HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Alberto Dassatti, Bruno
Da Rocha Carvalho

Laboratoire de High Performance Coding

semestre printemps 2025

Laboratoire 6 : Mesure de consommation énergétique

Temps à disposition : 2 périodes (1 séances de laboratoire).

1 Objectifs de ce laboratoire

L'objectif de ce laboratoire est de vous initier à une nouvelle dimension de mesure : celle de la consommation énergétique. En effet, dans certains contextes, la gestion de l'énergie revêt une importance capitale, et minimiser la consommation est une préoccupation constante. Ce tutoriel vous guidera à travers différentes méthodes d'évaluation de la consommation énergétique, vous permettant ainsi d'acquérir une compréhension approfondie de cette mécanique essentielle.

De manière générale, mesurer la consommation énergétique d'un système nécessite un support matériel ou des instruments de mesure externes. Cependant, ces outils ne permettent souvent pas de faire un lien suffisamment précis entre l'exécution du code et la consommation mesurée. C'est précisément pour pallier cette limitation que ce laboratoire a été conçu : il vous propose une approche intégrée permettant d'observer plus finement les impacts de vos choix logiciels sur la consommation énergétique.

Vous pouvez également, en complément de cette donnée de laboratoire, consulter cet article, qui donne accès à des exemples de code ainsi qu'à des explications plus approfondies.

2 Lexique

- PowerCap : C'est un sous-système du noyau linux conçu pour gérer et contrôler la consommation d'énergie des différents composants matériels d'un système informatique, tels que les processeurs, les mémoires et les périphériques.
- RAPL : Est l'acronyme de "Running Average Power Limit. C'est une technologie d'Intel, intégrée dans certains de leurs processeurs, qui permet de surveiller et de réguler la consommation d'énergie des CPU en temps réel.
- Zone package : Cette zone correspond à l'ensemble du package du CPU, c'est-à-dire l'ensemble des coeurs de calcul d'un processeur physique.
- Zone core : Cette zone correspond à des unités de calcul individuelles à l'intérieur du CPU.
- Zone uncore : Cette zone représente les composants du processeur qui ne sont pas des coeurs de calcul, tels que le cache, le contrôleur mémoire et dans certains cas, le processeur graphique.
- Zone DRAM : Cette zone correspond à la mémoire dynamique du système, c'est-à-dire la RAM.
- Zone système (pSys) ou platforme : Cette zone représente la consommation d'énergie totale du système, y compris tous les composents mentionnés ci-dessus (package, core, uncore, DRAM, etc.)

3 Mesures globales

3.1 perf

Avec perf, on est capable de mesurer la consommation énergétique globale d'un programme. Pour cela, perf met à disposition 5 événements prédéfinis :

- `power/energy-cores/`
- `power/energy-gpu/`
- `power/energy-pkg/`
- `power/energy-psys/`
- `power/energy-ram/`

Grâce à ces événements, on peut analyser la consommation énergétique d'un programme dans son ensemble.

3.1.1 Example d'utilisation

Pour illustrer cela, j'ai créé un programme simple. Ce programme crée un tableau de floats d'environ 70 Mo, rempli de 1.

Une fonction additionne tous les éléments de ce tableau et retourne le résultat, qui doit correspondre au nombre d'éléments du tableau.

Pour simplifier, j'analyse la consommation globale (pSys)

Voici l'analyse avec perf :

```
$ perf stat -e power/energy-psys/ ./build/global_power

Performance counter stats for 'system wide':

             1.11 Joules power/energy-psys/

       0.049966633 seconds time elapsed
```

3.2 likwid

Pour ceux chez qui likwid fonctionne, cet outil propose également une méthode de mesure, bien qu'il propose des domaines plus restreints selon votre architecture.

Cependant, une fonctionnalité que likwid permet, que perf ne permet pas nécessairement, est de profiler le maximum et le minimum de consommation d'une zone.

```
$ likwid-powermeter -i

--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
CPU type:       Intel Kabylake processor
CPU clock:      2.30 GHz
--------------------------------------------------------------------------------
Base clock:     2300.00 MHz
Minimal clock:  400.00 MHz
Turbo Boost Steps:
C0 4900.00 MHz
C1 4800.00 MHz
C2 4300.00 MHz
C3 4300.00 MHz
```

```
--------------------------------------------------------------------------------
Info for RAPL domain PKG:
Thermal Spec Power: 15 Watt
Minimum Power: 0 Watt
Maximum Power: 15 Watt
Maximum Time Window: 0 micro sec

Info for RAPL domain PLATFORM:
Thermal Spec Power: 736 Watt
Minimum Power: 0 Watt
Maximum Power: 736 Watt
Maximum Time Window: 976 micro sec

Info about Uncore:
Minimal Uncore frequency: 400 MHz
Maximal Uncore frequency: 4600 MHz

Performance energy bias: 6 (0=highest performance, 15 = lowest energy)

--------------------------------------------------------------------------------
```

On peut également faire une mesure ciblée sur un programme, comme avec perf : `likwid-powermeter ./build/global_power`

4 Mesures ciblées

Les mesures globales ne permettent pas toujours d'être précises. On souhaite souvent vérifier un algorithme spécifique ou une portion précise de code en évitant tout le reste.

4.1 likwid

Vous avez déjà utilisé Likwid dans le deuxième laboratoire, et évidemment, vous pouvez également utiliser des groupes de performance avec `likwid-perfctr` pour observer ces métriques.

Comme dans le deuxième laboratoire, vous pouvez cibler des portions de code avec les marqueurs `LIKWID_MARKER_START/STOP`.

Ensuite, comme dans le deuxième laboratoire, vous pouvez avoir des événements différents selon votre architecture. Vous pouvez simplement vérifier quels compteurs d'énergie vous avez avec la commande suivante :

```
$ likwid-perfctr -e | grep PWR

PWR0, Energy/Power counters (RAPL)
PWR1, Energy/Power counters (RAPL)
PWR2, Energy/Power counters (RAPL)
PWR3, Energy/Power counters (RAPL)
PWR4, Energy/Power counters (RAPL)
PWR_PKG_ENERGY, 0x2, 0x0, PWR0
PWR_PP0_ENERGY, 0x1, 0x0, PWR1
PWR_PP1_ENERGY, 0x4, 0x0, PWR2
PWR_DRAM_ENERGY, 0x3, 0x0, PWR3
PWR_PLATFORM_ENERGY, 0x5, 0x0, PWR4
```

Ces compteurs retournent tous des valeurs en joules.

4.2 perf

Nous allons créer nos propres marqueurs, comme avec likwid. Pour commencer, préparons une structure de données et des constantes :

```
typedef struct {
    int ctl_fd;
    int ack_fd;
    bool enable;
} PerfManager;

const char *enable_cmd = "enable";
const char *disable_cmd = "disable";
const char *ack_cmd = "ack\n";
```

La structure contient deux descripteurs de fichiers qui permettent de communiquer avec perf, et un booléen de contrôle. Ces deux descripteurs de fichiers pointeront vers des FIFOs.

Les constantes nous permettent de définir les commandes à envoyer à perf.

Maintenant, nous allons créer la fonction d'interaction qui enverra les commandes :

```
void send_command(PerfManager *pm, const char *command) {
    if (pm->enable) {
        write(pm->ctl_fd, command, strlen(command));
        char ack[5];
        read(pm->ack_fd, ack, 5);
        assert(strcmp(ack, ack_cmd) == 0);
    }
}
```

Ensuite, nous préparons notre structure avec toutes les informations nécessaires. *Nous reviendrons plus tard sur les variables d'environnement récupérées.*

```
void PerfManager_init(PerfManager *pm) {
    char *ctl_fd_env = getenv("PERF_CTL_FD");
    char *ack_fd_env = getenv("PERF_ACK_FD");
    if (ctl_fd_env && ack_fd_env) {
        pm->enable = true;
        pm->ctl_fd = atoi(ctl_fd_env);
        pm->ack_fd = atoi(ack_fd_env);
    } else {
        pm->enable = false;
        pm->ctl_fd = -1;
        pm->ack_fd = -1;
    }
}
```

Enfin, nous préparons les fonctions de pause et de reprise de perf :

```
void PerfManager_pause(PerfManager *pm) {
    send_command(pm, disable_cmd);
}

void PerfManager_resume(PerfManager *pm) {
    send_command(pm, enable_cmd);
}
```

Maintenant, il suffit d'initialiser notre structure dans le main, de demander de reprendre l'analyse avant la portion de code que nous souhaitons analyser et de l'arrêter après.

```
int main(int argc, char** argv) {
    PerfManager pmon;
    PerfManager_init(&pmon);

    PerfManager_resume(&pmon);
        // Code a mesurer
    PerfManager_pause(&pmon);

    return 0;
}
```

Pour pouvoir lancer notre application, nous devons préparer quelques éléments afin de pouvoir correctement communiquer et contrôler perf. Nous recommandons de tout mettre dans un script afin d'éviter de nombreuses commandes manuelles.

Nous devons d'abord créer deux FIFOs, une pour envoyer les commandes de contrôle et l'autre pour recevoir les réponses de perf.

```
mkfifo perf_fifo.ctl
mkfifo perf_fifo.ack
```

Ensuite, nous devons associer les deux FIFOs à des descripteurs de fichiers.

```
exec {perf_ctl_fd}<>perf_fifo.ctl
exec {perf_ack_fd}<>perf_fifo.ack
```

Enfin, pour faciliter l'accès aux deux descripteurs de fichiers depuis le code C, créons des variables d'environnement.

```
export PERF_CTL_FD=${perf_ctl_fd}
export PERF_ACK_FD=${perf_ack_fd}
```

Il ne reste plus qu'à lancer perf en liant les deux descripteurs de fichiers aux événements que nous souhaitons analyser, avec une option de délai qui permet de le lancer désactivé. Il attendra ensuite que nous lui demandions de commencer via le contrôleur.

```
perf stat \
    -e power/energy-psys/ \
    --delay=-1 \
    --control fd:${perf_ctl_fd},${perf_ack_fd} \
    -- ./exec
```

4.3 powercap

Powercap est un librairie qui exploite le sous-système du noyaux linux (powercap) pour pouvoir récupérer la consomations d'énergie des différents composants matériels.

Pour illustrer plus simplement l'utilisation on va observer un code pas à pas.

```
int main(int argc, char** argv){
    // On récupère le nombre de package disponible dans l'architecture
    uint32_t npackages = powercap_rapl_get_num_instances();

        uint64_t energy_uj1[npackages];
        uint64_t energy_uj2[npackages];

    // On créer une instance pour chaque package
    powercap_rapl_pkg pkg[npackages];

    // On défini la zone que l'on souhaite analyser, ici psys
    powercap_rapl_zone zone = POWERCAP_RAPL_ZONE_PSYS;

    // On vérifie si les packages supporte la zone qu'on souhaite analysée.
    bool supported[npackages];
    bool has_one_supported = false;
    for (size_t i = 0; i < npackages; i++) {
        supported[i] = powercap_rapl_is_zone_supported(&pkg[i], zone);

        if (supported[i] != 1){
            printf("RAPL is not supported on package %ld\n", i);
        } else {
            has_one_supported = true;
        }
    }

    if (!has_one_supported) {
        printf("No supported package for %s\n", to_string(zone));
        continue;
    }

        // Maintenant on lance la mesure de consommation énergétique
        for(size_t j = 0; j < npackages; ++j){
        if (supported[j]){
            if (powercap_rapl_get_energy_uj(&pkg[j], zone, &energy_uj1[j])) {
                printf("        Failed to get energy on package %ld\n", j);
                break;
            }
        }
```

```
    }

    // CODE MESURER


    // On remesure la consomation énergétique
    for (size_t j = 0; j < npackages; j++) {
        if (supported[j]){
            if (powercap_rapl_get_energy_uj(&pkg[j], zone, &energy_uj2[j])) {
                printf("        failed to get energy on package %ld\n", j);
                break;
            }
        }
    }


    for (size_t i = 0; i < npackages; i++) {
        powercap_rapl_destroy(&pkg[i]);
    }

    return EXIT_SUCCESS;
}
```

Finalement il suffirat de faire la différence entre la mesure après et celle avant le code a mesurer pour avoir la consommation en [uj].

5 Travail demandé

Dans le laboratoire SMID, vous avez eu l'opportunité d'améliorer un programme de segmentation d'image. Vous pouvez récupérer les deux versions, vectorisée et non vectorisée (fournie), et comparer leurs consommations en utilisant au moins trois des méthodes proposées.

N'oubliez pas que le compilateur peut parfois vectoriser des éléments sans que vous ne l'ayez explicitement demandé. L'utilisation de flag comme "no-tree-vectorize" peut être utile pour votre analyse.

Si vous n'êtes pas satisfait de votre version vectorisée de la segmentation d'image, vous pouvez créer un programme simple qui somme toutes les valeurs d'un grand tableau de floats (de préférence rempli de 1 pour éviter les overflows). Vous développerez une version non vectorisée et une version vectorisée.

Vous devez rendre un rapport présentant vos analyses et les différences obtenues.