysis and tools [96, 13, 64], the `affine`-IR (of MLIR) emerges as the natural choice to calculate `OI` and to obtain estimations of performance and power. This level of granularity enables detailed insights necessary for the effective application of frequency caps across various MLIR dialects. Owing to the composability and modularity of MLIR, `affine`-level analyses can be propagated and utilized within higher-level dialects such as `linalg` and `torch`.[11]

Granularity for Application The granularity at which frequency caps are applied plays a critical role in performance. Applying frequency caps at the `torch`-dialect is suboptimal due to the encapsulation of multiple CB/BB phases within a single `torch` Op, leading to coarse and imprecise control. On the other hand, capping at `affine`-dialect incurs excessive overhead from frequent changes at different `affine` loads, stores and `arith` operations.

Choosing `linalg`-dialect capping aligns well with stable computational/memory characteristics typically found within `linalg` ops, also offering an effective trade-off between control granularity and runtime overhead for efficient frequency management. We assume the current analysis is applied post-transformation on `linalg`/`affine` IR. In this context, all Polygeist-Pluto optimizations are activated at the affine IR, while optimizations based on LLVM IR utilizing the Polly framework remain disabled.

C. Searching for Frequency Caps (PolyUFC-Search)

Note that Eqns. 4, 10 are non-linear in f_c and $\mathcal{I}$. They induce a non-convex search space. In addition, the energy minimization or performance maximization (Eqn. 11) are also non-linear. Obtaining the frequency caps that give the desired improvements involves searching the above non-linear search space. Rather than applying convex relaxations [10], we employ a binary search strategy guided by bottleneck characterization, which enables identification of energy-efficient frequency settings. The search space is explored using performance and bandwidth estimates (Eqns. 5, 6), with optimization guided by energy-delay product (EDP) and the bandwidth/performance gains. EDP is computed using Eqns. 2 and 11, and a cost function is used to balance energy and execution time across different bottleneck types.

Search Our algorithm uses a simple binary search with 0.1 GHz step size to select frequency caps based on program type: for CB operations, it searches lower frequencies to prioritize energy efficiency when performance loss is within a tunable threshold ($< \epsilon$). Here, power scales *linearly* with f_c but *nonlinearly* with $\mathcal{I}$ (from $P_{f_c,\mathcal{I}} \propto \frac{f_c}{\mathcal{I}}$). Higher $\mathcal{I}$ reduces memory accesses, lowering uncore power consumption. For BB operations, it searches higher frequencies to prioritize performance when bandwidth and performance gains are aligned. Here, power scales *linearly* with both f_c and $\mathcal{I}$ (from $P_{f_c,\mathcal{I}} \propto f_c + \mathcal{I}$) as increased f_c raises uncore activity, while higher $\mathcal{I}$ increases computations per memory access. For each frequency setting, relative changes in performance, bandwidth, and EDP are evaluated as $\Delta Perf = \frac{Perf_{\text{new}}}{Perf_{\text{old}}}$, $\Delta BW = \frac{BW_{\text{new}}}{BW_{\text{old}}}$, and $\Delta EDP = \frac{EDP_{\text{new}}}{EDP_{\text{old}}}$, respectively. These metrics drive the search process, enabling architecture-aware tuning. The search terminates when the frequency stabilizes between iterations or the search space is exhausted. While the method *focuses on EDP*, it also supports *energy-only* or *performance-only* optimization objectives.

Tuning PolyUFC-Search**:** A tunable threshold ϵ guides PolyUFC-Search by comparing $Perf$ and BW changes. For CB programs, f_c is reduced if performance loss does not exceed BW loss by more than ϵ, enabling energy savings. For BB programs, f_c increases only when $Perf$ gains match BW gains within ϵ, ensuring efficient capping. Adjusting ϵ balances energy efficiency and $Perf$ across programs and μ-archs. Moreover, PolyUFC-Search can be extended to frequency scaling by tuning EDP and total power (Eqn. 10).

VII. Experimental Evaluation

In this section, we first give some implementation details of our PolyUFC system (Sec. VII-A): including our algorithms to calculate `OI`, our mathematical modeling, and some code generation details. We then explain our experimental setup in Sec. VII-B: this includes our experimental platforms of our two x86 micro-architectures. We carefully selected an illustrative set of benchmarks (Sec. VII-C) to show a variety in the compute/memory boundedness to evaluate our proposed uncore frequency capping technique. We show the results on characterization of benchmarks for performance and power (Sec. VII-D). Using the characterization and the proposed ML-PolyUFC in Sec. VI for uncore frequency capping, we compare the EDP of benchmarks with the available Intel uncore scaling driver on the target platforms (Sec. VII-E). Finally, we discuss the findings in relation to existing techniques and their broader implications (Sec. VII-F).

[11] Analysis on lower-level representations like `llvm-ir` is outside the scope of this work.

A. Implementation of POLYUFC

a) Operational Intensity ($\mathcal{I}$): We implemented Eqn. 1 calculations as analyses passes using Integer Set Library (`isl-0.27`) [96] and `barvinok` [98] (version 0.41.8) libraries[12]. We integrate them as MLIR pass within Polygeist [64, 63] compiler using `llvm-18`. This enables our framework to automatically analyze MLIR code containing affine control loops, facilitating precise static analysis of memory behaviors[13].

Implementation for **characterization and modeling** is written as a MLIR pass which take roofline variables[14] as inputs, parametric in f_c. In addition, it takes $\mathcal{I}$, and the metrics computed by POLYUFC-CM (Q_{DRAM}, $\rho^h_{c_i}$, $\rho^m_{c_i}$, Ω) as parameters. It analyzes the polyhedral IR and applies characterization on each polyhedral statement within an `affine` loop.

Code Generation Caps are applied based on the characterization of the top-level `affine` Op or `linalg` Op i.e, min (max) of all caps for statements for CB (BB). Using our performance, bandwidth, and power models parameterized by frequency f_c, we implement a search algorithm (Sec. VI-C) in MLIR that inserts `func` calls to set frequency caps before each top-level `affine for` Op. We use pattern-rewrite optimizations to remove redundant frequency caps.

We enable parallelism using the Polygeist-Pluto optimizer and `affine-parallelize` pass in MLIR, and then lower the code with the `scf` to `openmp` pass. Finally, the code is lowered and translated to `LLVM-IR`, which is then compiled for execution on the target machine.

TABLE II
BENCHMARKS: (A) SELECTED MLIR KERNELS (⊗) CONV2D ($nchw\_fchw$), LM-HEAD-MATMUL, AND SCALED DOT PRODUCT ATTENTION (SDPA). (B) POLYBENCH [79] WITH LARGE PROBLEM SIZE.

Prog.	Source	Problem Sizes
conv2d⊗	ALEXNET	1×3×224×224; 64×3×11×11
	CONVNEXT	1×384×28×28; 768×384×2×2
	WIDERESNET	64×1024×7×7; 2048×1024×1×1
sdpa⊗	BERT	2×12×128×64
	GEMMA2	1×16×7×256
matmul⊗	GPT2	4×768×50257
	LLAMA2	13×4096×32000

TABLE III
MICROARCHITECTURES USED AS PLATFORMS

Arch	Released	CPU	Core (GHz)	Uncore (GHz)
Broadwell (BDW)	2015	Xeon 1650-v4 (6C/12T)	1.2-4	1.2-2.8
Raptor Lake (RPL)	2023	Intel i5-13600 (14C/20T)	0.8-5	0.8-4.6

[12]As of the writing of our paper, `barvinok` library [98] is the *only* complete implementation for counting parametric integer sets and relations.

[13]Note that this method assumes unitary model, all operations are considered to have the same flop-count; it does not take into account the difference between individual low-level (`arith`-dialect/`math`-dialect) operators (like `addf`, `mulf`, `divf`) , and types (like `f16`, `f32`) of operands.

[14]We generate roofline microbenchmarks [19] for different intensities ranging from 0–10^6. Each PAPI [65, 47] event runs for 2^{10} iterations[1].

B. Experimental Setup

Our compiler baseline is parallel tiled kernels optimized with Pluto [15] (v0.11.4, using default tile-size = 32). Our hardware baseline is the default Intel uncore scaling driver. The platforms run on Ubuntu 24.04, with various modern Intel (x86) CPUs. In Tab. III, we show the target platforms for benchmarking and their core and uncore frequency ranges. We use Intel UFS driver [57] that also allows setting capping frequency f_c; this enables a limited set of uncore frequencies. For other frequency domains, we use existing hardware drivers like Intel P-state [56] for core with performance governor setting. For characterization (Sec. VII-D) as CB/BB, we disable data prefetchers and hyperthreading. For performance and energy numbers (Sec. VII-E), it is on by default.

C. Evaluation Benchmarks and Justification

We highlight the characterization of the selected kernels (CB and BB) in Tab. II. The selected affine programs present a diverse mix of CB/BB access patterns, allowing for a systematic study of how frequency capping impacts the energy and performance profiles. POLYBENCH [79] kernels (e.g., `gemm`, `mvt`, `2mm`) are widely used in benchmarking. Layers—like `conv2d` [52, 105, 58] (inference), `matmul` [24, 90] (language modeling head), and `sdpa` [82, 92] (self-attention)—are affine programs derived from important models (lowered to `linalg` Ops ⊗) taken from vision and language domains representing real-world deep learning applications. They are highly relevant because modern server and AI workloads are sensitive to both memory subsystem and interconnect performance, that are primarily governed by the uncore domain [34].

D. Characterization of Programs Using Rooflines

In Fig. 6, we present static roofline characterization obtained using POLYUFC, comparing them with the hardware numbers generated by Pluto tiled-parallel; at minimum core and maximum uncore frequency settings. We also show POLYUFC generated frequency-capped code along with the roofline estimates. We estimate performance using Eqns. 5 and 6. Power roofs for all uncore frequencies are constructed using Eqn. 8, and total power is estimated using Eqn. 10. We validate our characterization by comparing it with performance and power data obtained from PAPI [65, 47] performance counters and runtime measurements. Below are some key observations for the RPL platform:

- CB: The `OI` characterization matches hardware results for all benchmarks, classifying them as CB. For `conv2d` (⊗) from CONVNEXT, performance estimates differ by less than 7% from hardware. Among the 22 kernels, 13 are characterized as CB in POLYBENCH, primarily matrix–multiplication routines from `blas/kernels/solvers`, together with data-mining kernels and low-bandwidth stencils such as `jacobi-1d`.
- BB**:** Based on the `OI` characterization, all benchmarks are correctly classified as BB. Among the 22 kernels, nine are characterized as BB in POLYBENCH, mainly matrix–vector products from `blas/kernels/solvers`,

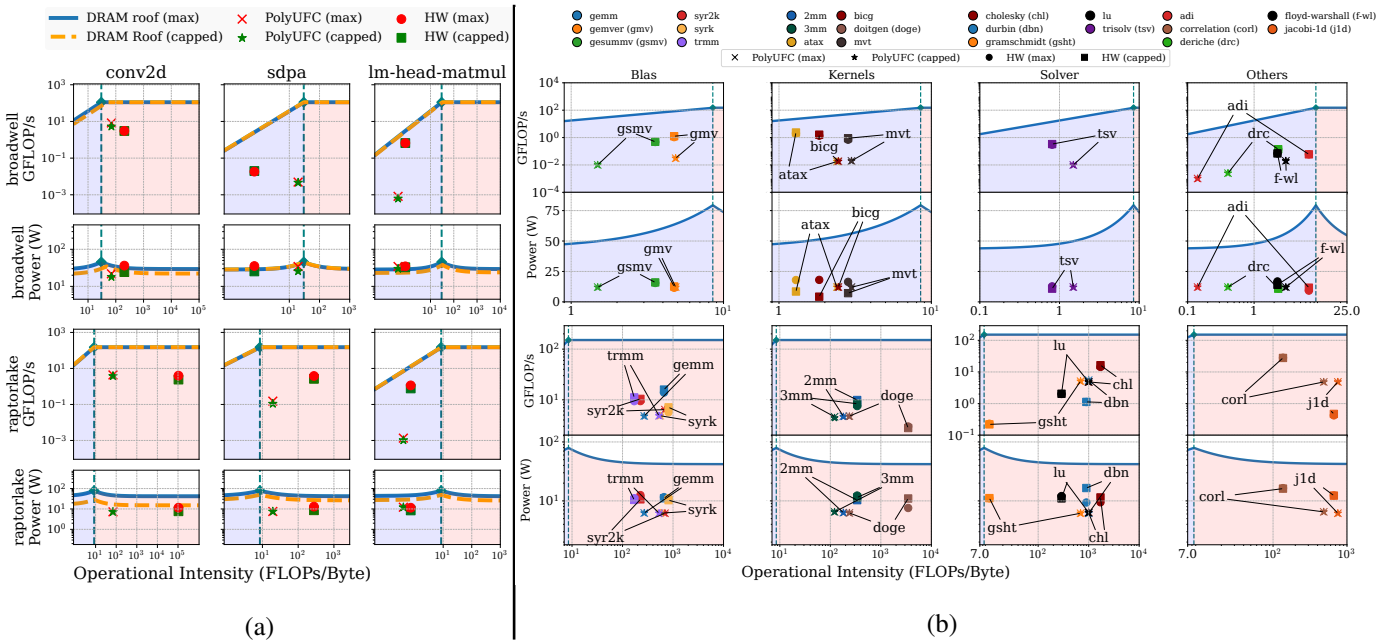


Fig. 6. Performance and Power characterization: (a) `conv2d` (CONVNEXT), `sdpa` (BERT), and `lm-head-matmul` (LLAMA2) on BDW/RPL. Vertically, from top (BDW) to bottom (RPL), the characterization of affine programs shifts from BB → CB due to larger cache sizes and higher bandwidth in the uncore subsystem of RPL. Horizontally, the shift towards BB behavior is driven by increased data movement and higher bandwidth demands. (b) POLYBENCH on RPL for different categories of kernels. Vertically, from top to bottom, the characterization of programs shifts from BB → CB due to higher `OI`.

along with memory-intensive kernels such as `deriche` and `adi`. RPL is better optimized for BB codes, as it provides a larger LLC cache and higher bandwidth. Kernels characterized by low `OI` exhibit reduced computational performance while demanding higher BW.

In summary, programs with high `OI` (`conv2d` (CONVNEXT), `sdpa` (BERT), `gemm`, `jacobi-1d`, `durbin`, `2mm` from POLYBENCH)are typically CB for large inputs, while those with low `OI` (`lm-head matmul` (LLAMA2) and `mvt`, `gemver`, `trisolv` from POLYBENCH) remain BB due to their data movement demands. POLYUFC effectively identifies these bottlenecks, guiding uncore search based on program characterization.*We note that all the evaluation benchmarks on RPL are classified correctly*. POLYUFC generated programs consume lesser power for CB, and almost same power for BB, as evident by the H/W measurements on RPL.[15]

E. Time, Energy, and EDP Comparison

In Fig. 7, we show EDP along with the performance and energy improvements. The EDP comparison across Intel micro-architectures (BDW and RPL) reveals that POLYUFC generated code yields notable EDP improvements across different characterizations. CB programs like `conv2d` benefit the most, with EDP improvements up to 13% for WIDERESNET, highlighting the effectiveness of uncore tuning for CB programs. Other CB programs such as `gemm`, `2mm`, `durbin`, and `sdpa` also experience measurable gains, with improvements upto 42%. BB programs, including `mvt` and `lm-head-matmul`, show improvements as well, achieving up to 27% and 12%, respectively on RPL. For POLYBENCH, the geomean EDP improves by 12% on BDW and by 10.6% on RPL, respectively. We set $\epsilon = 1 \times 10^{-3}$ for minimal tradeoff between $Perf/BW$ and $E_{f_c,\mathcal{I}}$. These results indicate that while CB programs see the highest relative gains, BB programs also profit significantly from uncore frequency optimizations, underscoring the broad applicability and impact of such techniques on both CB/BB programs.

Performance and Energy Tradeoff CB programs such as `conv2d`, `gemm`, and `2mm` show up to a 2.5% performance loss, but achieve energy savings of up to 14%. In contrast, BB programs like `mvt` and `sdpa` (GEMMA2) see performance gains of up to 20% and similar improvements in energy efficiency on RPL. For `lm-head-matmul` (GPT2) on BDW, performance declines due to underestimated bandwidth requirements.

F. Discussion

Intra-Kernel DVFS and Inter-Kernel Capping Each POLYUFC call on an operation/kernel results in setting the uncore cap, incurring an average overhead of 35μs on BDW and 21μs on RPL. In `sdpa` (GEMMA2), that is a multi-kernel benchmark, there are 28 kernels with 28 inter-kernel frequency-caps. On BDW, the overhead is $\approx$ 1ms, while on RPL it is $\approx$ 0.8ms (cumulative for all kernels). Fig. 7 demonstrates that inter-kernel *uncore* capping achieves equivalent or better performance than compared to only intra-kernel core/uncore DVFS/DUS, while offering a simpler, lower-overhead implementation. This validates inter-kernel capping as a practical

[15]For BDW, we are able to show only the total power (including both core and uncore). This is because of non-availability of *"energy zone"* for uncore using RAPL [23, 49] for BDW micro-architecture.

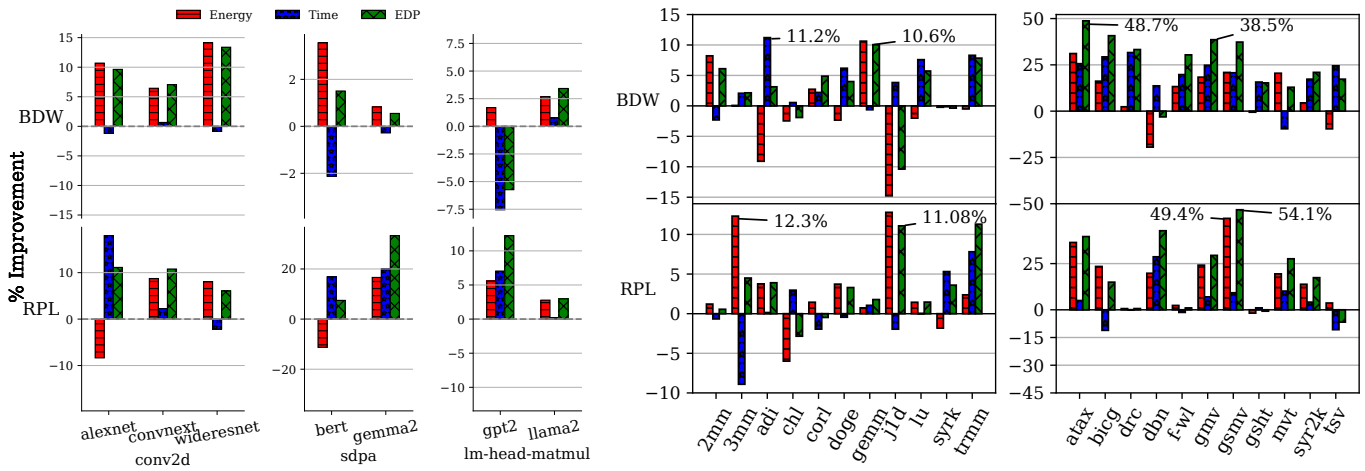


Fig. 7. Time, Energy and EDP comparison of affine programs on Intel micro-architectures. The baseline is Pluto tiled-parallel with the default Intel *uncore* UFS driver and the Intel *core* P-state driver.

TABLE IV

COMPILE-TIME BREAKDOWN FOR BENCHMARKS IN TAB. II OF POLYUFC FLOW: PREPROCESSING (ST. 2 EXTRACTION), PLUTO (ST. 2 OPTIMIZER), POLYUFC-CM (ST. 3A-3B), AND ST. 4–6 OF FIG. 3. TIME IS SHOWN IN MILLISECONDS FOR THE BDW CACHE CONFIGURATION.

Stages/Program	conv2d-AN	gpt2	sdpa-BERT	conv2d-WR	conv2d-CN	llama2	sdpa-G2	2mm	3mm	adi	atax	bicg	cholesky	correlation	deriche	doitgen	dbn	fwl	gemm	gemver	gesummv	gsht	jacobi-1d	lu	mvt	syr2k	syrk	trisolv	trmm
Preprocess	1	1	625	1	1	1	10	5	7	6	5	6	4	6	8	5	5	6	6	4	5	6	3	5	5	6	5	5	5
Pluto	454	88	1312	245	16196	190	39821	231	427	1092372	103	101	4172	374	49086	1227	323	797	116	96	93	491	100	6696	40	97	96	812	256
POLYUFC-CM	41753	30778	12794	1228	116200	26832	76224	8235	12157	181029	1594	1353	207300	11432	866	12376	934	1247000	3684	2942	949	8517	12772	1665118	6843	14922	5553	6280	5211
Steps 4-6	213	5	11	2	1	1	28	9	11	34	6	7	40	6	21	19	7	61	6	11	9	9	13	187	16	4	4	7	2
Total	42420	30872	14742	1476	132400	27024	116100	8486	12610	1273000	1715	1473	211600	11828	49995	13643	1276	1248000	3819	3059	1061	9031	12893	1672001	6911	15036	5666	7111	5482

and efficient approach for uncore power management, when combined with *core* DVFS using the P-state driver [56, 46].

Core Frequency Selection Given that hardware P-state drivers for *core* DVFS can override OS frequency settings, POLYUFC prioritizes uncore power management via compiler-directed caps. However, the POLYUFC remains adaptable and can be used to manage the core frequency domain.

Compilation Overhead In Table. IV, we show the compile time characteristics and the breakdown. For POLYBENCH, we show the numbers for only 22 benchmarks; for the others[16], POLYUFC-CM times out, with the timeout value set as 30mins. For kernels that overshoot, we reset the f_c to maximum. Presburger set manipulations can be computationally expensive [32, 39, 67] for arbitrary expressions, and `barvinok` [98] needs to count high-dimensional iteration domains returned by Pluto tiling[17].

EDP and Associativity In Fig. 8, we compare the estimated EDP of Pluto-tiled-parallel implementations (using default tile-sizes) for fully-associative and set-associative settings of POLYUFC-CM, with hardware measurements. The selected representative programs, `gemm` and `2mm`, are key computational kernels that incur high conflict misses and reveal variations in estimated EDP across different uncore frequencies, thereby necessitating a set-associative cache model. For `gemm` on BDW, the set-associative configuration yields a 4.65% EDP gain at $f_c = 1.7\,\text{GHz}$, while fully associativity results in 3% EDP gain at $f_c = 1.2\,\text{GHz}$. For `2mm` on RPL, set associativity yields a 0.55% EDP gain at $f_c = 2.2\,\text{GHz}$, while full associativity results in a 2.8% loss at $f_c = 1.1\,\text{GHz}$.

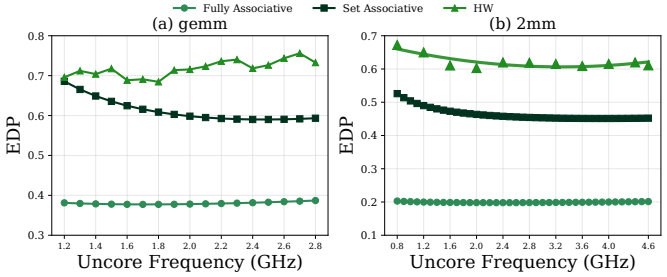


Fig. 8. EDP comparison for (a) `gemm` on BDW and (b) `2mm` on RPL over f_c. Estimated values using Eqns. 4, 11 with fully and set associative settings of POLYUFC-CM vs. HW measurements.

Search Precision We set the **search precision** to 100MHz increments within the permitted range of uncore frequencies. This choice reduces the search steps to ≈ 39, enabling the algorithm to converge more rapidly.

VIII. RELATED WORK

Frequency Capping: Li et al. [54] introduced frequency capping to address over-provisioning issues in conventional DVFS methods, and improve energy efficiency.

Cache Misses Various set-associative cache models [35, 17, 8] have been proposed that work for affine programs, though POLYUFC-CM is the *first cache model*, to the best of our knowledge, that is evaluated *on tiled parallel programs*.

[16]Polygeist [64] does not generate explicit auto-vectorization code as performed by Pluto optimizer, as the existing affine dialect super-vectorizer pass fails to generate vector IR.

[17]We use a custom duplicate elimination algorithm inside the reuse pair polynomial calculation over the union of maps. This improves the overall compile time by a 2.7x factor (geomean speedup).

PolyCache [8] provides multi-policy-aware exact modeling of set-associative cache misses of affine programs for single thread execution on private (upto L2) caches, offering high accuracy at the expense of increased compile-time.

In POLYUFC-CM, computation for compulsory misses is identical with [8]; for capacity/conflict misses, each cache set is evaluated independently, assuming fully-associative behavior within each set. This simplification enhances PolyUFC-CM's scalability by removing redundant reuse polytopes before symbolic counting of convex sets. On the other hand, using a modeling like [8]—that relies on k-fold set-difference operations—for high-associative shared-caches like LLC can trigger a combinatorial explosion in the number of pieces, and leads to quantifier eliminations. POLYUFC offers expressiveness comparable to the affine dialect with quasi-linear expressions, supporting input IR representation with constant-size tiling, parametric tiling restricted to hyper-rectangular regions with constant lower bounds, and non-parametric loop skewing.

Recent techniques [38, 85, 76] for estimating cache behavior of affine programs primarily target fully-associative caches, but their approximations may misestimate miss rates on practical set-associative caches (e.g., 16- or 20-way LLCs), affecting program characterization and EDP analysis.

Though POLYUFC can be improved in precision and scalability, it can be seen as a *post-scheduling polyhedral compiler flow*, that can be integrated with other established polyhedral compilers [80, 99, 95, 6] in a complementary manner.

Energy-Efficient Compilation has progressed from early static DVFS methods [103, 102] to hybrid static+dynamic and hardware-aware strategies [7], incorporating `OI` analysis, profiling, and software-controlled DVFS. Some methods are: race-to-sleep paradigm [104], decoupled access-execute [48, 51, 93], along with combining compiler-runtime [94, 66], and ML for power cap prediction [42], and coordinated core/uncore frequency scaling [89], fine-grained autotuning [86], and kernel-level optimization for heterogeneous systems [29].

Linux OS-based core scaling governors [28] (such as `ondemand` and `powersave`) incur high control-loop latency. On the other hand, compiler-based approaches, such as [7, 48], or POLYUFC provide fine-grained phase-aware guidance within/across loop-nest boundaries, thereby reducing control-loop latency by orders of magnitude.

Roofline-Aware Characterization POLYUFC uses a Performance+Power *characterization*[18] (using POLYUFC-CM, a multi-level cache model to estimate $Q_{c_i}, 1 \leq i \leq N$, Q_{DRAM}, and `OI` with Eqns. 4, 9) and a rooflines-based parametric mathematical model (Sec. V) to estimate average/peak performance and power. In comparison, [7] proposes program *classification* using profiling and an approximate static model PolyFeat [80] for `OI` calculation of affine programs.

Compiler-Driven Core/Uncore Scaling and Capping POLYUFC applies *uncore* capping that operates on-top-of *core* scaling supported by the Hardware P-State [21, 56], and *uncore* scaling of UFS [57] driver. Core scaling is orthogonal for the goals of POLYUFC; it is left for the default HWP to manage core-frequency. Capping necessitates knowledge of multi-domain (core/uncore) rooflines, both performance and power, that existing core scaling techniques [7, 48] lack.

Intra/Inter Loop-Nest Control POLYUFC performs inter loop-nest *uncore* capping as proposed in Sec. VI, aligning with the stable `OI` typically seen in single-phase ML loop-nests (e.g., `matmul`, `matvec`, `ReLU`). Although finer-grain control is possible in POLYUFC, empirical results on modern CPUs (like RPL) show loop-level capping is effective because of its low latency. In contrast, [7] proposes intra-loop-nest *core* scaling by modeling frequency variations within the loops.

We are not aware of any prior compiler-driven frequency capping approaches. It is possible to use techniques like DVFS for core [103, 48, 7, 87], and DUFS for uncore [37, 18, 34, 4] alongside POLYUFC to optimize for energy and performance.

Rooflines for Compiler Optimizations Elango et al. [27] used rooflines to theoretical define the `OI` upper-bound for a given program. Our goal is to estimate `OI` and statically characterize programs according to their boundedness.

IX. CONCLUSION

In this work, we introduced POLYUFC, the *first compilation flow* for uncore frequency capping in MLIR. By combining statically computed Operational Intensity with performance and power roofline analyses, POLYUFC uses characterization of programs to make frequency capping decisions. Experimental results on important affine programs—from NLP/Vision domains and POLYBENCH—to demonstrate that our approach effectively balances performance and energy consumption, and achieves improvements in EDP of up to 42% on CB and upto 54% on BB programs. POLYUFC framework enables further research in compiler-driven power optimizations across dialects, as demonstrated with ML-POLYUFC, and can be extended to new architectures and multiple optimization goals. Additional details are available at https://compilers.cse.iith.ac.in/projects/polyufc.

ACKNOWLEDGEMENTS

We are grateful to Govindarajan Ramaswamy, S. Venkata-Keerthy, and Siddharth Jain for their valuable feedback and discussions on the submitted version of this paper. We thank Albert Cohen and Dibyendu Das for providing feedback on early versions of the work. We acknowledge Charukesh V, Siddhartha Neyagapula, Ananya Varshney, Rajiv Shailesh Chitale, and Shrikar Anand Dongre for their contributions at various stages of this work. We are also grateful to the anonymous reviewers for their insightful and detailed comments that substantially improved the paper.

This work is supported by the Prime Minister's Research Fellowship (PMRF) programme, Government of India, with additional funding from faculty grants provided by AMD, and Qualcomm.

[18]The characterization provided by POLYUFC is more than classification; it gives the performance+power gaps to hardware peaks, as well as amount of reuse (distance to $\mathcal{B}^t_{DRAM}$) in FpB at LLC that the program achieves.

REFERENCES

[1] A. Abel and J. Reineke, "nanobench: A low-overhead tool for running microbenchmarks on x86 systems," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 34–46.

[2] M. A. Abella-González, P. Carollo-Fernández, L.-N. Pouchet, F. Rastello, and G. Rodríguez, "Polybench/python: benchmarking python environments with polyhedral optimizations," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 59–70. [Online]. Available: https://doi.org/10.1145/3446804.3446842

[3] A. Acharya, U. Bondhugula, and A. Cohen, "Polyhedral auto-transformation with no integer linear programming," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 529–542. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192401

[4] E. André, R. Dulong, A. Guermouche, and F. Trahay, "duf: Dynamic uncore frequency scaling to reduce power consumption," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 3, p. e6580, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6580

[5] J. Ansel *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. Association for Computing Machinery, 2024, p. 929–947. [Online]. Available: https://doi.org/10.1145/3620665.3640366

[6] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: a polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019.

[7] W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan, "Static and dynamic frequency scaling on multicore cpus," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, dec 2016. [Online]. Available: https://doi.org/10.1145/3011017

[8] W. Bao, S. Krishnamoorthy, L.-N. Pouchet, and P. Sadayappan, "Analytical modeling of cache behavior for affine programs," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158120

[9] C. Bastoul, "Openscop: A specification and a library for data exchange in polyhedral compilation tools," Paris-Sud University, France, Tech. Rep., September 2011.

[10] D. P. Bertsekas, "Nonlinear programming," *Journal of the Operational Research Society*, vol. 48, no. 3, pp. 334–334, 1997.

[11] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior." in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, IASTED*, 2001.

[12] K. Beyls and E. H. D'Hollander, "Generating cache hints for improved program efficiency," *Journal of Systems Architecture*, vol. 51, no. 4, pp. 223–250, 2005.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 101–113. [Online]. Available: https://doi.org/10.1145/1375581.1375595

[14] U. Bondhugula, A. Acharya, and A. Cohen, "The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, Apr. 2016. [Online]. Available: https://doi.org/10.1145/2896389

[15] U. Bondhugula *et al.*, "Pluto compiler, version 0.11.4," https://github.com/bondhugula/pluto/commit/8e24aaddf4d2acde638335afe1215b22aa559adb, 2023.

[16] J. A. Butts and G. S. Sohi, "A static power model for architects," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: Association for Computing Machinery, 2000, p. 191–201. [Online]. Available: https://doi.org/10.1145/360128.360148

[17] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, "Exact analysis of the cache behavior of nested loops," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 286–297, 2001.

[18] H.-Y. Cheng, J. Zhan, J. Zhao, Y. Xie, J. Sampson, and M. J. Irwin, "Core vs. uncore: The heart of darkness," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[19] J. W. Choi, "A roofline model of energy ubenchmarks," https://github.com/jeewhanchoi/a-roofline-model-of-energy-ubenchmarks, 2020.

[20] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 661–672.

[21] I. Corporation, "Enhanced power management for low-latency workloads," Intel Corporation, Tech. Rep., 2021, [Online]. [Online]. Available: https://builders.intel.com/docs/networkbuilders/power-management-enhanced-power-management-for-low-latency-workloads-technology-guide-1617438252.pdf

[22] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[23] H. David *et al.*, "Rapl: memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 189–194. [Online]. Available: https://doi.org/10.1145/1840845.1840883

[24] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[25] Edler, Jan, "Dinero IV trace-driven uniprocessor cache simulator," 1994, http://pages.cs.wisc.edu/~markhill/DineroIV/.

[26] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "On characterizing the data movement complexity of computational dags for parallel execution," in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 296–306. [Online]. Available: https://doi.org/10.1145/2612669.2612694

[27] V. Elango, N. Sedaghati, F. Rastello, L.-N. Pouchet, J. Ramanujam, R. Teodorescu, and P. Sadayappan, "On using the roofline model with lower bounds on data movement," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, Jan. 2015. [Online]. Available: https://doi.org/10.1145/2693656

[28] Elsevier, ""advanced configuration and power interface"," ScienceDirect Topics, 2025, accessed: 2025, https://www.sciencedirect.com/topics/computer-science/advanced-configuration-and-power-interface.

[29] K. Fan, M. D'Antonio, L. Carpentieri, B. Cosenza, F. Ficarelli, and D. Cesarini, "Synergy: Fine-grained energy-efficient heterogeneous computing for scalable energy saving," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. ACM, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607055

[30] P. Feautrier, "Some efficient solutions to the affine scheduling problem: I. one-dimensional time," *Int. J. Parallel Program.*, vol. 21, no. 5, p. 313–348, Oct. 1992. [Online]. Available: https://doi.org/10.1007/BF01407835

[31] ——, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *International journal of parallel programming*, vol. 21, pp. 389–420, 1992.

[32] M. J. Fischer and M. O. Rabin, "Super-exponential complexity of presburger arithmetic," in *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 1998, pp. 122–135.

[33] X. Fu, X. Wang, and C. Lefurgy, "How much power oversubscription is safe and allowed in data centers," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 21–30. [Online]. Available: https://doi.org/10.1145/1998582.1998589

[34] N. Gholkar, F. Mueller, and B. Rountree, "Uncore power scavenger: a runtime for uncore power conservation on hpc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356150

[35] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: a compiler framework for analyzing and tuning memory behavior," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, p. 703–746, Jul. 1999. [Online]. Available: https://doi.org/10.1145/325478.325479

[36] A. Guermouche, "Combining uncore frequency and dynamic power capping to improve power savings," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022.

[37] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa, "The forgotten 'uncore': on the energy-efficiency of heterogeneous cores," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 34.

[38] T. Gysi, T. Grosser, L. Brandner, and T. Hoefler, "A fast analytical model of fully associative caches," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 816–829. [Online]. Available: https://doi.org/10.1145/3314221.3314606

[39] C. Haase, "A survival guide to presburger arithmetic," *ACM SIGLOG News*, vol. 5, no. 3, p. 67–82, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3242953.3242964

[40] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the intel haswell processor," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 896–904.

[41] J. Haj-Yahya, M. Alser, J. Kim, A. G. Yağlıkçı, N. Vijaykumar, E. Rotem, and O. Mutlu, "Sysscale: exploiting multi-domain dynamic voltage and frequency scaling for energy efficient mobile processors," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 227–240. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00029

[42] M. Hao, W. Zhang, Y. Wang, G. Lu, F. Wang, and A. V. Vasilakos, "Fine-grained powercap allocation for power-constrained systems based on multi-objective machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, 2021.

[43] W. Heirman *et al.*, "Sniper: Scalable and accurate parallel multi-core simulation," in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network, 2012.

[44] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[45] D. L. Hill, D. Bachand, S. Bilgin, R. Greiner, P. Hammarlund, T. Huff, S. Kulick, and R. Safranek, "The uncore: A modular approach to feeding the high-performance cores." *Intel Technology Journal*, vol. 14, no. 3, 2010.

[46] Intel Corporation, "Intel power management technology overview," Intel Corporation, Tech. Rep. 637748-v2, Feb. 2022, [Online]. [Online]. Available: https://cdrdv2-public.intel.com/637748/Power%20Management%20-%20Technology%20Overview%20TechGuide_637748v2.pdf

[47] H. Jagode, A. Danalis, G. Congiu, D. Barry, A. Castaldo, and J. Dongarra, "Advancements of papi for the exascale generation," *The International Journal of High Performance Computing Applications*, vol. 39, no. 2, pp. 251–268, 2024. [Online]. Available: https://doi.org/10.1177/10943420241303884

[48] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 262–272. [Online]. Available: https://doi.org/10.1145/2581122.2544161

[49] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, mar 2018. [Online]. Available: https://doi.org/10.1145/3177754

[50] S. C. Kleene, "Representation of events in nerve nets and finite automata," *Automata Studies*, p. 3–42, 1956.

[51] K. Koukos, P. Ekemark, G. Zacharopoulos, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Multiversioned decoupled access-execute: the key to energy-efficient compilation of general-purpose programs," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 121–131. [Online]. Available: https://doi.org/10.1145/2892208.2892209

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[53] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[54] X. Li, G. Yan, Y. Han, and X. Li, "Smartcap: User experience-oriented power adaptation for smartphone's application processor," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 57–60.

[55] Linux Kernel development community, "amd_pstate cpu performance scaling driver," https://www.kernel.org/doc/html/v6.8/admin-guide/pm/amd-pstate.html, 2021, [Online].

[56] ——, "intel_pstate cpu performance scaling driver," https://www.kernel.org/doc/html/v6.8/admin-guide/pm/intel_pstate.html [Online], 2017.

[57] ——, "Intel_uncore_frequency cpu uncore frequency scaling driver," https://www.kernel.org/doc/html/v6.8/admin-guide/pm/intel_uncore_frequency_scaling.html [Online], 2023.

[58] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 11 976–11 986.

[59] LLVM, "MLIR Affine Dialect Documentation," https://mlir.llvm.org/docs/Dialects/Affine/, LLVM, 2025, [Online].

[60] ——, "MLIR Linalg Dialect Documentation," https://mlir.llvm.org/docs/Dialects/Linalg/, LLVM, 2025, [Online].

[61] LLVM, "torch-mlir: Pytorch ecosystem compiler support in mlir," https://github.com/llvm/torch-mlir, 2025.

[62] S. Malla *et al.*, "Coordinated priority-aware charging of distributed batteries in oversubscribed data centers," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[63] W. S. Moses *et al.*, "Polygeist compiler," 2025, accessed: Sept 2025. [Online]. Available: https://github.com/llvm/Polygeist/commit/77c04bb2a7a2406ca9480bcc9e729b07d2c8d077

[64] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising c to polyhedral mlir," in *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '21. IEEE Press, 2021, p. 45–59. [Online]. Available: https://doi.org/10.1109/PACT52795.2021.00011

[65] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710. University of Tennessee, 1999.

[66] L. Narmour, T. Yuki, and S. Rajopadhye, "(when) do multiple passes save energy?" in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu, M. Jung, and M. Reichenbach, Eds. Cham: Springer International Publishing, 2022, pp. 451–466.

[67] D. Nguyen Luu, "The computational complexity of presburger arithmetic," Ph.D. dissertation, University of California, Los Angeles, 2018. [Online]. Available: https://www.proquest.com/dissertations-theses/computational-complexity-presburger-arithmetic/docview/2061552413/se-2

[68] NVIDIA, "nvml," https://developer.nvidia.com/management-library-nvml, 2025, [Online].

[69] A. Olivry, J. Langou, L.-N. Pouchet, P. Sadayappan, and F. Rastello, "Automated derivation of parametric data movement lower bounds for affine programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: ACM, 2020, p. 808–822.

[70] A. Olivry, G. Iooss, N. Tollenaere, A. Rountev, P. Sadayappan, and F. Rastello, "Ioopt: automatic derivation of i/o complexity bounds for affine programs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. USA: ACM, 2021, p. 1187–1202. [Online]. Available: https://doi.org/10.1145/3453483.3454103

[71] OpenMP, "Openmp api specification 6.0," https://www.openmp.org, 2025, accessed: 2025-09-11.

[72] K. O'Leary, I. Gazizov, A. Shinsel, R. Belenov, Z. Matveev, and D. Petunin, "Intel advisor roofline analysis," *THE CHANGING HPC LANDSCAPE STILL LOOKS THE SAME*, p. 56, 2017.

[73] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim, "Synergistic cpu-gpu frequency capping for energy-efficient mobile games," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 2, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3145337

[74] P. Patel, E. Choukse *et al.*, "Characterizing power management opportunities for llms in the cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming*

Languages and Operating Systems, Volume 3, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 207–222. [Online]. Available: https://doi.org/10.1145/3620666.3651329

[75] L. Piga, I. Narayanan, A. Sundarrajan *et al.*, "Expanding datacenter capacity with dvfs boosting: A safe and scalable deployment experience," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 150–165. [Online]. Available: https://doi.org/10.1145/3617232.3624853

[76] A. Pitchanathan, K. Grover, and T. Grosser, "Falcon: A scalable analytical cache model," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656452

[77] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache, "Iterative optimization in the polyhedral model: Part i, one-dimensional time," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 144–156. [Online]. Available: https://doi.org/10.1109/CGO.2007.21

[78] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," *SIGPLAN Not.*, vol. 43, no. 6, pp. 90–100, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375594

[79] L.-N. Pouchet *et al.*, "Polybench benchmarks," http://sourceforge.net/projects/polybench/, 2025.

[80] L.-N. Pouchet, "Pocc version 1.6.0-alpha," http://pocc.sf.net, 2022, accessed 27-Aug-2025.

[81] H. Qiu, L. Zhang, C. W. H. Franke, Z. T. Kalbarczyk, and R. K. Iyer, "Parm: Adaptive resource allocation for datacenter power capping," in *Machine Learning for Systems Workshop at the Annual Conference on Neural Information Processing Systems (NeurIPS 2023)*, 2023.

[82] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[83] S. Ramesh, S. Perarnau, S. Bhalachandra, A. D. Malony, and P. Beckman, "Understanding the impact of dynamic power capping on application progress," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 793–804.

[84] V. Sakalkar, V. Kontorinis, D. Landhuis, S. Li *et al.*, "Data center power oversubscription with a medium voltage power plane and priority-aware capping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 497–511. [Online]. Available: https://doi.org/10.1145/3373376.3378533

[85] N. R. Shah, A. Misra, A. Miné, R. Venkat, and R. Upadrasta, "Bullseye: Scalable and accurate approximation framework for cache miss calculation," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 1, Nov. 2022. [Online]. Available: https://doi.org/10.1145/3558003

[86] M. Sourouri, E. B. Raknes, N. Reissmann, J. Langguth, D. Hackenberg, R. Schöne, and P. G. Kjeldsberg, "Towards fine-grained dynamic tuning of hpc applications on modern multi-core architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. ACM, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126945

[87] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green governors: A framework for continuously adaptive dvfs," in *2011 International Green Computing Conference and Workshops*, 2011, pp. 1–8.

[88] B. Subramaniam and W.-c. Feng, "Towards energy-proportional computing for enterprise-class server workloads," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13. Association for Computing Machinery, 2013, p. 15–26. [Online]. Available: https://doi.org/10.1145/2479871.2479878

[89] V. Sundriyal, M. Sosonkina, B. Westheimer, and M. Gordon, "Core and uncore joint frequency scaling strategy," *Journal of Computer and Communications*, vol. 06, pp. 184–201, 01 2018.

[90] G. Team *et al.*, "Gemma 2: Improving open language models at a practical size," *arXiv preprint arXiv:2408.00118*, 2024.

[91] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[92] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv:2307.09288*, 2023.

[93] K.-A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: look-ahead compile-time scheduling," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. IEEE Press, 2017.

[94] K.-A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras, "Swoop: software-hardware co-design for non-speculative, execute-ahead, in-order cores," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 328–343. [Online]. Available: https://doi.org/10.1145/3192366.3192393

[95] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018. [Online]. Available: https://arxiv.org/abs/1802.04730

[96] S. Verdoolaege, "Isl: An integer set library for the polyhedral model," in *Proceedings of the Third International Congress Conference on Mathematical Software*, ser. ICMS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 299–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=1888390.1888455

[97] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, vol. 141, 2012.

[98] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Counting integer points in parametric polytopes using barvinok's rational functions," *Algorithmica*, vol. 48, no. 1, pp. 37–66, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1007/s00453-006-1231-0

[99] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, Jan. 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400713

[100] Z. Wang, Y. Zhang, F. Wei, B. Wang, Y. Liu, Z. Hu, J. Zhang, X. Xu, J. He, X. Wang, W. Dou, G. Chen, and C. Tian, "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*, ser. ASPLOS '25. ACM, 2025, p. 1118–1132. [Online]. Available: https://doi.org/10.1145/3669940.3707231

[101] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, apr 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[102] Q. Wu, V. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D. Clark, "A dynamic compilation framework for controlling microprocessor energy and performance," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005.

[103] F. Xie, M. Martonosi, and S. Malik, "Compile-time dynamic voltage scaling settings: opportunities and limits," *SIGPLAN Not.*, vol. 38, no. 5, p. 49–62, may 2003. [Online]. Available: https://doi.org/10.1145/780822.781138

[104] T. Yuki and S. Rajopadhye, "Folklore confirmed: Compiling for speed = compiling for energy," in *Languages and Compilers for Parallel Computing*, C. Caşcaval and P. Montesinos, Eds. Cham: Springer International Publishing, 2014, pp. 169–184.

[105] S. Zagoruyko and N. Komodakis, "Wide residual networks," *CoRR*, vol. abs/1605.07146, 2016. [Online]. Available: http://arxiv.org/abs/1605.07146

[106] C. Zhang *et al.*, "Flex: high-availability datacenters with zero reserved power," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 319–332. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00033

[107] H. Zhang, A. Nukada, and Q. Liao, "Fcufs: Core-level frequency tuning for energy optimization on intel processors," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, 2024.

[108] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," *SIGPLAN Not.*, vol. 51, no. 4, p. 545–559, mar 2016. [Online]. Available: https://doi.org/10.1145/2954679.2872375