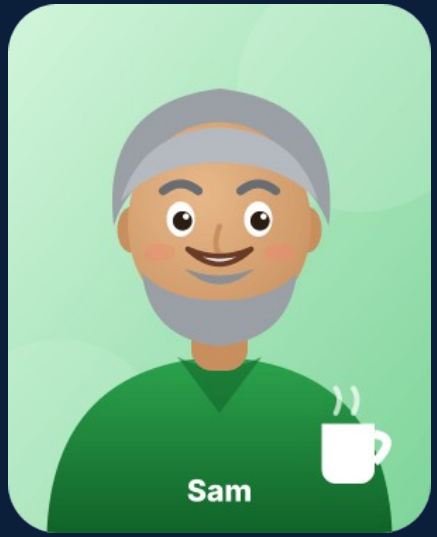
HEF-130 · MISSION 03 / 10

Profilage CPU



Ne devine pas où le temps passe : demande au profileur. L'intuition se trompe presque toujours sur l'endroit du goulot.



Badge : Chasseur de points chauds

100 min · avancé

L'histoire

Kodi est sûr que la génération du rapport mensuel est lente à cause de la base de données. Il passe une journée à optimiser les requêtes. Gain : 3 %. Sam lance le profileur : 82 % du temps est passé dans une fonction de mise en forme de texte, appelée 400 000 fois.



> *« L'intuition nous dit "c'est la base", ou "c'est ce qui est compliqué", dit Sam. Le profileur dit où le CPU est réellement consommé. Il coûte cinq minutes et fait économiser des jours. Et il suit une règle : on optimise ce qui est chaud, jamais ce qui est froid. »*

Tu apprends à profiler avec `cProfile`, à lire un profil, à distinguer temps propre et temps cumulé, à bâtir ton propre profileur par échantillonnage, et à corriger un point chaud en vérifiant le gain.

Ta mission

À la fin, je serai capable de…

1. utiliser `cProfile` et `pstats` pour identifier les fonctions les plus coûteuses
2. distinguer temps propre (tottime) et temps cumulé (cumtime)
3. expliquer la différence entre profilage par instrumentation et par échantillonnage
4. corriger un point chaud (algorithme, structure de données, appels répétés) et mesurer le gain
5. appliquer Amdahl à une série d'optimisations

Défi d'échauffement

Défi d'échauffement

Une cuisine met 45 minutes à servir un plat. Le chef pense que la cuisson est trop lente. Un observateur chronomètre chaque étape : préparation 5 min, cuisson 8 min, attente d'un ingrédient qu'on va chercher en cave 30 min, dressage 2 min. Où faut-il agir ? Qu'aurait-on fait en suivant l'intuition du chef ?

Le concept

Pourquoi profiler

Les programmeurs devinent mal où le temps passe. Les loi de Pareto s'applique : **80 % du temps dans 20 % du code** (souvent 95 % dans 5 %). Optimiser du code froid est du temps perdu et rend le code plus complexe pour rien. Citation célèbre de Knuth : *« l'optimisation prématurée est la racine de tous les maux »* — optimiser **avant d'avoir mesuré**.

Deux familles de profileurs

	Instrumentation (déterministe)	Échantillonnage (statistique)
Principe	enregistre chaque appel et retour de fonction	regarde périodiquement (100 fois/s) quelle fonction s'exécute
Précision	exacte (nombre d'appels, temps)	approximative, mais statistiquement fiable

Le concept

Deux familles de profileurs (suite)

	Instrumentation (déterministe)	Échantillonnage (statistique)
Coût	élevé (ralentit le programme, fausse les petites fonctions)	faible (utilisable en production)
Exemples	`cProfile`, `profile`	`py-spy`, `perf`, profileurs de production

`cProfile` : le profileur standard de Python

```
python -m cProfile -s cumtime mon_script.py        # en ligne de commande
```

Ou dans le code :

```
profil = cProfile.Profile()
profil.enable(); ma_fonction(); profil.disable()
stats = pstats.Stats(profil).sort_stats("tottime")
stats.print_stats(10)
```

Le concept

Lire un profil

Colonne	Sens
`ncalls`	nombre d'appels (`10/3` : 10 appels dont 3 primitifs, la différence venant de la récursion)
`tottime`	temps **propre** : passé dans la fonction elle-même, sans ses sous-appels
`percall`	temps par appel
`cumtime`	temps **cumulé** : la fonction et tout ce qu'elle appelle

- Un `**tottime** élevé` désigne le code qui calcule : c'est le **point chaud**.
- Un `**cumtime** élevé` avec un faible `tottime` désigne un chef d'orchestre : cherche plus bas.
- Un `**ncalls** énorme` avec un petit `percall` : le coût vient du nombre d'appels (peut-être un appel dans une boucle qui pourrait sortir de la boucle).

Le concept

Les points chauds fréquents

Cause	Symptôme dans le profil	Remède
Algorithme quadratique (HEF-030)	`tottime` croissant plus vite que les données	meilleur algorithme / structure de données
Recalcul identique	même fonction avec les mêmes arguments, `ncalls` élevé	mémoïsation, sortir de la boucle
Concaténation de chaînes en boucle	`tottime` dans le code de concaténation	`join`
Recherche dans une liste	`in` sur liste dans une boucle	`set` / `dict`
Conversions inutiles	`str`, `int`, `list(...)` dans la boucle	éviter
Expressions régulières recompilées	`re.compile` appelée en boucle	compiler une fois

Le concept

Les points chauds fréquents (suite)

Cause	Symptôme dans le profil	Remède
Boucles Python sur des données numériques	boucle pure	fonctions natives, bibliothèques vectorisées

Méthode en 5 étapes

1. **Baseline** : mesurer le temps total sur une charge représentative.
2. **Profiler** sur cette charge.
3. **Identifier** le point chaud (celui qui a le plus fort `tottime`).
4. **Corriger** une seule chose.
5. **Remesurer** et vérifier que le **résultat est identique**. Recommencer, jusqu'à atteindre le budget.

Le concept

Amdahl appliqué à l'optimisation

Si une fonction représente une part `f` du temps et qu'on la rend `s` fois plus rapide, l'accélération globale est `1 / ((1 − f) + f / s)`. Rendre ×100 plus rapide un code qui ne pèse que 5 % du total donne au mieux ×1,05. Il faut **viser le plus gros morceau** d'abord, et recommencer : le point chaud suivant devient visible.

Profileur par échantillonnage maison

En Python, `sys._current_frames()` donne la pile de chaque thread. Un thread secondaire qui l'interroge 200 fois par seconde et compte les fonctions rencontrées construit un profil statistique à très faible coût.

Le concept

Profileur par échantillonnage maison (suite)

🧠 ***Astuce** : profile avec des **données réalistes**. Un profil sur trois lignes de données ment sur le comportement avec trois millions.*

Exemple pas à pas

Problème : profiler un générateur de rapport lent, identifier le point chaud avec `cProfile`, le corriger, et vérifier que le résultat est identique et que le gain correspond à la prédiction d'Amdahl.

```
import cProfile, io, pstats, time

# ---------------- le programme lent ----------------
def charger(n):
    return [{"membre": f"m{i % 800}", "cours": f"HEF-{(i * 7) % 90:03d}", "montant
   ": (i * 37) % 200} for i in range(n)]

def est_membre_premium(membre, premiums):
    return membre in premiums                                  # premiums est une
    LISTE : recherche en O(n)

def formater_ligne(e):
    texte = ""
    for morceau in (e["membre"], " ", e["cours"], " ", str(e["montant"]), "€"):
        texte += morceau                                       # concaténation ré
```

Exemple pas à pas (suite)

```
   pétée
    return texte

def rapport_lent(evenements, premiums):
    total_premium, lignes = 0, []
    for e in evenements:
        if est_membre_premium(e["membre"], premiums):
            total_premium += e["montant"]
        lignes.append(formater_ligne(e))
    return total_premium, "\n".join(lignes)

EVENEMENTS = charger(30000)
PREMIUMS = [f"m{i}" for i in range(0, 800, 3)]                    # ~267 membres pre
   mium

# ---------------- profilage ----------------
profil = cProfile.Profile()
profil.enable()
resultat_lent = rapport_lent(EVENEMENTS, PREMIUMS)
```

Exemple pas à pas (suite)

```
profil.disable()

sortie = io.StringIO()
stats = pstats.Stats(profil, stream=sortie).sort_stats("tottime")
total = stats.total_tt
lignes = sorted(((v[2], k[2]) for k, v in stats.stats.items() if k[0] == "<string>
   " or "est_membre" in k[2] or "formater" in k[2] or "rapport" in k[2]), reverse=
   True)
print(f"temps total profilé : {total:.2f} s")
for tottime, nom in lignes[:4]:
    print(f"  {nom:28s} tottime = {tottime:6.3f} s ({tottime / total:5.1%})")
chaud = lignes[0][1]
assert chaud == "est_membre_premium"                          # le point chaud
   n'est PAS le formatage « laborieux »

# ---------------- correction : une liste → un ensemble ----------------
def rapport_v2(evenements, premiums):
    premiums = set(premiums)                                  # recherche en O(1
   )
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    total_premium, lignes = 0, []
    for e in evenements:
        if e["membre"] in premiums:
            total_premium += e["montant"]
        lignes.append(formater_ligne(e))
    return total_premium, "\n".join(lignes)

def temps(f, *args, n=5):
    meilleur = float("inf")
    for _ in range(n):
        d = time.perf_counter(); f(*args); meilleur = min(meilleur, time.perf_coun
   ter() - d)
    return meilleur

t0, t1 = temps(rapport_lent, EVENEMENTS, PREMIUMS), temps(rapport_v2, EVENEMENTS,
   PREMIUMS)
assert rapport_v2(EVENEMENTS, PREMIUMS) == resultat_lent        # RÉSULTAT IDENTIQ
   UE
print(f"v1 {t0 * 1000:7.1f} ms → v2 {t1 * 1000:7.1f} ms  (x{t0 / t1:.1f})")
```

Exemple pas à pas

Exemple pas à pas (suite)

```
assert t0 / t1 > 3

# ---------------- le point chaud suivant ----------------
profil = cProfile.Profile(); profil.enable(); rapport_v2(EVENEMENTS, PREMIUMS); pr
   ofil.disable()
stats = pstats.Stats(profil, stream=io.StringIO())
suivant = max(((v[2], k[2]) for k, v in stats.stats.items() if k[2] in ("formater_
   ligne", "rapport_v2", "<listcomp>")), default=(0, ""))
print("nouveau point chaud :", suivant[1])

def rapport_v3(evenements, premiums):
    premiums = set(premiums)
    total_premium = sum(e["montant"] for e in evenements if e["membre"] in premium
   s)
    return total_premium, "\n".join(f"{e['membre']} {e['cours']} {e['montant']}€"
   for e in evenements)

t2 = temps(rapport_v3, EVENEMENTS, PREMIUMS)
assert rapport_v3(EVENEMENTS, PREMIUMS) == resultat_lent
```

Exemple pas à pas (suite)

```
print(f"v3 {t2 * 1000:7.1f} ms  (x{t0 / t2:.1f} par rapport à la version initiale)
   ")
assert t2 < t1

# ---------------- Amdahl : prévoir le gain avant de coder ----------------
part_chaude, acceleration_locale = 0.80, 50
print(f"si un point chaud pèse {part_chaude:.0%} et devient x{acceleration_locale}
    plus rapide : gain global x{1 / ((1 - part_chaude) + part_chaude / acceleratio
   n_locale):.1f}")
assert round(1 / ((1 - part_chaude) + part_chaude / acceleration_locale), 1) == 4.
   6
```

Lecture :

1. Le profil désigne `est_membre_premium` (recherche dans une liste) comme point chaud — pas la mise en forme que l'intuition aurait suspectée.
2. Passer la liste en **ensemble** donne un gain de plusieurs fois, **résultat identique** vérifié.

Exemple pas à pas (suite)

3. Après correction, le profil change : un nouveau point chaud apparaît (le formatage). On recommence — c'est un cycle.
4. **Amdahl** prédit le gain : un point chaud de 80 % rendu 50 fois plus rapide ne donne que ×4,6 global. Les 20 % restants deviennent la limite.

À toi, avec aide

Compare quatre façons de construire une grande chaîne à partir de 100 000 petits morceaux, avec `timeit` (ou ton banc de la mission 2) :

1. `texte += morceau` dans une boucle ;
2. `"".join(liste)` ;
3. `io.StringIO` avec `write` ;
4. une liste de morceaux puis `"".join` sur un générateur.

Classe-les, calcule le rapport entre la plus lente et la plus rapide, et assure-toi que les quatre produisent le même texte. Conclus : quelle serait ta règle générale ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Prépare `morceaux = [str(i) for i in range(100000)]` une fois.

Indice 2

Utilise `timeit.timeit(fonction, number=5)`.

Indice 3

Vérifie l'égalité des résultats avec `assert`.

À toi, avec aide · réponse

Réponse

```
import io, timeit
morceaux = [str(i) for i in range(100000)]

def concat():
    t = ""
    for m in morceaux: t += m
    return t
def jointure(): return "".join(morceaux)
def flux():
    s = io.StringIO()
    for m in morceaux: s.write(m)
    return s.getvalue()
def generateur(): return "".join(m for m in morceaux)

assert concat() == jointure() == flux() == generateur()
resultats = {f.__name__: min(timeit.repeat(f, number=3, repeat=5)) / 3 for f in (c
   oncat, jointure, flux, generateur)}
for nom, t in sorted(resultats.items(), key=lambda kv: kv[1]):
    print(f"{nom:10s} {t * 1000:7.2f} ms")
```

À toi, avec aide · réponse

Réponse (suite)

```
assert min(resultats, key=resultats.get) == "jointure"
```

`"".join(liste)` est la plus rapide et la plus lisible. `StringIO` est très proche. La concaténation répétée peut sembler correcte en CPython (une optimisation interne la rend souvent linéaire) mais elle n'est **pas garantie** dans toutes les implémentations ni dans tous les contextes. **Règle générale** : construire une collection de morceaux, puis assembler **une seule fois** à la fin.

À toi, en solo

Écris ton propre profileur par échantillonnage. Avec `threading` et `sys._current_frames()` :

1. un thread échantillonneur interroge la pile du thread principal toutes les 2 ms et note le nom de la fonction en haut de pile, ainsi que la chaîne d'appelants ;
2. à l'arrêt, il produit un tableau : fonction, nombre d'échantillons, pourcentage, avec distinction « propre » (en haut de pile) et « cumulé » (présente dans la pile) ;
3. teste-le sur un programme qui passe ~70 % de son temps dans `fonction_a`, ~25 % dans `fonction_b` et ~5 % ailleurs ;
4. vérifie que ton profil s'approche de ces proportions (à 15 points près) ;
5. compare avec `cProfile` sur le même programme.

À toi, en solo · réponse

Pistes de correction

```
import cProfile, collections, io, pstats, sys, threading, time

class Echantillonneur:
    def __init__(self, intervalle=0.002):
        self.intervalle, self.propre, self.cumule, self.total = intervalle, collec
   tions.Counter(), collections.Counter(), 0
        self._stop, self._cible = threading.Event(), threading.get_ident()
        self._thread = threading.Thread(target=self._boucle, daemon=True)
    def _boucle(self):
        while not self._stop.is_set():
            frame = sys._current_frames().get(self._cible)
            if frame is not None:
                noms = []
                while frame:
                    noms.append(frame.f_code.co_name); frame = frame.f_back
                self.propre[noms[0]] += 1
                for n in set(noms): self.cumule[n] += 1
                self.total += 1
            time.sleep(self.intervalle)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def __enter__(self): self._thread.start(); return self
    def __exit__(self, *a): self._stop.set(); self._thread.join()
    def tableau(self):
        return [(n, c, c / self.total) for n, c in self.propre.most_common()]

def travailler(duree):
    fin = time.perf_counter() + duree
    x = 0
    while time.perf_counter() < fin:
        x += 1
    return x

def fonction_a(): travailler(0.35)
def fonction_b(): travailler(0.125)
def programme():
    fonction_a(); fonction_b(); travailler(0.025)

with Echantillonneur() as e:
    programme()
```

Pistes de correction (suite)

```
for nom, c, part in e.tableau()[:4]:
    print(f"{nom:12s} {c:4d} échantillons ({part:5.1%})")
cumul_a, cumul_b = e.cumule["fonction_a"] / e.total, e.cumule["fonction_b"] / e.to
   tal
assert abs(cumul_a - 0.70) < 0.15 and abs(cumul_b - 0.25) < 0.15
assert e.cumule["programme"] / e.total > 0.95                          # programme
    englobe tout (temps cumulé)

pr = cProfile.Profile(); pr.enable(); programme(); pr.disable()
st = pstats.Stats(pr, stream=io.StringIO())
cum = {k[2]: v[3] for k, v in st.stats.items()}
print(f"cProfile (cumulé) : a = {cum['fonction_a'] / cum['programme']:.0%}, b = {c
   um['fonction_b'] / cum['programme']:.0%}")
assert abs(cum["fonction_a"] / cum["programme"] - cumul_a) < 0.15
```

Les deux méthodes s'accordent. L'échantillonneur n'enregistre qu'**environ 500 points par seconde** et ralentit à peine le programme ; `cProfile` intercepte chaque appel et fausse davantage les petites fonctions fréquentes. C'est pourquoi les outils de production utilisent l'échantillonnage.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Optimiser par intuition	On améliore ce qui est froid	Profiler d'abord, optimiser le point chaud
Profiler sur des données minuscules	Le profil ne reflète pas la charge réelle	Données représentatives
Changer plusieurs choses à la fois	On ne sait pas ce qui a aidé	Un changement, une mesure
Oublier de vérifier le résultat	Une version plus rapide mais fausse	Comparer les sorties avant et après

Mini-boss

Série d'optimisations guidée par le profil. On te donne (ou tu écris) un programme `rapport.py` de 80 lignes avec **quatre** points chauds cachés (liste au lieu d'ensemble, recalcul dans une boucle, concaténation de chaînes, expression régulière recompilée) sur 50 000 enregistrements.

1. mesure la baseline avec le banc de la mission 2 ;
2. profile avec `cProfile`, désigne le point chaud n°1, prédis le gain avec Amdahl, corrige, mesure ;
3. recommence jusqu'à avoir traité les quatre ;
4. à chaque étape : tableau (étape, correction, temps, accélération cumulée, accélération prédite par Amdahl) ;
5. vérifie à chaque étape que la sortie est **strictement identique** ;
6. trace un graphique texte du temps après chaque étape et explique pourquoi le gain par étape diminue ;

Mini-boss (suite)

7. conclus : à quel moment aurais-tu arrêté d'optimiser, et pourquoi ?

Mini-boss · réponse

Pistes de correction

Un bon travail : un tableau du type « étape 1 : liste→ensemble, ×6 ; étape 2 : sortir le calcul de la boucle, ×1,9 ; … » avec des gains décroissants (loi des rendements décroissants : chaque étape traite une part plus petite du temps restant), la prédiction d'Amdahl proche de la mesure, des sorties identiques à chaque étape. L'arrêt se justifie par l'atteinte du budget ou par le fait que le prochain gain est inférieur au coût de maintenance du code plus complexe.

Quiz éclair

Quiz éclair

1. Dans un profil, `tottime` mesure :

A. le temps total du programme
B. le temps passé dans la fonction elle-même, sans ses sous-appels
C. le nombre d'appels

2. La première étape d'une optimisation est de :

A. réécrire le code en C
B. ajouter du cache partout
C. profiler pour identifier le point chaud

3. Un profileur par échantillonnage :

A. observe périodiquement quelle fonction s'exécute, avec un faible coût

Quiz éclair (suite)

B. enregistre chaque appel sans coût

C. remplace les tests

4. 🔁 *Révision mission 02* : Pour décrire la latence ressentie par les moins chanceux, on regarde :
A. la moyenne uniquement

B. les percentiles élevés (p95, p99)

C. le nombre de serveurs

5. 🔁 *Révision mission 02* : Si les intervalles de confiance de deux versions se chevauchent largement :
A. la version B est meilleure

B. la version A est meilleure

C. la différence n'est pas démontrée

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

1. Quelle partie de ton code crois-tu lente par intuition, sans l'avoir profilée ?
2. À partir de quel gain estimé arrêterais-tu d'optimiser un point chaud ?

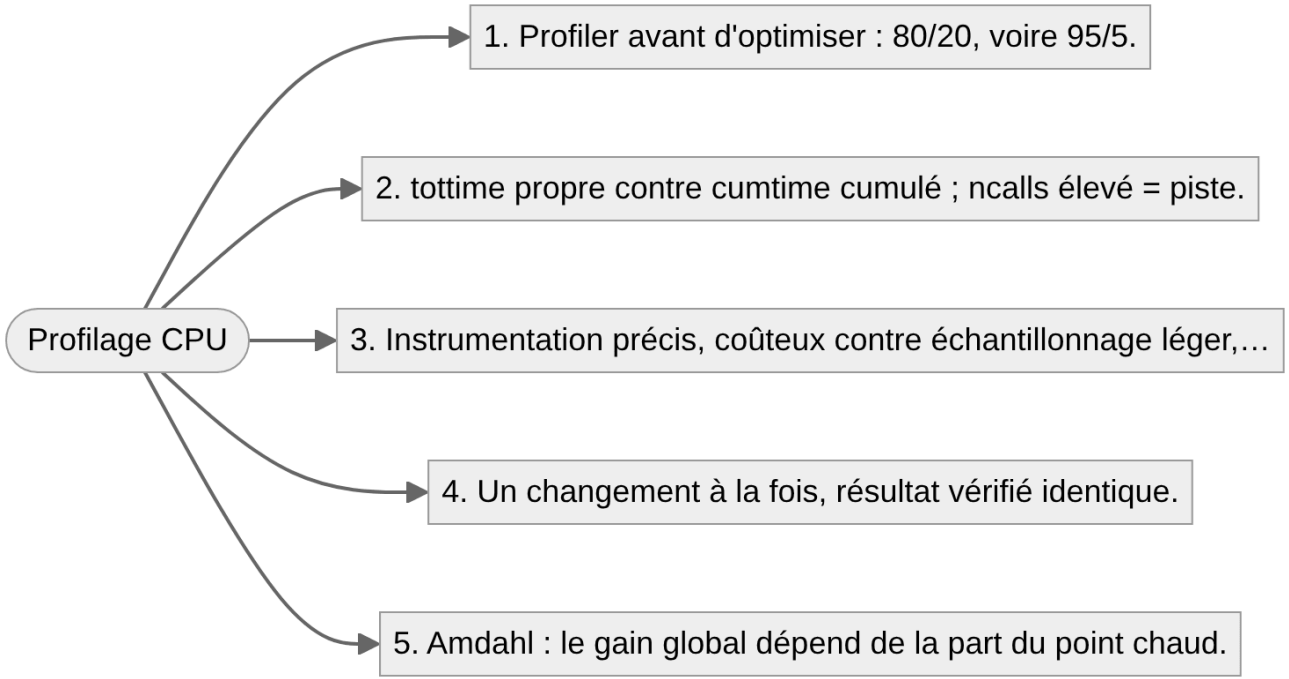
À retenir

À retenir

1. Profiler avant d'optimiser : 80/20, voire 95/5.
2. tottime (propre) contre cumtime (cumulé) ; ncalls élevé = piste.
3. Instrumentation (précis, coûteux) contre échantillonnage (léger, statistique).
4. Un changement à la fois, résultat vérifié identique.
5. Amdahl : le gain global dépend de la part du point chaud.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Chasseur de points chauds

Prochaine mission : Mémoire et allocations

