Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 COMPUTER SCIENCE

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Оптимизированный гетерогенный планировщик, управляемый графами переходов состояний / Optimized Heterogeneous Scheduler Driven by State-Transition Graphs**

Работу выполнил / Thesis is executed by	**Муталапов Арсен Ильдарович / Mutalapov Arsen Ildarovich**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Бурмяков Артём Сергеевич / Burmyakov Artem Sergeevich**	подпись / signature

Иннополис, Innopolis, 2025

Contents

List of Tables

List of Figures

Abstract

An increasing use of heterogeneous mobile platforms forces the design of more efficient schedulers. Such platforms are comprised of processing units that differ in time and energy performance. A scheduler then aims at finding a suitable trade-off between runtime and energy consumption. We propose new optimal and suboptimal scheduling techniques based on the notion of state-transition graphs, which model possible programs execution schedules over different processing unit types. As a scheduling objective we chose Energy-Delay Product metric that aggregates time and energy consumption. Our schedulers are implemented by a C++ Scheduler tool, which is available on demand.

Chapter 1

Introduction

In a classical uniform multicore[1] scheduling, an operating system scheduler allocates available uniform cores to dynamically arriving programs [1]. Once the number of pending programs exceeds the number of cores, a scheduler prioritizes programs by either static priorities set by a system designer, or specific dynamically computed priorities. Schedulers objectives are the minimization of the average programs execution time, latency, migrations and preemptions number, and other. We emphasize that the problem of an optimal multicore scheduling is shown to be NP-hard, and thus the heuristic approaches are typically used to find a suboptimal solution.

1.1 Heterogeneous scheduling

Even further, unlike the classical homogeneous case, heterogeneous scheduling is significantly complicated by the presence of multiple types of processing units for a computing heterogeneous hardware platform [2]. Some of these heterogeneous units are slow but energy efficient, while others are fast but too energy

[1] We do not distinguish terms “multiprocessor" and “multicore"

consuming (see Fig. 1.1). A well-known example of a heterogeneous platform is ARM big.LITTLE architecture [3], conceptually depicted in Fig. 1.2a. It consists of:

- a cluster of fast (“big") but energy-hungry cores;
- a cluster of slow (“LITTLE") but energy-efficient cores;
- GPU and
- NPU devices.

Observe the performance comparison of “big" (Cortex-A15) and “LITTLE" (Cortex-A7) cores in Fig. 1.2b. There is an intersecting performance range for these core types, where the use of “LITTLE" cores leads to much lower energy consumption than the use of “big" cores. An appropriate allocation of these processing units depends on programs activity. For example, read of 32 bytes from an 8KB SRAM consumes 125 times less energy than from a DRAM [4]. Considering a memory-intensive program, processing units optimized for memory access, for example, by providing larger caches, significantly reduce energy consumption at little performance degradation. Thus, a heterogeneous scheduler not only aims at optimizing time-related aspects, but also the overall energy consumption. This problem of an optimized heterogeneous scheduling is especially relevant for systems with autonomous power supplies. We elaborate on heterogeneous hardware in Section 3.3.

1.2 Toy schedule example

Let us next provide a toy example of scheduling three programs over a heterogeneous hardware platform with one “slow" and one “fast" core. We assume

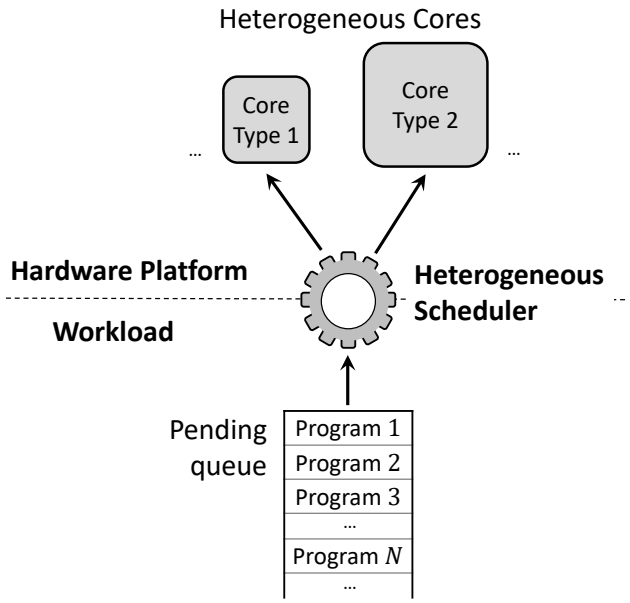


Fig. 1.1. Heterogeneous scheduling: key components

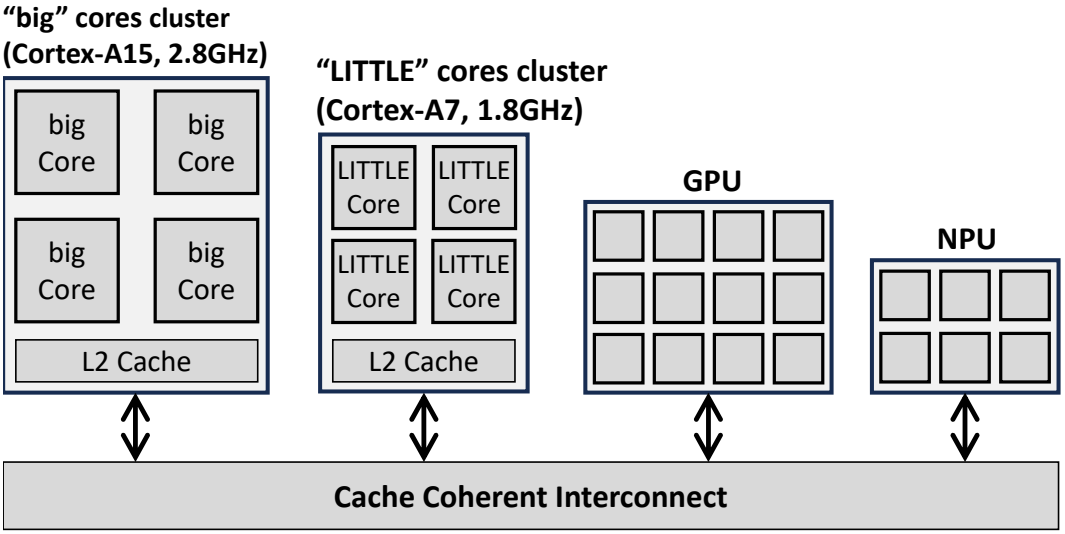
that programs are composed of individual logically separated code blocks[2] with their execution requirements listed in Table 1.1. In Fig. 1.4a and 1.4c we depict cores load and programs execution for two possible schedules S1 and S2, assuming programs arrive asynchronously at times 0, 1, and 3 respectively. For example, in S1 the "slow" Core 1 is allocated to the Programs 1 and 3, while in S2 this core solely executes all blocks of Program 2.

The figures, as well as the rest of the manuscript, rely on the graphical notation in Fig. 1.3.

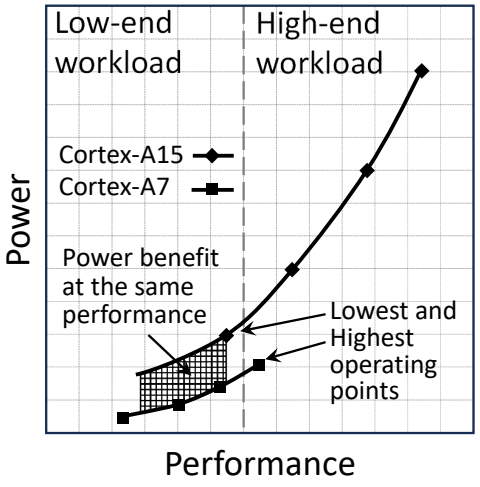
1.3 Contribution

We propose a novel approach for heterogeneous scheduling. It is based on traversing state-transition graphs of executed programs. More specifically, our contributions are:

[2]The notion of a block is not yet defined and to be elaborated later. So far, we assume that program blocks execute sequentially.



(a) The ARM big.LITTLE architecture: overview



(b) Power/performance trends

Fig. 1.2. Heterogeneous architecture: example with Cortex-A15 and Cortex-A7

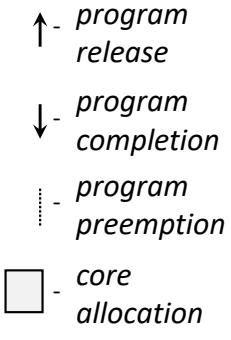


Fig. 1.3. Graphical notation for Fig. 1.4, 3.1, 3.3, and 4.1

TABLE 1.1
Programs execution requirements per their block

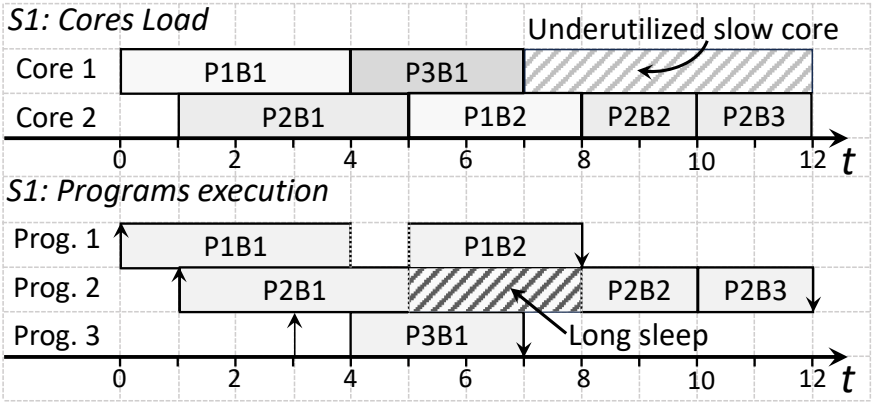
Programs Blocks		Execution time, sec		Energy use, Joules	
		Core 1 (slow)	Core 2 (fast)	Core 1	Core 2
P1	B1	4	3	8	9
	B2	5	3	3	4
P2	B1	5	4	6	10
	B2	3	2	3	7
	B3	3	2	3	6
P3	B1	3	2	6	7

- A theoretically optimal heterogeneous scheduler. Unlike other existing schedulers, our one addresses not only programs response time, but also energy consumption;
- Scheduling heuristics, which make scheduling decisions significantly faster than an optimal scheduler. We also demonstrate that some heuristic is near-optimal;
- A prototype C++ tool implementing all our schedulers;
- A thorough comparison of all proposed schedulers.

At the moment our schedulers do not account for preemption and migration overheads. However, the proposed state-transition graph methodology is easily extendable to such more realistic scenarios.

1.4 Organization

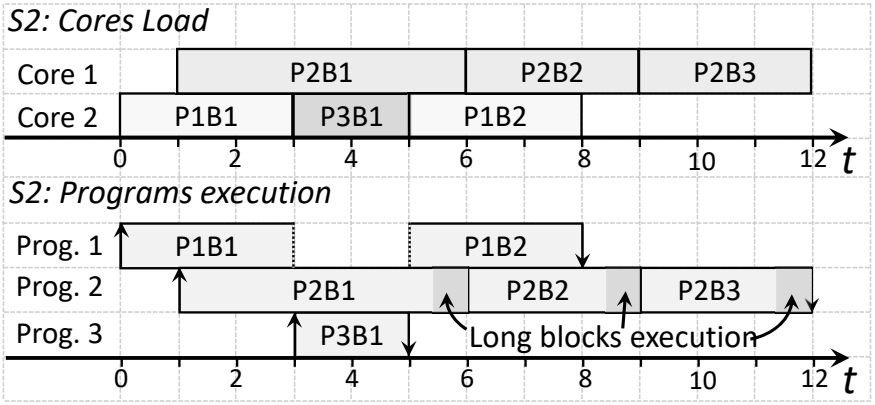
The rest of the paper is organized as follows. In the first sections we describe the necessary background for our analysis. After introducing the heterogeneous



(a) Schedule S1: programs execution

Schedule	Program	Latency	Execution Time	Response Time	Average Execution Time	Average Response Time
S1	P1	0	7	8	**6**	**7.67**
	P2	0	8	11		
	P3	1	3	4		

(b) S1: time performance metrics



(c) Schedule S2: programs execution

Schedule	Program	Latency	Execution Time	Response Time	Average Execution Time	Average Response Time
S2	P1	0	6	8	**6.33**	**7**
	P2	0	11	11		
	P3	0	2	2		

(d) S2: time performance metrics

Fig. 1.4. Sample execution schedules for programs from Table 1.1 (no migrations and preemptions considered)

scheduling problem and a toy example in Chapter 1, we next, in Chapter 2, introduce the system model for a heterogeneous computing platform and programs processing requirements. Then, in Chapter 3 we revise multiprocessor scheduling strategies, which serve as the basement for our refined scheduler. We dedicate Chapter 4 to discuss various programs performance metrics to be considered in the analysis, both reflecting time and energy consumption. After providing the necessary preliminaries, we derive our optimized heterogeneous scheduler. In Chapter 5 we describe analytically our optimized scheduling driven by the notion of a state-transition graph. After that, in Chapter 7, we report the evaluation results for our approach through a set of experiments. Finally, in Chapter 8, we conclude our work by discussing potential directions for optimizing our heterogeneous scheduler.

Chapter 2

System Model

Let us formally describe the necessary definitions and assumptions for an addressed heterogeneous scheduling problem. We consider scheduling over ARM BIG.LITTLE platform, which is introduced in Chapter 1. However, to keep our analysis simpler, for now we restrict the platform to the case of two cores only:

- The BIG core - fast but energy-hungry;
- The LITTLE core - slow but energy-efficient;

We assume every core to execute only one program at a time. We also consider no hardware-level parallelism and other optimisations at the moment.

2.1 Mathematical notations

We have N programs, which are denoted by $P_1, \ldots, P_N$, to be executed concurrently over a shared BIG.LITTLE platform of $\mathcal{M}$ cores[1]. These programs arrive asynchronously into a pending queue as depicted in Fig. 1.1. Then, at scheduling

[1] In our analysis, $\mathcal{M} = 2$: one BIG and one LITTLE cores

time instants a heterogeneous scheduler examines this queue and allocates available cores to the programs. The scheduler aims at optimizing a scheduling objective, which we define later.

Every program $P_i = \{b_{i1}, \ldots, b_{in_i}\}$, with each block representing a sequence of instructions. In our analysis, a program is executed block by block as depicted in Fig. 1.4a and 1.4c. Block's b_j processing requirements, if executing over an m-type core are modeled by:

- T_{jm} - block execution time;
- E_{jm} - block energy consumption,

These metrics are typically collected by profiling tools, such as Perf. Thus, we consider these metrics to be known prior to our analysis, e.g. like an example in Table 1.1.

For a given program P_i, parameters T_m and E_m denote its total runtime and energy consumption over m-type core, which are computed over all its blocks by:

$$\mathsf{T}_m^i = \sum_{j=1}^{n} T_{jm} \qquad (2.1) \qquad\qquad \mathsf{E}_m^i = \sum_{j=1}^{n} E_{jm} \qquad (2.2)$$

Considering execution requirements of every $P_1, \ldots, P_n$ programs are known, our key objective is to minimize the so-called "energy-delay product" metric [5], which is denoted by ρ and computed by:

$$\rho = \mathsf{T} * \mathsf{E} \qquad (2.3)$$

where T and E denote the aggregated runtime and energy consumption of all simultaneously executed programs:

$$\mathsf{T} = \sum_{i=1}^{N} T^{i}_{\left(m_1 \ldots m_{n_i}\right)} \qquad (2.4) \qquad \mathsf{E} = \sum_{i=1}^{N} E^{i}_{\left(m_1 \ldots m_{n_i}\right)} \qquad (2.5)$$

where $(m_1 \ldots m_{n_i})$ denotes a sequence of n_i blocks of program P_i executed over cores $m_j \in \{1, \ldots, \mathcal{M}\}$. In fact, we aim at determining all those sequences, which result in a program execution with an optimal trade-off between runtime and energy consumption.

We also study scheduling efficiency for minimum average response time and energy consumption in isolation in Chapter 4.

Additionally, for the analysis simplicity, we assume that:

- Granularity unit for our analysis is one program block;
- Two programs cannot reuse the same core at a time;
- Blocks within a given program execute sequentially;
- Hardware overheads are insignificant (see Section 3.4);
- Block execution can be preempted and migrated between cores (see Section 3.4).

We aim at gradually relaxing these assumptions in the future.

Chapter 3

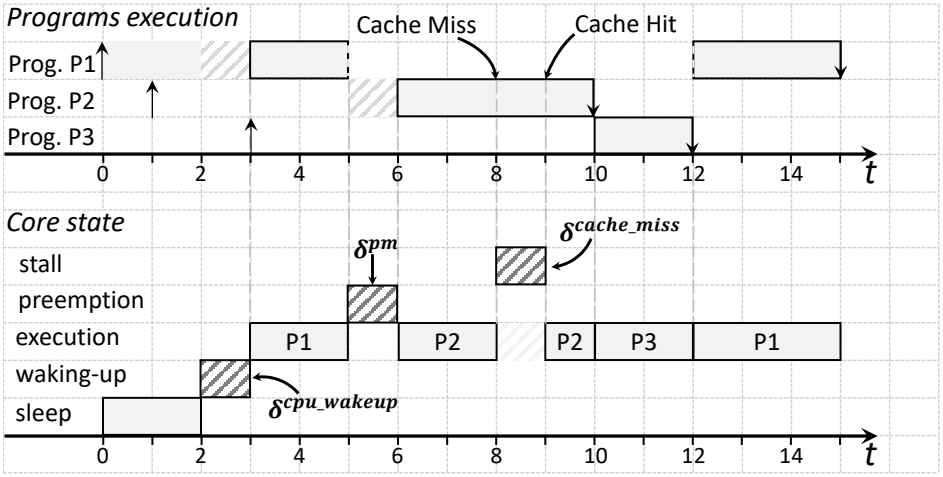
Processor Scheduling Strategies

We provide the necessary preliminaries for our heterogeneous scheduling analysis. We start with a homogeneous scheduling case with identical processors in Section 3.1 and 3.2. Then, we describe the basic heterogeneous scheduling principles in Section 3.3. We especially focus on various scheduling overheads in regard to these principles in Section 3.4.

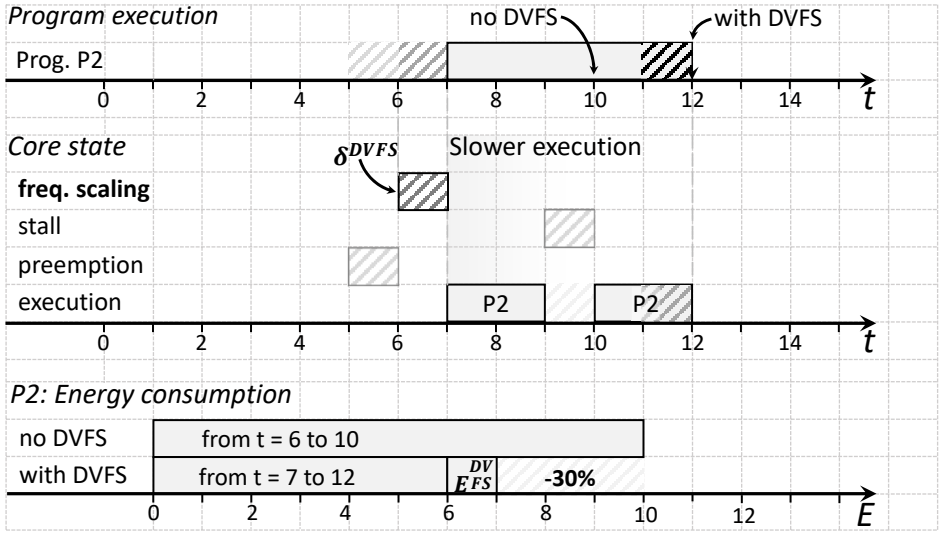
3.1 Single processor case

Consider an execution schedule of Fig. 3.1a. Three programs labeled P1, P2 and P3 execute over a same shared single core. Programs execution requirements are listed in Fig. 3.2. The scenario of Fig. 3.1a assumes programs P1, P2 and P3 to arrive at times $t = 0$, 1 and 3 correspondingly.

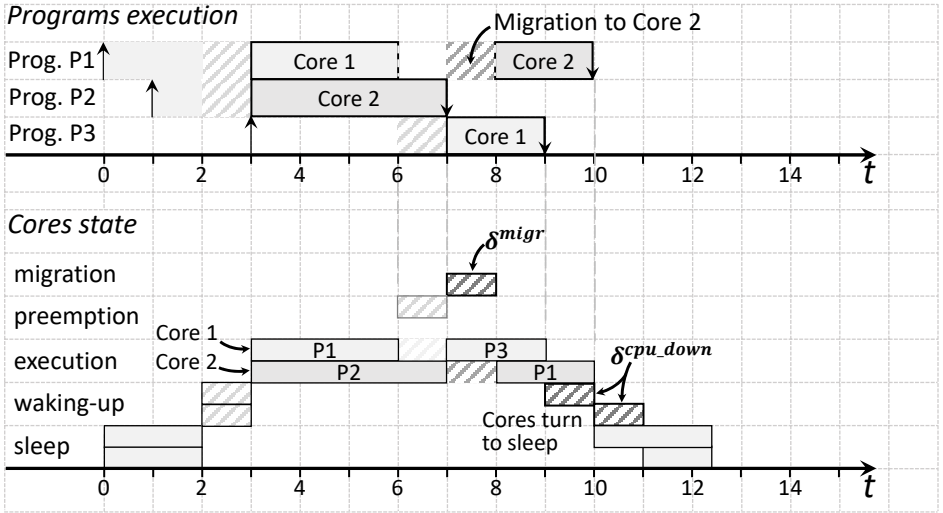
Observe that in the presented execution schedule more than 30% of its duration is wasted on core overheads due to its transitions between execution and other states.



(a) Single processor executing three programs with requirements in Fig. 3.2.



(b) Program P_2 execution with DVFS enabled



(c) Programs execution over 2 “no DVFS" cores with requirements in Fig. 3.2

Fig. 3.1. Single and multi-processor programs execution

Programs	Execution time, ms		Energy use, mJ	
	Core (no DVFS)	Core (slower)	Core (no DVFS)	Core (slower)
P1	5	7	9	8
P2	4	5	10	6
P3	2	3	7	6

Fig. 3.2. Programs requirements for a single core

In fact, this is a highly relevant problem for realistic systems, so that it must be carefully accounted when making scheduling decisions.

Modern cores actually support a varying execution speed due to so-called Dynamic Voltage and Frequency Scaling (DVFS). However, DVFS again imposes additional performance overheads. As an illustration, consider an execution in Fig. 3.1b, assuming programs requirements for a slower core are listed in Fig. 3.2. Starting from $t = 6$, the execution is as following:

$t = 6$: Core's frequency decreases;

$t = 7$: P2 executes;

$t = 9$: The core stalls (see Fig. 3.1a);

$t = 12$: P2 completes;

Below we summarize energy consumed for P2 execution. Note that use of DVFS significantly reduced it from 10 to 6 units at moderate performance degradation.

3.2 Homogeneous multiprocessor case

A sample multiprocessor execution [6]–[10] is provided in Fig. 3.1c, assuming cores are homogeneous that is they provide same execution speed without DVFS. The figure illustrates the following:

$t = 0$: P1 arrives, both cores sleep;

$t = 1$: P2 arrives;

$t = 2$: Cores wake up;

$t = 3$: P1 starts on Core 1, P2 - on Core 2, P3 arrives;

$t = 6$: P3 preempts P1 on Core 1;

$t = 7$: P2 completes, P1 migrates to Core 2;

$t = 9$: P3 completes, Core 1 turns down;

$t = 10$: P1 completes, Core 2 turns down;

Compared to a single processor case, the programs are completed at time 10 instead of 15. Also, a multiprocessor allowed programs to migrate between cores and to execute in a single core mode (from $t = 9$ in Fig. 3.1c) by turning down cores. Although, both migrations and awake/sleep transitions impose additional time overheads.

3.3 Heterogeneous multiprocessor case

Heterogeneous hardware (HW) contains processing units of different types, such as CPU, GPU, TPU and NPU [11]. Some of them are designed for general-

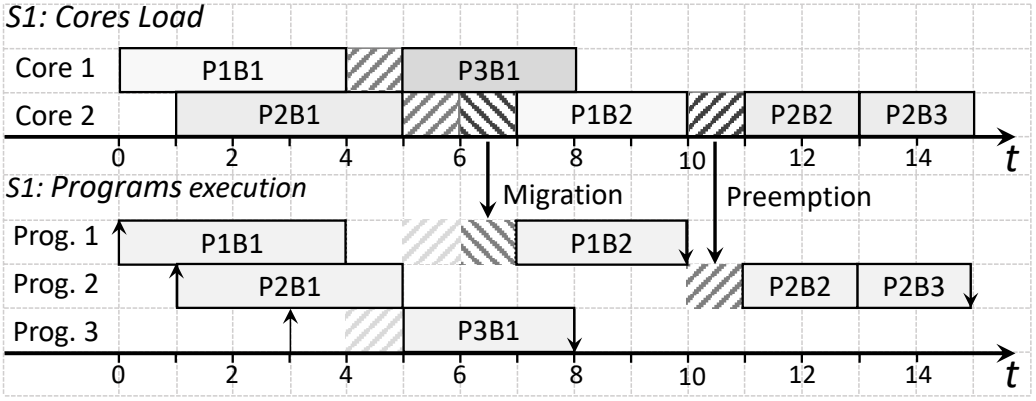


Fig. 3.3. Schedule S1 with preemptions and migrations considered

purpose computational tasks, such as single- or multicore Central Processing Units (CPU) used for operating systems management, while others - for application-specific tasks, such as Graphic, Tensor or Neural Processing Units (GPU, TPU, NPU) used for Artificial Intelligence training and inference tasks.

Despite significant variation in execution speed and energy consumption, these PUs share common HW organizational principles originally coming from a classical single processor case described in Section 3.1. In fact, the major key differences are in the number of cores and supported instruction sets. For example, in Fig. 1.2a the “big" cluster provides only four cores, while GPU and NPU provide thousands and hundreds respectively.

Common execution principles

We illustrate a concurrent programs execution upon different PU types, considering preemptions and migrations, in Fig. 3.3, which extends the Schedule S1 of our toy example. The programs execute as following:

$t = 0$: P1 starts on Core 1;

$t = 1$: P2 starts on Core 2;

$t = 4$: P3 preempts P1;

$t = 5$: P1 preempts P2;

$t = 6$: P1 migrates to Core 2;

$t = 7$: P1 resumes.

Both preemptions and migrations require a context switch, which is to save the state of a preempted program and load the preemptee's. Such a context switch impose significant performance overhead, which in reality vary from tens to hundreds microseconds [12]–[14].

3.4 Scheduling types and overheads

In our toy example we assume preemptions and migrations are possible. In reality, this is not always a case. Next, we describe different schedules types as well as summarize time overheads relevant for our analysis.

Scheduling types

We distinguish scheduling types from the preemptiveness and migrativeness perspectives. A program execution is:

- Preemptive: it can be preempted by another program;
- Non-preemptive: once started, it cannot be preempted;
- Migrative: it can migrate between processing units (PUs);
- Non-migrative: it is statically assigned to a specific PU.

A preemption might occur either at any time or at a completion time of a program block[1]. For example, in Fig. 3.3 preemptions occur at $t = 4$ and 5 after completion of the first blocks of Prog. P1 and P2 correspondingly, while migration - at $t = 6$ when P1 switched from Core 1 to Core 2.

We note that execution preemptions and migrations on one side provide scheduling flexibility[2], but however at additional overhead costs. As an example, consider the execution schedule in Fig. 3.3: Core 2 spends 11 time units for programs execution and 3 for context switches, which is 20% of its operational time. Anyway, both preemptions and migrations are widely used and incomparable from the performance perspective. Preemptions offer increased processor utilization, while migrations - reduced programs execution time. For example, in Fig. 1.4 Schedule S2 has no migrations and greater total execution time comparing to S1, which has 1 migration.

Performance overheads

Any scheduler imposes various hardware-related execution overheads, which are related to principles discussed in Chapter. 3, such as preemptiveness and migrativeness, cache misses, and other:

- Preemptive and migrative programs execution:

  δ^{pm}- preemption overhead;

  δ^{migr}- migration overhead.

- Processing unit mode:

[1]The definition of a program block is to be elaborated later

[2]The flexibility in programs prioritization policy, different scheduling types and other improves hardware utilization

$\delta^{\text{cpu\_wakeup}}$- time overhead related to an operational mode switch;

$\delta^{\text{cpu\_down}}$- time overhead related to a sleep mode switch.

- Dynamic Voltage and Frequency Scaling (DVFS):

  δ^{DVFS}- time overhead related to the processing frequency adjustment.

- Cache hits and misses:

  $\delta^{\text{cache\_miss}}$ - time overhead related to a cache miss.

3.5 Existing heterogeneous schedulers

Several heterogeneous schedulers are proposed in the literature, including Heterogeneous Earliest-Finish-Time (HEFT) [15]–[18], Critical-Path-on-a-Processor (CPOP) [15], [16], StarPU [19], Fast Load Balancing (FLP) [2], [20], Fast Critical Path (FCP) [2], [21], and Heterogeneity-Aware Signature-Supported scheduler (HASS) [22]. These schedulers are however provide suboptimal solutions based on different heuristics. Some of them employ fixed priority assignments for programs, although without any knowledge about the optimality of those priorities. Other schedulers use greedy time optimization, static cores allocations (also known as partitioned scheduling), and other.

Chapter 4

Performance Metrics of Program Execution

A widely-established metric to balance time and energy performance of a program execution is Energy-Delay Product (EDP) [5], [8], [23], [24] defined by:

$$\rho = \mathsf{E} * \mathsf{T}^{\mathsf{RT}} \tag{4.1}$$

where E is energy consumed by a processing unit for a program execution and T^{RT} is program response time discussed below. EDP metric is similar to power-delay product (PDP) used in digital electronics[1].

4.1 Time performance metrics

The major time performance metric is program response time denoted by T^{RT}, which is the time from its invocation until completion. For example, in a non-preemptive scenario of Fig. 4.1a, the program response time is 12, and in a

[1] https://wikipedia.org/wiki/Power-delay_product

preemptive in Fig. 4.1b it is 15 time units. In turn, T^{RT} is subject to various other metrics, including preemption and migration overheads discussed in Section 3.4, latency and execution time, discussed next.

Consider a 3-blocks program execution in Fig. 4.1b. The time since program's release until its actual start is latency (Δt^{l}), which is 2 time units in the figure. Next 4 time units are program's first block execution time, which is the time since a block start until its completion. The sum of all blocks execution times is program execution time (t^{exec}). While, the sum of blocks preemption times is program preemption time (t^{pm}), which is the time when the program is preempted by other programs.

With these metrics we express program response time as:

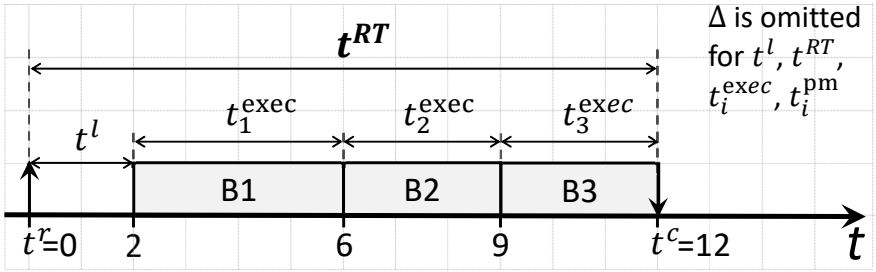
$$\mathsf{T}^{\mathsf{RT}} = t^{\mathsf{c}} - t^{\mathsf{r}} = \Delta t^{\mathsf{l}} + \Delta t^{\mathsf{pm}} + \Delta t^{\mathsf{exec}} * 1.25 \tag{4.2}$$

where t^{r} and t^{c} are program release and completion times. We note that performance overheads are not yet considered. Instead, we assume that they take at most 25% of program execution time.

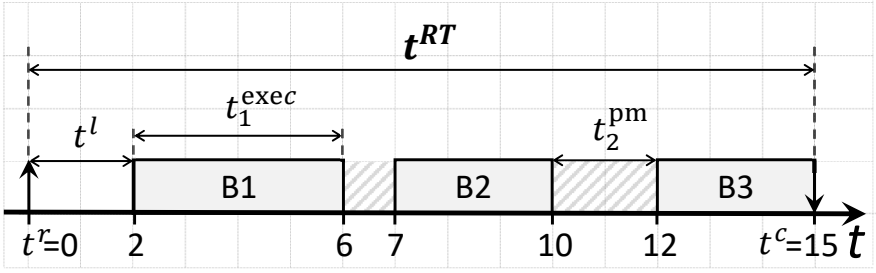
For our toy example, below in Fig. 1.4 we compute average program response and execution times for two schedules. Notice that schedule S1 has higher average response time, but lower average execution time than S2. Also, S1 preemption time is twice larger than in S2.

4.2 Energy performance metrics

Nowadays energy efficiency deserves attention and is considered equally with program response time. However, unlike response time, energy consumption is difficult to measure. Typical direct measurement approaches with specialized devices,



(a) Non-preemptive execution



(b) Preemptive execution at blocks completion

Fig. 4.1. Time metrics of an N-blocks program execution

such as wattmeter and multimeter, which are attached to an operating hardware, are often impracticals. This motivates the development of theoretical techniques to model energy consumption. This modeling mostly relies on the analysis of different program execution events collected from the following sources[2]:

- Hardware: Performance monitor counters (PMCs);
- Operating system: Kernel counters and tracepoints;
- Tracing: Dynamic instrumentation of user-level software.

Specifically, PMC represented as registers collect statistics of CPU and memory units. While tracing in software collects the statistics on a higher level program execution. Then, this statistics, such as number of clock cycles, integer or floating point instructions, branch mispredictions, cache misses, page faults, context switches and migrations, is aggregated into a value representing energy consumption of a program execution. Some studies expect that a modeling error is within

[2]https://www.brendangregg.com/perf.html

5-10% compared to the direct measurements[25], [26] or cycle-accurate simulation [27].

Energy consumption denoted by E is defined by:

$$\mathsf{E} = \mathsf{P} * \mathsf{T}^{\mathsf{RT}} \tag{4.3}$$

where P is average power in Watts used by hardware for a program execution, and T^{RT} is program response time in seconds. Hence, E is measured in Watt-seconds, which are actually Joules denoted by J.

Moreover, various other energy performance metrics are available, such as Energy per Instruction (EPI), Energy per Operation (EPO), which are however not yet considered in our analysis.

4.3 Optimization objective

Our key optimization objective is a so-called Energy-Delay Product (EDP) metric [5], [24], which we denote by ρ. It aggregates both time and energy performance metrics of all executing programs

For an individual program EDP is computed by:

$$\rho = \mathsf{E} * \Delta t^{\mathsf{RT}} \tag{4.4}$$

Here, both energy and time are considered equally, while in alternatives to EDP, such as $\mathsf{ED}^2\mathsf{P}$ or $\mathsf{ED}^3\mathsf{P}$, time is critical:

$$\rho^2 = \mathsf{E} * \Delta t^{\mathsf{RT}^2} \tag{4.5}$$

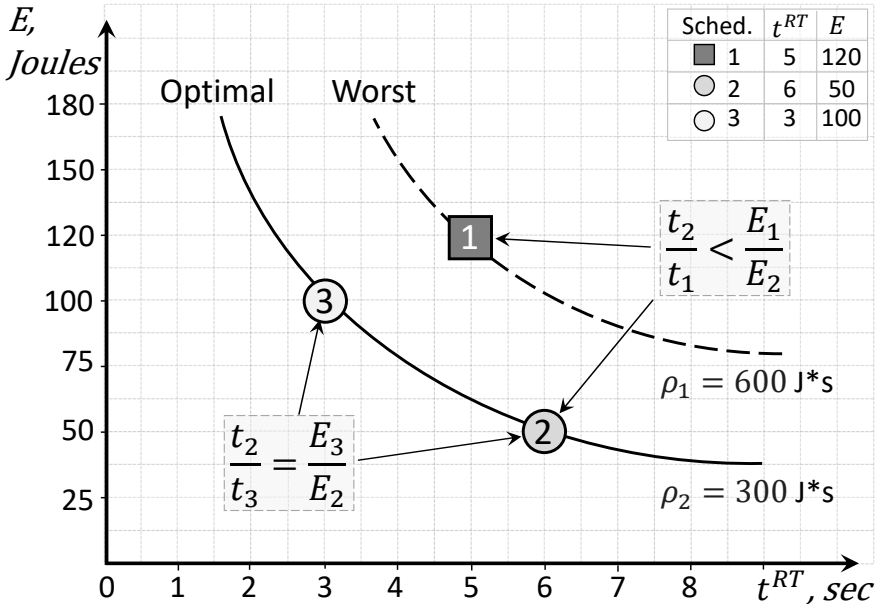


Fig. 4.2. Trade-off between response time and energy consumption. An optimal line (Sched. 2 and 3) has better trade-off.

$$\rho^3 = E * \Delta t^{RT^3} \tag{4.6}$$

Consider in Fig. 4.2, which depicts energy consumption vs. average program response of three abstract schedules. Schedule 1, represented as a dark square, has EDP of 300 Joules-second, which is indicated by the hyperbolic dashed-line. While schedules 2 and 3, represented by two lighter circles, have equal EDP of 150. Note the difference in ratios of times to energy between schedule 1-2 and 3-2. Compared to schedule 1, the execution time of schedule 2 increased less than energy consumption decreased. While between schedules 2 and 3 these time/energy ratios are the same. Hence, schedules 2 and 3 are equal and better than schedule 1 from the EDP-perspective.

The EDP metric is extensively used later in our analysis starting from Section 5.3.

Chapter 5

Optimized Heterogeneous Scheduling Driven by State-Transition Graphs

We are now ready to describe our optimized heterogeneous scheduler. First we provide the necessary background on state-transition graphs, which are the core of our scheduler, with Section 5.1 discussing the case of a single program, Section 5.2 describing the composition of graphs for multiple programs, and Section 5.3 summarizes the overall scheduling decisions mechanism.

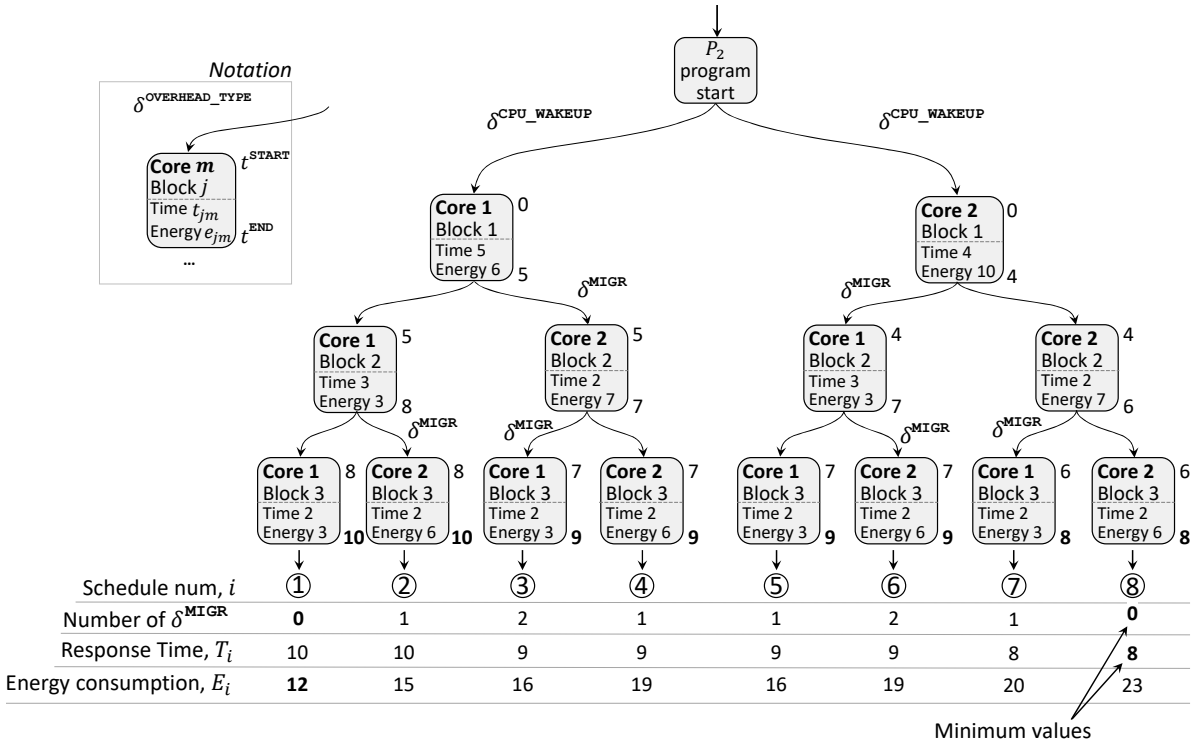


Fig. 5.1. The state-transition graph of the program P_2 from Table 1.1

5.1 Step 1: Individual program schedules

To represent all possible execution schedules for a given program we use a state-transition graph (STG). Graph states correspond to different allocation scenarios of heterogeneous hardware to program blocks. A simple graph example is provided in Fig. 5.1, which corresponds to the P_2 program with execution requirements from Table 1.1. Next we formalize the notion of such a state-transition graph.

Consider a program of n blocks, denoted by $P = [b_1, \ldots, b_n]$. Formally, its state-transition graph is defined according to the next principles.

Graph state is a tuple:

$$x = \left(m, j^{\text{block}}, t^{\text{start}}, t^{\text{end}}\right), \tag{5.1}$$

where:

1. $m \in \{1, \ldots, \mathcal{M}\}$ is an allocated core type;
2. $j^{\mathsf{block}} \in \{1, \ldots, n\}$ is index of an executing block;
3. $t^{\mathsf{start}}, t^{\mathsf{end}}$ are start and end times of an interval allocated for block execution, counted from the initial state start.

Graph transition defines, for a given state x, a set $\hat{X}(x) = \{\hat{x}_1, \ldots, \hat{x}_{\mathcal{M}}\}$ of its reachable successors[1], which is computed by:

$$\begin{matrix} \hat{x}_m \in \hat{X}(x) \\ \forall\, m \in \{1, \ldots, \mathcal{M}\} \end{matrix} \Leftrightarrow \begin{cases} \hat{j}^{\mathsf{block}} = j^{\mathsf{block}} + 1 \\ \hat{t}^{\mathsf{start}} = t^{\mathsf{end}} \\ \hat{t}^{\mathsf{end}} = \hat{t}^{\mathsf{start}} + \left(t^{\mathsf{end}} - t^{\mathsf{start}}\right) \end{cases} \tag{5.2}$$

Set $\hat{X}$ construction is similar to [28].

The execution requirements for time t_{jm} and energy use e_{jm} are derived from t^{start} and t^{end} as following:

$$t_{jm} = t^{\mathsf{end}} - t^{\mathsf{start}}, \tag{5.3}$$

$$e_{jm} = \frac{t^{\mathsf{end}} - t^{\mathsf{start}}}{T_{jm}} * E_{jm} \tag{5.4}$$

Above, T_{jm} and E_{jm} are initial time and energy requirements of block j^{block} to be executed over an m-type core.

Graphical notation for visualisation, also shown in Fig. 5.1, is the following:

- States are represented by rectangles with curved angles;

[1] For briefness, the term “successors" refers to “neighbor successors"

- The state at the top having an incoming arrow is a program initial state, which precedes execution start;
- Curved lines between states are transitions corresponding to some scheduling events;
- $\delta^{\mathsf{overhead\_type}}$, which is placed near a transition arrow, denotes some performance overhead caused by this transition, e.g. due to migrations between cores.

For more descriptive representation, we also show t_{jm} and e_{jm}, which are derived from math notation above.

The state-transition graph of an individual program P assumes an entire hardware platform to be dedicated to the execution of exactly this program P with no concurrent competitors considered. In fact, the state-transition graph is a very flexible and extensible method for modeling various scheduling problems [29].

A program schedule is a graph branch from the initial state to some leaf state. In case of k branches a set of schedules $\mathcal{S}$ for an n-block program's STG is defined as following:

$$\mathcal{S} = \{s_1, s_2, \ldots, s_k\} \tag{5.5}$$

Considering that every graph state corresponds to a certain program block execution, we say that some schedule s_k is a sequence of block states:

$$s_k = [x_k(b_1), x_k(b_2), \ldots, x_k(b_n)] \tag{5.6}$$

where each block state $x_k(b_j)$ is defined by (5.1).

5.2 Step 2: A merged state-transition graph

Typically multiple programs execute concurrently over a shared heterogeneous platform. In this case the scheduling problem is to determine an efficient allocation of heterogeneous processing units to those programs, according to chosen scheduling objectives. Moreover, it must be considered that programs arrive asynchronously and their execution over the same processing unit yields different performance gain: for example, a matrices multiplication program executes drastically faster over a GPU, while strongly sequential non-parallelisable programs typically do not benefit from GPU use in any way, despite its computational power and provided parallelism.

For our efficient scheduling we model possible concurrent executions of given programs. For such modeling we merge STGs of individual programs into a so-called joint state-transition graph (jSTG). An example of a simple jSTG is depicted in Fig. 5.2, and its formal description is given below.

Graphs state of a jSTG, called a system state, is a set of program states $\mathcal{P} = \{y_1, \ldots, y_N\}$. Each program state y_k is a tuple similar to a state x of an individual program's STG described above, i.e. describes an execution of a program block $x.j^{block}$. The state represents an entire or partial block $y_k.j^{block}$ execution. In case of the entire execution, a program state is the same as a state in its STG. In case of partial execution, a program state spreads across l consequent system states Y such that the core type $y.m$ is the same $\forall\, y \in Y$ and $\sum_{k=1}^{l} y_k.t^{end} - y_k.t^{start} = T_{y_k.j^{block},m}$, i.e. the overall time allocated to block j execution equals to required time for an m type core.

Overall, the jSTG state $y_k(\mathcal{P})$ of a programs state set $\mathcal{P} = \{x_1, \ldots, x_N\}$ is

defined as following:

$$y_k(\mathcal{P}) = \{x_1, \ldots, x_N\} \Leftrightarrow x_a.m_{l1} \neq x_b.m_{l2} \\ \forall\, l1, l2 \in \{1, \ldots, \mathcal{M}\} \mid l1 \neq l2 \tag{5.7}$$

Graphical notation of jSTG depicted in Fig. 5.2 provides:

1. Top black circle - the initial state;
2. Dashed rectangles - combined program states[2];
3. Horizontal dashed lines - programs arrivals;
4. “Idle" keywords - no-program state;
5. Paths from initial state to graph leafs - schedules..

The color of program states distinguishes programs, which corresponds to programs colors in Fig. 1.4. The example illustrates schedules S1 and S2 from Fig. 1.4.

The joint state-transition graph of concurrent programs assumes:

- An entire heterogeneous platform is available;
- A single core executes at most one program at a time;
- Performance overheads are insignificant.

The example with three programs is directly extendable to multiple concurrent programs by analogy.

[2]The programs set excludes programs without an allocated core

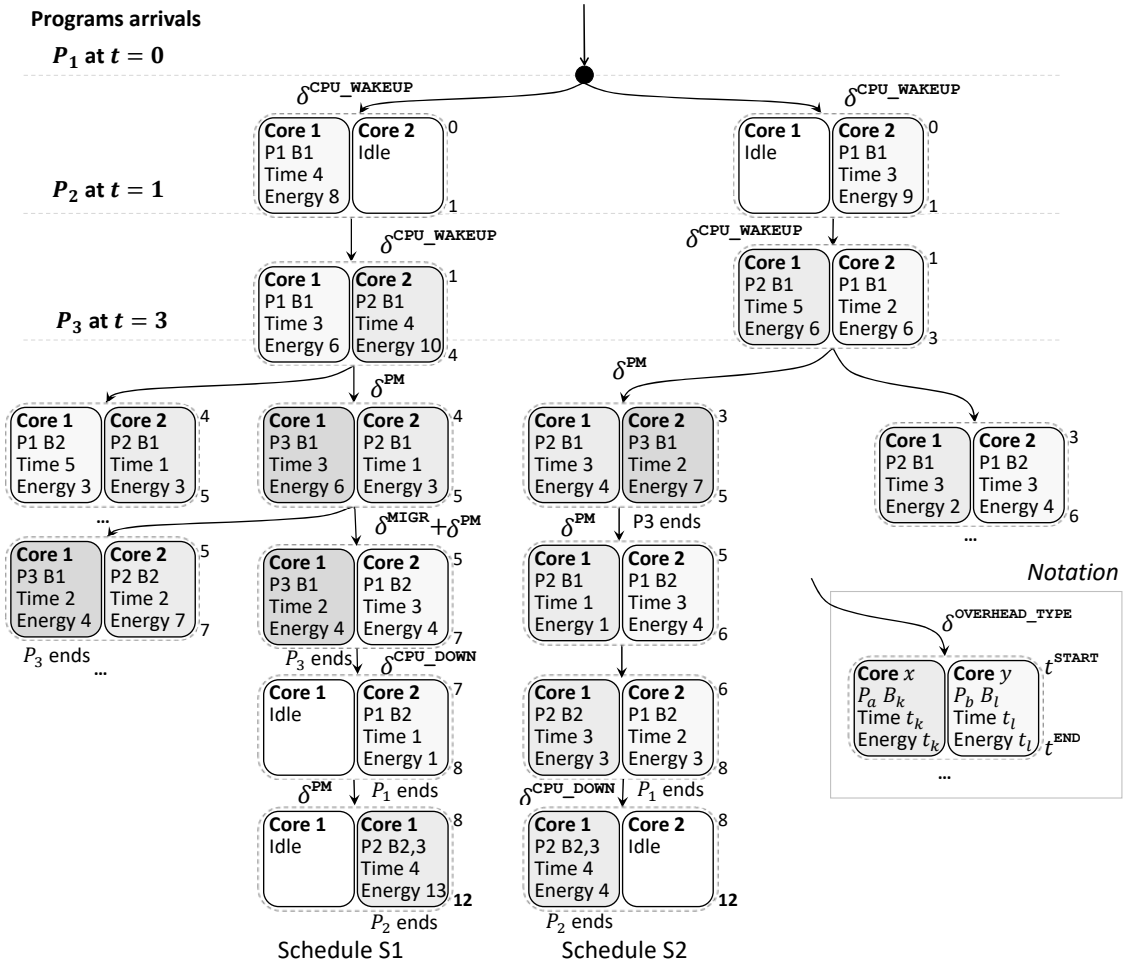


Fig. 5.2. A joint state-transition graph with schedules from Fig. 1.4. The case of non-preemptive scheduling.

The graph in Fig. 5.2 models the execution of three programs P_1, P_2, P_3, with their parameters listed in Table 1.1 over $m = 2$ core types. Observe that programs arrive asynchronously at time instants $t = 0,\ 1$, and 3.

A schedule for a joint-STG represents a sequence of graph system states. Where a system state is a set of program states defined above. A set of such schedules $\mathcal{S}^{\text{joint}}$ is defined by:

$$\mathcal{S}^{\text{joint}} = \{s_1^{\text{joint}} \ldots, s_k^{\text{joint}}\} \tag{5.8}$$

Considering a single schedule s_k^{joint} to be a sequence of l system states for a set of programs $\mathcal{P} = \{p_1, \ldots, p_N\}$. Such a sequence is defined as follows:

$$s_k^{\text{joint}} = [y_{1,k}(\mathcal{P}), \ldots, y_{l,k}(\mathcal{P})] \tag{5.9}$$

where $y_l, k(\mathcal{P})$ is a graph system state defined in (5.7).

5.3 Scheduling procedure: Putting the pieces together

We are finally ready to formalize the procedure of our optimized heterogeneous scheduling. Its key steps are outlined in Fig.5.3 and are the following:

Step 1: A compiler generates STGs from programs source code (Section 5.1);

Step 2: A scheduler merges STGs of programs to be concurrently executed (Section 5.2);

Step 3: The scheduler allocates processing units to programs based on the con-

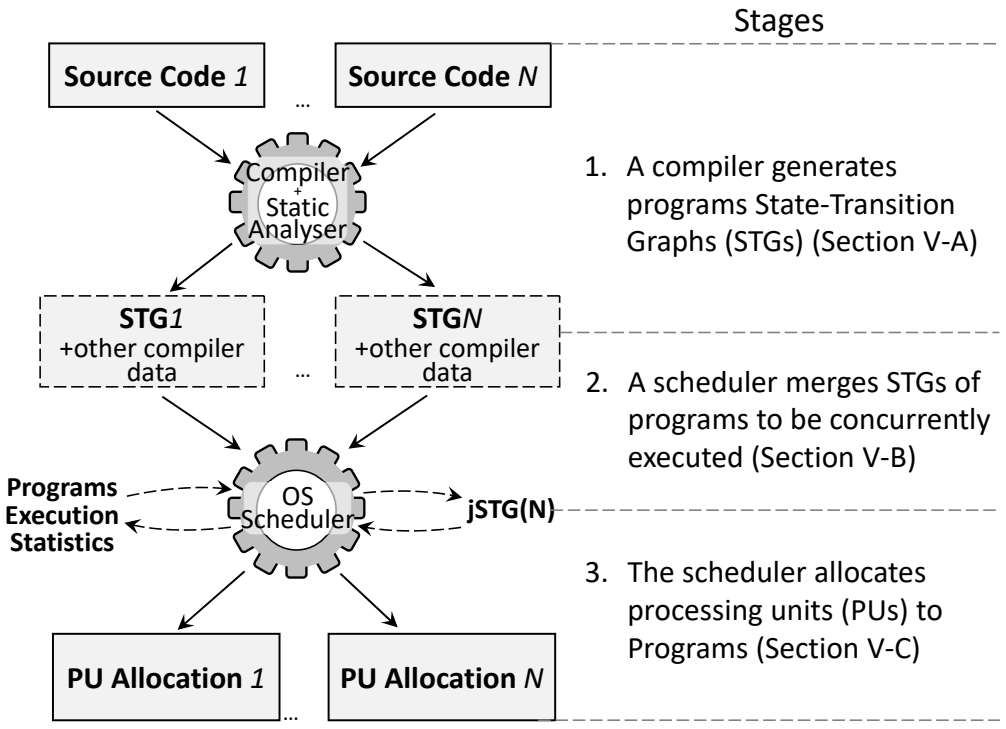


Fig. 5.3. Proposed extension to heterogeneous scheduling

structed joint-STG as discussed below);

To make efficient heterogeneous sheduling decisions, a scheduler analyzes total response times and energy consumptions of different execution schedules[3], which are illustrated below in Fig. 5.1. An operating system scheduler then examines summaries of these schedules aiming at an optimal trade-off between schedule response time and energy consumption, which is discussed in Section 4.3.

[3]The schedules examination procedure can be optimized with state-space pruning techniques [30], which avoids non-optimal transitions in a state-transition graph

Chapter 6

Faster scheduling heuristics

Determination of an optimal schedule requires traversal of an entire state-transition graph for programs, what takes an exponential computation time with a number of programs and other parameters. To make schedules computation faster, we also propose two heuristics: “random walk" and “greedy" - which however result in suboptimal schedules. Anyway, later in Chapter 7 we demonstrate that some heuristic performs almost as an optimal approach, but takes drastically less time to be computed.

Fig. 6.1 demonstrates greedy and random walk applied to a joint-state-transition graph traversal. Below we provide some details on these heuristics, and later in Chapter 7 we evaluate their scheduling performance.

6.1 Random walk

Using a random walk heuristic all scheduling decisions are made randomly without any particular knowledge about graph states. For a given graph state, the next state to be examined is chosen randomly among successors. This procedure

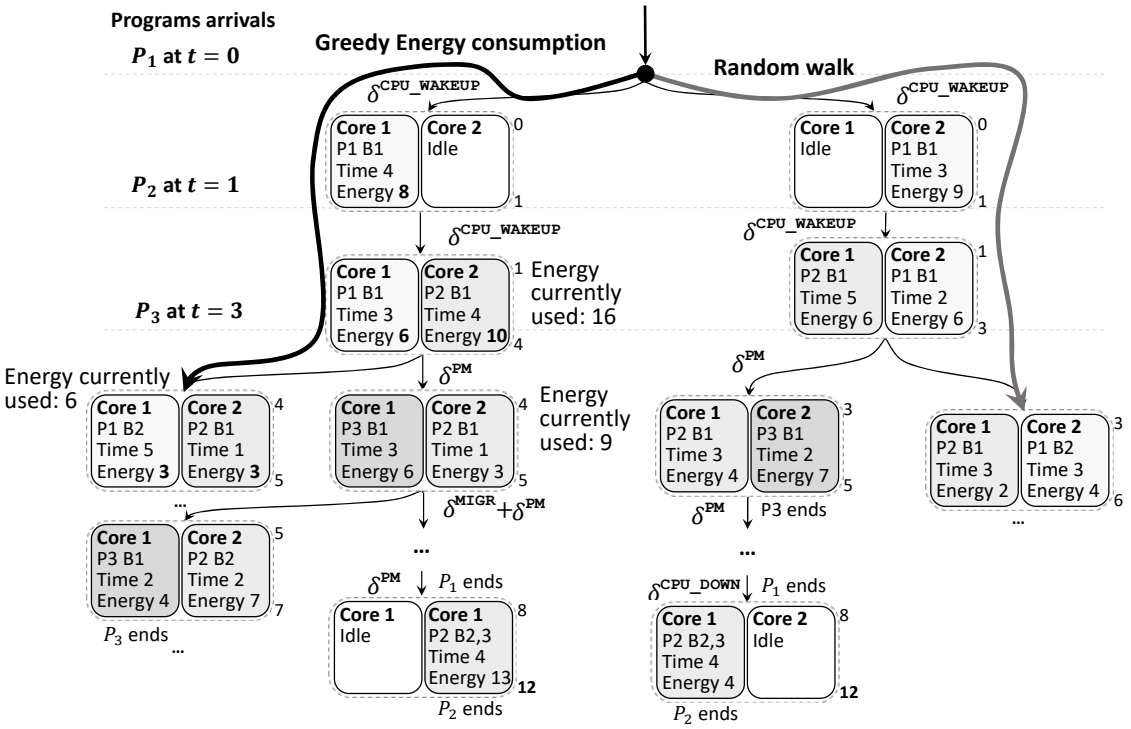


Fig. 6.1. Greedy energy consumption and random walk traversals of a joint state-transition graph. Non-preemptive case.

continues until a graph leaf state is reached producing a certain schedule. In a K-random walks heuristic, this random walk is repeated K times producing K schedules. As a result the most optimal schedule is chosen.

6.2 Greedy walk

In a greedy walk locally optimal scheduling decisions are made. Given a graph state, the neighbouring state with a certain greedy-metric optimized is picked. Such a metric of a graph state can be its total blocks required execution time, energy consumption, number of migrations and preemptions, energy-delay product, and other. The greedy walk completes at a graph leaf state resulting in a certain suboptimal schedule. As a greedy-metric we chose graph state completion time, which minimizes of both total response time and energy-delay product.

Chapter 7

Evaluation

We are now ready to evaluate the efficiency of our schedulers for heterogeneous systems, which are proposed in Chapter 5 and 6. First, in Section 7.1 we describe the default experimental settings and the generation procedure for input programs parameters. Then, in Section 7.2 we describe our C++ Scheduler tool, which we implemented to evaluate our schedulers. By using this tool, we conduct various experiments with numerous input programs and report the results in Section 7.3. In addition, to estimate the tractability scale for our tool, we also plot its computation time along with the experiments.

Experiments are run on a system with next specifications:

- Processor: AMD Ryzen 5 4600h @ 3.4GHz;
- Cores: 6;
- Caches:
  - L1: 64 KB (per core);
  - L2: 512 KB (per core);

TABLE 7.1
Key parameter values

Parameter	Values
Cores number, $m^{\mathsf{slow}} + m^{\mathsf{fast}}$	$1 + 1$
Programs number, N	3
Total blocks number, N^{blocks}	5
Block time, Core {1,2}, t_{jm}	1: [220; 250] 2: [210; 230]
Block energy use, Core {1,2}, e_{jm}	1: [120; 130] 2: [125; 150]

  - L3: 8 MB (shared between cores);

- Memory (RAM): 16 GB DDR4 @ 3200 MHz;
- Operating system: Linux Mint 22 LTS;
- Compiler: g++ 13.3;
- Compiler flag: -O3 (all optimizations are enabled).

7.1 Inputs generation

Input programs are generated for the key parameters listed in Table 7.1, along with their default values and ranges. Their choice considers the assumption for the difference between cores speed and energy consumption. By “Core 1" we denote a slow but energy-efficient core, and “Core 2" - faster but energy-hungry. Then, the generation procedure for one input is the following:

1. Cores number is fixed to m;
2. Programs number N is randomly picked;

3. Total blocks number for all programs in one input is fixed to N^{blocks}, so that

4. N^{blocks} is randomly distributed between N programs;

5. For every program P_i, its block b_j, core m, this block's requirements t_{jm} and e_{jm} for runtime and energy are randomly picked.

All parameter values specified by ranges are picked according to a uniform distribution.

We note that programs requirements, like the ones listed in Table 7.1, in practice are typically collected by using a suitable profiler tool, such as perf[1]. In each experiment the value of one of the parameters is varied, while other parameters are fixed to defaults. For each value of a varying parameter we generate at least 100 inputs and plot the average output value as a datapoint. All experiments assume a synchronous invocation of all programs at $t = 0$.

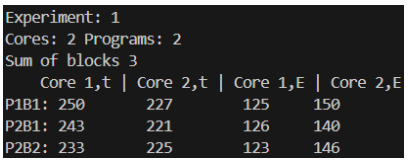
7.2 Scheduler implementation

All our schedulers are implemented by a C++ Scheduler tool. As an input, it takes values of programs parameters, as discussed above. Then it traverses the state-transition graph in a depth-first way, and at the end produces optimal and suboptimal schedules. The tool outputs various summarized statistics regarding computed schedules. Currently the tool does not support migration and preemption overheads, which are to be included.

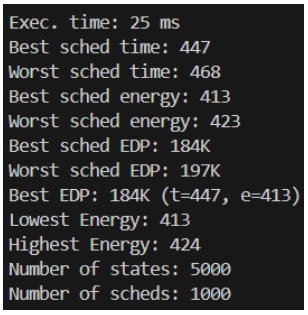
In Fig. 7.1 we depict the screenshots of using our tool through the Terminal:

1. Specification of input parameters: Fig. 7.1a;

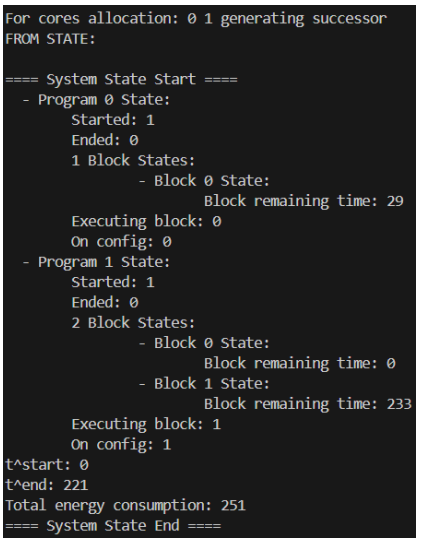
[1] https://perfwiki.github.io



(a) Input specification



(b) Output statistics



(c) Intermediate graph state

Fig. 7.1. Scheduler tool use: screenshots

2. Verbosed graph states traversal: Fig. 7.1c;

3. Results output: Fig. 7.1b.

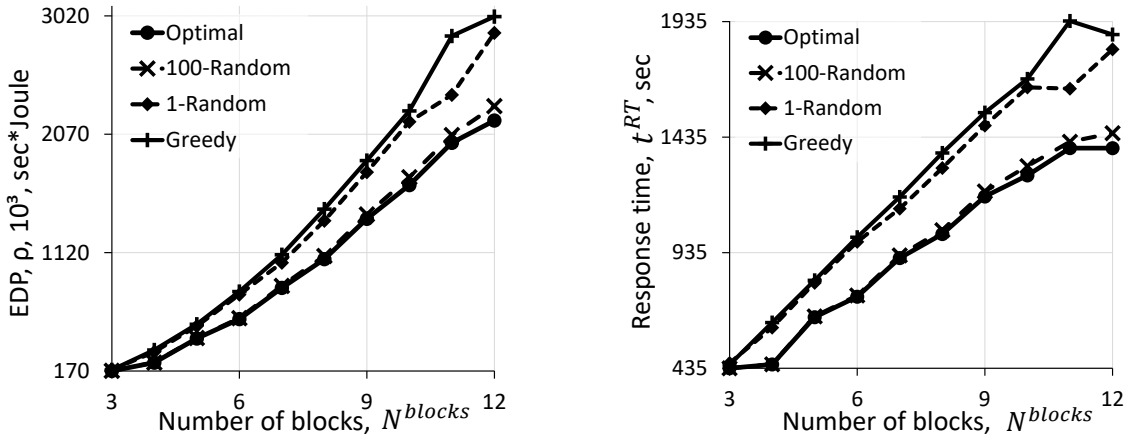
Our Scheduler tool is available on demand.

7.3 Schedulers evaluation

Finally we compare the performance of our schedulers:

- optimal, which is determined by brute-forcing (Chapter 5);
- a random-walk with a customizable number of runs (Section 6.1),
- a greedy walk (Section 6.2).

The key performance metric for schedulers comparison is EDP ρ (see Section 4.3) and programs response times t^{RT} (see Section 4.1). We also assess the Scheduler tool runtime t.



(a) EDP vs. blocks number

(b) Response time vs. blocks number

Fig. 7.2. Varying blocks number: greedy, random walk and optimal schedules EDP and response times

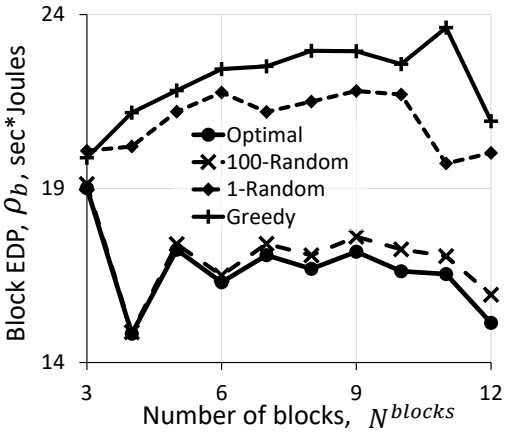
Experiment: Varying blocks number

The first experiment is for a varying total number of programs blocks N^{blocks}. In fact, the N^{blocks} parameter makes the most significant impact on the performance difference between schedulers.

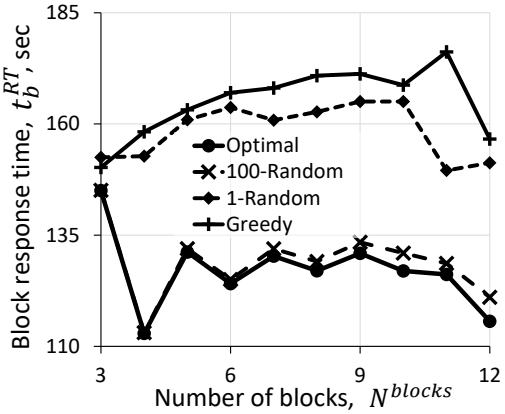
In Fig. 7.2a and 7.3a we report the dependency between EDP ρ and N^{blocks}. We observe that the 100-random walk heuristic is near-optimal, but computed significantly faster, as seen in Fig. 7.4a. Regarding other schedules, the single random walk and the greedy heuristics result in significantly higher EDP, meaning that the schedules they produce consume higher energy. The deviation between the EDP most and least efficient schedules is up to 40-50% for $N^{\text{blocks}} = 12$, and seems to increase further with N^{blocks}.

In turn, computed average programs response times are in Fig. 7.2b and 7.3b. We observe nearly the same trends as for EDP.

The computation time for an optimal schedule is exponential, unlike heuristics, according to Fig. 7.4a (note the logarithmic scale for the time axis).

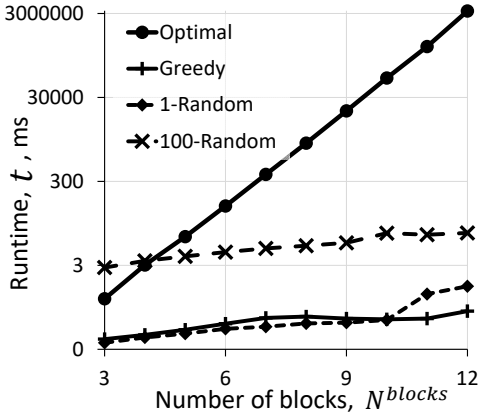


(a) Per block EDP vs. blocks number

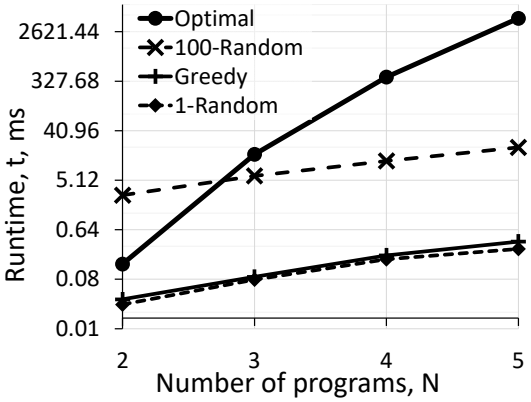


(b) Per block response time vs. blocks number

Fig. 7.3. EDP and runtime per block vs number of blocks. Continuation of Fig. 7.2.

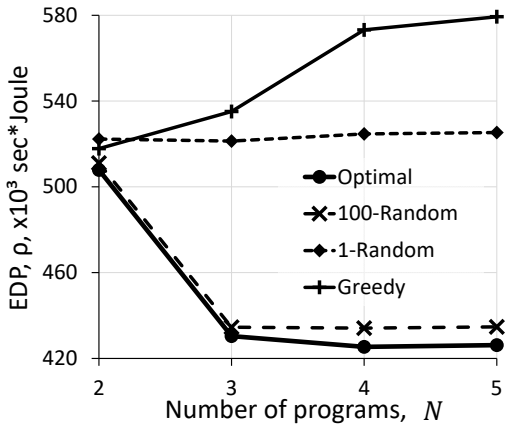


(a) Runtime vs. blocks number

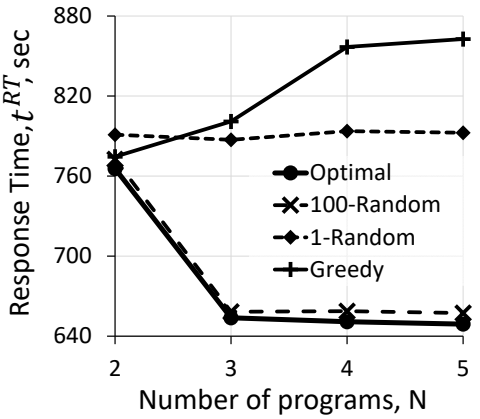


(b) Runtime vs. programs number

Fig. 7.4. Tool runtime dependency of greedy, random walk and optimal schedules



(a) EDP vs. blocks number



(b) Response time vs. programs number

Fig. 7.5. EDP and programs response time dependencies of greedy, random walk and optimal schedules on number of programs

Experiment: Varying program number

Fig. 7.5a illustrates the dependency between EDP ρ and number of programs N. We observe a similar result regarding EDP efficiency: 100-random walk is near-optimal, unlike the single random walk and greedy. The deviation between schedules EDP is again up to 35-40%. Regarding programs response time in Fig 7.5b, the results are similar to programs EDP.

Chapter 8

Conclusion

We tackled the problem of an efficient heterogeneous scheduling by considering both programs response times and energy consumption. We proposed several schedulers, which are an optimal one, and heuristics: random walk and greedy. Our key contribution is the consideration of energy consumption, unlike other existing schedulers, such as HEFT and HASS.

8.1 Detailed contribution

Behind our schedulers is the traversal of a programs state-transition graph. An optimal scheduler needs to traverse an entire huge in size graph, requiring a significant computation time. Heuristics traverse only selected graph fragments aiming at computation time reduction, but produce less efficient schedules. To examine schedulers efficiency, we implemented a C++ Scheduler tool to model concurrent execution of input programs. We also implemented the tool for generating input programs parameters to run experiments. The schedulers performance is evaluated through a series of experiments in terms of energy-delay product (EDP), which

accounts for both programs response times and consumed energy. We show that 100-random walk is near to an optimal schedule, while greedy shows the worst efficiency. Regarding the runtime, an optimal schedule computation takes orders of magnitude larger time due to an exponential complexity compared to heuristics. The EDP deviation for schedules is 40-50%, for selected experimental settings. We do not yet consider migration and preemption overheads, what is to be included.

The C++ tools used in this work are available on demand.

Bibliography cited

[1] J. Anderson, J. Calandrino, and U. Devi, "Real-time scheduling on multicore platforms," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, 2006, pp. 179–190. doi: 10.1109/RTAS.2006.35.

[2] A. Radulescu and A. van Gemund, "Fast and effective task scheduling in heterogeneous systems," in *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556)*, 2000, pp. 229–238. doi: 10.1109/HCW.2000.843747.

[3] E. L. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. O. Alexandre Navaux, and J.-F. Méhaut, "Performance/energy trade-off in scientific computing: The case of arm big.little and intel sandy bridge," *IET Computers & Digital Techniques*, vol. 9, no. 1, pp. 27–35, 2015. doi: https://doi.org/10.1049/iet-cdt.2014.0074. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-cdt.2014.0074. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-cdt.2014.0074.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, isbn: 0128119055.

[5] I. Ratković, N. Bežanić, O. S. Ünsal, A. Cristal, and V. Milutinović, "Chapter one - an overview of architecture-level power- and energy-efficient design techniques," in ser. Advances in Computers, A. R. Hurson, Ed., vol. 98, Elsevier, 2015, pp. 1–57. doi: https://doi.org/10.1016/bs.adcom.2015.04.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245815000303.

[6] P. Gepner and M. Kowalik, "Multi-core processors: New way to achieve high system performance," in *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, 2006, pp. 9–13. doi: 10.1109/PARELEC.2006.54.

[7] L. Chai, Q. Gao, and D. K. Panda, "Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, 2007, pp. 471–478. doi: 10.1109/CCGRID.2007.119.

[8] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 81–92. doi: 10.1109/MICRO.2003.1253185.

[9] B. Nayfeh and K. Olukotun, "A single-chip multiprocessor," *Computer*, vol. 30, no. 9, pp. 79–85, 1997. doi: 10.1109/2.612253.

[10] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005. doi: 10.1109/MC.2005.160.

[11] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Scientific Pro-*

gramming, vol. 18, no. 1, p. 540 159, 2010. doi: https://doi.org/10.3233/SPR-2009-0296. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.3233/SPR-2009-0296. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.3233/SPR-2009-0296.

[12] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," in *ASPLOS IV*, 1991. [Online]. Available: https://api.semanticscholar.org/CorpusID:9051432.

[13] F. M. David, J. C. Carlyle, and R. H. Campbell, "Context switch overheads for linux on arm platforms," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07, San Diego, California: Association for Computing Machinery, 2007, 3–es, isbn: 9781595937513. doi: 10.1145/1281700.1281703. [Online]. Available: https://doi.org/10.1145/1281700.1281703.

[14] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '96, San Diego, CA: USENIX Association, 1996, p. 23.

[15] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, 1999, pp. 3–14. doi: 10.1109/HCW.1999.765092.

[16] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," 3, vol. 13, 2002, pp. 260–274. doi: 10.1109/71.993206.

[17] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, "Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algo-

rithm," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 27–34. doi: 10.1109/PDP.2010.56.

[18] Y. Samadi, M. Zbakh, and C. Tadonki, "E-heft: Enhancement heterogeneous earliest finish time algorithm for task scheduling based on load balancing in cloud computing," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, 2018, pp. 601–609. doi: 10.1109/HPCS.2018.00100.

[19] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 863–874, isbn: 978-3-642-03869-3.

[20] A. Radulescu and A. van Gemund, "Flb: Fast load balancing for distributed-memory machines," in *Proceedings of the 1999 International Conference on Parallel Processing*, 1999, pp. 534–541. doi: 10.1109/ICPP.1999.797442.

[21] A. Rădulescu and A. J. C. van Gemund, "On the complexity of list scheduling algorithms for distributed-memory systems," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS '99, Rhodes, Greece: Association for Computing Machinery, 1999, pp. 68–75, isbn: 158113164X. doi: 10.1145/305138.305162. [Online]. Available: https://doi.org/10.1145/305138.305162.

[22] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, *et al.*, "Hass: A scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, Apr. 2009, issn: 0163-5980. doi: 10.1145/1531793.1531804. [Online]. Available: https://doi.org/10.1145/1531793.1531804.

[23] A. Venkat and D. M. Tullsen, "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 121–132, Jun. 2014, issn: 0163-5964. doi: 10.1145/2678373.2665692. [Online]. Available: https://doi.org/10.1145/2678373.2665692.

[24] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 9, pp. 1277–1284, 1996. doi: 10.1109/4.535411.

[25] R. Joseph and M. Martonosi, "Run-time power estimation in high performance microprocessors," in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ser. ISLPED '01, Huntington Beach, California, USA: Association for Computing Machinery, 2001, pp. 135–140, isbn: 1581133715. doi: 10.1145/383082.383119. [Online]. Available: https://doi.org/10.1145/383082.383119.

[26] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00, Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 83–94, isbn: 1581132328. doi: 10.1145/339647.339657. [Online]. Available: https://doi.org/10.1145/339647.339657.

[27] T. Li and L. K. John, "Run-time modeling and estimation of operating system power consumption," ser. SIGMETRICS '03, San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 160–171, isbn: 1581136641. doi: 10.1145/781027.781048. [Online]. Available: https://doi.org/10.1145/781027.781048.

[28] A. Burmyakov, E. Bini, and E. Tovar, "An exact schedulability test for global fp using state space pruning," Nov. 2015, pp. 225–234. doi: 10.1145/2834848.2834877.

[29] A. Burmyakov and B. Nikolic, "An exact comparison of global, partitioned, and semi-partitioned fixed-priority real-time multiprocessor schedulers," *J. Syst. Archit.*, vol. 121, p. 102 313, 2021. doi: 10.1016/J.SYSARC.2021.102313. [Online]. Available: https://doi.org/10.1016/j.sysarc.2021.102313.

[30] A. Burmyakov, E. Bini, and C. Lee, "Towards a tractable exact test for global multiprocessor fixed priority scheduling," *IEEE Trans. Computers*, vol. 71, no. 11, pp. 2955–2967, 2022. doi: 10.1109/TC.2022.3142540. [Online]. Available: https://doi.org/10.1109/TC.2022.3142540.