



# Profilage(s)

Hadrien Grasland

2023-10-20



# **Profilage(s) : Rencontre des fonctions chaudes de ta région**

Hadrien Grasland

2023-10-20

# Avertissement

- Sujet trop vaste pour tout couvrir en 1 webinaire !
- On va donc mettre l'accent sur...
  - L'outillage **Linux**, en particulier **perf**
  - Les codes **CPU- et memory-bound**
  - La compilation **Ahead-of-Time (AoT)\***
  - Les processeurs **x86\_64**

\* Stratégie d'implémentation la plus courante pour C/++, Fortran, Rust..., où les programmes sont transformés en binaires standardisés avant exécution → Outillage standardisé !



# Plan d'attaque

- 1. Vue d'ensemble du système**
2. Métriques CPU fines
3. Isolation du code « chaud »
4. Et ensuite ?

# Penser système entier

- Un programme interagit avec de nombreuses **ressources**
  - Matérielles (CPU, RAM, stockage, réseau, GPU...)
  - Logicielles (OS, serveurs distants ex : bases de données...)
- Chacune de ces ressources a des **limites**
  - Liées au matériel ou à une politique d'administration
- Objectif : comprendre **quelles limites affectent le code**
  - Pour ensuite diminuer la pression ou augmenter la limite

# Méthode USE

1. Enumérer les ressources systèmes potentiellement utilisées\*
    - Composants internes, services externes, interconnexions...
  2. Pour chacune de ces ressources, étudier...
    - **Utilisation** (temps relatif/absolu passé sur des requêtes)
    - **Saturation** (nombre de requêtes en attente d'être traitées)
    - **Erreurs** (problèmes empêchant le traitement de requêtes)
- Origine : <https://www.brendangregg.com/usemethod.html>

\* Il faut un peu d'entraînement pour ne pas en oublier → S'aider des experts locaux !

# Conseils pratiques

- Penser à étudier les coeurs CPU, disques... **individuellement**
  - Votre code ne sait pas forcément en utiliser plusieurs
- Penser aux **interconnexions** (CPU-RAM, CPU-GPU, réseau...)
  - Leur perf dépend beaucoup de la granularité des messages  
→ Mesurer à la fois volumétrie/s et messages/s
- Penser au **monde extérieur** (stockage partagé, hyperviseurs...)
  - Vérifier les métriques hôtes + quotas avec les admins

# Quelques outils Linux\*

- **GUI** : Idéalement ksysguard (configurable et mature)
  - Ubuntu 22 : Remplacé un peu vite par Plasma SysMon :-)
- **CLI** :
  - Pseudo-graphique : zenith, nmon, glances, btop++...
  - Tabulaire : sar (aka « sysstat »), dstat, vmstat...
- **Web** : Demander un accès à l'UI de monitoring (grafana ou équivalent) des serveurs de calcul que vous utilisez

\* Sous Windows, le taskmgr standard a pas mal de possibilités, notamment via son mode avancé « resource monitor ». Sous macOS, une référence est Xcode Instruments.





# Demo : Monitoring système



# Plan d'attaque

1. Vue d'ensemble du système
- 2. Métriques CPU fines**
3. Repérage du code « chaud »
4. Et ensuite ?

# Mieux comprendre

- Les métriques système globales sont ambiguës
  - Ex : Comment interpréter une utilisation CPU à 100 % ?
  - Frontend ? ALUs ? Bus CPU <-> RAM ? Autre chose ?
- Pour aller plus loin, on peut utiliser des **métriques à grain fin**
  - Clarifie la nature de la saturation observée
  - Suffit parfois à comprendre son origine
- Très nombreuses ! → Exemples **ciblés** avec l'outil **perf**

# Les ASR détestent cette ruse !

- Les mesures fines avec perf ont des **prérequis assez lourds**
  - **Noyau Linux assez récent** (date sortie  $\geq$  CPU, idéal.  $\geq 5.x$ )
  - **Privilèges utilisateur** (attention aux VMs et conteneurs)
- Peut être **difficile à négocier** sur une machine partagée... :(
  - Prévoir un job simplifié qui tourne sur une machine de dev
  - Vérifiez bien qu'il reste représentatif !

# Exécution de code efficace ?

- **Métriques** : Instructions / cycle + fréquence d'horloge
- **Mesure** : Fait partie de « perf stat <commande> »\*
- **Interprétation** :
  - IPC < 1 toujours suspect mais IPC  $\geq$  1 est ambigu
  - Multi-coeur : On veut clock  $\geq$  spécification « base clock »
  - Séquentiel : Il devrait y avoir du turbo, si pas désactivé
  - Spécifications via <https://ark.intel.com/> & similaire

\* Mesure minimale donnant cette information : -e instructions,cycles,task-clock

# Limité par la hiérarchie mémoire ?

- Version simple :
  - **perf stat -d** inclut les caches posant souvent problèmes
  - -dd ajoute des caches moins « problématiques »
  - Problème : Mesures peu précises (compteurs nombreux)
- Version avancée : « perf list » → Etudier les possibilités
- Mettre les nombres absolus en regard de « instructions »
  - Les pourcentages de petits nombres ne veulent rien dire !

# Limité par les branches ?

- Présent dans la sortie de « perf stat » sans arguments\*
- Mettre « branches » en regard de « cycles »
  - Si ils sont similaires, **limite de 1 branche / cycle** atteinte...
  - Solution : Utiliser moins de branches
- « **branch-misses** » ~ « **branches** » → **Exécution imprévisible**
  - Très coûteux (~dizaines de cycle à chaque ratage)
  - Solutions : Moins de branches ou branches plus prévisibles

\* Mesure minimale donnant l'information : -e instructions,branches,branch-misses

# Limité par les FLOPs ? (spécifique Intel)

- **Métrique « perf stat -M FLOPc » ou équivalent explicite\***
  - Opérations flottantes / cycle
  - Décomposition en scalaire, SSE, AVX...
- **Interprétation :**
  - Max 2 en f64 scalaire, 4 en SSE, 8 en AVX, 16 en AVX-512
  - Limite x2 si f32, x2 si instruction FMA utilisée
  - Si vous atteignez cette limite (bravo!), la seule optim possible est diminuer le nombre de calculs effectués...

\* Exemple pour la simple précision : perf stat -e cycles,fp\_arith\_inst\_retired.scalar\_single,fp\_arith\_inst\_retired.128b\_packed\_single,fp\_arith\_inst\_retired.256b\_packed\_single,fp\_arith\_inst\_retired.512b\_packed\_single



# Bref...

- **Commencer par « perf stat -d »** (problèmes classiques)
- Pour aller plus loin, prendre le temps d'étudier « **perf list** »
- Ne pas hésiter à formuler des hypothèses et les tester
- Limites de perf stat (si utilisé avec les PMCs comme ici) :
  - Compteurs spécifiques à certains matériels (Intel >> AMD)
  - Trop de mesures simultanées → Précision réduite
  - Aggrégé sur toute l'exécution (--interval peut aider)
  - **N'a pas de sens si le CPU n'est pas la ressource limitante !**



# Demo : perf stat



# Plan d'attaque

1. Vue d'ensemble du système
2. Métriques CPU fines
- 3. Repérage du code « chaud »**
4. Et ensuite ?

# Cibler l'effort

- Ce serait bien de tout optimiser au maximum mais...
  - Pas assez d'heures dans une journée
  - Compromis avec d'autres objectifs\*
- La loi de Pareto peut nous aider
  - Souvent >80 % du temps est passé dans <20 % du code
  - **Avantage** : L'effort bien ciblé est très efficace
  - **Inconvénient** : L'effort mal ciblé est très inefficace
- Enjeu : Trouver les « fonctions chaudes » critiques pour la perf

\* Le code parfait est aussi facile à écrire et maintenir, lisible par d'autres que son auteur, flexible, extensible, ergonomique, conforme à diverses politiques, sécurisé, etc.

# Influence du langage

- Outils standardisés ? Dépend du modèle d'exécution !
  - **AoT** (C/++, Fortran, Rust, Go...) => **Généralement oui**
    - Nécessite des **symboles de debug (app + deps !)**
  - **JiT** (Java, C#, Julia, Numba, ROOT...) => **Peut-être**
    - Parfois il faut compiler le JIT avec un flag, le patcher...
    - Parfois, ce n'est pas possible du tout
  - **Interprété** (CPython, bash...) => **Non, outils spécifiques**
- Ici, on utilise **perf record** → Plutôt pour de l'AoT

# Perf record = Acquisition de données

- **Exemple** : « perf record -F 100 ma\_commande --arguments »
- On dit à perf quel **événement** on veut localiser dans le code
  - Par défaut, passage du temps (« cycles »)
- Toutes les N occurrences de cet événement, perf mesure **où on est dans le code** (IP ou données pour stack trace)
  - N ajusté pour une fréquence d'échantillonnage ~constante
- On en déduit **dans quel code l'événement arrive souvent**

# Les graphes d'appels

- **Question** : Quel code appelle les utilitaires (malloc, rand...) ?
- On utilise la **mesure de piles d'appels** avec `-call-graph=xyz`
  - Config de base recommandée : « `--call-graph=dwarf` »
  - Plus fiable mais plus coûteux : « `dwarf,64000` »
  - Fonctionne en copiant les octets supérieurs de la pile
- **Coûteux** : Baisser freq. échantillonnage avec `-F`, idéalement mettre la sortie sur ramdisk avec `-o /dev/shm/perf.data`

# Perf report = Visualisation TUI

- Principe = **Histogramme** configurable des échantillons : par binaire source, fonction, ligne de code source...
- En présence de piles d'appel, il est **hiérarchique**
  - % du temps passé dans main(), ses enfants, leurs enfants
  - On peut zoomer sur les enfants, récursivement...
- C'est à ce moment là qu'on a besoin des **infos de debug**
  - ...mais c'est une TUI, donc utilisable sur un serveur
  - Par contre, ça a ses limites ergonomiques (ex : threads !)



# Firefox Profiler = Visualisation GUI

- <https://profiler.firefox.com/> = plate-forme permettant la **visualisation locale et le partage cloud** de profils de perfs
  - A la base, pensé pour l'optimisation de Firefox
  - Mais accepte les données perf après digestion :  
« perf script -F +pid > /tmp/test.perf »
- **Intérêts** : Plus ergonomique & puissant que la TUI, et permet le partage facile avec des collègues n'utilisant pas perf.
  - ...et nettement moins buggé que l'alternative hotspot.



# Demo : perf record, FF profiler



# Plan d'attaque

1. Vue d'ensemble du système
2. Métriques CPU fines
3. Repérage du code « chaud »
- 4. Et ensuite ?**

# Que faire du « code chaud » ?

- D'abord, **extraire des fragments** à étudier davantage
  - Microbenchmarks à conserver pour le suivi des perfs !
- Ensuite, essayer de **comprendre ce qui le limite**
  - Examen manuel de l'ASM avec Cutter...
  - Analyse statique avec -fopt-info, MAQA0...
  - Analyse dynamique avec perf annotate, Intel PT...
  - Instrumentation manuelle si besoin...
- Enfin, optimiser en **vérifiant l'impact** sur les perfs

# Conclusion

- **Comprendre ce qui se passe** est le nerf de la guerre
- Commencer grossier, puis raffiner avec l'outil adapté
  - Attention, une erreur de compréhension précoce coûte cher (inutile d'accélérer le CPU 30x si c'est l'I/O qui limite)
- Les outils sont nombreux mais très puissants
  - Apprenez-les progressivement, selon les besoins
  - Utilisez-les souvent pour les maîtriser

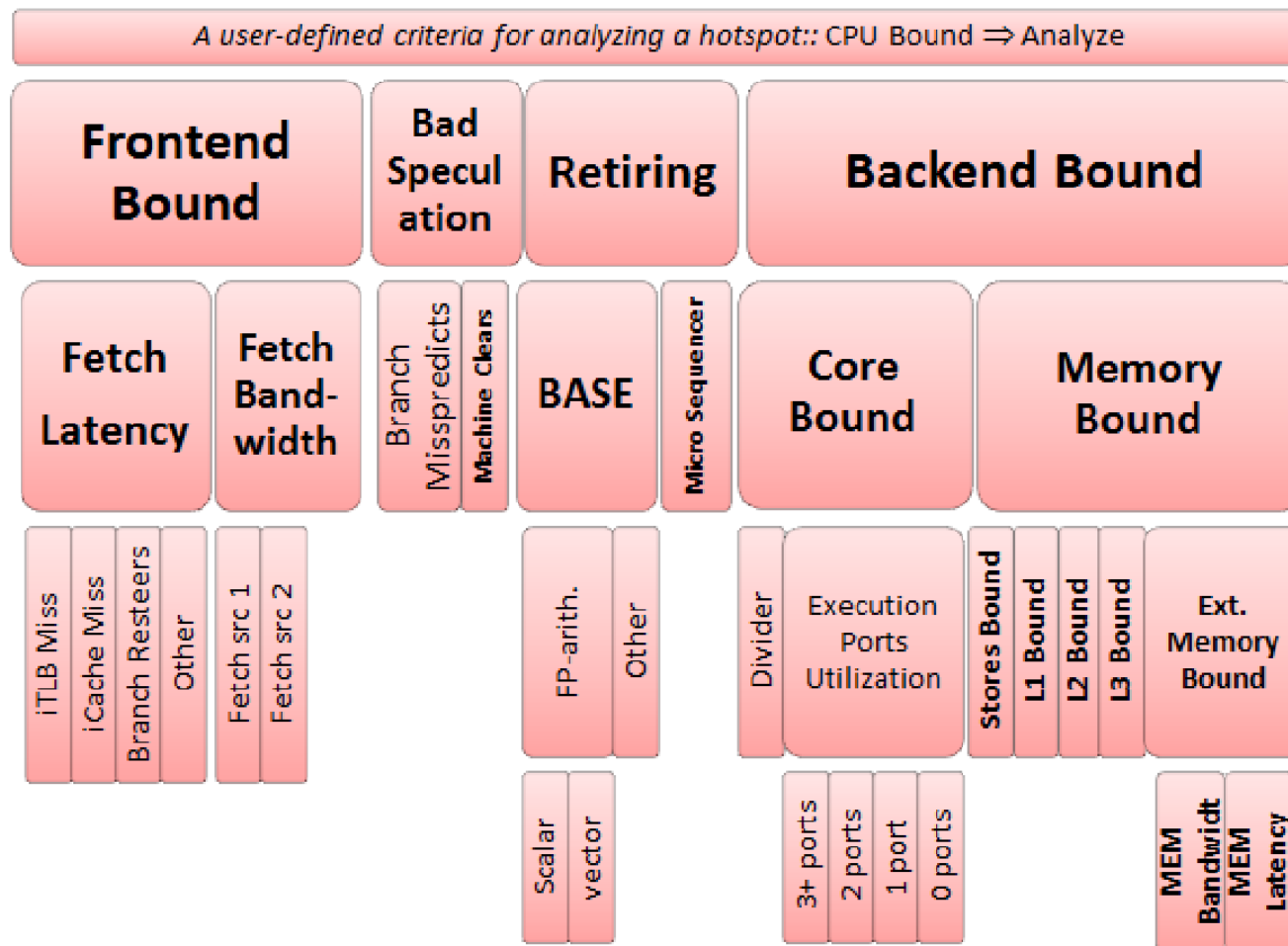
**Merci de votre attention !**

# Pourquoi perf ? (sous Linux\*)

	(i)gprof, gperftools...	callgrind	VTune	perf
Support multi-thread	⚠	×	✓	✓
Activité micro-architecture	×	⚠	✓	✓
Granularité instruction ASM	×	✓	✓	✓
Activité libc (mutex, malloc...)	⚠	⚠	⚠	✓
Instrumentation de code	×	×	×	✓
Analyse activité OS	×	×	⚠	✓
Utilisable via SSH	⚠	⚠	✓	✓
Biais de mesure faible	✓	×	✓	✓
Support matériel AMD, ARM...	✓	✓	⚠	✓

\* Sous Windows et macOS, je conseille respectivement WPA et Xcode Instruments.

# Partie du CPU limitante ? (Intel-only)



- Analyse « **top-down** »
  - Retiring = Exec. normale
  - Bad spec = Le code est imprévisible pour le CPU
  - FE bound = Le code est indigeste pour le CPU
  - BE bound = Limité par core ou mem → ambigu !
- Mesure « perf stat -M TopDownL1 »
  - Puis cf schéma