RESEARCH-ARTICLE

Energy-aware performance portability with OpenMP dynamic variants

AYMAN BOURRAMOUSS EL MAACH, Barcelona Supercomputing Center, Barcelona, Barcelona, Spain

ADRIAN MUNERA, Barcelona Supercomputing Center, Barcelona, Barcelona, Spain

SARA ROYUELA, Barcelona Supercomputing Center, Barcelona, Barcelona, Spain

Open Access Support provided by:

Barcelona Supercomputing Center

Published: 16 November 2025

Citation in BibTeX format

SC Workshops '25: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis
November 16 - 21, 2025
MO, St Louis, USA

Conference Sponsors:
SIGHPC

Energy-aware performance portability with OpenMP dynamic variants

Ayman Bourramouss el Maach
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
ayman.bourramouss@bsc.es

Adrian Munera
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
adrian.munera@bsc.es

Sara Royuela
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
sara.royuela@bsc.es

Abstract

Performance portability in HPC and embedded systems is often limited by power and thermal constraints. The OpenMP programming model offers a compile-time mechanism known as variants, allowing different function specializations. Previous research extended this concept to the runtime level, enabling dynamic variant selection. We build on these foundations with an energy-aware runtime that augments variant selection with low-overhead power and temperature instrumentation and a multi-criteria policy balancing power caps, thermal headroom, and performance. Implemented in LLVM and publicly available, our mechanism profiles per-variant energy and thermal behavior, selecting specializations at runtime based on user-defined thresholds and live system state. Validation on HPC and embedded platforms shows the runtime enforces dynamic power caps with 98% compliance on a workstation (versus 67% unconstrained). On thermally constrained edge devices, proactive CPU/GPU migration beats hardware throttling, cutting execution time by 39% while maintaining stability. In a simulated battery-limited mission, energy-aware selection extends battery lifetime.

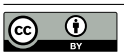
CCS Concepts

• **Computing methodologies** → **Parallel programming languages**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → **Power management**; **Runtime environments**; *Concurrent programming languages*; *Software performance.*

Keywords

OpenMP, performance portability, energy-aware runtime, thermal constraints, function variants

ACM Reference Format:
Ayman Bourramouss el Maach, Adrian Munera, and Sara Royuela. 2025. Energy-aware performance portability with OpenMP dynamic variants. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3731599.3767495



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767495

1 Introduction

The relentless pursuit of exascale computing has exacerbated two critical challenges in modern High-Performance Computing (HPC): extreme hardware heterogeneity and escalating energy constraints. While architectures diversify to include CPUs, GPUs, and specialized accelerators, power consumption and thermal management have emerged as fundamental limits to sustained performance. Therefore, achieving true performance portability demands adaptability not only to computational resources but also to dynamic power and thermal boundaries.

The convergence of HPC and embedded systems has further intensified the need for performance portability under extreme physical constraints. Modern High-Performance Embedded Computing (HPEC) systems utilize parallel and heterogeneous architectures in thermally restricted and power-constrained environments, such as those found in vehicles and edge servers. In these scenarios, energy scarcity and thermal limits supersede raw performance as critical bottlenecks, since exceeding power budgets risks system failure, while thermal accumulation forces throttling and the subsequent performance loss or hardware damage.

OpenMP is a pivotal solution for performance portability, offering a high-level API that enables the seamless exploitation of parallelism across various architectures. It provides a wide variety of mechanisms to exploit data (e.g., `for`) and task-parallelism (e.g., `task`) in CPUs, as well as offloading (e.g., `target`) and parallelism (e.g., `teams distribute`) features for GPUs. Although easy to use, the OpenMP specification encompasses a wide range of options for tuning the application's execution.

In addition, OpenMP specification offers the `declare variant` and `metadirective` mechanisms to enable compile-time user conditioned selection of specialized code paths (functions or directives) tailored to specific hardware traits or application semantics. Although this mechanism offers adaptability to different static system conditions, it lacks responsiveness to real-time system states, such as device workload imbalance or thermal conditions, which limits their adaptability under evolving operational conditions and hardware changes.

A recent work [10] introduced runtime adaptability to system workload and memory states, exploiting OpenMP variants. However, this method ignores the physical realities of embedded HPC: (i) battery or supply limits prohibit energy-inefficient variants, and (ii) overheating risks due to passive cooling, and confined form factors require temperature-aware scheduling.

To close the existing gap between the need for performance portability and adaptability and the capabilities offered by current

frameworks in terms of flexibility, simplicity, and heterogeneity, this work introduces the following contributions:

(1) A multi-criteria optimization OpenMP runtime mechanism that integrates novel energy and thermal-aware heuristics with existing speed and resource utilization for optimal variant selection.
(2) A low-overhead instrumentation mechanism based on a modular sensor-adapter interface, enabling portable and accurate energy/thermal profiling across heterogeneous hardware, from embedded SoCs with integrated sensors to HPC nodes with external power meters.
(3) Validation in HPC and HPEC scenarios, ensuring the system adheres to system-wide or per-device power caps, preventing thermal throttling by avoiding hotspots, and showcasing the exploitation of energy-performance trade-offs.

2 OpenMP function variants

Function variants are alternate implementations of a given function, each optimized for different conditions such as hardware characteristics, workload patterns, or system constraints. The primary objective of this feature is to enhance performance portability and efficiency adaptability across heterogeneous and dynamic environments. This capability is particularly important in modern systems, where hardware diversity and dynamic resource contention make static optimizations brittle and suboptimal.

OpenMP included support for function variants in its version 5.0 [3]. The `declare variant` directive includes clauses to define the *traits* that must occur for a variant implementation to be selected. These traits include the directive names in the calling context (i.e., `construct` selector), the characteristics of the targeted device (e.g., `kind` and `isa` selectors), implementation-defined selectors, and user-defined conditions. In its current definition, traits are evaluated at compile time, when static conditions decide which version is executed for each call to a function that includes variants.

Figure 1 shows an OpenMP example using the variants feature for implementing a vector product. The code defines two variants for the sequential version of the *vxv* vector product operation: *p_vxv*, which uses the `for` worksharing to implement parallelism in the host, and *t_vxv*, which uses `simd` extensions and the `distribute` construct to implement parallelism in the GPU device. The traits used rely on the *construct* used in the calling context. In the *main* function, each call to *vxv* results in calling a different implementation.

The OpenMP variants feature enables (i) *performance portability*, as it allows for the automatic dispatching of the best-suited code in different scenarios; (ii) *maintainability*, as it keeps a single source code while isolating platform-specific optimizations and (iii) *extensibility*, as new variants for emerging architectures can be added incrementally without modifying existing logic.

Nonetheless, this mechanism lacks responsiveness to dynamic physical constraints, as traits are evaluated at compile time and fixed during execution. This behavior prevents variants from adapting to fluctuating conditions within the system, such as power budgets or temperature variations, a characteristic relevant in traditional HPC systems for environmental reasons, and especially critical in embedded and edge environments, due to their restricted resources.

```
1  void p_vxv(int *v1,int *v2,int *v3,int n) {
2    #pragma omp for
3    for (int i= 0; i< n; i++) v3[i] = v1[i] * v2[i]*3;
4  }
5
6  void t_vxv(int *v1,int *v2,int *v3,int n) {
7    #pragma omp distribute simd
8    for (int i= 0; i< n; i++) v3[i] = v1[i] * v2[i]*2;
9  }
10
11 #pragma omp declare variant(p_vxv) \
12                         match(construct={parallel})
13 #pragma omp declare variant(t_vxv) \
14                         match(construct={target})
15 void vxv(int *v1,int *v2,int *v3,int n) {
16   for (int i= 0; i< n; i++) v3[i] = v1[i] * v2[i];
17 }
18
19 int main() {
20   int v1[N], v2[N], v3[N];
21   for(int i=0; i<N; i++) {
22     v1[i]=(i+1); v2[i]=-(i+1); v3[i]=0;
23   }
24
25   #pragma omp parallel
26   vxv(v1,v2,v3,N); // Call to p_vxv
27
28   #pragma omp target teams map(to: v1[:N], v2[:N]) \
29                            map(from: v3[:N])
30   vxv(v1,v2,v3,N); // Call to t_vxv
31
32   vxv(v1,v2,v3,N); // Call to vxv
33
34   return 0;
35 }
```

Figure 1: OpenMP variants example with a vector multiplication (from OpenMP 6.0 examples [4]).

3 Related Work

Several attempts exist to include energy and thermal considerations in OpenMP (or similar) parallel programs. This section highlights the most relevant works, stating their benefits and limitations, and exposes a gap between the needs of modern systems and the capabilities of the proposed techniques, thereby motivating the work presented in this paper.

In 2013, Planas et al. [11] proposed a framework built on top of the OmpSs[1] programming model that includes syntax support for defining task specializations and runtime support for gathering performance metrics of each variant and then selecting among the different options. This work was an early precursor to OpenMP variants. It is, however, limited to performance metrics and does not consider multi-criteria optimizations, energy caps, and thermal issues.

The same year, Porterfield et al. [12] utilized hardware performance counters to measure energy and power consumption in OpenMP programs, enabling concurrency throttling, i.e., dynamically varying the number of active threads. However, this work is limited to Intel processors, as it relies on Intel's Running Average Power Limit (RAPL) interface. Furthermore, it can only react to system conditions by varying the amount of parallelism, disregarding options like available hardware resources and function variants that can reduce bottlenecks and enhance load balance.

[1] https://pm.bsc.es/ompss

In 2015, Alessi et al. [1] proposed OpenMPE, an extension to OpenMP with directives for per-region power hints to express energy vs. performance trade-offs at the application level. Their prototype framework, comprising compiler and runtime support, achieved 15% energy savings across nine benchmarks with no loss in quality of service (QoS). This work, however, (1) employs complex annotations that are highly coupled with changes to the application's code, negatively impacting usability, maintainability and (2) lack of portability across heterogeneous systems, as the framework is restricted to OpenMP-based CPU parallelism and cannot exploit other accelerators.

The same year, Shafik et al. [13] proposed a scalable controller that applies DVFS and dynamic core allocations in OpenMP programs. This work employs an iterative learning algorithm that utilizes performance counters to adjust hardware resources, achieving 17% energy savings on an Intel Xeon using NAS benchmarks. However, this work only operates at the hardware level, without consideration of algorithmic variants, power caps, and thermal impacts. Furthermore, it focuses on CPUs, making it inadequate in modern heterogeneous MPSoCs and larger HPC systems.

In 2017, Meyer et al. [7] presented a prototype of an energy-aware scheduling plugin for the OmpSs runtime, implementing an intelligent agent that monitors task workload to apply DVFS, balancing energy consumption against application performance. This work can reduce dynamic power by up to 20% in an 8-core Intel Xeon machine. However, it focuses only on CPUs and relies solely on DVFS, ignoring other aspects such as heterogeneous platforms and resource utilization.

In 2019, Mirka et al. [8] introduced an online profiler for OpenMP workloads that computes Chunks per Second (CpS) as a performance indicator and Chunks per Joule (CpJ) as an energy-efficiency indicator. It then employs an auto-encoder to detect phase changes at runtime, enabling energy-efficiency analysis without code annotations. This work uses an analytical approach that enables offline optimizations based on application phases but does not facilitate automatic execution adjustments in response to dynamic system conditions.

In 2020, Mirka et al. [9] extended their previous analytical work to incorporate a concurrency throttling mechanism. Although adaptable in terms of energy consumption and performance, this proposal lacks awareness of other key metrics, such as resource utilization and thermal constraints, as well as application semantics and hardware-specific adjustments, thereby failing to adapt to complex, heterogeneous systems.

In 2024, Munera et al. [10] introduced a runtime mechanism for selecting OpenMP function variants based on system metrics, including CPU and GPU load, as well as memory usage. This work improves CPU utilization by up to 4x and avoids memory crashes in various HPC kernels. However, this work omits energy and thermal metrics, preventing optimization for power caps or thermal throttling.

Overall, existing energy-aware OpenMP techniques either (a) focus on standard system-level controls, like DVFS and power throttling, that can only act without knowledge of an application's semantics, often leading to suboptimal tradeoffs, or (b) rely on static hints or compile-time choices that lack real-time responsiveness, or (c) prioritize energy aspects over other relevant system metrics. Furthermore, implementations are either obsolete or not public, making reproducibility impossible. This scenario leaves a gap for an open-source, lightweight runtime that unifies performance, resource utilization, energy, and thermal management within a holistic, multi-criteria, dynamic framework.

4 Energy and thermal-aware OpenMP runtime

To address the limitations of existing tools, we propose a new energy- and thermal-aware OpenMP framework that combines system-level instrumentation with dynamic variant selection[2]. The goal of this framework is to enhance adaptability in managing energy and temperature more effectively. By accounting for dynamic system conditions, it aims to reduce total energy usage, limit peak power consumption, prevent thermal throttling, and extend the life of nodes and batteries.

Our implementation in the LLVM 20.0.0 compilation framework builds upon the work of Munera et al. [10], which extends the OpenMP 5.0 declare variant mechanism by providing a dynamic variant framework that enables runtime selection based on system metrics and user-defined thresholds, rather than the standard mechanism that selects variants statically at compile-time. This paper further enhances the dynamic variant framework by incorporating energy and thermal awareness, enabling the runtime to choose the most suitable variant based on (i) user-specified energy and thermal constraints, (ii) variant statistics, and (iii) current system conditions.

The energy and thermal-aware runtime operates in two phases, each corresponding to a separate binary execution: (1) an *instrumentation and profiling phase*, during which the power and temperature characteristics of different function specializations defined through OpenMP variants are collected; and (2) a *variant selection phase*, in which the OpenMP runtime selects the most suitable specialization based on current system conditions, user-defined thresholds, and the previously gathered profiling data.

To instruct the OpenMP framework to proceed with the different stages, the variant compilation supports two modes: (1) the `-dynamic-variant-warmup` Clang flag directs the compiler to insert runtime hooks through an LLVM pass. These hooks enable the runtime to execute all functions marked with the dynamic variants extension and collect runtime data to save a profile for future selection (2) the `-dynamic-variant` flag directs the compiler to generate the final binary that performs variant selection based on the stored profile, user thresholds and live system state.

To specify dynamic variants, we propose the following OpenMP extension syntax:

```
#pragma omp declare variant(variant_name)
    match(implementation=extension(dynamic))
```

where `variant_name` in the `variant` clause is the name of the variant implementation (as in standard variants), and the `implementation` selector in the `match` clause includes the `extension` trait with the new value `dynamic`, marking the variant for runtime selection.

[2]The framework is publicly available at https://gitlab.bsc.es/ppc-bsc/research/p3hpc_artifacts where we have provided a README with instructions to install the compiler on a Jetson Orin

4.1 Profiling phase

The dynamic variant selection mechanism relies on accurate performance and energy profiles of each function variant. To obtain these metrics, we extend the dynamic variants to add a comprehensive profiling phase that characterizes the behavior of each variant under real execution conditions. Figure 2 illustrates the complete workflow for processing user source code containing functions with dynamic variant annotations, resulting in profiled variant data. This process involves three key stages: (1) compilation introducing instrumentation hooks for annotated function variants using an LLVM transformation pass, (2) runtime monitoring during execution, and (3) data persistence for variant selection use.

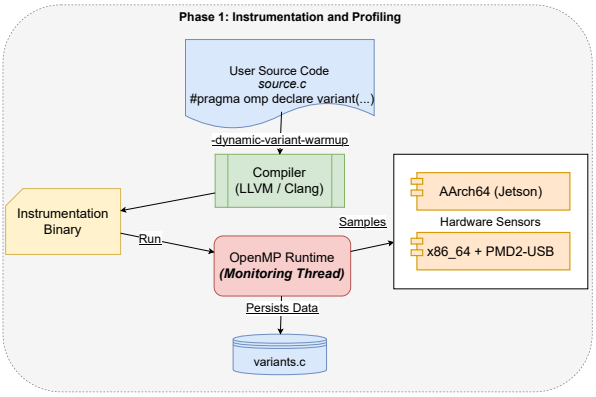


Figure 2: Overview of the profiling phase workflow.

The profiling phase begins when the user compiles the application with the `-dynamic-variant-warmup` flag. This flag instructs the LLVM pass to scan for all function calls annotated with a `declare variant` with the `extension(dynamic)`[3] trait and then insert instrumentation hooks wrapping the code of each function variant. These hooks will signal the runtime to begin and end profiling for each variant execution. When the binary is dispatched for execution, the runtime generates a profile by collecting relevant metrics for each variant. The runtime ensures that all variants are executed at least once by transparently re-executing the application's main entry point and selecting a different variant in round-robin fashion until profiling has been completed for all of them.

During the execution of the instrumented binary, the OpenMP runtime spawns a dedicated *monitoring thread* that runs concurrently with the application. This monitoring thread serves as a central data collection mechanism, continuously sampling hardware sensors throughout the execution of the variant.

The monitoring thread abstracts away device-specific details behind a pluggable *sensor-adapter* interface (Fig. 2). We ship two ready-made adapters: one for the on-board power/thermal monitors of an **NVIDIA Jetson AGX Orin** and another for an external **PMD2** power meter on an x86 host. Because the LLVM pass itself is ISA-agnostic, porting to new measurement hardware only requires writing an adapter that implements the same OpenMP sampling API.

During execution the monitoring thread:

(a) samples the sensors every `SAMPLE_RATE` ms (override with `SAMPLE_RATE` environment variable, default 100 ms);
(b) aggregates results across repeated calls or runs, producing one representative profile per variant; and
(c) optionally explores different thread counts when `OMP_VARIANTS_NUM_THREADS` is set (e.g. `1,2,4`), storing a separate profile for each configuration.

Table 1 summarizes the power, thermal, and resource metrics collected during instrumentation. The profiler also records execution time, CPU load, and heap/stack usage, giving us a comprehensive basis for multi-objective optimization. After profiling, the monitoring thread packs each variant's metrics into a compact record, stores the records in an array, and emits them to the generated source file `variants.c`.

During the variant selection phase, the profiling file generated from the warm up phase is compiled and linked against the variant selection binary, in order to load the profiles in-memory, the variant selection binary calls a method that returns the profiling data to optimize system performance and power efficiency, as well as to provide performance portability through adaptability to hardware specifications and system conditions.

Table 1: Metrics collected per variant during instrumentation.

Category	Metric	Unit
Power rail	Peak power	W
	Energy consumed	J
	Energy-delay product	J·s
Temperature	CPU temperature	°C
	GPU temperature	°C
	BOARD temperature	°C
Performance	Execution time	ms
	CPU usage	%
	GPU usage	%
Memory	Peak heap	MB
	Thread stack	MB

4.2 Data acquisition on heterogeneous platforms

The compiler instrumentation layer described in Section 4.1 is completely platform-independent; only the *data-acquisition back-end* changes from one machine to the next. Each back-end maps the generic OpenMP sampling API to the vendor's native power- and temperature-monitoring interface, so porting to new hardware means writing a thin adapter rather than touching the LLVM pass or runtime logic.

We currently provide two reference back-ends, one for a conventional CPU+GPU HPC node and one for an embedded HPEC SoC.

[3]We acknowledge that "dynamic" is an existing keyword in OpenMP for loop scheduling. Should this work proceed towards standardization, a different keyword would be proposed to avoid ambiguity.

4.2.1 HPC platform – x86_64 CPU + NVIDIA GPU. For power measurements on the x86_64 platform, we employed the ElmorLabs PMD2[4][5] external power measurement device. This approach was essential because the NVIDIA GT 1030 GPU used in this work lacks the on-board circuitry for real-time power reporting through standard software interfaces like nvidia-smi. As a low-power GPU, it draws all its power directly from the PCIe slot, making external measurement the only viable method for obtaining accurate power data. To capture this, we used a PMD PCI-E Slot Power Measurement Adapter[6], which isolates and measures the power delivered through the slot. The specific channels monitored in this setup are detailed in Table 2.

We acknowledge that the GT 1030 is an older, low-power GPU. It was the hardware available to us. Our goal here is to validate the runtime and the measurement pipeline.

The direct hardware measurement approach aligns with the best practices for achieving the highest measurement fidelity for the evaluation. As highlighted by recent research from Yang et al. [17], even higher-end GPUs (A100/H100) that support software monitoring can exhibit significant sampling gaps in nvidia-smi, potentially impacting the accuracy of energy measurement. Our use of an external meter provides a ground-truth measurement that avoids such issues entirely.

It is important to note that this external hardware configuration was used for higher accuracy, but our framework is designed to be fully functional and effective using standard, widely available software tools as its default methods, that includes Intel's Running Average Power Limit (RAPL) for CPU power, nvidia-smi for nvidia or rocm-smi for amd GPUs, ensuring broad applicability.

On the other hand, in the HPC platform we did not instrument thermal sensors: while the device exposes temperature channels and policies (e.g., via `lm-sensors`/`nvidia-smi`) were left disabled in this build because temperature was not part of the HPC experiments.

Table 2: PMD2 power channels used in the x86_64 platform.

Channel	Description
ATX24_12V	12V rail from ATX24 connector
ATX24_5V	5V rail from ATX24 connector
ATX24_5VSB	5V standby rail
ATX24_3V3	3.3V rail from ATX24 connector
EPS1	CPU power rail 1
EPS2	CPU power rail 2
PCIE3	GPU power via PCIe adapter

4.2.2 HPEC platform – Jetson AGX Orin. As a representative HPC embedded platform, we use the NVIDIA Jetson AGX Orin 32GB development kit. This platform provides two INA3221 current-sensing integrated circuits (ICs). INA3221 is a triple-channel shunt and bus voltage monitor IC that can simultaneously measure the voltage and current of up to three different power channels, or rails, by sensing the voltage across a shunt resistor. This IC features direct kernel driver support for saving measurements in the *sysfs* virtual file system via the Inter-Integrated Circuit (I2C) communication bus, enabling the SoC to read power measurements in real-time. Our implementation polls the power rails specified in Table 3.

The temperature in the Jetson Orin is managed through the Linux kernel for on-board thermal sensors and the board and management processor (BPMP) for on-chip sensors. The temperature readings are gathered from multiple thermal zones (i.e., Areas equipped with their own temperature sensors and cooling devices) to monitor the system's thermal state, which is crucial for managing heat dissipation and ensuring safe operation. Table 4 summarizes the sensors used for this work.

4.3 Hardware sampling limits

Unless otherwise stated, we use a software polling period of `SAMPLE_RATE`= 100 ms to keep host overhead low and make results comparable across platforms. This is a configurable *software* default, not a hardware limit. On the workstation, the external PMD2 power meter supports up to ≈ 5 kHz sampling (0.2 ms samples) [17], and we downsample raw readings to the chosen `SAMPLE_RATE` (mean power per window; energy by trapezoidal integration). On Jetson AGX Orin, the on-board INA3221 rails (via `/sys/class/hwmon`) expose an `update_interval` of ≈ 1 ms on our boards, so polling faster does not yield new values; we therefore keep 100 ms by default for portability and lower it (e.g., to 1–10 ms) when finer timing is needed.

4.4 Variant selection phase

After warm-up, the compiler has produced a profile record for every function variant, capturing power, energy, temperature, and performance metrics for each rail or sensor (Tables 3–4) for each variant. Re-compiling the application with `-dynamic-variant` links this record into the final binary. At variant selection the OpenMP runtime loads the profile, samples the current power/thermal state through the data-acquisition back-end (Section 4.2), and verifies both against any user-supplied thresholds and current system state (Section 4.4.1). The runtime variant selection process then chooses, at each annotated call site, the variant that best satisfies these constraints. Figure 3 gives an overview of this build-time and run-time workflow.

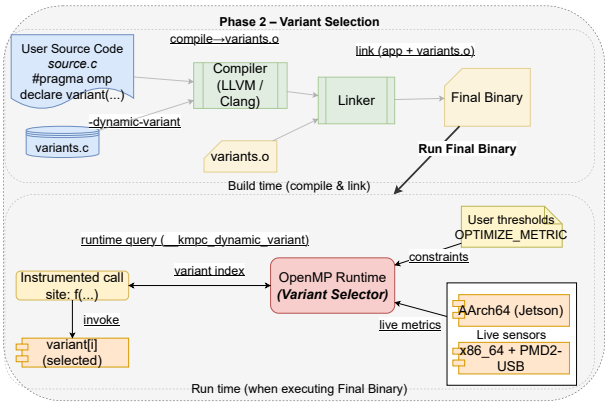


Figure 3: Overview of the variant selection workflow.

[4]https://elmorlabs.com/product/elmorlabs-pmd2/
[5]https://github.com/ElmorLabs/PMD2-Python
[6]https://elmorlabs.com/product/pmd-pci-e-slot-power-measurement-adapter/

Table 3: I2C channels used for power measurement in the Jetson AGX Orin series.

I2C addr.	Channel num.	Channel name	Description
0x40	1 (Physical)	VDD_GPU_SOC	Total power consumed by GPU and SoC core, which supplies the memory subsystem and various engines.
0x40	2 (Physical)	VDD_CPU_CV	Total power consumed by CPU and computer vision (CV) cores, i.e., the deep learning accelerator (DLA) and the programmable vision accelerator (PVA).
0x40	3 (Physical)	VIN_SYS_5V0	Power consumed by the system 5V rail, which supplies various IOs (e.g., HDMI, USB, DDR).
0x40	4 (Logical)	SUMMATION	Sum of VDD_GPU_SOC and VDD_CPU_CV.
0x41	2 (Physical)	VDDQ_VDD2_1V8AO	Power consumed by DDR core, DDR IO and 1V8AO.

Table 4: Thermal sensors used for temperature measurement in the Jetson AGX Orin series.

Sensor	Zone name	Description
TEGRA234_THERMAL_ZONE_CPU	CPU-therm (thermal_zone0)	CPU temperature.
TEGRA234_THERMAL_ZONE_GPU	GPU-therm (thermal_zone1)	GPU temperature.
TMP451	Tboard_tegra (thermal_zone9)	Board temperature (overall system).

The runtime variant selection is driven by:

4.4.1 User-Defined Optimization Policies. Users can run their application, guiding the selection of function specializations based on system resource constraints. To that end, the dynamic variants framework supports the definition of thresholds for limiting:

- **Peak Power (W)**: The highest power consumption recorded at any single moment during the variant execution.
- **Energy Consumption (J)**: discrete-time sum of power samples.
- **Energy-Delay Product (EDP, J·s)**: energy-performance tradeoff.
- **Peak Temperature (°C)**: thermal threshold, CPU, GPU, and board.

Power, energy and energy-delay product can be applied over *any measurement channel* enumerated in Tables 3 and 2 whereas temperature thresholds can be applied over Table 4 sensors.

Thresholds are set via `OPTIMIZE_METRIC=metric:threshold:weight` [10], where `metric` is a runtime-recognized metric–channel (e.g., `POWER_PEAK_CPU`, `PEAK_GPU_TEMPERATURE`), `threshold` is a hard cap, and `weight` optionally sets relative importance for multi-criteria optimization.

Following this syntax, the metric environment variable for a dynamic-variants runtime execution can be set as:

```
OPTIMIZE_METRIC=
    "POWER_PEAK_GPU:100:1,EDP_SUMMATION:200:2"
```

This configuration will cause the variant selection algorithm to first filter out variants and select only the ones that violate the smallest possible number of thresholds, i.e., `POWER_PEAK_GPU>100W` and `EDP_SUMMATION>200J·s`. If multiple variants remain without any threshold violations, the dynamic variant mechanism will always default to the fastest variant in terms of execution time. If instead there are multiple variants that violate the same nonzero minimum number of thresholds, each variant is scored as described in the following Section 4.4.2 to select the most appropriate one. The weights give more importance to a certain optimization metric, in this instance, `EDP_SUMMATION` has a weight of 2, which gives it twice the importance in the scoring.

4.4.2 Hierarchical variant selection algorithm. Thresholds act as hard constraints that gate the next stage of selection. The dynamic variant scheduler filters variants that violate thresholds and applies a heuristic to select the most suitable one among those that violate the minimum amount of thresholds. The decision logic for those consists in a scoring-based mechanism where user-defined weights influence the final ranking.

The selection algorithm uses a hierarchical, multi-tier process to ensure threshold constraints are respected before attempting any optimization. Selection occurs at each invocation of a function with variants marked with the extension `dynamic` in the `match` clause.

Tier 1: Threshold filtering. For each variant and each user-defined metric triplet, the runtime checks whether the variant violates the corresponding threshold. The number of threshold violations is stored separately for each variant. The runtime prefers variants with the fewest violations.

Notation: Throughout the threshold checking process, we use the following notation:

- I denotes instantaneous metric values (power or temperature measurements at a specific point in time).
- T represents user-defined thresholds for each metric.
- M denotes cumulative metrics computed over the entire execution.
- $P[k]$ represents power samples at discrete time step k.
- Δt is the sampling interval.
- Subscripts indicate the measurement context: *peak* (maximum during profiling), *idle* (baseline at rest), *live* (current real-time measurement), and *profile* (value obtained during warm-up profiling).

For **instantaneous metrics**, such as power or temperature on a specific channel, the profiled values are used to determine violations:

- **Instantaneous metrics** (Power, Temperature).
  During the *warm-up phase*, the runtime stores the profiled reference values I_{peak} and I_{idle}. During the actual variant invocation, it measures the current instantaneous value I_{live} from the system. A violation occurs if:

$$\left(I_{\text{peak}} - I_{\text{idle}}\right) + I_{\text{live}} > T,\ I \in \{\text{Power},\ \text{Temperature}\}. \quad (1)$$

- **Cumulative metrics** (Energy, EDP).
  The warmup phase computes discrete-time energy from each power trace ($\sum_k P[k]\,\Delta t$) and stores the resulting Energy and EDP per variant. At runtime, the check reduces to:

$$M_{\text{profile}} > T, \qquad M \in \{\text{Energy},\ \text{EDP}\}. \quad (2)$$

Tier 2: Score-based ranking. When multiple variants have the same (minimal) nonzero number of violations, they are ranked using a conservative, weight-based scoring system. Each such variant is assigned a normalized score for each metric, scaled to the range $[0, 1]$, where higher is better. For metrics where lower values are preferable (e.g., power or energy), the normalized score of metric m is calculated using min-max scaling, assuming a minimum of zero:

$$normalized\_score_m = 1 - \frac{value_m}{max\_value_m} \quad (3)$$

Where the *max_value* is the maximum metric value across profiled variants. To prioritize specific metrics, the user may assign higher weights, w. The final score for a variant v is computed as the weighted sum of the normalized scores of its metrics:

$$final\_score_v = \sum_m w_m \cdot normalized\_score_m \quad (4)$$

Note that the purpose of this ranking is to choose the variant with the smallest threshold excess (i.e. the variant whose metric values are nearest to the limits), because lower raw values yield higher normalized scores.

Tier 3: Final selection. The variant selection system chooses, from all profiled variants, the one with the fewest threshold violations. Among those, it selects either the variant with the highest weighted aggregate score to reduce threshold excess or the one with fastest execution time in case no thresholds are violated. This hierarchical selection process ensures threshold compliance while still allowing user-controlled optimization through scoring preferences.

4.5 Example: From instrumentation to selection

To utilize the dynamic variants framework presented above, users must tag their functions with the proposed `declare variant` OpenMP pragmas with the `dynamic` extension. Figure 4 shows a portable AXPY operation. This is a classic operation in numerical linear algebra implementing $y = ax + y$, where a is a scalar and x, y are vectors.

```
// CPU implementation
void axpy_cpu(float* x, float* y, float a, int n) {
  #pragma omp parallel for
  for (int i = 0; i < n; i++) y[i] = a * x[i] + y[i];
}
// GPU implementation
void axpy_gpu(float* x, float* y, float a, int n) {
  #pragma omp target teams distribute parallel for
  for (int i = 0; i < n; i++) y[i] = a * x[i] + y[i];
}
// Declare variants
#pragma omp declare variant(axpy_cpu) \\
          match(extension(dynamic))
#pragma omp declare variant(axpy_gpu) \\
          match(extension(dynamic))
void axpy_seq(float* x, float* y, float a, int n) {
  for (int i = 0; i < n; i++) y[i] = a * x[i] + y[i];
}
// Main function
int main() {
  axpy_seq(x, y, a, n);
}
```

Figure 4: Axpy operation using the dynamic variants feature.

4.5.1 Instrumentation Phase: Profiling Each Variant. When compiled with the `-dynamic-variant-warmup` flag, the enhanced LLVM compiler replaces variant calls with instrumentation logic, as shown in Figure 5.

```
// Instrumented main function
int main() {
  // Signal instrumentation start
  __kmpc_instrument_variant_start(...);

  // Choose which variant to profile
  int variant = __kmpc_dynamic_variant(
                    "axpy", "axpy_cpu", "axpy_gpu");
  // Run selected variant
  switch(variant) {
     case 0: axpy_cpu(x, y, a, n); break;
     case 1: axpy_gpu(x, y, a, n); break;
     default: axpy(x, y, a, n);
  }
  // Signal instrumentation end
  __kmpc_instrument_variant_end(...);
}
```

Figure 5: AXPY code after instrumentation.

When running the instrumented binary, the runtime collects power and performance metrics for each variant and stores them to a profiling file, as shown in Figure 6.

```
// variants.c (auto-generated)
variant_data_t variants[] = {
  {"axpy_seq", .power_peak_cpu=30.0, .energy_cpu=300.0,
  .exec_time= 120.0
  ...},
  {"axpy_cpu", .power_peak_cpu=100.9, .energy_cpu=1000.0,
  .exec_time=60.0
  ...},
  {"axpy_gpu", .power_peak_cpu=20.0, .energy_cpu=300.0,
  .exec_time=30.0
  ...}
};
```

Figure 6: AXPY profile data.

Once profiling is complete, the source code is compiled with the generated profile and the `-dynamic-variant` flag. At runtime the binary uses the stored metrics to select the best variant on the fly. The process unfolds as follows:

(1) The runtime loads the profiled variant metrics.
(2) It evaluates each variant against user thresholds and current system state.
(3) The best variant under constraints is selected, using the hierarchical variant selection algorithm in Section 4.4.2.

As an illustration, consider the following scenario:

- With `OPTIMIZE_METRIC="ENERGY_CPU:500:1"`, run the instrumented binary.
- *axpy_seq* consumes 300 J → satisfies threshold.
- *axpy_gpu* consumes 300 J → satisfies threshold.
- *axpy_cpu* consumes 1000 J → does not satisfy it.
- *axpy_gpu* is selected, because among those that pass the threshold, the execution time is the lowest.

5 Evaluation

Modern HPC and HPEC platforms operate under hard *power*, *thermal*, and *energy* constraints. Pushing for peak performance can exceed node or rack-level power budgets [15], trigger temperature-driven throttling that depresses sustained throughput [2], or accelerate hardware wear-out and reduce reliability [14]. In practice, operators enforce power caps and thermal trip points, while battery-powered edge deployments must optimize total energy (J) rather than instantaneous power (W). To cope with these regimes, the proposed *Dynamic-Variants Runtime (DVR)* selects among variants to balance performance against resource limits. This section evaluates DVR across three scenarios: (i) peak-power compliance under dynamic caps on an HPC workstation, (ii) thermal control on an edge device where proactive migration prevents throttling as temperatures approach a setpoint, and (iii) battery-limited operation where energy-aware variant selection maximizes tasks executed under a fixed energy-budget while minimizing energy consumption per-task exectuion.

5.1 Experimental setup

Benchmark suite. To evaluate our framework across a range of computational patterns, we use a suite of representative and publicly available benchmarks. For thermal-aware on the HPEC platform, we use (1) the **XSBench** algorithm, a simplified version of a macroscopic neutron cross-section lookup kernel, which is one of the core operations in a Monte Carlo neutron transport simulation [7]. For multi-criteria, power-constrained evaluation on the HPC platform we use three benchmarks from the On-Board Processing Benchmark (OBPMark) suite [16]: (2) the **AES Encryption** algorithm implements the Advanced Encryption Standard (AES) encryption method and the related CCSDS standard, with 128-, 196- and 256-bit key-lengths (3) the **Radar** algorithm is intended to cover the generation of images from raw radar data, following the range-Doppler algorithm (4) the **Image** algorithm, an image calibration and correction method that combines multiple frames into an image using temporal binning.

[7] https://github.com/ANL-CESAR/XSBench

Hardware platforms. **HPEC Edge device.** An NVIDIA Jetson AGX Orin Dev Kit with a Tegra-23× SoC (12 × Cortex-A78AE @ 2.2 GHz), 32 GB unified LPDDR5, and an integrated Ampere GPU (CC 8.7)[8]. **HPC workstation.** An AMD Ryzen Threadripper 7980X (64 C / 128 T), 128 GB DDR5-5600 ECC, and an ASUS GeForce GT 1030 2 GB (CC 6.1)[9]

5.2 Performance under power peak constraints

To evaluate the effectiveness of DVR in adapting application performance under varying power peak constraints, we conduct an experiment using representative HPC workloads from the OBPMark suite, specifically, we execute a workflow comprising Image, AES-256, and Radar benchmarks.

The workflow is executed using two parallel processes P1 and P2. This parallel execution serves two purposes: (1) create sufficient load to observe meaningful power consumption differences between variants, and (2) demonstrate DVR's ability to adapt when power limits change at runtime. The specific ordering of benchmarks within the workflow is arbitrary; our focus is on the aggregate power behavior and DVR's response to dynamic power cap adjustments rather than individual benchmark characteristics. To ensure our results are statistically robust, each experimental scenario was executed multiple times to account for variability.

Our prototype instruments *whole binaries* at variant boundaries, we do not yet expose task-level (intra-process) semantics. We therefore co-run two processes to induce a controlled, time-varying node load so the runtime can demonstrate on-the-fly decisions in the presence of a co-runner. This is a methodological choice: in classic HPC a single application typically owns the node (e.g., SLURM). The same decision logic would apply to a single process with overlapping kernels/phases once task-level semantics are available. In addition, the runtime implementation remains the same, since the runtime polls whole system resource usage.

For each benchmark the DVR chooses one implementation variant: CUDA, Sequential, or OpenMP (from 4 to 128 threads in powers of 2), applying the selected variant uniformly across all kernels within that benchmark. Variant parameters used for the binaries in warm-up and selection phases are shown in Table 5.

Table 5: OBPMark benchmarks and their configuration for the power-cap study.

Benchmark configuration
Image v1.1: 10,000 × 10,000 px, 8-tap filter; Variants per run: CUDA, sequential CPU, OpenMP 4/8/16/32/64/128 threads
AES-256 v3.1: 675 MB plaintext, 256-bit key; Variants per run: CUDA, sequential CPU, OpenMP 4/8/16/32/64/128 threads
Radar v1.2: 8 look angles, 8,000 × 10,000 samples; Variants per run: CUDA, sequential CPU, OpenMP 4/8/16/32/64/128 threads

Due to the limited 2 GB VRAM of the GT 1030 GPU, concurrent execution of GPU-intensive benchmarks is not feasible. To avoid out of memory errors, we manually enforce serial execution of GPU workloads. Consequently, the DVR scheduler automatically

[8] Jetson runs Ubuntu 20.04, JetPack 5.1.1 (L4T 35.3.1), CUDA 11.8.89.
[9] Workstation runs Ubuntu 24.04.2, NVIDIA driver 575.51.03, CUDA 12.0.140; default governors unless stated.

disqualifies CUDA variants when concurrent GPU utilization is detected, thus ensuring stable execution.

To test DVR's adaptability, we defined a two-phase power cap that dynamically tightens from 250 W to 200 W mid-execution. For the baseline, we execute without DVR using manually selected optimal variants per benchmark (CUDA, OpenMP-64t, or OpenMP-16t as shown in Figure 7). These thread counts represent actual performance optima. Using OpenMP 128t would degrade performance due to excessive threading overhead while simultaneously increasing power consumption, making it both slower and less efficient. Despite using these performance-optimal variants, the static execution still produces frequent power violations, demonstrating that even well-tuned static configurations cannot match DVR's dynamic adaptation to power constraints. Figure 7 shows a representative power trace of this unconstrained scenario, where the runtime consistently chooses the fastest variants, causing frequent and large power spikes.

The aggregated statistics over multiple runs, presented in Table 6, quantify this behavior. On average, the baseline execution achieved only **66.9% ± 0.6%** compliance with the hypothetical power cap. The p99 power[10] reached a mean of **371.7 ± 8.9 W**, and the average maximum overshoot above the threshold was **231.8 ± 69.1 W**. These results confirm that without DVR the system consistently and significantly violates the power budget.

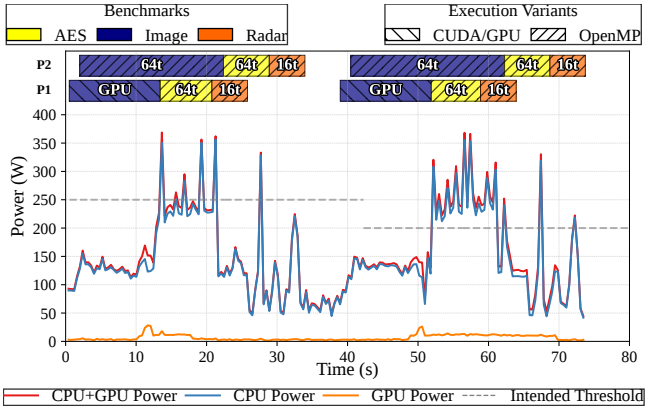


Figure 7: Baseline scenario (no power limits), two parallel processes executing OBPMark workloads. (Single run)

In contrast, Fig. 8 demonstrates how the DVR adapts when explicit power constraints are defined. Initially we clamp the `POWER_PEAK_SUMMATION` channel at 250 W for the first phase. During runtime we further decrease the cap to 200 W for the second phase, testing the DVR's responsiveness under tighter constraints. The statistical results in Table 7 confirm the effectiveness of this adaptive strategy across all multiple runs. The DVR achieved a mean overall compliance rate of **98.5% ± 0.7%**. Concretely, p99 power drops from 371.7 ± 8.9 **W** (baseline) to 218.4 ± 15.1 **W** (DVR), violation time from 12.4 ± 0.2 **s** to 0.8 ± 0.3 **s**, and max overshoot from 231.8 ± 69.1 **W** to 32.3 ± 12.0 **W**.

[10]The 99th percentile of power samples, a robust metric for near-peak behavior.

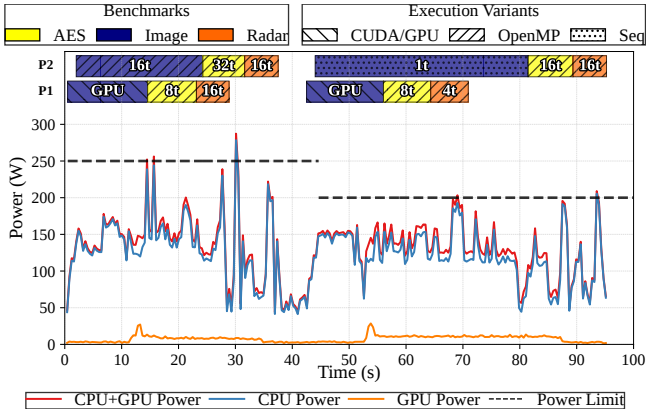


Figure 8: Power-constrained scenario (250 W → 200 W), DVR dynamically adapts variant selection. (Single run)

Table 6: Cap-compliance and overshoot metrics for multiple unlimited baseline runs.

Metric	Mean ± Std
Overall compliance [%]	66.9 ± 0.6
Violation time [s]	12.4 ± 0.2
Phase 1 compliance (250 W) [%]	70.8 ± 0.8
Phase 2 compliance (200 W) [%]	63.0 ± 0.8
Max overshoot [W]	231.8 ± 69.1
Mean overshoot [W]	79.4 ± 4.5
p99 power [W]	371.7 ± 8.9

Table 7: Cap-compliance and overshoot metrics for multiple limited runs (250 W → 200 W).

Metric	Mean ± Std
Overall compliance [%]	98.5 ± 0.7
Violation time [s]	0.8 ± 0.3
Phase 1 compliance (250 W) [%]	99.7 ± 0.7
Phase 2 compliance (200 W) [%]	97.4 ± 1.0
Max overshoot [W]	32.3 ± 12.0
Mean overshoot [W]	17.5 ± 9.0
p99 power [W]	218.4 ± 15.1

These experiments clearly demonstrate that DVR effectively balances performance and power efficiency by dynamically selecting the optimal variant to meet real-time power constraints.

5.3 Thermal control: run-time migration beats throttling

While the previous experiment demonstrates the DVR ability to respect power peak budgets in HPC environments, modern computing systems, especially those used in HPEC environments face a additional physical constraints that manifest as thermal limits. These two types of constraints, power and thermal, are fundamentally related through the physics of semiconductor operation, yet

they present distinct challenges that require different runtime adaptation strategies. Power caps typically arise from infrastructure limitations (e.g., data power delivery, electricity costs) and affect the entire system uniformly. In contrast, thermal constraints emerge from local hot spots and inadequate heat dissipation, particularly in edge computing platforms such as an NVIDIA Jetson where compact form factors limit cooling capacity.

To demonstrate the DVR's versatility in handling both constraint types within a unified framework, we now shift our evaluation to a thermally-constrained edge device.

The following experiment shows how DVR's variant selection mechanism can prevent thermal throttling, a performance degrading hardware response where processors automatically reduce their operating frequency to stay within safe temperature ranges. By proactively migrating computations between processing units before thermal limits are reached, DVR maintains higher sustained performance than traditional throttling-based thermal management. In order to evaluate the DVR's thermal management capabilities, we set GPU-therm diode to trip at 49 °C via the thermal management subsystem provided by Linux.

In the Jetson platform, each major component (CPU, GPU, and others) has its own dedicated cooling mechanism and operates largely independently in terms of thermal regulation. This independence allows targeting thermal management policies toward a specific component without affecting the others. In our case, we configure the system to throttle only the GPU rather than imposing a board-wide limit, enabling us to focus on the GPU's thermal behavior and performance under controlled conditions.

Experiments were conducted on an NVIDIA Jetson AGX Orin dev-kit configured for `MAXN` mode. The GPU runs at 1.30 GHz and the CPU cluster at 2.21 GHz. The active cooler of the dev-kit remained on cool mode. We execute XSBench sequentially three times, using the parameters shown in table 8.

Table 8: XSBench and the configuration for the thermal study.

Benchmark configuration
XSBench: grid size: large, grid type: hash; Variants per run: CUDA, OpenMP Offload, OpenMP 12 threads

Figure 9 shows the baseline case without DVR where the thermal policy stair-steps the GPU clock down to 300 MHz to stay below 49 °C.

The thermal management system keeps lowering the GPU frequency in order to cool the GPU at the expense of adding more execution time per execution.

In contrast, Fig. 10 uses DVR with the threshold environment variable `PEAK_GPU_TEMPERATURE:49:1`. Whenever the GPU temperature hits the target of 49 °C, the DVR will automatically dispatch to the compute unit that is more conservative or satisfies the threshold. Fig 10 demonstrates that the dynamic policy migrates work to the CPU once the GPU temperature sensor hits 49 °C. We can observe how the GPU frequency goes down when the DVFS is triggered, after the first benchmark has finished executing, the runtime notices that the temperature of the GPU is above 49 °C and dispatches the next benchmarks to the CPU until the temperature goes below the threshold.

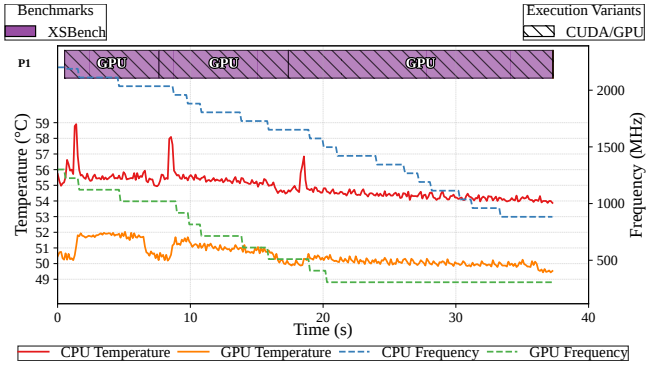


Figure 9: Temperature and frequency timeline with GPU throttling at 49 °C. (Single run)

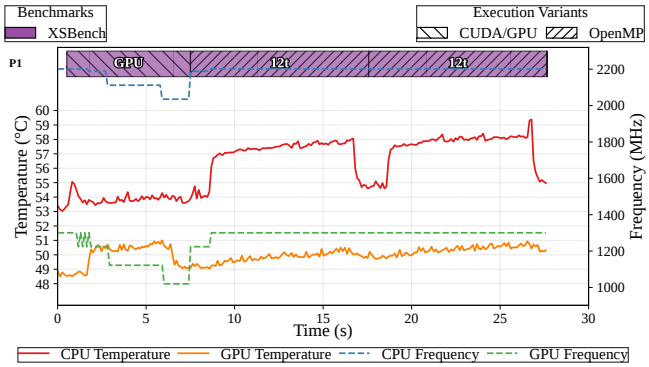


Figure 10: Temperature and frequency timeline under dynamic GPU→CPU migration when GPU hits 49 °C. (Single run)

Run-time migration therefore cuts execution time by **26 %** (37.5 to 27.7 s) compared with static throttling at 49 °C as shown in Figures 9 and 10.

Across multiple trials, the results show that run-time migration reduces execution time by **29.4% ± 6.9%** (39.5 ± 3.8 s $\rightarrow$ 27.9 ± 0.4 s) compared with hardware throttling at 49 °C. The dynamic policy not only accelerates execution, it also exhibits an order-of-magnitude lower run-to-run variability ($\sigma_{\text{dyn}} = 0.4$ s vs. $\sigma_{\text{thr}} = 3.8$ s), eliminating the frequency "flapping" typical of DVFS-governed GPUs. When thermal headroom is scarce, *moving the work* is therefore both safer and faster than throttling it, an approach that also helps mitigate long-term device degradation [6] and avoids the frequency flapping documented in prior work on embedded GPUs[5].

5.4 Battery-limited operation: smart beats fast

While the previous experiments demonstrate DVR's effectiveness in managing instantaneous power and thermal constraints typical of grid-connected systems, many edge computing deployments face a different challenge: finite energy budgets. Battery-powered systems in remote sensing, autonomous vehicles, and space applications must optimize not just for performance or instantaneous power, but for total mission accomplishment under strict energy constraints.

Unlike grid-connected systems where power caps protect infrastructure, battery-limited systems must balance the competing demands of task throughput, execution speed, and energy efficiency to maximize operational lifetime. This challenge is different from managing power peaks, because the energy consumption over time dictates battery life. Power peaks may be momentary and manageable, energy consumption is accumulated over time as the sum of instantaneous power values multiplied by the sample rate. Even brief bursts of high power can add up, depleting the battery faster. As such, battery-powered systems need to focus on optimizing this total energy consumption to ensure mission completion, especially when recharge opportunities are intermittent or constrained, such as with solar-powered edge devices that must survive through periods of darkness until the next charging window.

To evaluate DVR's effectiveness in battery-constrained scenarios, we conduct a mission-level simulation that models realistic operational constraints faced by autonomous edge systems. While we abstract away cell-level battery dynamics (e.g., voltage sag, internal resistance, cut-off curves) to isolate the runtime policy effects, our model captures the essential trade-offs between performance and energy consumption that determine mission success. The simulation operates under a finite battery capacity of $E_{\text{bat}} = 27.8$ kJ, with a single recharge window providing 30 W for 2 minutes at $t \in [22, 24]$ min of 3.6kJ to emulate opportunistic harvesting (e.g., a brief solar exposure under a cloudy environment) and to stress the policy decision.

We evaluate DVR using the OBPMark **Image** benchmark (10,000 x 10,000 pixels, 8 frames, 8-tap filter). The warmup is done with 4 to 12 threads in increments of 2 using the OpenMP variants. Tasks execute back-to-back with no idle intervals, simulating continuous operational requirements typical of monitoring applications.

In this study, profiling is restricted to OpenMP variants with a varying degree of thread counts. Including the CUDA in the profiling process would result in its selection, as it exhibits superior performance and minimal energy consumption. On the other hand, the sequential implementation exhibits the lowest performance, with longer execution times and the highest energy consumption. A similar performance degradation is observed for configurations employing between one and four threads.

As described in the variant selection subsection 4.4, when all variants exceed the specified threshold, the runtime selects the variant whose value is closest to that threshold. This behavior allows for variant selection within the Pareto frontier.

When we optimize for two goals at once, time and energy, some variants are clearly worse than others. A variant is Pareto-optimal if no other variant is both faster and more energy-efficient. The set of non-dominated variants is the Pareto frontier. Points off the frontier are dominated, meaning there exists another variant that is at least as good on both axes and strictly better on one.

Figure 11 reveals the trade-offs between execution time and energy consumption across different variants. The time-energy Pareto frontier shows that OpenMP with 6 threads (OMP-6) dominates OMP-12 on energy efficiency, achieving nearly identical performance while consuming less energy. The OMP-12 averages $T = 26.37$ s and $E = 593.16$ J per-task, while OMP-6 averages $T = 26.58$ s and $E = 502.64$ J. This represents a 15.2% reduction in energy consumption at less than 1% performance penalty.

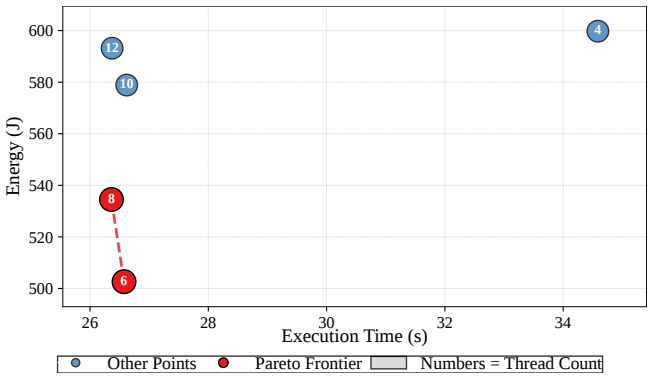


Figure 11: Per-execution time–energy Pareto for OBPMark Image (10k×10k, 8 frames) on Jetson.

The image benchmark is dominated by per-pixel passes, so time improves with threads until the platform hits a "knee" where additional threads no longer reduce runtime. The 4-thread case sits below this knee. It exposes less concurrency and runs longer than the 6–12-thread configurations. Since we report *total system* energy and several Jetson rails (e.g., `VIN_SYS_5V0`, DDR) draw roughly steady power while the program runs, the longer run also consumes more energy. Hence OMP-4 is dominated and appears off the Pareto front.

To demonstrate the practical impact of energy-aware variant selection on mission success, we compare two scenarios: **FAST**, which operates without DVR and consistently selects the (OMP-12), and **ENERGY-CAPPED**, which uses DVR with `OPTIMIZE_METRIC="ENERGY_TOTAL_SYSTEM:0:1"` to select OMP-6 based on energy consumption. Here we intentionally set a threshold of 0, because all variants exceed the energy threshold, hence DVR selects the variant that minimizes energy consumption, always choosing thread counts that are on the Pareto frontier. If we had not profiled with 6 threads, the runtime would have selected the 8 thread variant instead. In this case, since the energy consumption is the integral of instantaneous power over a time sample, the variant selection is minimizing the execution time and energy consumption tradeoff. Both policies execute identical workloads against the same battery budget and recharge opportunity.

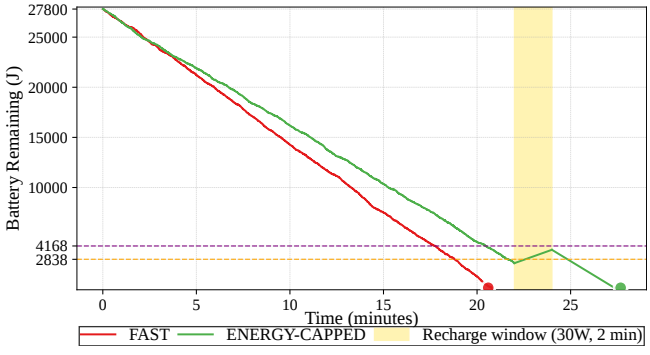


Figure 12: Battery state-of-charge over time for FAST (OMP-12) vs. ENERGY-CAPPED (OMP-6). The shaded band indicates the 30 W, 2 min recharge window. (Single run)

Figure 12 compares the mission outcomes of the two policies. The **FAST** policy depletes the battery at **20 min and 35 sec** (**20:35.9**) *before* the 22–24 min recharge window and therefore cannot harvest energy. It shuts down after completing **46 full tasks** (46.87 total). The slightly faster per-task latency of OMP-12 comes with a higher average draw ($\approx$ 22.49 W), producing an energy burn that exhausts the battery ahead of the charging opportunity.

In contrast, **ENERGY-CAPPED** (OMP-6) reduces per-task energy by ~90.5 J (593.16→502.64 J) at essentially identical latency (26.37 s vs. 26.58 s), lowering average system power to $\approx$ 18.91 W. This keeps the system alive to the recharge window: at **22:00** it has **2,838 J** remaining and has completed **49 full tasks** (49.66 total). Across the **2 min** window, the charger delivers a gross **3,600 J**, but because tasks continue running, the *net* battery gain is only **1,331 J** (30 W − 18.91 W) × 120 s. The battery thus rises to **4,169 J** at **24:00**, yielding **3 min 40 sec** (**3:40.5**) of additional runtime and a final shutdown at **27 min 40 sec** (**27:40.5**). By that point, ENERGY-CAPPED has completed **62 full tasks** (62.47 total), substantially more than FAST, illustrating how choosing the Pareto-efficient variant extends both mission lifetime and total work accomplished.

6 Conclusions

The increasing heterogeneity and strict power constraints of modern computing systems demand adaptive software solutions that provide performance portability. Standard compile-time optimizations for OpenMP are insufficient to handle the dynamic conditions of HPC and HPEC environments. To address this gap, we have designed, implemented, and validated an energy and thermal-aware runtime framework that extends OpenMP's declare variant feature. Our system uses a two-phase, profile and select approach. An offline instrumentation and profile phase characterizes each function variant's performance, power, and thermal profile. At runtime, a hierarchical selection algorithm uses this data, combined with live system state for instantaneous constraints, to select the optimal variant based on user-defined, multi-criteria policies. Our experimental evaluation demonstrates the practical benefits of this approach. On an HPC workstation, the runtime successfully enforced dynamic power caps with over 98% compliance by intelligently selecting lower-thread-count variants. On a thermally-constrained embedded platform, it improved performance by 39% over traditional hardware throttling by migrating workloads from the GPU to the CPU. In a battery-limited mission simulation, our framework increased task throughput by choosing a Pareto-optimal, energy-efficient variant that allowed the system to survive long enough to reach a critical recharge window while saving 15% of energy consumption.

Beyond the best-effort guarantees shown here, as future work we plan to (a) add progress-aware and preemptible scheduling via OMPT hooks, (b) replace offline profiling with online profiling and uncertainty guards, (c) pair selection with active actuators (RAPL, NVML/rocm-smi, CPUfreq, cgroups), (d) complete temperature providers on x86 for unified thermal policies, (e) lift policies to the cluster level (SLURM/Kubernetes) to coordinate node/rack budgets (f) kernel level variants where the runtime automatically does memory transfers and selects the optimal hardware (g) prune the warm-up search space using cost-aware heuristics and lightweight learned models, keeping profiling overhead bounded as the number of variants grows.

7 Discussion

It is important to point out that DVR employs a best-effort approach. If users set unrealistically low power thresholds, DVR may be unable to guarantee compliance. Furthermore, external applications consuming power simultaneously may also affect DVR's ability to maintain the specified power cap.

Because selection is based on a snapshot of system state, conditions may change between selection and execution (e.g., another application starts on a different core or the GPU), invalidating the original choice. This is common on multi-tenant, multi-core nodes. To reduce this effect, we recommend using short-lived variants.

A further limitation is the lack of progress awareness during the variant selection phase. Once a variant is dispatched, control transfers to the operating system, so the OpenMP scheduler has no visibility into what is currently running or how much time remains. As a result, decisions are work-conserving but nearsighted. The selector chooses the best variant for the current conditions without considering that a faster resource may become available shortly, which can yield suboptimal makespan.

Our prototype operates at whole-binary scope, which aligns with common HPC practice (one MPI+OpenMP binary per node or GPU partition). On large nodes with many cores/GPUs this still applies: selection remains local to the controlled binary, using its profiled variants and current sensor readings to satisfy user thresholds. When nodes are shared, interference can shift feasibility; the best-effort and snapshot properties above already cover this case. To move beyond a single node without changing application code, budgets can be coordinated job-wide (e.g., via SLURM/Kubernetes) while each node keeps running the same selection logic—i.e., node-local decisions under a job-level envelope.

Our warm-up phase executes all declared variants once to obtain representative time and resource-usage profiles. For highly parameterized applications/kernels this exhaustive pass can be costly as the search space grows. Practical mitigations include: (i) online profiling with early stopping once time/energy stabilize within a tolerance, (ii) skipping regions shorter than the sensor period and assigning them the current selection, (iii) grouping very short calls into metered epochs, and (iv) cost-aware or learned pruning so only promising variants are sampled. Profiles are cached per (binary, hardware, policy) so subsequent runs incur no warm-up.

We prototype with `match(implementation=extension(dynamic))`. To avoid confusion with OpenMP's `schedule(dynamic)`, an implementation-defined trait such as `extension(dynselect)` would be clearer. A pragmatic path is to ship as an extension, gather experience on thresholds, scoring, and telemetry semantics, and then socialize a proposal to the OpenMP ARB for runtime-aware variant selection with a neutral trait name.

Acknowledgments

This work is funded by the HiPART project, with reference PID2023-148117NA-I00, financed by MICIU/AEI /10.13039/501100011033 and FEDER, UE, as well as the ASCENDER project, financed by the UNICO I+D Cloud program (Ministry for Digital Transformation and Public Administration and the European Union – NextGenerationEU, within the framework of the PRTR and the MRR).

References

[1] Ferdinando Alessi, Peter Thoman, Giorgis Georgakoudis, Thomas Fahringer, and Dimitrios S Nikolopoulos. 2015. Application-level energy awareness for OpenMP. In *Proceedings of the 11th International Workshop on OpenMP*. Springer, 219–232.

[2] Ghazanfar Ali, Lowell Wofford, Christopher Turner, and Yong Chen. 2022. Automating CPU dynamic thermal control for high performance computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 514–523.

[3] OpenMP ARB. 2018. OpenMP Application Programming Interface version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[4] OpenMP ARB. 2024. OpenMP Application Programming Interface examples version 6.0. https://www.openmp.org/wp-content/uploads/openmp-examples-6.0.pdf

[5] James RB Bantock, Bashir M Al-Hashimi, and Geoff V Merrett. 2020. Mitigating interactive performance degradation from mobile device thermal throttling. *IEEE Embedded Systems Letters* 13, 1 (2020), 5–8.

[6] C-H Hsu and Wu-Chun Feng. 2005. *Reducing overheating-induced failures via performance-aware CPU power management.* Technical Report. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

[7] Jan Christian Meyer, Thomas B Martinsen, and Lasse Natvig. 2017. Implementation of an energy-aware OmpSs task scheduling policy.

[8] Maxime Mirka, Guillaume Devic, Florent Bruguier, Gilles Sassatelli, and Abdoulaye Gamatié. 2019. Automatic energy-efficiency monitoring of OpenMP workloads. In *14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip*. IEEE, 43–50.

[9] Maxime Mirka, Gilles Sassatelli, and Abdoulaye Gamatié. 2020. Online learning for dynamic control of OpenMP workloads. In *9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 1–6.

[10] Adrian Munera, Guerau Dasca, Eduardo Quiñones, and Sara Royuela. 2024. Adaptive parallelism in OpenMP through dynamic variants. In *International Conference on High Performance Computing*. Springer, 17–30.

[11] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesús Labarta. 2013. Self-adaptive OmpSs tasks in heterogeneous environments. In *IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 138–149.

[12] Allan K Porterfield, Stephen L Olivier, Sridutt Bhalachandra, and Jan F Prins. 2013. Power measurement and concurrency throttling for energy reduction in OpenMP programs. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 884–891.

[13] Rishad A Shafik, Anup Das, Sheng Yang, Geoff Merrett, and Bashir M Al-Hashimi. 2015. Adaptive energy minimization of OpenMP parallel applications on many-core systems. In *Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*. 19–24.

[14] Pardeep Shahi, Amith Mathew, Satyam Saini, Pratik Bansode, Rajesh Kasukurthy, Dereje Agonafer, et al. 2022. Assessment of reliability enhancement in high-power CPUs and GPUs using dynamic direct-to-chip liquid cooling. *Journal of Enhanced Heat Transfer* 29, 8 (2022).

[15] Christian Simmendinger, Marcel Marquardt, Jan Mäder, and Ralf Schneider. 2024. Powersched-managing power consumption in overprovisioned systems. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. IEEE, 1–8.

[16] David Steenari, Leonidas Kosmidis, Ivan Rodríguez-Ferrández, Alvaro Jover-Alvarez, and Kyra Förster. 2021. OBPMark (On-Board Processing Benchmarks)-open source computational performance benchmarks for space applications. In *European Workshop on On-Board Data Processing (OBDP)*. 14–17.

[17] Zeyu Yang, Karel Adamek, and Wesley Armour. 2024. Accurate and Convenient Energy Measurements for GPUs: A Detailed Study of NVIDIA GPU's Built-In Power Sensor. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–17. doi:10.1109/SC41406.2024.00028