Supplemental: Minimization of Execution Time and Energy Consumption of Parallel Applications on Heterogeneous Hybrid Parallel Computers through Optimal Combination of Workload Distribution and Operating Voltage and Frequency

Hesam Nejati Sharifaldin, *Member, IEEE*, Sourodip Ghoshdastidar, Ravi Reddy Manumachu, *Member, IEEE*, and Alexey Lastovetsky, *Member, IEEE*

I. Introduction

The supplementary material includes:

- Definition of static and dynamic energy consumptions.
- Background on bi-objective optimization.
- A study of DVFS governors for multicore CPU processors in Linux OS and DVFS tools for Nvidia GPU processors.
- A study of Likwid tools to obtain performance events including voltage of a processor core for Intel multicore CPU processors.
- Statistical methodology to obtain an experimental data point. Steps to prevent noise in energy measurements.
- Detailed description of the continuous algorithm merging the piecewise linear pareto fronts.
- Formulation of the discrete optimization problem HEPOPTGD and an exact algorithm HEPOPTAGD solving the problem.
- Description of the HEPOPTA algorithm invoked by HEPOPTAGD solving the discrete optimization problem.
- Efficient construction of models of two applications, DGEMM and 2D-FFT, for processor configuration represented by DVFS state.
- Dynamic energy and execution time profiles for the two hybrid parallel applications, 2D fast Fourier transform and mixed precision matrix multiplication, on the heterogeneous platform comprising an Intel Icelake multicore CPU and two Nvidia A40 GPUs.
- Workflow of the optimization process of a data-parallel application for performance and energy on a heterogeneous hybrid server.

II. Static and Dynamic Energy Consumptions

There are two types of energy consumption, static and dynamic. The static energy consumption is the idle power of the platform multiplied with the execution time of the application. The dynamic energy consumption is calculated by subtracting the static energy consumption from the platform's total energy consumption during the application execution. If P_S is the idle power consumption of the platform, E_T is the total energy consumption of the platform during the execution of the application, which takes T_E seconds, then the dynamic energy E_D is

$$E_D = E_T - (P_S \times T_E) \tag{1}$$

Futhermore, the dynamic energy consumption during an application execution is the average dynamic power consumption multiplied by the execution time of the application.

H. Sharifaldin, S. Ghoshdastidar, R. R. Manumachu, and A. Lastovetsky are with the School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland.
E-mail: hesam.nejatisharifaldin@ucdconnect.ie, sourodip.ghoshdastidar@ucdconnect.ie,ravi.manumachu@ucd.ie, alexey.lastovetsky@ucd.ie

III. Background on Bi-objective Optimization

In mathematical terms, a bi-objective optimization problem can be formulated as [1], [2]:

$$\begin{aligned} &minimize \quad \{T(\vec{x}), E(\vec{x})\} \\ &\text{Subject to} \quad \vec{x} \in \mathcal{X} \end{aligned}$$

There are two objective functions, $T : \mathbb{R}^k \rightarrow \mathbb{R}$ and $E : \mathbb{R}^k \rightarrow \mathbb{R}$ denoted by a vector, $\mathcal{F}(\vec{x}) = (T(\vec{x}), E(\vec{x}))^T$. The decision variable vectors, $\vec{x} = (x_1, ..., x_k)^T$, belong to the feasible set, $\mathcal{X} \subset \mathbb{R}^k$. The feasible objective set, which is the image of $\mathcal{X}$, is given by $\mathcal{F}(\mathcal{X}) \subset \mathbb{R}^2$. The goal of the problem is to minimize both objective functions simultaneously. The solutions to the problem are called Pareto-optimal solutions. The set of Pareto-optimal solutions is called the Pareto front.

Definition 1. *A decision variable vector $\vec{x}^* \in \mathcal{X}$ is Pareto optimal if there does not exist another decision variable vector $\vec{x} \in \mathcal{X}$ such that $T(\vec{x}) \leq T(\vec{x}^*), E(\vec{x}) \leq E(\vec{x}^*)$ and either $T(\vec{x}) < T(\vec{x}^*)$ or $E(\vec{x}) < E(\vec{x}^*)$ or both.*

Any point in the feasible objective set, $\mathcal{F}(\mathcal{X})$, that is not on the Pareto front is a bad solution. Along the Pareto front, moving from one Pareto-optimal solution to the other will require a trade-off between the two objectives.

There are several classifications for methods solving bi-objective optimization problems [1], [2]. Since the set of Pareto optimal solutions is partially ordered, one classification is based on the involvement of the decision-maker in the solution method to select specific solutions. This involvement can be classed as follows:

- **No preference**: No preference information is provided. In this case, the solution method can return one or more solutions that the decision-maker can accept or reject.
- **A posteriori**: The solution method determines the Pareto-optimal set of solutions and presents it to the decision maker, who selects the most preferred one. These methods usually have high computational complexity.
- **A priori**: The decision maker provides his or her preferences before the solution process. The most common approach is called the aggregation method, where the objective functions are combined into a single objective function using the utility function, which is provided in a mathematical form by the decision-maker. The single objective function is then optimized using traditional optimization methods.
- **Interactive**: There is repeated interaction between the decision maker and the solution method. The basic steps are [1]: a). Find an initial feasible solution b). Interact with the decision maker c). Obtain a new solution. If the new solution or one of the previous solutions is acceptable to the decision-maker, stop. Otherwise, go to step (b).

The algorithms solving bi-objective optimization problems can be divided into two major categories, *exact methods* and *metaheuristics*. While branch-and-bound (B&B) is the dominant technique in the first category, genetic algorithm (GA) is popular in the second category.

IV. Dynamic Voltage and Frequency Scaling (DVFS) in Multicore CPU and Nvidia GPU Processors

Modern processors can operate in a number of clock frequency and voltage configurations, often referred to as P-states in ACPI terminology. Briefly, ACPI defines the power state of system processors while in the G0 working state as being either active executing or sleeping [3]. The processor power states are designated $C_0, C_1, \cdots, C_n$. The C_0 power state is an active power state where the CPU executes instructions. The C_1 through C_n power states are processor sleeping states where the processor consumes less power and dissipates less heat than the processor in the C_0 state. In the C_0 state, ACPI allows scaling the CPU frequency up or down in order to save power or improve performance through transitions into multiple performance states (P-states), $P_0, P_1, \cdots, P_m$. The CPU frequency scaling can be done automatically in response to system load or programmatically in user space programs.

A. DVFS for Multicore Processors in Linux

The Linux kernel supports CPU frequency scaling through the *CPUFreq* subsystem. The subsystem provides three main layers of abstraction:

- The *core* provides the user space interfaces to support CPU frequency scaling. It establishes the fundamental framework for the other two components [4].
- *Scaling governors* that implement the algorithms to compute the desired CPU frequency (potentially based on CPU utilization [5]).
- *Scaling drivers* provide information on the available P-states to the scaling governors and access platform-specific hardware interfaces to change CPU P-states as requested by the scaling governors.

All available scaling governors can be used with every scaling driver because the scaling algorithm can select a P-state in a platform-independent way and request the scaling driver to set the state. Therefore, the same scaling governors should be suitable for every supported platform.

However, some performance scaling algorithms employ information provided by the hardware using feedback registers specific to the hardware interface. For this case, CPUFreq allows scaling drivers to bypass the governor layer and implement these performance scaling algorithms. For instance, *intel_pstate* scaling driver adopts this approach.

CPUFreq provides generic scaling governors that can be used with all scaling drivers. Each scaling governor implements a pre-configured power scheme within the operating system kernel. The standard scaling governors are:

- The *performance* governor allows to maximize performance by setting CPU frequencies to the maximum available level. However, it results in higher power consumption.
- The *powersave* governor maximizes power savings by setting CPU frequencies to their minimum available values. However, it can potentially impact performance, resulting in prolonged response time.
- The *ondemand* governor dynamically adjusts CPU frequency based on CPU utilization (CPU load). It continuously monitors utilization, and when it surpasses a specific threshold, the governor maximizes the CPU frequency. Conversely, if utilization falls below the threshold, it lowers the CPU frequency to conserve power.
- The *conservative* governor also dynamically adjusts CPU frequency based on CPU load. It estimated the CPU load using the same approach as the *ondemand* governor. However, it uses a different CPU frequency selection algorithm. It avoids changing the frequency significantly over short intervals by changing it in relatively small steps, depending on whether or not the estimated CPU load has exceeded a (configurable) threshold.
- The *userspace* governor empowers the user to control the CPU frequency. The user can manually set the CPU frequency to a specific value among those supported by the processor, allowing balancing performance and power consumption according to the user's specific needs.
- The *schedutil* governor uses the CPU utilization metric employed by the CPU scheduler. It is regarded as a replacement for the *ondemand* and *conservative* governors since it is simpler in design and tightly integrated with the CPU scheduler.

Additionally, modern CPUs include a feature known as *Turbo Boost* which can propel a processor from a base frequency to the maximum Turbo frequency to boost the performance of the processor's fastest cores. More details can be found in [6].

In this work, we utilize CPUFreq with the *acpi_cpufreq* scaling driver since this driver allows the user to set the CPU frequencies using the *userspace* scaling governor. We employ the *cpupower* tool to set a processor frequency.

We employ all the available frequencies (P-states) when using the *userspace* scaling governor, ranging from 1.0 GHz to 2.1 GHz, with an increment of 0.1 GHz. Specifically, the tested frequencies were: $\{1.0, 1.1, 1.2, ..., 2.0, 2.1\}GHz$.

The scripts containing the commands to set and view the frequencies can be accessed at the public repository [7].

B. DVFS for Nvidia GPU Processors in Linux

The user can set the main memory and graphics clock frequencies of an Nvidia GPU using the *nvidia-smi* tool. The user can set the applications clocks (memory and graphics) using the *-lgc* option. The user can reset the application clocks using the *-rgc* option. The scripts containing the commands to set and reset the application clocks are provided in the public repository [7].

V. Overview of Performance Monitoring Tools Providing Hardware Events

Likwid [8] and PAPI [9] are two widely used performance profiling and monitoring tools that allow access to low-level hardware events, such as cache misses, branch mispredictions, and floating-point operations.

These two tools differ significantly in their design goals, features, and use cases. PAPI provides portable, cross-platform, fine-grained performance monitoring that can be easily integrated into application code. In contrast, Likwid offers built-in tools for rapid profiling that require minimal modifications to the code.

The primary reason for using Likwid in this work is its capability to provide the voltage readings of the logical CPU cores through the performance event labeled *VOLTAGE_CORE*, which is part of the performance monitoring group known as *VTG0*. PAPI, on the other hand, does not provide direct voltage measurements for either logical or physical cores as part of its standard hardware event monitoring capabilities.

Likwid is a well-established tool for obtaining performance event counts. It offers c number of core-level and socket-wide counters and e performance events where c and e vary from one processor architecture to another and e is much greater than c. Furthermore, the values of c and e increase with every processor generation. The counters are further divided into core-level and socket-wide counter groups of a specific hardware component of the architecture. For example, there are 160 counters, {CBOX0C0,...,CBOX0C3,...,CBOX39C0,...,CBOX39C3}, that belong to the CBOX group that can capture the counts of activities of the different caching agents in the Intel IceLake multicore CPU processor. A caching agent is a per-core unit inside the cores that maintains the cache coherency between multiple processor cores. Therefore, each group of counters can store the counts of performance events of that group. Furthermore, all the counters can gather as many performance event counts as possible in one application run.

We employ a lightweight Likwid tool, *likwid-perfctr*, which is a command line tool to configure and read out hardware performance monitoring data on supported x86 processors. *likwid-perfctr* allows to measure the average voltage of a core during an application execution. The script containing the commands to obtain the voltage readings are provided in the public repository [7].

VI. Statistical Methodology to Measure Execution Time and Energy Consumption

Consider a hybrid application, *app*, consisting of three data-parallel kernels, *Kernel_cpu*,*Kernel_gpu1*, and *Kernel_gpu2*, which run in parallel. The goal is to measure the execution time and the dynamic energy consumption of kernels in the application. To do this, we instrument the application, as shown in Algorithm 1. The instrumented application returns the execution time of each kernel and the energy consumption of all three kernels.

Algorithm 1 Instrumentation of a sample application (*app*) consisting of three kernels, executing on CPU and two GPUs simultaneously.

```
1: HCL_WATTSUP_START( )
2: #pragma parallel
3: Begin
4:     te_cpu1 ← gettimeofday()
5:         KERNEL_CPU( )
6:     te_cpu2 ← gettimeofday()
7: End
8: Begin
9:     te_gpu11 ← gettimeofday()
10:        KERNEL_GPU1( )
11:    te_gpu12 ← gettimeofday()
12: End
13: Begin
14:    te_gpu21 ← gettimeofday()
15:        KERNEL_GPU2( )
16:    te_gpu22 ← gettimeofday()
17: End
18: energy_app ← HCL_WATTSUP_STOP( )
19: te_cpu ← te_cpu2 − te_cpu1
20: te_gpu1 ← te_gpu12 − te_gpu11
21: te_gpu2 ← te_gpu22 − te_gpu21
22: return (te_cpu, te_gpu1, te_gpu2, energy_app)
```

1) Methodology to Measure Execution Time: We instrument each kernel in the hybrid application (*app*) using the member function *gettimeofday()* of the Linux library *sys/time.h* to measure its execution time separately. As shown in Algorithm 1, the execution times are stored in variables te_{cpu}, te_{gpu1}, and te_{gpu2} and output at the end of the application execution.

2) Methodology to Measure the Energy Consumption: The heterogeneous hybrid node has one WattsUp Pro power meter that sits between the wall A/C outlets and the node's input power sockets. The power meters capture the total power consumption of the node. They have data cables connected to one USB port of the node. A Perl script collects the data from the power meter using the serial USB interface. The execution

of these scripts is non-intrusive and consumes insignificant power.

The power meters are periodically calibrated using an ANSI C12.20 revenue-grade power meter, Yokogawa WT210. The maximum sampling speed of the power meters is one sample every second. The accuracy specified in the data-sheets is $\pm 3\%$. The minimum measurable power is 0.5 watts. The accuracy at 0.5 watts is ± 0.3 watts.

We use *HCLWattsUp* API [10], which gathers the readings from the power meters to determine the average power and energy consumption during the execution of an application for the whole node. *HCLWattsUp* API also provides two macros: *HCL_WATTSUP_START* and *HCL_WATTSUP_STOP*. The *HCL_WATTSUP_START* macro starts gathering power readings from the power meter using the Perl script, whereas the *HCL_WATTSUP_STOP* stops gathering and return the total energy as a sum of these power readings.

To measure the amount of energy consumed by the application, we invoke *HCL_WATTSUP_START* and *HCL_WATTSUP_STOP* macros, as shown in Algorithm 1. The consumed energy is stored in the variable, $energy_{app}$, that is output at the end of the application execution.

A. Methodology to Ensure Reliability of Experimental Results

As explained in Section VI, each application is instrumented for measuring its performance and energy consumption. The measured execution times and consumed energy in each run of the application are stored in the variables te_{cpu}, te_{gpu1}, te_{gpu2}, and $energy_{app}$, which are output (Sample algorithm 1).

We keep running the application until the sample means of the measured execution times and energy consumption of the application lie within a given confidence interval, and a given precision is achieved. For this, we employ a script, which is named MEANUSINGTTEST. Algorithm 2 presents the pseudocode of this script. It executes the application *app* repeatedly until one of the following three conditions is satisfied:

1) The maximum number of repetitions ($maxReps$) is exceeded (Line 4).
2) The sample means of all devices (kernel execution times and the application energy consumption) fall in the confidence interval (or the precision of measurement eps is achieved) (Lines 11-15).
3) The elapsed time of the repetitions of application execution exceeds the maximum allowed time ($maxT$ in seconds) (Lines 16-18).

MEANUSINGTTEST returns the sample means of the execution times for each abstract processor (i.e. $time_{cpu}$, $time_{gpu1}$, $time_{gpu2}$) and the energy consumption of the kernels (i.e. $energy$). The input parameters are the minimum and maximum number of repetitions, $minReps$, and $maxReps$. These parameter values differ based on the workload size solved. For small workload sizes ($32 \leq n \leq 1024$), these values are set to 10000 and 100000. For medium workload sizes ($1024 < n \leq 5120$), these values are set to 100 and 1000. For large workload sizes ($n > 5120$), these values are set to 5 and 50. The values of $maxT$, cl, and eps are set to 3600, 0.95, and 0.1. If the precision of measurement is not achieved before the maximum number of repeats is exceeded, we increase the number of repetitions and the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in the experiments.

Algorithm 2 Script determining the mean of an experimental run using student's t-test.

1: **procedure** MEANUSINGTTEST($app, minReps, maxReps, maxT, cl, eps, reps\#, elapsedTime, time_{cpu}, time_{gpu1}, time_{gpu2}, energy$)
Input:
The application to execute, app
The minimum number of repetitions, $minReps \in \mathbb{Z}_{>0}$
The maximum number of repetitions, $maxReps \in \mathbb{Z}_{>0}$
The maximum time allowed for the application to run, $maxT \in \mathbb{R}_{>0}$
The required confidence level, $cl \in \mathbb{R}_{>0}$
The required accuracy, $eps \in \mathbb{R}_{>0}$
Output:
The number of experimental runs actually made, $reps\# \in \mathbb{Z}_{>0}$
The elapsed time, $elapsedTime \in \mathbb{R}_{>0}$
The mean execution times, $time_{cpu}, time_{gpu1}, time_{gpu2} \in \mathbb{R}_{\geq 0}$
The mean consumed energy, $energy \in \mathbb{R}_{>0}$

2: $reps \leftarrow 0$; $stop \leftarrow 0$; $etime \leftarrow 0$
3: $sum_{cpu} \leftarrow 0$; $sum_{gpu1} \leftarrow 0$; $sum_{gpu2} \leftarrow 0$; $sum_{eng} \leftarrow 0$
4: **while** ($reps < maxReps$) and (!$stop$) **do**
5: $(t_{cpu}[reps], t_{gpu1}[reps], t_{gpu2}[reps], eng[reps]) \leftarrow$ EXECUTE(app)
6: $sum_{cpu} += t_{cpu}[reps]$
7: $sum_{gpu1} += t_{gpu1}[reps]$
8: $sum_{gpu2} += t_{gpu2}[reps]$
9: $sum_{eng} += eng[reps]$
10: **if** $reps > minReps$ **then**
11: $stop_{cpu} \leftarrow$ CALACCURACY($cl, reps + 1, t_{cpu}, eps$)
12: $stop_{gpu1} \leftarrow$ CALACCURACY($cl, reps + 1, t_{gpu1}, eps$)
13: $stop_{gpu2} \leftarrow$ CALACCURACY($cl, reps + 1, t_{gpu2}, eps$)
14: $stop_{eng} \leftarrow$ CALACCURACY($cl, reps + 1, t_{eng}, eps$)
15: $stop \leftarrow stop_{cpu} \wedge stop_{gpu1} \wedge stop_{gpu2} \wedge stop_{eng}$
16: **if** $\max\{sum_{cpu}, sum_{gpu1}, sum_{gpu2}\} > maxT$ **then**
17: $stop \leftarrow 1$
18: **end if**
19: **end if**
20: $reps \leftarrow reps + 1$
21: **end while**
22: $reps\# \leftarrow reps$
23: $elapsedTime \leftarrow \max\{sum_{cpu}, sum_{gpu1}, sum_{gpu2}\}$
24: $time_{cpu} \leftarrow \frac{sum_{cpu}}{reps}$; $time_{gpu1} \leftarrow \frac{sum_{gpu1}}{reps}$; $time_{gpu2} \leftarrow \frac{sum_{gpu2}}{reps}$
25: $energy \leftarrow \frac{sum_{eng}}{reps}$
26: **return** $(reps\#, elapsedTime, time_{cpu}, time_{gpu1}, time_{gpu2}, energy)$
27: **end procedure**

Algorithm 3 shows the pseudocode of the helper functions CALACCURACY, which is used by MEANUSINGTTEST. It returns one if the sample mean of a given reading lies in the 95% confidence interval (cl), and a precision of 0.025 ($eps = 2.5\%$) is achieved. Otherwise, it returns 0.

Algorithm 3 Algorithm Calculating Accuracy

1: **function** CALACCURACY($cl, reps, Array, eps$)
2: $clOut \leftarrow$ fabs(gsl_cdf_tdist_Pinv(cl, $reps - 1$)) × gsl_stats_sd($Array$, 1, $reps$) / sqrt($reps$)
3: **if** $clOut \times \frac{reps}{\sum_{i=0}^{reps-1} Array[i]} < eps$ **then**
4: **return** 1
5: **end if**
6: **return** 0
7: **end function**

If the precision of measurement is not achieved before the maximum number of repeats is exceeded, we increase the number of repetitions and the maximum elapsed time allowed.

However, we observed that condition (2) is always satisfied before the other two in the experiments.

B. Steps to Prevent Noise in Energy Measurements

Several steps are taken to measure energy consumption to eliminate any potential interference (noise) from components not involved in data transfer between the host CPU and a GPU.

Energy consumption of the data transfer between the host CPU and a GPU may include contributions from SSDs, NIC, and fans. Therefore, we ensure that these contributions are minimal using the steps below:

- The program performing the data transfer between the host CPU and a GPU does not invoke any file I/O functions. We monitor the disk consumption during the program run and ensure that negligible I/O is performed by the program using tools such as *sar* and *iotop*.
- The sizes of the data transferred between the host CPU and a GPU do not exceed the CPU's and GPU's main memory. Therefore, we ensure that no swapping (paging) occurs during the program run on the CPU side and that there are no failures on the GPU side.
- The program performing the data transfer between the host CPU and a GPU does not invoke any network I/O functions. We monitor the network activity using tools such as *sar* and *atop* and ensure it is not significant.

Fans are significant contributors to energy consumption. Our hybrid server platform controls fans in zones: a) zone 0: CPU or System fans, b) zone 1: Peripheral zone fans. There are four levels to control the speed of fans:

- Standard: Baseboard management controller (BMC) controls both fan zones, with the CPU and Peripheral zones set at speed 50%;
- Optimal: BMC sets the CPU zone at speed 30% and the Peripheral zone at 30%;
- Heavy IO: BMC sets the CPU zone at speed 50% and the Peripheral zone at 75%;
- Full: All fans running at 100%.

To rule out fans' contribution to dynamic energy consumption, we set the fans at full speed before running the programs. Therefore, the energy consumption by fans is included in the static power consumption of the server.

Furthermore, during the application run, we monitor the server's temperatures and the fans' speeds with the help of Intelligent Platform Management Interface (IPMI) sensors. We make sure that there are no changes in the temperatures and the fans' speeds.

VII. Algorithm for Merging Piecewise Linear Pareto Fronts

We now describe our continuous algorithm *MergeParetoFrontsC* to determine the globally optimal Pareto front given k piecewise linear Pareto fronts. *MergeParetoFrontsC* outputs two Pareto fronts. One is a globally optimal Pareto front, which can be discontinuous and contains breaks. The other is a locally optimal Pareto front containing the globally optimal Pareto front with breaks filled by locally Pareto-optimal solutions.

We illustrate the algorithm using Figure 1 for the case of three heterogeneous processors solving a workload. We consider a small set of nine platform configurations for clarity in our exposition. Therefore, the input to the algorithm is a set of nine locally optimal Pareto fronts. Each Pareto front corresponding to a platform configuration is piecewise linear comprising two linear segments.

The algorithm commences from the left by finding the point with the least execution time. If there is more than one such point, it selects one with the least dynamic energy. Figure 1a shows this point as a green sphere. The algorithm then draws a vertical line passing through this point. It moves this vertical line to the right continuously.

At each instant, it finds the Pareto-optimal point on the vertical line, which is the lowest point on the line. It draws a horizontal line intersecting the vertical line at the Pareto-optimal point. The two lines divide the objective space into four disjoint quadrants, I, II, III, and IV. By the definition of Pareto optimality, quadrant I will be empty. Quadrant III comprises sub-optimal points that are removed. Figure 1a shows the sub-optimal points in quadrant III removed in Figure 1b. Quadrant IV will contain the partially built globally optimal Pareto front. Figure 1b shows the objective space at the end of an instant.

Therefore, the algorithm proceeds in this fashion, pruning the objective space at each instant. Figures 1c-1h show the different instants of the algorithm's progress. From Figure 1h onwards, the origin is shifted upwards (axes scales are reduced)) to show the Pareto fronts in better detail. The algorithm terminates in Figure 1h where there are no points to explore in quadrant II. Quadrant IV contains the final globally optimal Pareto front. It is discontinuous with breaks. The algorithm fills the breaks with locally Pareto optimal solutions. We do not present how this filling is done in this paper due to space limitations.

VIII. Formulation of the Discrete Optimization Problem and An Exact Algorithm Solving the Problem

This section formulates HEPOPTGD, the discrete version of the bi-objective optimization problem minimizing the execution time and energy consumption of parallel applications executing on a heterogeneous hybrid parallel computers comprising p heterogeneous processors through optimal combination of platform and application configurations.

The problem has two decision variables: a vector of p processor configurations (platform configuration) and a vector of p workload sizes (workload distribution representing the application configuration). The workload size solved by a processor is assumed to be a multiple of a basic computation unit that does not differ during the application execution.

HEPOPTGD considers the execution of an application workload of size $n \in \mathbb{Z}_{\geq 0}$ on a heterogeneous hybrid parallel computers with base/idle power consumption, P_s, and comprising p heterogeneous processors. The set, $\mathbb{M}_{\mathbb{D}}$, contains the processor models, which are discrete execution time and dynamic energy functions (with arbitrary shape and

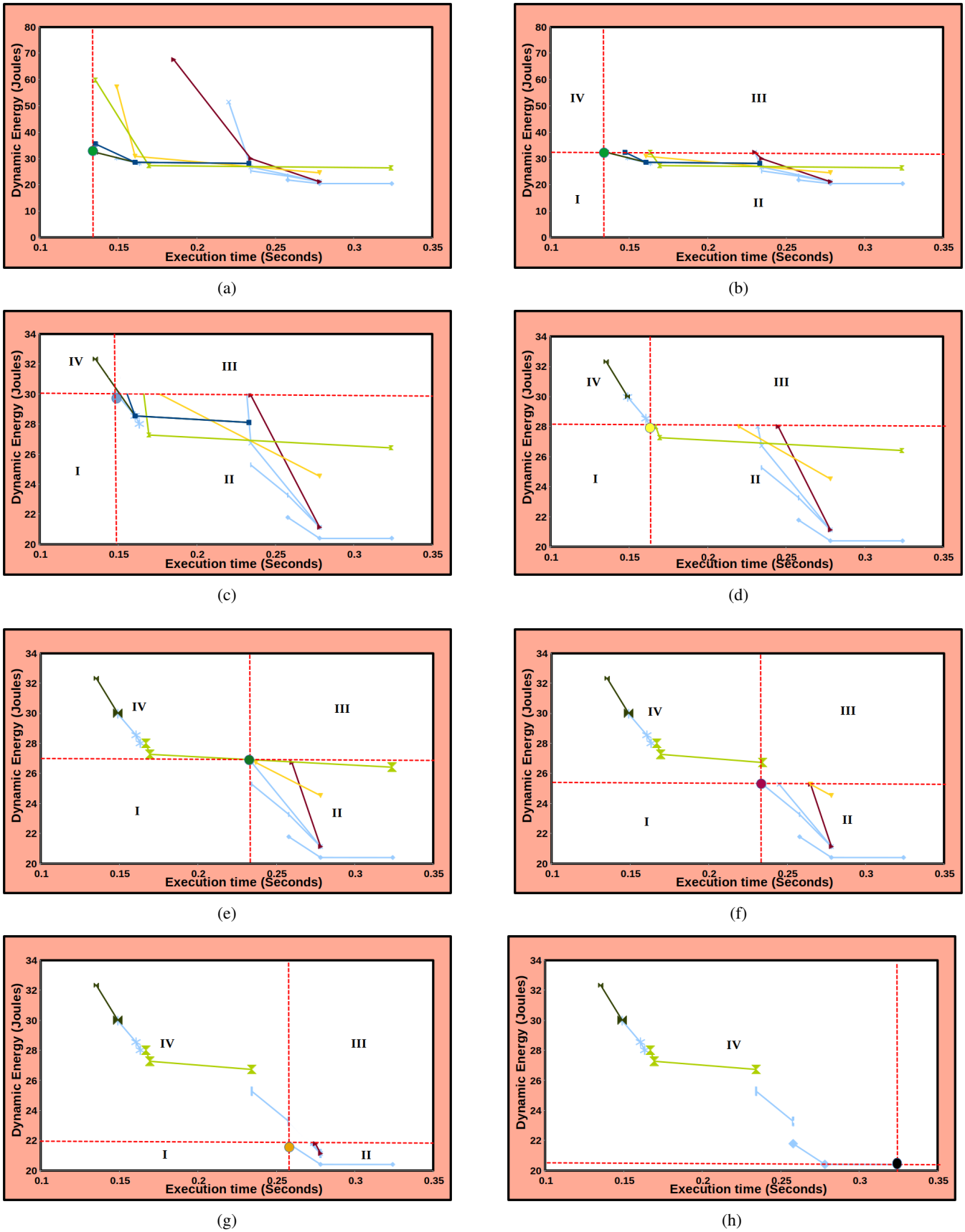


Fig. 1: Illustration of the continuous algorithm to determine the globally optimal Pareto front given a set of piecewise linear Pareto fronts. The algorithm commences by finding the point with the least execution time represented by a green sphere. It draws a vertical line through this point that is moved continuously to the right. At each instant, a horizontal line is drawn intersecting the vertical line in a Pareto-optimal point. The two lines allow the objective space to be pruned. Quadrant IV will contain the globally optimal Pareto front at the end of each instant. The figure is zoomed in (c) by shifting the origin upwards to show the Pareto fronts in better detail. The figures from (c) to (h) show the explored and unexplored objective space at the end of different instants in the execution of the algorithm. Figure (h) shows the final globally optimal Pareto front.

represented by a set of points) of workload size. Specifically, the sets T_i and E_i contain c_i discrete execution time and dynamic energy functions of processor i where the function, $e_{ij}(w) : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$, $j \in \{1, \cdots, c_i\}$ gives the amount of dynamic energy consumed by P_i to execute the workload size $w \in \mathbb{Z}_{\geq 0}$ employing the processor configuration, j, and $t_{ij}(w) : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ gives the execution time.

HEPOPTGD is formulated as follows:

HEPOPTGD$(n, p, \mathbb{C}, \mathbb{M}_{\mathbb{D}}, P_s)$:

$$G_T(\sigma, \mathbb{W}) = \max_{i=1}^{p} t_{i\sigma_i}(w_{i\sigma_i})$$

$$G_E(\sigma, \mathbb{W}) = \sum_{i=1}^{p}(P_s \times t_{i\sigma_i}(w_{i\sigma_i}) + \sum_{i=1}^{p} e_{i\sigma_i}(w_{i\sigma_i}))$$

$$\underset{(\sigma, \mathbb{W})}{\textbf{minimize}} \quad \{G_T(\sigma, \mathbb{W}), G_E(\sigma, \mathbb{W})\}$$

$$\textbf{subject to:} \sum_{i=1}^{p} w_{i\sigma_i} = n, w_{i\sigma_i} \in \mathbb{Z}_{\geq 0}$$
$$i \in \{1, \cdots, p\}, \sigma_i \in \{1, \cdots, c_i\}$$

The two objective functions are $G_T(\sigma, \mathbb{W})$ and $G_E(\sigma, \mathbb{W})$. The objective function $G_T(\sigma, \mathbb{W})$ gives the parallel execution time of the workload employing the platform and application configurations, $\{(\sigma_1, w_{i\sigma_1}), \cdots, (\sigma_p, w_{i\sigma_p})\}$ where $\{\sigma_1, \cdots, \sigma_p\}$ represents the platform configuration and $\{w_{1\sigma_1}, \cdots, w_{p\sigma_p}\}$ is the workload distribution signifying the application configuration. The objective function $G_E(\sigma, \mathbb{W})$ gives the total energy consumption during the execution of the workload. $G_T \times G_E : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}$ denotes the objective space of this problem.

HEPOPTGD returns a discrete set of globally Pareto-optimal solutions for execution time and total energy. The set contains elements with four components, $\{G_T(\sigma_{opt}, \mathbb{W}_{opt}), G_E(\sigma_{opt}, \mathbb{W}_{opt}), \sigma_{opt}, \mathbb{W}_{opt}\}$, where the first two components are two scalars giving the optimal execution time and total energy and following two represent the Pareto-optimal decision variable. The projection of the set onto the objective space gives the Pareto front. The Pareto-optimal decision variable, $\{\sigma_{opt}, \mathbb{W}_{opt}\}$, represents a set of pairs, $\{(\sigma_{opt_1}, w_{1\sigma_{opt_1}}), \cdots, (\sigma_{opt_p}, w_{p\sigma_{opt_p}})\}$, where the pair $(\sigma_{opt_i}, w_{i\sigma_{opt_i}})$ gives the optimal processor and application configurations for processor i. If the input parameter P_s is 0, HEPOPTGD returns a set of globally Pareto-optimal solutions for execution time and dynamic energy.

A. Algorithm Solving the Discrete Optimization Problem

We propose an algorithm HEPOPTAGD that solves HEPOPTGD. Unlike the two sub-algorithms proposed for solving HEPOPTGC, HEPOPTAGD is a single algorithm that outputs a discrete set of points (globally optimal Pareto front) where each point comprises the optimal platform and application configurations and the optimal objective function values.

The inputs to HEPOPTAGD are the workload size, n; the number of heterogeneous processors, p; the set $\mathbb{C} = \{c_1, \cdots, c_p\}$ where c_i is the number of configurations of the processor i; the functional models of the processors, $\mathbb{M}_{\mathbb{D}}$; the static power consumption of the hybrid server comprising the p processors, P_s.

HEPOPTAGD outputs a discrete set, $\mathbb{P}$, containing elements with four components, $\{G_T(\sigma_{opt}, \mathbb{W}_{opt}), G_E(\sigma_{opt}, \mathbb{W}_{opt}), \sigma_{opt}, \mathbb{W}_{opt}\}$. The globally Pareto-optimal decision variable vector, $(\sigma_{opt}, \mathbb{W}_{opt})$, is a set of tuples, $\{(\sigma_{opt_1}, w_{1\sigma_{opt_1}}), \cdots, (\sigma_{opt_p}, w_{p\sigma_{opt_p}})\}, \sigma_{opt_i} \in \{1, \cdots, c_i\}$, giving the optimal platform and application configurations. The pair $(\sigma_{opt_i}, w_{i\sigma_{opt_i}})$ gives the processor and application configurations for processor i. $G_T(\sigma_{opt}, \mathbb{W}_{opt})$ and $G_E(\sigma_{opt}, \mathbb{W}_{opt})$ are the objective function values associated with the globally Pareto-optimal decision variable vector, $(\sigma_{opt}, \mathbb{W}_{opt})$.

Algorithm 4 illustrates the pseudocode of HEPOPTAGD.

Algorithm 4 HEPOPTAGD: Algorithm solving the optimization problem HEPOPTGD for worload size n using p heterogeneous processors represented by models, $\mathbb{M}_{\mathbb{D}}$, and $\mathbb{C}$ containing the number of processor configurations.

1: **procedure** HEPOPTAGD(n, p, $\mathbb{C}$, $\mathbb{M}_{\mathbb{D}}$, P_s)
Input:
The workload size, n
The number of heterogeneous processors, p
The set containing the number of configurations of the processors, $\mathbb{C} = \{c_1, \cdots, c_p\}$
The discrete functional models of the processors, $\mathbb{M}_{\mathbb{D}} = \{(T_{d_1}, E_{d_1}), \cdots, (T_{d_p}, E_{d_p})\}$
The static power consumption, P_s
Output:
The globally optimal Pareto front, $\mathbb{P}$

2: $\mathbb{O} \leftarrow \{\}$
3: **for all** $\sigma \in \mathbb{G}$ **do**
4: $\mathbb{T}_{d\sigma} \leftarrow \{t_{1\sigma_1}, \cdots, t_{p\sigma_p}\}$
5: $\mathbb{E}_{d\sigma} \leftarrow \{e_{1\sigma_1}, \cdots, e_{p\sigma_p}\}$
6: $\mathbb{O} \leftarrow \mathbb{O} \cup$ **HEPOPTA**$(n, p, \mathbb{T}_{d\sigma}, \mathbb{E}_{d\sigma}, P_s)$
7: **end for**
8: **return MergeParetoFrontsD**$(\mathbb{O})$
9: **end procedure**

Like HEPOPTAGC, HEPOPTAGD loops over the set, $\mathbb{G}$ (Line 2). The set $\mathbb{G}$ of size $c_1 \times c_2 \times \cdots c_p$ comprises all platform configurations. Each platform configuration is represented by the vector σ of size p.

HEPOPTAGD then invokes an algorithm *HEPOPTA* [11] (Line 6) with the inputs, the workload size, n; the number of heterogeneous processors, p; the sets, $\mathbb{T}_{d\sigma} = \{t_{1\sigma_1}(w), \cdots, t_{p\sigma_p}(w)\}$ and $\mathbb{E}_{d\sigma} = \{e_{1\sigma_1}(w), \cdots, e_{p\sigma_p}(w)\}$, containing the p discrete execution time and dynamic energy functions of workload size; and the static power consumption of the hybrid server, P_s. The output of HEPOPTA is a discrete Pareto front given by a set containing elements with four components, $G_E(\sigma, \mathbb{W}_{opt}), G_T(\sigma, \mathbb{W}_{opt}), \sigma$ and $\mathbb{W}_{opt}$, where $\mathbb{W}_{opt}$ is a vector of size p giving the optimal workload distribution, $G_E(\sigma, \mathbb{W}_{opt})$ is the optimal total energy, and $G_T(\sigma, \mathbb{W}_{opt})$ is the optimal execution time for a given platform configuration, σ. HEPOPTA is detailed below.

The Pareto front obtained for each platform configuration is inserted into the set, $\mathbb{O}$ (Line 6). Finally, the routine *MergeParetoFrontsD* is invoked to determine the globally optimal Pareto front given the Pareto fronts in $\mathbb{O}$ (Line 7).

B. Algorithm for Merging Discrete Pareto Fronts

The inputs to *MergeParetoFrontsD* are k Pareto fronts in the set $\mathbb{S} = \{s_1, \cdots, s_k\}$. Each Pareto front is set of (e, t) pairs. *MergeParetoFrontsD* outputs the Pareto Front of non-dominated (e, t) pairs in $\mathbb{P}$.

Algorithm 5 MergeParetoFrontsD: Algorithm determining globally optimal Pareto front given k discrete Pareto fronts.

1: **procedure** HEPOPTAGD(k, $\mathbb{S}$)
Input:
The number of discrete Pareto fronts, k
The set of Pareto fronts, $\mathbb{S} = \{s_1, \cdots, s_k\}$
Output:
The merged final Pareto front, $\mathbb{P}$

2: $\mathbb{U}, \mathbb{P} \leftarrow \Phi$
3: **for all** $i \leftarrow 1, k$ **do**
4: $\mathbb{U} \leftarrow \mathbb{U} \cup s_i$
5: **end for**
6: $Sort_{\downarrow t}(\mathbb{U})$
7: $\mathbb{P} \leftarrow \mathbb{P} \cup \mathbb{U}[1]$
8: $tmpe \leftarrow \mathbb{U}[1].e$; $tmpt \leftarrow \mathbb{U}[1].t$
9: **for all** $i \leftarrow 2, |\mathbb{U}|$ **do**
10: **if** $(U[i].e < tmpe)$ **then**
11: **if** $(U[i].t == tmpt)$ **then**
12: $\mathbb{P} \leftarrow \mathbb{P} - (tmpe, tmpt)$
13: **end if**
14: $\mathbb{P} \leftarrow \mathbb{P} \cup \mathbb{U}[i]$
15: $tmpe \leftarrow \mathbb{U}[i].e$; $tmpt \leftarrow \mathbb{U}[i].t$
16: **end if**
17: **end for**
18: **return** $\mathbb{P}$
19: **end procedure**

The pseudocode of *MergeParetoFrontsD* is given in Algorithm 5. In Lines 3-5, the elements of all the input sets are inserted into a single set, $\mathbb{U}$. The set $\mathbb{U}$ is then sorted in ascending order of t values (Line 6). In Lines 9-17, the set $\mathbb{U}$ is iterated and all dominating solutions are added to the set, $\mathbb{P}$. The two conditions in Lines 10 and 11 ensure that if there are several points with the same execution time (t), then the point with the least energy is selected. Finally, the algorithm outputs the Pareto front, $\mathbb{P}$, in Line 18.

C. HEPOPTA: A Solution Method Solving the Bi-objective Optimization Problem for a Given Platform Configuration

HEPOPTA is a branch-and-bound algorithm that organizes workload distributions, $\{x_1, \cdots, x_p\}$, efficiently as a tree while considering only distributions that sum to n to determine the Pareto front [11].

The inputs to HEPOPTA are the workload size, n; the number of heterogeneous processors, p; the sets, $\mathbb{T}_{d\sigma} = \{t_{1\sigma_1}(w), \cdots, t_{p\sigma_p}(w)\}$ and $\mathbb{E}_{d\sigma} = \{e_{1\sigma_1}(w), \cdots, e_{p\sigma_p}(w)\}$, containing the p discrete execution time and dynamic energy functions of workload size; and the static power consumption of the hybrid server, P_s. The output of HEPOPTA is a discrete Pareto front given by a set containing elements with four components, $G_E(\sigma, \mathbb{W}_{opt}), G_T(\sigma, \mathbb{W}_{opt}), \sigma$ and $\mathbb{W}_{opt}$, where $\mathbb{W}_{opt}$ is a vector of size p giving the optimal workload distribution, $G_E(\sigma, \mathbb{W}_{opt})$ is the optimal total energy, and $G_T(\sigma, \mathbb{W}_{opt})$ is the optimal execution time for a given platform configuration, σ. The solutions in the Pareto front are arranged in an increasing order of total energy. Assume that $m = \max_{i=0}^{k} m_i$ is the maximum of the cardinalities of the sets in $\mathbb{T}_{d\sigma}$.

The building block of HEPOPTA is HDePOPTA, which solves the bi-objective optimization problem for performance and *dynamic* energy. We describe HDePOPTA using a simple example. Suppose there are four heterogeneous processors ($p = 4$) executing a given workload size $n = 4$. The other inputs are four discrete dynamic energy functions, $E = \{e_0(x), \cdots, e_3(x)\}$, as well as four discrete time functions, $T = \{t_0(x), \cdots, t_3(x)\}$, shown in Figure 2. The functions are sorted by dynamic energy in non-decreasing order.

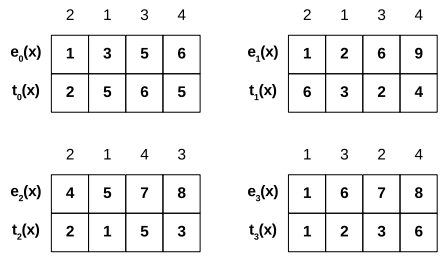


Fig. 2: Sample dynamic energy and execution time functions sorted in non-decreasing order of dynamic energies.

A naive algorithm explores the full solution tree and finds all possible workload distributions to construct the Pareto front for execution time and dynamic energy. Figure 3 shows the tree constructed by such an algorithm. Due to lack of space, only a partial tree is shown.

The tree consist of 4 levels $\{L_0, L_1, L_2, L_3\}$ where all workload sizes given to processor P_i are examined in level L_i. Each node in L_i, $i \in \{0, 1, 2, 3\}$, is labelled by a positive value representing the workload size that is distributed between processors $\{P_i, \cdots, P_3\}$. Each edge connecting a node at level L_i to its ancestor is labelled by a triple $(w, e_i(w), t_i(w))$ where w is the workload size assigned to P_i, along with its consumed dynamic energy $(e_i(w))$ and its execution time $(t_i(w))$.

The exploration process begins from the root to find all distributions for the workload size four between four processors $\{P_0, P_1, P_2, P_3\}$. Five workload sizes, including all data points in the function $e_0(x)$ and a zero workload size, are assigned to the processor P_0 one after another. Assigning the workload sizes $\{0, 2, 1, 3, 4\}$ to P_0 expands the root into 5 children at L_1 representing the remaining workload to be distributed between processors $\{P_1, P_2, P_3\}$. For instance, the edge $(2, 1, 2)$, highlighted in blue in Figure 3, indicates that a workload size 2 with a dynamic energy consumption of 1 and

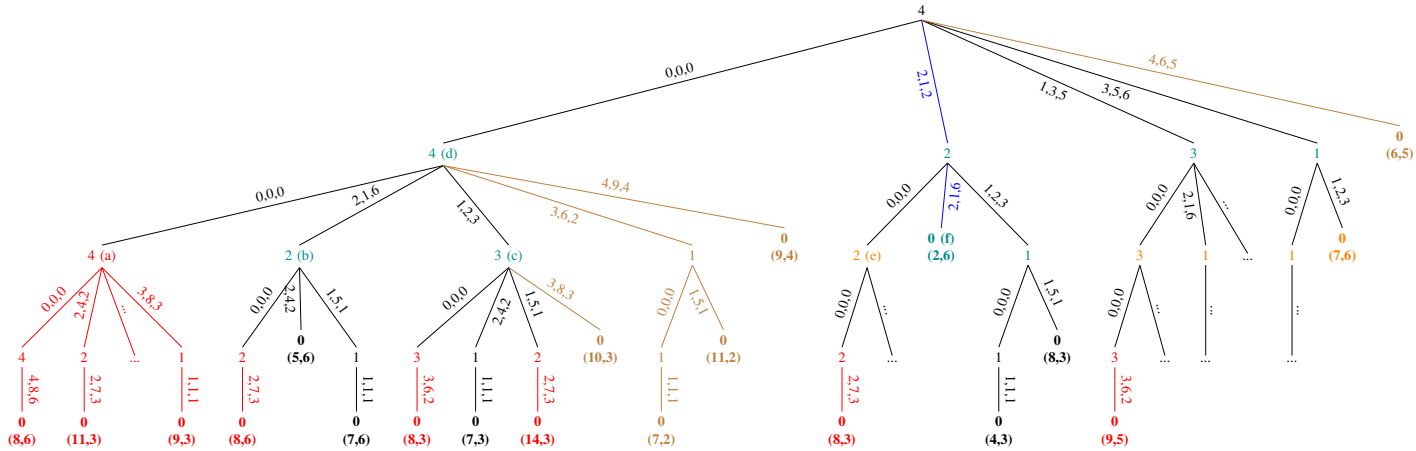


Fig. 3: For a workload $n = 4$ on four processors, the naive algorithm explores the full solution tree to find all the workload distributions to construct the Pareto front.

an execution time of 2 is given to P_0, and its child is labelled by 2 which equals the remaining size distributed at the level L_1.

In the same manner, each node in levels $\{L_1, L_2, L_3\}$ is expanded towards the leaves. Any leaf node, labelled by 0, illustrates a solution that its dynamic energy consumption is the summation of dynamic energy consumptions, and its execution time is the maximum of the execution times labelling the edges in the path from the root to the leaf. For example, the blue path $\{(2,1,2),(2,1,6)\}$ in the tree highlights a solution distributing the workload 4 on two processors P_0 and P_1. The dynamic energy consumption of this workload distribution is 2 $(= 1 + 1)$, and the corresponding execution time is 6 $(= \max\{2,6\})$. The other two processors $\{P_2, P_3\}$ are assigned a zero workload size.

Due to lack of space, we have not shown the branches that do not provide any solution. In a non-solution branch, the summation of workload sizes labelling the edges from the root to its leaf is greater than 4.

In this example, each internal node in the solution tree has either 5 children (or $m+1$ in the general case) or just one child in which case the child is always a leaf. There are two types of leaves: *solution* leaves, labelled by 0 along with its dynamic energy consumption and execution time beneath it, and *no-solution* leaves, eliminated from, and therefore, not shown in the tree. Each internal node at level L_i, labelled by a positive number w, becomes a root of a solution tree for distribution of the workload w between processors $\{P_i, \cdots, P_3\}$ and is therefore constructed recursively.

Once a solution is found, the algorithm updates the Pareto front. In the end, the Pareto front includes three members, $\{(\langle 2,6\rangle, \{2,2,0,0\}), (\langle 4,3\rangle, \{2,1,0,1\}), (\langle 5,2\rangle, \{2,0,2,0\})\}$. The Pareto front member, $(eng, time, \{x_0, \cdots, x_3\})$, gives the dynamic energy consumption (eng) and the execution time $(time)$ of the workload distribution $\{x_0, \cdots, x_3\}$.

The naive algorithm has exponential complexity. HDePOPTA is an efficient recursive algorithm to determine the Pareto-optimal solutions for data-parallel applications executing on heterogeneous processors. It has polynomial time complexity. HDePOPTA reduces the search space by utilizing three optimizations to avoid exploring whole subtrees in the solution tree. The optimizations are a). *Cut*, b). *SavePareto*, and c). *ReadParetoMem*. HDePOPTA eliminates sub-trees from the search space using the operation *Cut* that employs two upper bounds for energy and workload size. It memorizes solutions for each node using *SavePareto* and retrieves the saved solutions using *ReadParetoMem*.

HEPOPTA takes as input the Pareto front for execution time and dynamic energy determined by HDePOPTA and finds the Pareto front for execution time and total energy. The time complexity of both HDePOPTA and HEPOPTA is $\mathcal{O}(m^3 \times p^3 \times \log_2(m \times p))$.

D. Complexity of Algorithm for Merging Discrete Pareto Fronts

Proposition VIII.1. *Consider k sets with cardinalities $\{m_1, \cdots, m_k\}$, $m = \max_{i=0}^{k} m_i$ and containing (e,t) pairs. The time complexity of MergeParetoFrontsD is $\mathcal{O}(k \times m \times \log_2(k \times m))$.*

Proof. The time complexity of loop in Lines 2-4 is $\mathcal{O}(k \times m)$. The time complexity of *sort* in Line 5 is $\mathcal{O}(k \times m \times \log_2(k \times m))$. The time complexity of loop in Lines 8-16 is $\mathcal{O}(k \times m)$.

Therefore, the time complexity of *MergeParetoFrontsD* is $\mathcal{O}(k \times m \times \log_2(k \times m))$. □

E. Complexity of HEPOPTAGD

Proposition VIII.2. *The time complexity of HEPOPTAGD is $\mathcal{O}(\mathbb{L}^p \times m^3 \times p^3 \times \log_2(m \times p))$ where $m = \max_{i=0}^{k} m_i$ is the maximum of the cardinalities of the sets in $\mathbb{T}_{d\sigma}$ and $\mathbb{L} = \max(c_1, \cdots, c_p)$.*

Proof. Consider the loop (Lines 2-6). The number of loop iterations is equal to $c_1 \times c_2 \times \cdots c_p$ (size of $\mathbb{G}$). The time complexity of HEPOPTA is $\mathcal{O}(m^3 \times p^3 \times \log_2(m \times p))$. Therefore, the time complexity of the loop is $\mathcal{O}(c_1 \times c_2 \times \cdots c_p \times m^3 \times p^3 \times \log_2(m \times p)) = \mathcal{O}(\mathbb{L}^p \times m^3 \times p^3 \times \log_2(m \times p))$.

The time complexity of *MergeParetoFrontsD* is $\mathcal{O}(c_1 \times c_2 \times \cdots c_p \times m \times p \times \log_2(c_1 \times c_2 \times \cdots c_p \times m \times p)) = \mathcal{O}(\mathbb{L}^p \times m \times p^2 \times \log_2(\mathbb{L} \times m \times p))$ since the maximum number of Pareto-optimal solutions for each platform configuration is equal to $m \times p$ (the maximum number of distinct execution times in the input).

Therefore, the time complexity of HEPOPTAGD is dominated by the HEOPTA invocations in the main loop and is given by $\mathcal{O}(\mathbb{L}^p \times m^3 \times p^3 \times \log_2(m \times p))$. □

F. Scalability and Search-Space Reduction

HEPOPTAGC and HEPOPTAGD construct one local Pareto front for each platform configuration and then merge these local fronts to obtain the global Pareto front. Since the set of platform configurations has size $c_1 \times c_2 \times \cdots \times c_p$, where c_i is the number of configurations of processor i, the number of local fronts grows with both the number of processors and the number of configurations per processor. Therefore, the search space may become large when many DVFS states or many processors are considered. In the discrete case, the number of local Pareto points also depends on the cardinality of the discrete workload-size models, which is affected by the workload size and the chosen discretization granularity.

In practice, this cost can be reduced by filtering the configuration set before optimization. DVFS states can be limited to user-controllable and application-relevant operating points. Configurations that are dominated over the profiled workload range can also be removed when such dominance can be established. User constraints, such as a maximum execution time, an energy budget, or a restricted frequency range, can further reduce the feasible configuration set. These reductions change the configuration set considered by the algorithm; therefore, exactness is preserved with respect to the reduced or constrained configuration set.

Moreover, the local Pareto front for each platform configuration is constructed independently. Thus, the loop over platform configurations is naturally parallelizable. This makes the exact method suitable for offline profiling and optimization workflows. For very large systems, applying the same algorithms to a reduced configuration set lowers the computational cost while preserving exactness with respect to that reduced set.

G. Robustness to Modeling Inaccuracies

The proposed optimization algorithms are exact with respect to the supplied execution-time and dynamic-energy functions. Therefore, their optimality guarantees hold for the input models used by the algorithms. Modeling inaccuracies may change the estimated objective values and may affect the ordering of closely located Pareto-optimal solutions in the time–energy space. In such cases, the point selected from the estimated Pareto front may differ from the point that would be selected using exact measurements, while still providing a nearby trade-off.

However, modeling errors do not make the generated platform configurations or workload distributions infeasible, since feasibility is determined by the workload-partitioning constraints and the available processor configurations. Thus, the main effect of modeling error is on the ranking and selection of Pareto-optimal trade-offs, rather than on the validity of the generated configurations.

This effect is mitigated by statistically reliable profiling, validation against ground-truth measurements, and, in practical use, by selecting Pareto-optimal points with sufficient time or energy margins when such margins are required. In our methodology, execution time and energy measurements are obtained from repeated runs using Student's t-test until the required confidence and precision are achieved. The scaling models are also validated against ground-truth measurements across multiple workloads and frequency levels.

IX. Efficient Construction of Models of Parallel Applications for Processor Configurations Represented by DVFS State

A. *Scaling Solution for Processor Consisting of One Independently-powered Processor Component*

we propose the following circuit-level Dennard scaling power model, an extension of the original Dennard scaling model for the dynamic power consumption of a CMOS transistor:

$$P_{avg}(w, V, f) = C(w) \times V^2 \times f, \tag{2}$$

where $P_{avg}(w, V, f)$ is the average dynamic power consumption of the processor during the execution of the application workload size, w, V and f are the operating voltage and frequency of the processor component. $C(w)$ is the workload-specific dynamic effective capacitance and signifies the cumulative activity of CMOS gates during the execution of an application workload.

We perform extensive experiments with several different application workloads to validate our hypothesis. Figure 4 shows the plot of average dynamic power consumption against $V^2 \times f$ for workloads of two multithreaded applications based on 2D fast Fourier transform (2D-FFT) and matrix multiplication (DGEMM). The 2D-FFT application executes on an Intel Icelake multicore processor comprising 64 logical cores. It employs 64 OpenMP threads and computes a 2D fast Fourier transform of a dense square signal matrix. The DGEMM application employs 64 threads and multiplies two dense matrices of size $N \times N$.

We use the Likwid tool [8] to collect the voltage readings from the logical CPU cores on our heterogeneous platforms. Likwid is a well-established performance profiling and monitoring tool that provides access to low-level hardware events, including cache misses, branch mispredictions, and floating-point operations. The voltage readings are obtained through the performance event labeled *VOLTAGE_CORE*, which is part of a performance monitoring group known as *VTG0*.

As model (2) assumes deterministic execution of the application workload, the experimental setup is designed to make the execution of the applications as deterministic as possible. The application workload is equally distributed among the threads. There is no communication or synchronization between the threads. The mapping of the threads to the cores

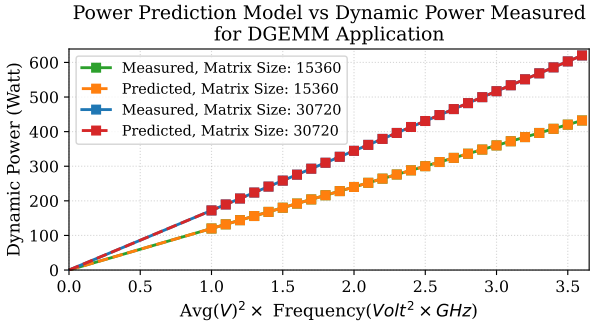
is fixed and does not change neither during the execution nor between different executions.

The above assumption may be less accurate in real systems when the execution is affected by non-deterministic OS activity, memory stalls, irregular synchronization, significant uncore activity, or thermal throttling. In such cases, the workload-dependent parameters may vary across DVFS states, and the scaling method should be applied only after workload-specific profiling and validation. In our experiments, these effects are reduced by using a fixed workload configuration and fixed thread mapping, avoiding paging/swapping, monitoring disk and network activity, setting fans to full speed, and monitoring temperatures and fan speeds during execution.

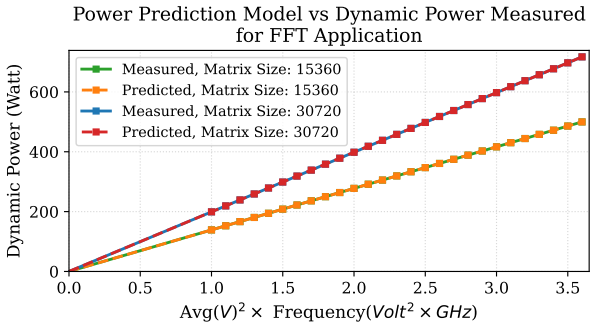
The DVFS state of the multicore processor is set to maintain uniform frequencies (f) across all the processor cores. The values of f are systematically varied from 800 MHz to 3.60 GHz in increments of 100 MHz, providing a comprehensive range of data for analysis.

The plots demonstrate that the average dynamic power consumption is linearly proportional to $V^2 \times f$.

Figure 5 shows the plots of execution time against the reciprocal of frequency, $\frac{1}{f}$, for workloads of the four applications. The DVFS state of the multicore processor is set to maintain uniform frequencies (f) across all the processor cores. The values of f are varied from 3.60 GHz to 800 MHz in decrements of 100MHz. The plots clearly demonstrate that the execution time is linearly proportional to the reciprocal of the processor core frequency, $\frac{1}{f}$.



(a) DGEMM



(b) 2D-FFT

Fig. 4: Average dynamic power consumption against $V^2 \times f$ for two workloads of each of two multithreaded applications executing on an Intel Icelake multicore processor comprising 64 logical cores. (a) DGEMM: two workloads with square matrix sizes 15360×15360 and 30720×30720. (b) 2D-FFT: two workloads with signal matrix sizes 15360×15360 and 30720×30720. All the processor cores are set to the same frequency f. The values of f are varied from 800 MHz to 3.60 GHz in increments of 100MHz, and the dots in the graphs represent experimental points.

B. Application of the Scaling Solution for Processor with One Indepenedently-powered Component to Multicore CPU Processors

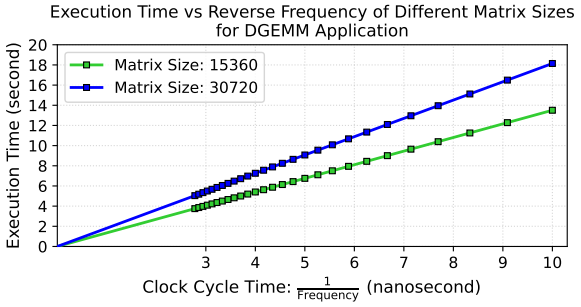
We experimentally validate our scaling models and method for an Intel multicore CPU processor. The dynamic energy consumption for the multicore CPU processor is obtained using the *ground-truth* method. The execution time is measured using processor clocks. The processor voltage is obtained using the performance monitoring counters provided for core voltages by the Likwid performance monitoring tool [8].

Figure 6 shows the plots of dynamic energy consumption estimated by the scaling method against the ground-truth method for several workloads of four multithreaded applications DGEMM, and 2D-FFT on an Intel Icelake multicore processor comprising 64 logical cores. The application configurations employ 64 threads. The DVFS state of the processor has all the processor cores set to the same frequency, f. The plots are shown for three frequencies (1.4, 1.6, and 1.8 GHz). The estimation accuracy of the scaling method is close to 100% for the DGEMM workloads, in the range of 95-98% for the 2D-FFT workloads. The estimation accuracy is representative of the other frequencies.

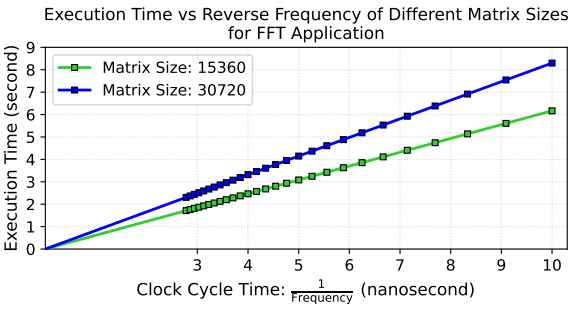
Figure 7 shows the plots of execution time estimated by the scaling method against the measured execution time for a range of workloads of four multithreaded applications DGEMM, and 2D-FFT on an Intel Icelake multicore processor comprising 64 logical cores. The application configurations employ 64 threads. The DVFS state of the processor has all the processor cores set to the same frequency, f. The plots are shown for three frequencies (1.4, 1.6, and 1.8 GHz). The estimation accuracy of the scaling method is close to 100% for the DGEMM workloads, in the range of 94-98% for the 2D-FFT workloads. The estimation accuracy is representative of the other frequencies.

C. Scaling Solution for Processor with Two Independently-powered Processor Components, only One of Which is Scaled

While we can directly apply the scaling solution developed for a processor with one indepenedently powered component to mainstream multicore CPUs, it will not work for mainstream GPUs. The Nvidia A40 GPU processor (Table I) used in our experiments comprises a graphics component with 144 Streaming Multiprocessors (SMs) and a main memory (GDDR6 SRAM) component. Both components are independently powered with different supply voltages (V_g and V_{mem}). V_{mem} is 1.35 V and does not vary. The main memory supports

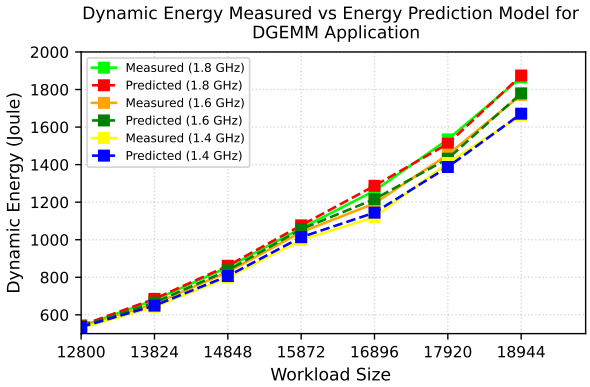


(a) DGEMM

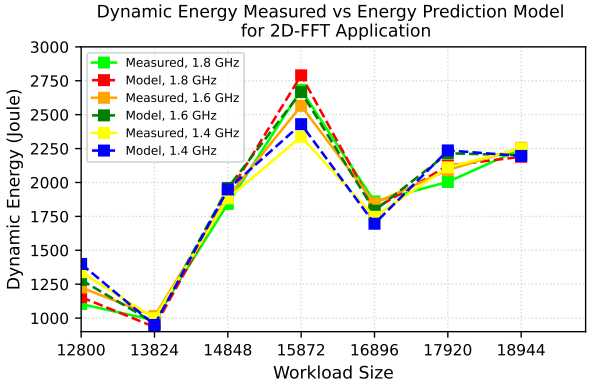


(b) 2D-FFT

Fig. 5: Execution time vs clock period, the reciprocal of frequency, for two workloads of each of two multithreaded applications on the Intel Icelake multicore processor comprising 64 logical cores. (a) DGEMM: two workloads with square matrix sizes 15360×15360 and 30720×30720. (b) 2D-FFT: two workloads with signal matrix sizes 15360×15360 and 30720×30720. All the processor cores are set to the same frequency, f. The values of f are varied from 3.60 GHz to 800 MHz in decrements of 100MHz, and the dots on the graphs represent experimental points. The execution time of the workload is a linear function of the reciprocal of f.



(a) DGEMM



(b) 2D-FFT

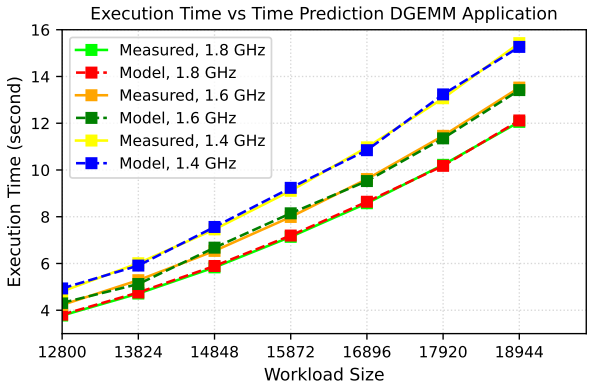
Fig. 6: Experimentally built and calculated dynamic energy profiles of two multithreaded applications on the Intel Icelake CPU, shown for three frequencies (1.4, 1.6, and 1.8 GHz). (a) DGEMM multiplies two dense square matrices of size $N \times N$ where $N \in \{12800, 13824, 14848, 15872, 16896, 17920, 18944\}$. (b) 2D-FFT computes a 2D fast Fourier transform of a dense signal matrix of size $N \times N$ for the same values of N.

two (V_{mem}, f_{mem}) combinations, where the frequencies are 7251 MHz and 405 MHz.

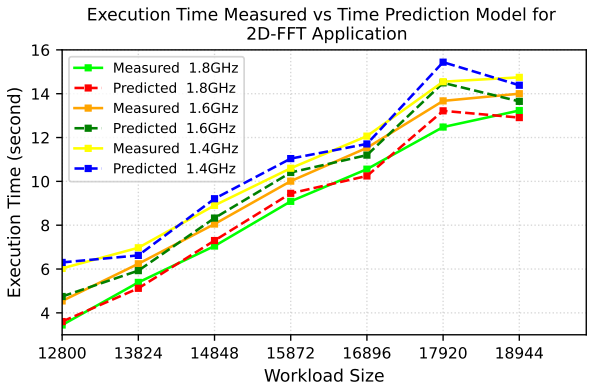
We observe that the dynamic power consumption of the main memory component for the (V_{mem}, f_{mem}) pair with f_{mem} equal to 405 MHz is very low, to the extent that it can not be measured reliably. Therefore, in the case of f_{mem}= 405 MHz, we can realistically ignore the main memory dynamic energy consumption and apply the scaling models and the scaling method developed for a processor with one independently powered component as we did for a CPU processor. However, when f_{mem} = 7251 MHz, the main memory dynamic energy consumption is significant and cannot be ignored. Therefore, to address this case, we need new average dynamic power and dynamic energy models that would account for two independently powered components of the processor. As the second component, the main GPU memory in our case, is always powered with the same voltage and frequency, we do not need V_{mem} and f_{mem} as variables in these new models, which therefore take the following form:

$$
\begin{aligned}
P_{avg}(w, V_g, f_g) &= I(w) + C_g(w) \times V_g^2 \times f_g \\
t(w, V_g, f_g) &= K_g(w) \times \frac{1}{f_g} \\
E_d(w, V_g, f_g) &= P_{avg}(w, V_g, f_g) \times t(w, f_g) \\
&= I(w) \times K_g(w) \times \frac{1}{f_g} \\
&\quad + C_g(w) \times K_g(w) \times V_g^2 \\
&= I_1(w) \times \frac{1}{f_g} + I_2(w) \times V_g^2
\end{aligned}
$$

In the power model, the intercept $I(w)$ is a workload-specific constant representing the dynamic energy consumption of the second independently powered processor component. As its voltage and frequency are fixed, they do not appear as variables in the models, and the DVFS state of the processor

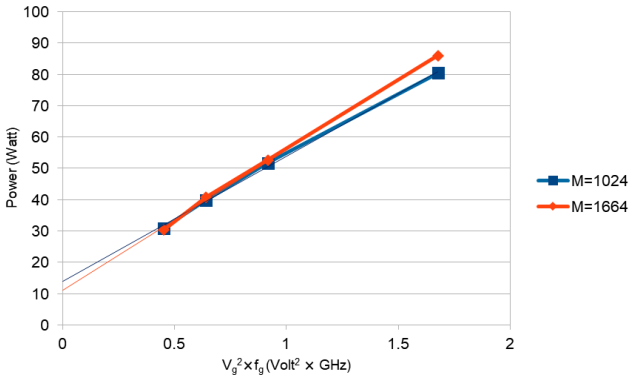


(a) DGEMM

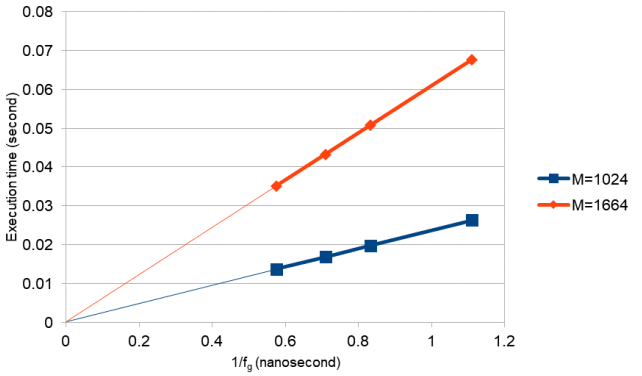


(b) 2D-FFT

Fig. 7: Experimental and calculated time profiles of two multithreaded applications on the Intel Icelake CPU, shown for three frequencies (1.4, 1.6, and 1.8 GHz). (a) DGEMM multiplies two dense square matrices of size $N \times N$ where $N \in \{12800, 13824, 14848, 15872, 16896, 17920, 18944\}$. (b) 2D-FFT computes a 2D fast Fourier transform of a dense signal matrix of size $N \times N$ for the same values of N.



(a)



(b)

Fig. 8: Average dynamic power consumption against $V_g^2 \times f_g$ (top) and execution time against $\frac{1}{f_g}$ (bottom) for two workloads of the 2D-FFT application invoking the cuFFT routine on the Nvidia A40 GPU. 2D-FFT computes a 2D fast Fourier transform of a dense signal matrix of size $M \times 51200$ for $M = 1664$ (red) and $M = 1024$ (blue). The values of f_g used in the experiments are 900 MHz, 1200 MHz, 1410 MHz, 1740 MHz, and the dots in the graphs represent experimental points.

is thus represented by the voltage, V_g, and frequency, f_g, of the first component only.

We perform experiments with several application workloads executing on GPUs to validate these models. Figure 8 shows the plots of average dynamic power consumption against $V_g^2 \times f_g$ and execution time against $\frac{1}{f_g}$ for two workloads of the 2D-FFT application executing on an Nvidia A40 GPU. The application invokes the cuFFT routine to compute a 2D fast Fourier transform of a dense signal matrix of size $M \times 51200$ for $M = 1664$ and $M = 1024$.

The plots clearly demonstrate that the execution time is linearly proportional to the reciprocal of the CUDA core frequency, $\frac{1}{f_g}$, approaching zero as the frequency goes to infinity, and the average dynamic power consumption is linearly proportional to $V_g^2 \times f_g$ approaching a workload-specific intercept as the frequency or voltage supply approaches zero.

These models reveal that the processor with two independently powered components is qualitatively different from the processor consisting of one independently powered component, even in the simple case when the voltage and frequency of the second component are the same for all DVFS states of the processor. Namely, the dynamic energy consumption of the two-component processor will depend not on voltage only as for the single-component processor but on both voltage and frequency of its first component, making it more complex for performance/energy management.

*1) **Execution Time and Dynamic Energy Scaling Models of a Two-component Processor**:* Now from these time and dynamic energy models we can derive the following scaling models, relating the time and dynamic energy of the execution of a given workload w for two reference DVFS states (V_{g1}, f_{g1}) and (V_{g2}, f_{g2}), and those for an arbitrary DVFS state (V_g, f_g):
As

$$t(w, V_{g1}, f_{g1}) = K(w) \times \frac{1}{f_{g1}}, \quad t(w, V_g, f_g) = K(w) \times \frac{1}{f_g},$$

then

$$t(w, V_g, f_g) = t(w, V_{g1}, f_{g1}) \times \frac{f_{g1}}{f_g}. \tag{3}$$

$$E_d(w, V_g, f_g) = I_1(w) \times \frac{1}{f_g} + I_2(w) \times V_g^2,$$

$where$

$$I_1(w) = \frac{E_d(w, V_{g1}, f_{g1}) \cdot V_{g1}^2 \cdot f_{g1}^2 - E_d(w, V_{g2}, f_{g2}) \cdot V_{g2}^2 \cdot f_{g2}^2}{V_{g1}^2 \cdot f_{g1}^2 - V_{g2}^2 \cdot f_{g2}^2}$$

$$I_2(w) = \frac{E_d(w, V_{g1}, f_{g1}) \times f_{g1} - E_d(w, V_{g2}, f_{g2}) \times f_{g2}}{V_{g1}^2 \times f_{g1} - V_{g2}^2 \times f_{g2}} \quad (4)$$

*2) **Scaling Method to Determine The Complete Set of Time and Energy Profiles of a Two-component Processor**:* Unlike the scaling method for a one-component processor, the method for a two-component processor needs not one but two experimentally obtained energy profiles to build the complete set of time and energy profiles. It stll uses one time profile though.

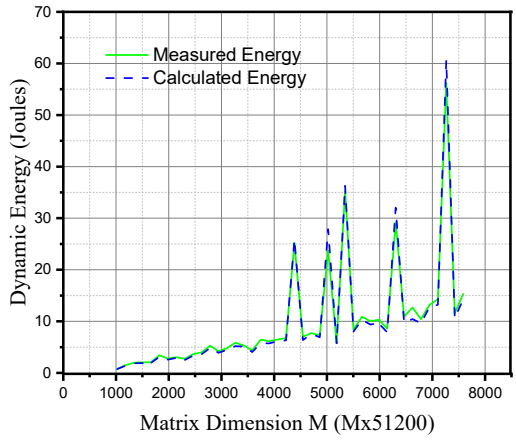
The steps of the scaling method for a two-component processor are as follows:

1) For DVFS states $(V_{g1}, f_{g1}), (V_{g2}, f_{g2}) \in \mathbb{VF}$, experimentally build the execution time and dynamic energy profiles, $t(w, V_{g1}, f_{g1})$, $E_d(w, V_{g1}, f_{g1})$, and $E_d(w, V_{g2}, f_{g2})$. Calculate time profile $t(w, V_{g2}, f_{g2})$ using formula (3).
2) For each DVFS state $(V_g, f_g) \in \mathbb{VF} \setminus \{(V_{g2}, f_{g2}), (V_{g1}, f_{g1})\}$, calculate the execution time profile $t(w, V_g, f_g)$ using formula (3) and the dynamic energy profile $E_d(w, V_g, f_g)$ using formula (4).

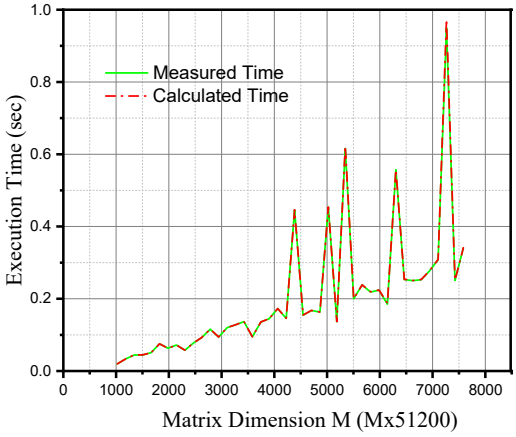
D. Application of the Scaling Solution for Processor with Two Independently-powered Components to Nvidia A40 GPU Processor

The proposed models and scaling method for a processor with two independently powered processor components can be applied to multithreaded workloads executing on a Nvidia A40 GPU processor given the following constraints are satisfied:

- The main memory operates at the higher frequency, 7251 MHz. If the memory operates at the lower frequency, 405 MHz, its dynamic energy consumption will be insignificant and can be ignored, and therefore the models and scaling method for a one-component processor can be used.
- We only consider DVFS states of the GPU processor where all CUDA cores operate at the same frequency.
- The dynamic energy consumption of all other independently powered components is insignificant in comparison with that of the graphics processor and the main memory and therefore can be ignored.
- The configuration of the workload should be fixed and be as deterministic as possible to guarantee the same operation of the hardware during different runs of the workload.
- The dynamic power consumption for DVFS states (V_{g1}, f_{g1}) and (V_{g2}, f_{g2}) must be high enough to be accurately and reliably measured by the ground-truth method. While this constraint is typically satisfied by workloads running on CPUs, it is more common for accelerators to consume so low dynamic power during the execution of some workloads (even when operating at the highest frequency) that it is impossible to reliably measure this power using the ground-truth method employing the power meter.



(a)



(b)

Fig. 9: Experimentally built and calculated dynamic energy and time profiles (top and bottom, respectively) of the 2D-FFT application invoking the cuFFT routine on the Nvidia A40 GPU for graphics frequency, 1200 MHz.

We experiment with workloads of the 2D fast Fourier transform application, 2D-FFT, executing on a Nvidia A40 GPU and satisfying the above applicability constraints. The application invokes the cuFFT library routine to compute a 2D fast Fourier transform of a dense signal matrix of size $M \times 51200$. In our experiments M varies from 1024 to 7584 in increments of 160.

We experimentally build a time profile for the DVFS state with frequency f_{g1} = 1740 MHz and two dynamic energy profiles for the DVFS states with frequencies f_{g1} = 1740 MHz and f_{g2} = 1410 MHz. All other time and dynamic energy profiles are calculated using formulas (3) - (4).

Figure 9a shows the experimentally built and calculated dynamic energy profiles for the DVFS state with graphics frequency, 1200 MHz. Figure 9b shows the experimentally built and calculated time profiles for the same DVFS state.

One can see that the experimentally built and calculated profiles are practically indistinguishable. Such a high accuracy of the scaling method in this case is explained by deterministic implementation of the cuFFT routine. The method will not be that accurate for non-deterministic routines, where two identical invocations of the routine on the same hardware may result in significantly different execution flows.

E. Applicability and Limitations of the Scaling Method

The proposed optimization formulation and algorithms are not restricted to FFT and GEMM applications. The algorithms require, as input, execution-time and dynamic-energy functions for each processor configuration and workload size. Therefore, they can be applied to other data-parallel applications once these functions are obtained either by direct profiling or by a validated scaling method.

The scaling method is most suitable for workloads whose execution behaviour remains sufficiently stable across DVFS states. This includes workloads with fixed input size, fixed thread mapping, limited non-deterministic operating-system interference, and repeatable computation and data-transfer patterns. These conditions are used in the experimental methodology to make the measured execution-time and dynamic-energy profiles reproducible.

For workloads with irregular computation, irregular synchronization, communication phases, or non-deterministic execution paths, the applicability of the scaling method depends on whether the workload-specific parameters remain stable across DVFS states. If irregular memory stalls, synchronization delays, communication costs, operating-system activity, or thermal effects significantly change the execution behaviour, the scaling assumptions may become less accurate. In such cases, the proposed optimization algorithms can still be used with directly measured or workload-specifically validated execution-time and dynamic-energy functions, while the scaling method should be applied only after profiling confirms acceptable prediction accuracy.

For HGEMM, the manuscript explicitly includes CPU-to-GPU data transfers before GPU computation and GPU-to-CPU data transfers after GPU computation in the execution-time and dynamic-energy functional models. The scaling method is not separately validated in this work for hybrid applications with frequent CPU–GPU synchronization or repeated CPU–GPU data movement during kernel iterations. For such applications, the method should be applied only when these costs are included in the measured profiles and workload-specific validation confirms reproducible execution-time and dynamic-energy profiles across DVFS states.

X. Experimental Results

We demonstrate the efficiency and accuracy of our solution in optimizing two heterogeneous hybrid applications, 2D fast Fourier transform and mixed precision matrix multiplication, on a heterogeneous platform comprising an Intel Icelake multicore CPU and two Nvidia A40 GPUs (Table I).

TABLE I: Specifications of the heterogeneous hybrid server containing a single-socket Icelake multicore CPU and two Nvidia A40 GPUs.

Intel Platinum 8362 Icelake	
No. of cores per socket	32
No. of threads per core	2
Socket(s)	1
L1d cache, L1i cache	1.5 MiB, 1 MiB
L2 cache, L3 cache	40 MiB, 48 MiB
Total main memory	64 GB DDR4-3200
TDP	265 W
NVIDIA A40 GPU	
No. of GPUs	2
No. of Ampere cores	10,752
Total board memory	48 GB GDDR6 (with ECC)
Memory bandwidth	696 GB/sec
TDP	300 W

A. Hybrid Parallel 2D fast Fourier Transform application (HFFT)

The hybrid parallel 2D fast Fourier Transform application (HFFT) computes the 2D-FFT of a signal matrix S of size $N \times N$. The workload size is $5 \times N^2 \times \log_2 N$. The application comprises three software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). Each component *i* is assigned a number of rows of S provided as a parameter to the application. The 2D FFT accomplished in four steps: (1) Computing 1D FFTs on *N* rows of the signal matrix N. (2) Transpose of the signal matrix. (3) 1D FFTs on *N* rows of the signal matrix N and (4) Transpose of the signal matrix.

Figure 10a depicts the hybrid parallel 2D fast Fourier Transform application (HFFT) computing the 2D-FFT of a signal matrix S of size $N \times N$. Figure 10b shows the hybrid parallel mixed precision matrix multiplication application (HGEMM) that computes the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$.

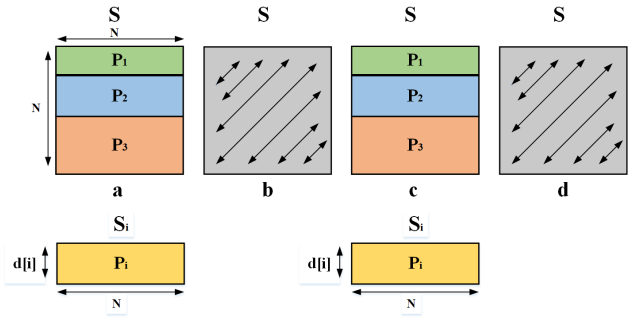
B. Dynamic Energy and Execution time Profiles for HFFT

Figure 11 shows the dynamic energy consumption and execution time functions of matrix dimension x for the CPU and GPU processors for the DVFS states employed for validation for the HFFT application. For the sake of brevity, we only show the processor core frequency f_{core} in the DVFS state for the CPU processor and the f_{gra} graphics frequency for the A40 GPU processor in the plots.

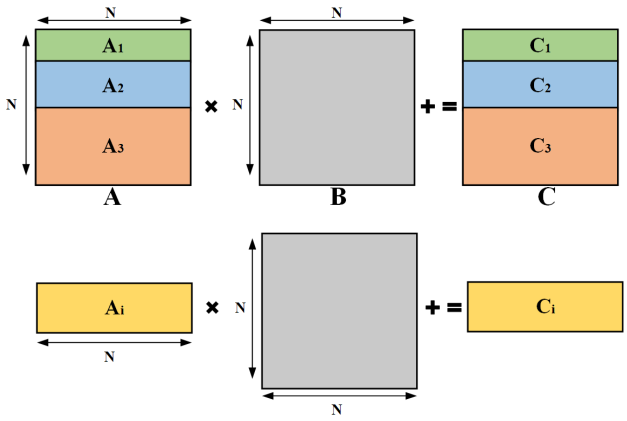
Figure 12 shows the linear profiles (and linear approximations) of dynamic energy and execution time versus matrix dimension x for the CPU and GPU processors. Each profile is a function of workload size representing the number of 1D-FFTs (x) of size N in the 2D-FFT of a signal matrix, $x \times N$ where N is 30720 and x varies from 1024 to 15360 in steps of 1024.

C. Hybrid Parallel Mixed Precision Matrix Multiplication (HGEMM)

The hybrid parallel mixed precision matrix multiplication application (HGEMM) computes the matrix product ($C+ =$

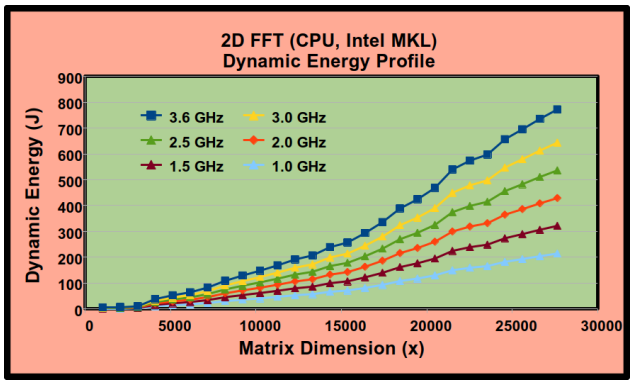


(a) Heterogeneous hybrid parallel 2D fast Fourier Transform application (HFFT) computing the 2D-FFT of a signal matrix S of size $N \times N$. The application comprises three software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). Each component i is assigned a number of rows of S provided as a parameter to the application. The 2D FFT accomplished in four steps: (1) Computing 1D FFTs on N rows of the signal matrix N. (2) Transpose of the signal matrix. Steps (3) and (4) are repetition of steps (1) and (2), respectively.

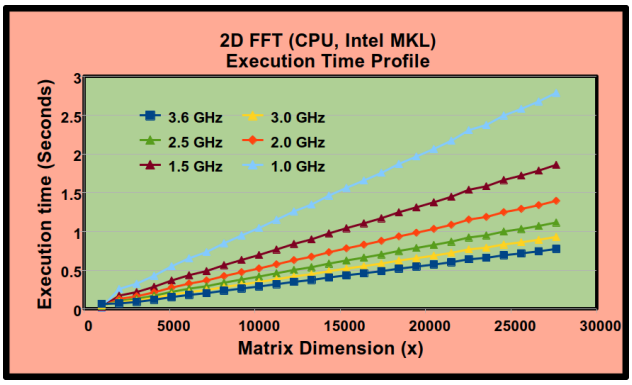


(b) Matrix partitioning between the software components in hybrid parallel matrix multiplication application (HGEMM) computing the matrix product ($C+ = A \times B$) of two dense square matrices A and B of size $N \times N$. The matrix B is shared by all the components. The application comprises 3 software components executed in parallel (one CPU component, one GPU_1 component, and one GPU_2 component). Each software component i computes the matrix product, $C_i = A_i \times B$.

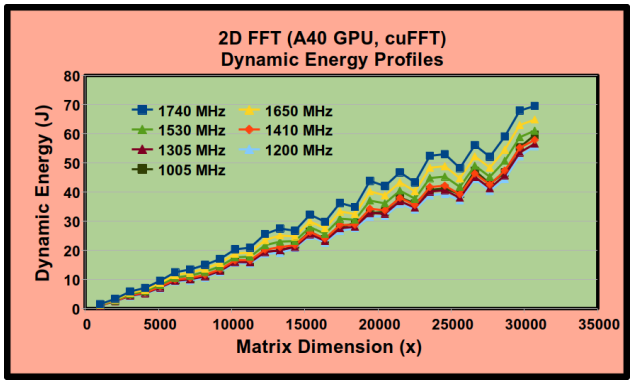
Fig. 10: Description of the two heterogeneous hybrid applications.



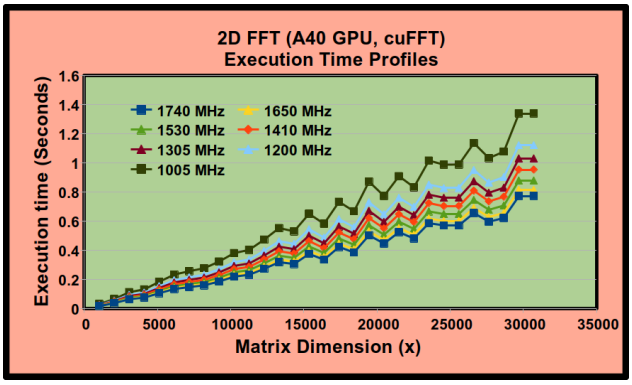
(a)



(b)



(c)



(d)

Fig. 11: Dynamic energy consumption and execution time functions of matrix dimension x for the CPU and GPU processors for the DVFS states employed for validation for the HFFT application. For the sake of brevity, we only show the processor core frequency f_{core} in the DVFS state for the CPU processor and the f_{gra} graphics frequency for the A40 GPU processor in the plots.

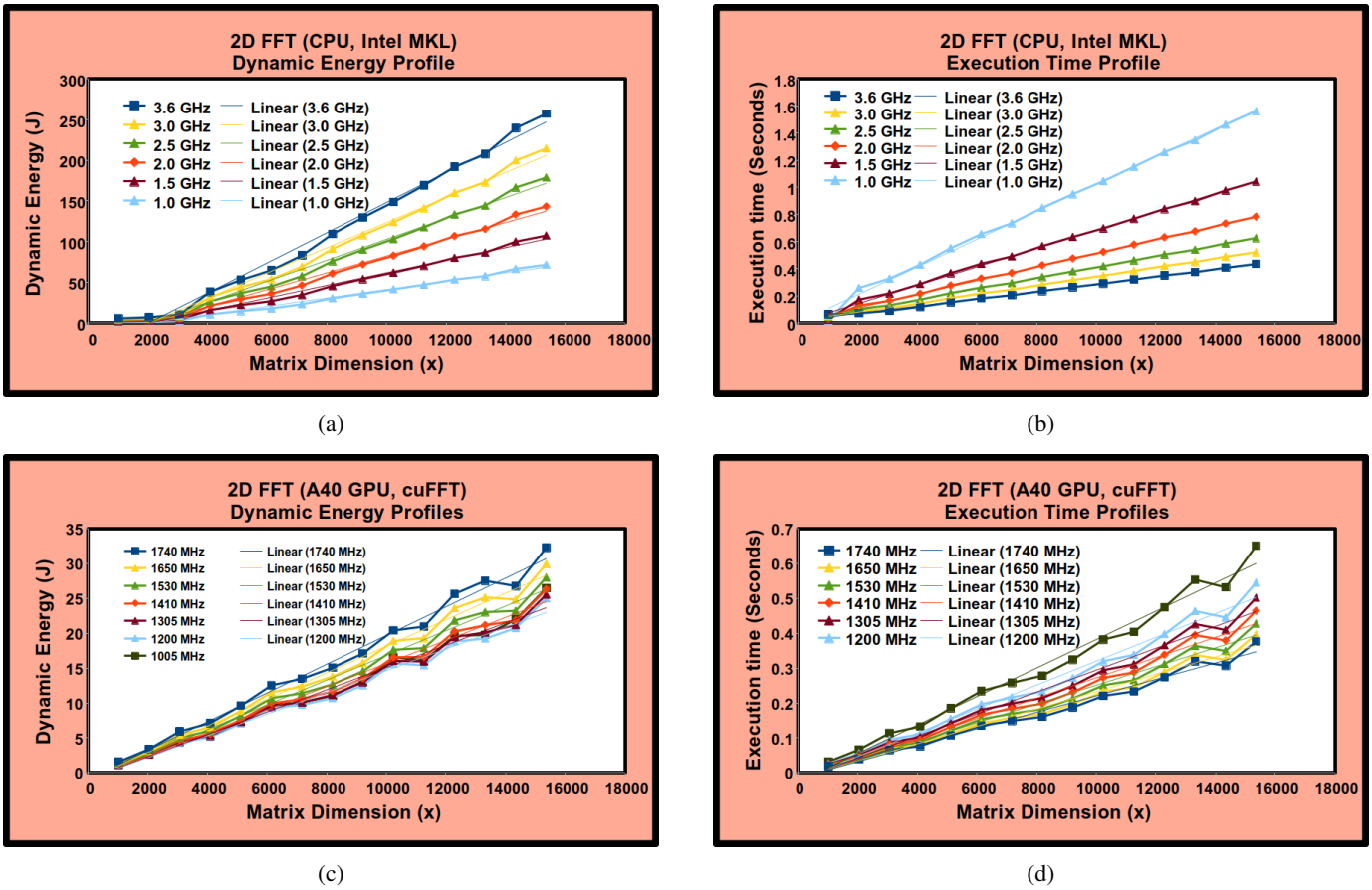


Fig. 12: Linear profiles (and linear approximations) of dynamic energy and execution time versus matrix dimension x for the CPU and GPU processors for the DVFS states employed for validation for the HFFT application.

$A \times B$) of two dense square matrices A and B of size $N \times N$. The application has three software components: CPU, A40 GPU_1, and A40 GPU_2. All the components share the matrix B. The matrices A and C are horizontally partitioned so that each software component is assigned several contiguous rows of A and C provided as input parameters to the application. The CPU software component is assigned N_1 rows of A and C. The A40 GPU_1 software component is assigned N_2 rows of A and C. Finally, the A40 GPU_2 software component is assigned the remaining $(N - N_1 - N_2)$ rows of A and C. Each software component i ($i \in \{1, 2, 3\}$) computes the matrix product, $C_i = A_i \times B$.

D. Dynamic Energy and Execution time Profiles for HGEMM

Figure 13 shows the dynamic energy consumption and execution time functions of matrix dimension x for the CPU and GPU processors for the DVFS states employed for validation for the mixed precision HGEMM application.

Figure 14 shows the linear profiles (and linear approximations) of dynamic energy and execution time versus matrix dimension x for the CPU and GPU processors for the DVFS states. Each profile is a function of workload size representing the matrix product of $x \times N$ and $N \times N$ matrices where N is 32768 and x varies from 1024 to 25600 in steps of 1024.

XI. Workflow of Optimization Process

Figure 15 illustrates the core steps of the proposed methodology for optimization of a data-parallel application for performance and energy on a heterogeneous hybrid server. While the methodology is applicable to an arbitrary hybrid server equipped with accelerators of different architectures, for the sake of simplicity of presentation we assume that the server consists of one CPU and several identical GPUs (our experimental server has two identical A40 GPUs).

The optimization process begins with the user selecting one CPU frequency and two GPU frequencies. Energy (E) and execution time (T) profiles of the CPU and one GPU are then experimentally obtained for these initial settings. Using the scaling formulas for CPU and GPU, the method constructs energy and performance profiles for all other frequency levels without requiring further measurements.

Next, Pareto fronts are generated for all possible combinations of CPU and GPU frequencies in the server, capturing the trade-offs between energy consumption and execution time for each possible DVFS configuration of the server. These fronts are then merged into a single global Pareto front, from which the user selects the most suitable Pareto-optimal trade-off (a point on the global Pareto front). The final step employs our algorithm HEPOPTADV to determine both the frequency

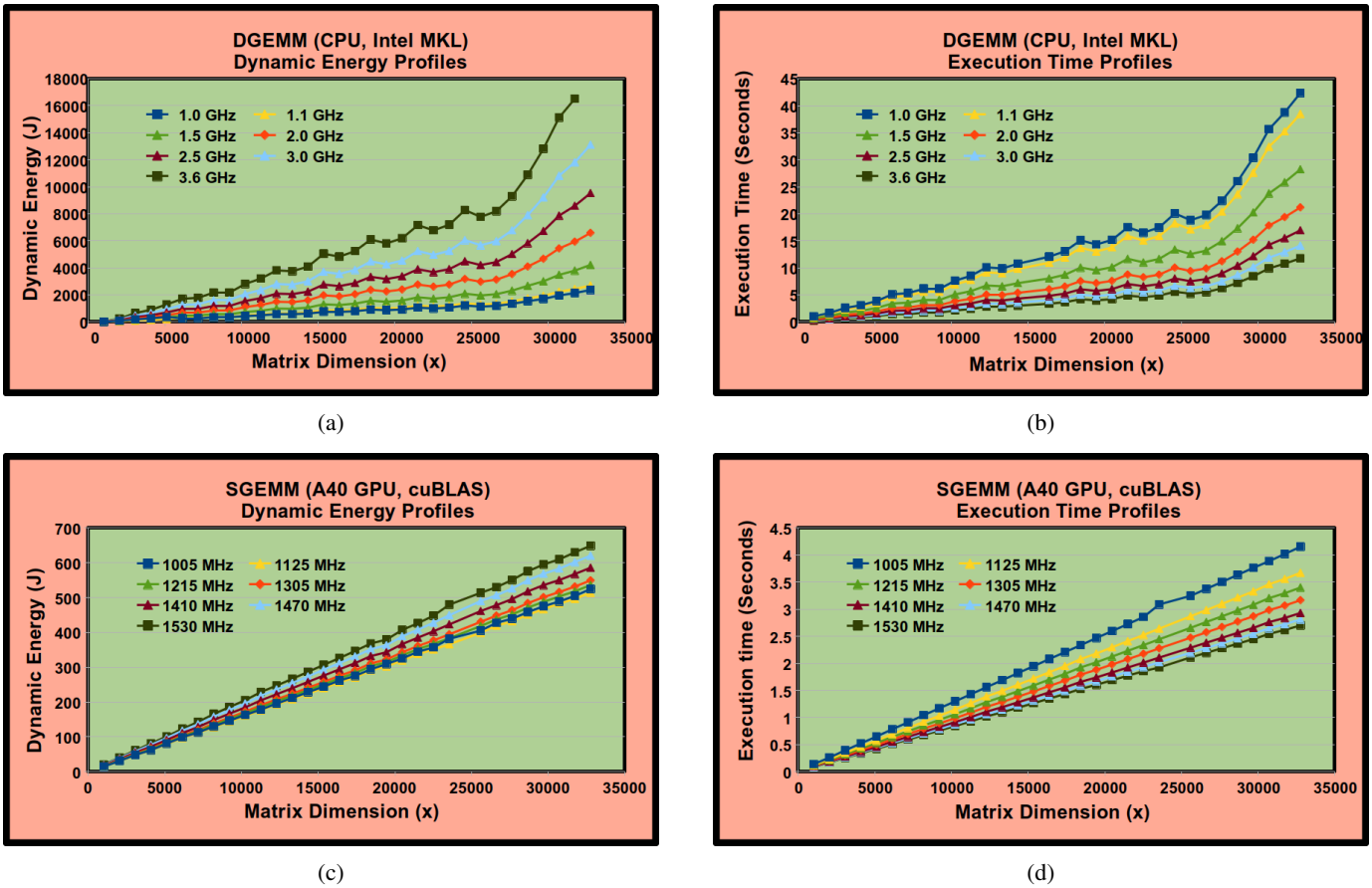


Fig. 13: Dynamic energy consumption and execution time functions of matrix dimension x for the CPU and GPU processors for the DVFS states employed for validation for the mixed precision HGEMM application. For the sake of brevity, we only show the processor core frequency f_{core} in the DVFS state for the CPU processor and the f_{gra} graphics frequency for the A40 GPU processor in the plots.

settings and the workload distribution across devices for the selected Pareto-optimal trade-off.

ACKNOWLEDGMENTS

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under the SFI Frontiers for the Future Programme 20/FFP-P/8683.

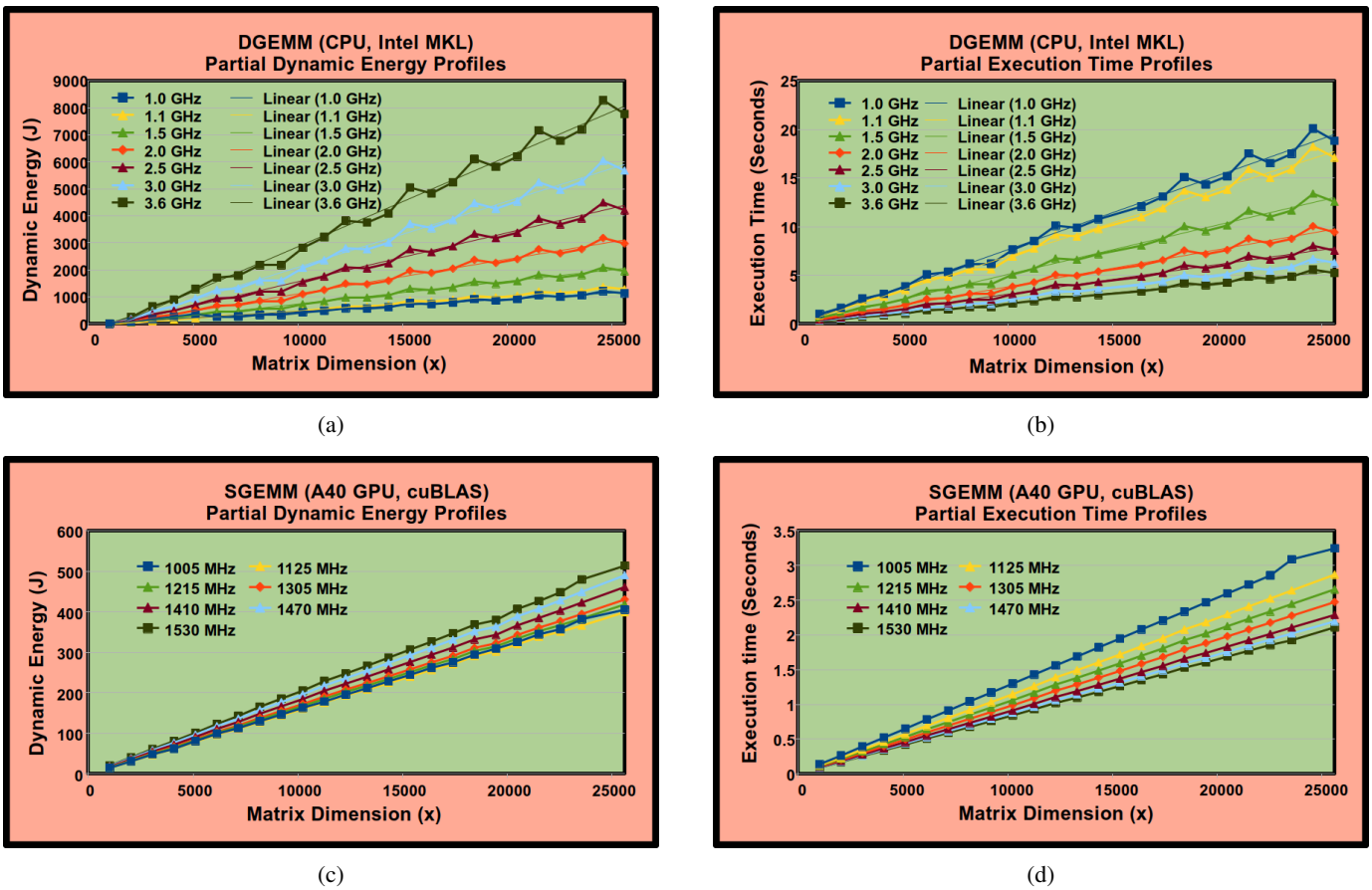


Fig. 14: Linear profiles (and linear approximations) of dynamic energy and execution time versus matrix dimension x for the CPU and GPU processors for the DVFS states employed for validation for the HGEMM application.

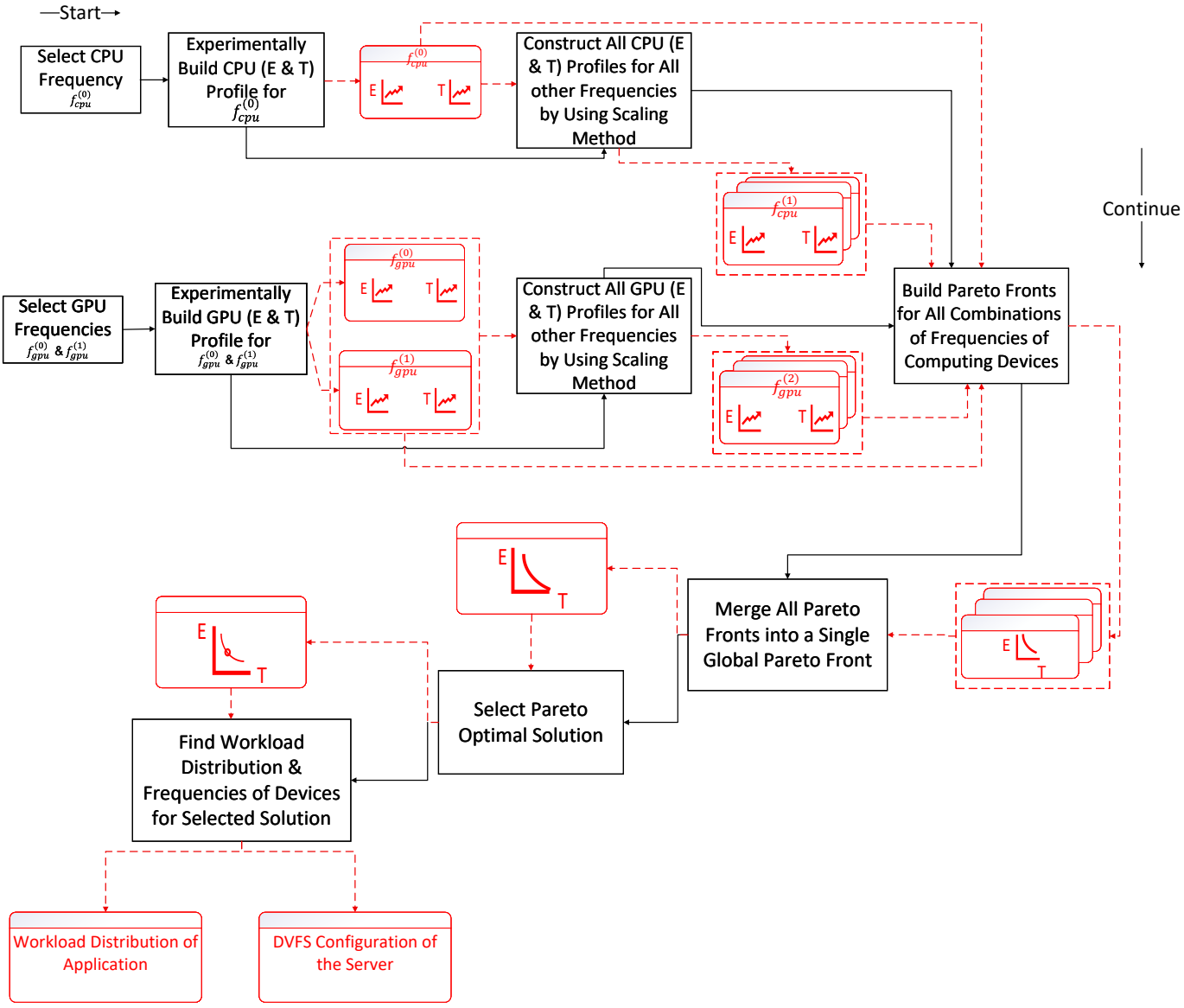


Fig. 15: Overview of the proposed methodology for energy-time trade-off optimization on a heterogeneous hybrid system consisting of one CPU and several identical GPUs. The process begins with user selection of a CPU frequency and two GPU frequencies, followed by experimental profiling of execution time (T) and energy consumption (E) at those frequencies. Using scaling methods, performance and energy profiles are calculated for all other frequencies. Pareto fronts are then constructed for all combinations of frequencies of the devices and then merged into a global Pareto front. Finally, the user selects a Pareto-optimal solution and invokes our algorithm HEPOPTADV to determine device frequencies and workload distribution.

REFERENCES

[1] K. Miettinen, *Nonlinear multiobjective optimization*. Kluwer, 1999.

[2] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.

[3] Nvidia, "Advanced configuration and power interface specification, version 6.2," 05 2017. [Online]. Available: https://uefi.org/sites/default/files/resources/ACPI_6_2.pdf

[4] Z. Zhuang, H. Ramachandra, and B. Sridharan, "SLA-Aware dynamic CPU scaling in business cloud computing environments," in *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015, pp. 836–843.

[5] S. Ibrahim, T.-D. Phan, A. Carpen-Amarie, H.-E. Chihoub, D. Moise, and G. Antoniu, "Governing energy consumption in Hadoop through CPU frequency scaling: An analysis," *Future Generation Computer Systems*, vol. 54, pp. 219–232, 2016.

[6] D. Brodowski, N. Golde, R. J. Wysocki, and V. Kumar, "CPU frequency and voltage scaling code in the Linux kernel," *Linux kernel documentation*, p. 66, 2013.

[7] Heterogeneous Computing Laboratory, "Linux scripts for DVFS for CPU and Nvidia GPU processors and Likwid tools," https://csgitlab.ucd.ie/manumachu/openh/-/tree/master/Articles/IEEETC2025?ref_type=heads, 2025.

[8] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th international conference on parallel processing workshops*. IEEE, 2010, pp. 207–216.

[9] P. J. Mucci, K. London, and S. Moore, "Cross-platform component performance monitoring using PAPI," in *Proceedings of the International Conference on Computational Science*. Springer, 2001, pp. 127–134.

[10] M. Fahad and R. R. Manumachu, *HCLWattsUp: energy API using system-level physical power measurements provided by power meters*. Heterogeneous Computing Laboratory, University College Dublin, May 2022. [Online]. Available: https://csgitlab.ucd.ie/manumachu/hclwattsup

[11] H. Khaleghzadeh, M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on heterogeneous HPC platforms for performance and energy through workload distribution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 543–560, 2021.



Hesam Nejati SharifAldin received his Master's degree in software engineering from the Islamic Azad University of Mashhad, Iran in 2017. He is pursuing a PhD degree from the School of Computer Science, University College Dublin. His research interests includes parallel computing, high-performance computing, and energy-efficient computing.



Sourodip Ghoshdastidar received a B.E degree from Jadavpur University, India in 2015, and a M.Tech degree from Indian Institute of Science, Bangalore in 2023. He is currently pursuing a PhD degree from the School of Computer Science, University College Dublin. His research interests span high performance computing, energy-efficient computing and molecular dynamics simulations.



Ravi Reddy Manumachu received a B.Tech degree from I.I.T, Madras in 1997 and a PhD degree from the School of Computer Science, University College Dublin (UCD) in 2005. He is currently Assistant Professor in the School of Computer Science at UCD. His main research interests include high performance heterogeneous computing and energy-efficient computing.



Alexey Lastovetsky received the Ph.D. degree from the Moscow Aviation Institute in 1986, and the Doctor of Sciences degree (Habilitation) from the Russian Academy of Sciences in 1997. His main research interests include heterogeneous parallel computing and energy-efficient parallel computing. He authored the monographs Parallel computing on heterogeneous networks (Wiley, 2003) and High performance heterogeneous computing (Wiley, 2009). He is an inaugural ScholarGPS Highly Ranked Scholar - Lifetime (ranked within the 0.05% of all scholars in Parallel Computing).