

Techniques et Outils d'Ingénierie Logicielle

...

John Gliksberg

Temporalité

- Cours 1 :
 - Maîtriser un shell et ses automatisations
 - Avoir un environnement de développement productif
 - Savoir compiler un programme (lib, makefile, gcc, search paths)
 - Comprendre les phases de la compilation
- Cours 2 :
 - Cycle de vie d'un programme
 - Gestion de code source (Versioning) == GIT
 - Principe de forge (Issue, MR, fork, Pull)
 - Notion d'open-source
 - Gestion des conflits de code source
- Cours 3 :
 - Débogage de programmes
 - Principe et structure des débogueurs
 - Utilisation de GDB
 - Structure d'un binaire
 - Structure d'un binaire lors de l'exécution
 - Débogage mémoire
- Cours 4 :
 - **Optimisation de code**
 - **Outils de mesure**
 - **Méthodes de mesure**
 - Optimisation d'algorithmes

Objet du cours

- Profilage de code
 - Principes et méthodes
- Compréhension des rapports
 - Flat profile / bottom-top / top-bottom
 - FlameGraph
- Outils
 - Callgrind + KCacheGrind
 - perf (Linux) + Hotspot
- Limites

Principe du profilage

Objectifs du développement, dans un ordre subjectif :

le programme doit être correct

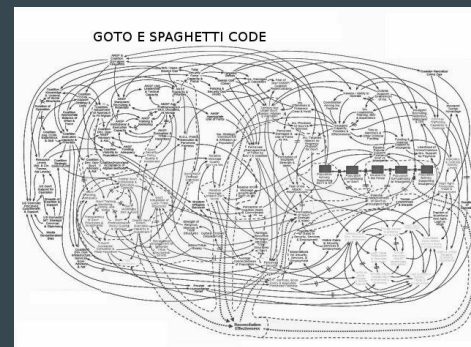
- comment savoir s'il l'est ?
 - intégration continue (CI) & développement piloté par les tests (TDD)
 - tests unitaires, tests fonctionnels, fuzzing (sécurité)
- s'il ne l'est pas, revenir au cours 3 « debug »

le code doit être maintenable (« génie logiciel »)

- cf. cours 2 « git », et peut-être cours 5 « no spoil »

le programme doit se comporter convenablement

- vitesse d'exécution
- consommation mémoire
- &c.



Principe du profilage

Profilage du temps d'exécution

Premier outil : `time(1)` (/ bash builtin)

- real — temps réel « wall »
- user — temps CPU appli (/proc)
- sys — temps CPU noyau (/proc)

Temps “on-CPU” = user + sys

Temps “off-CPU” = temps réel - (user + sys) ÷ nprocs

Instrumentation du code : `clock_gettime(2)`

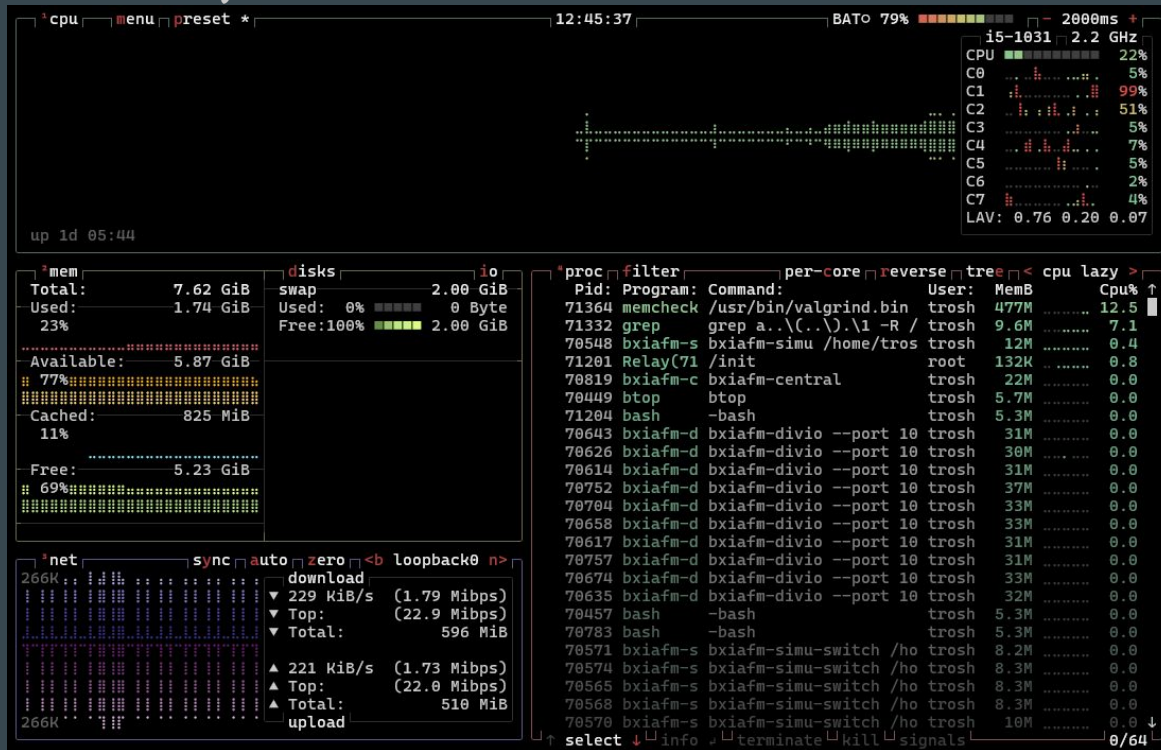
- quelle horloge, quelle résolution ? Voir `time(7)`

Principe du profilage

Profilage de l'utilisation des ressources systèmes

Premier outil : **top**(1)

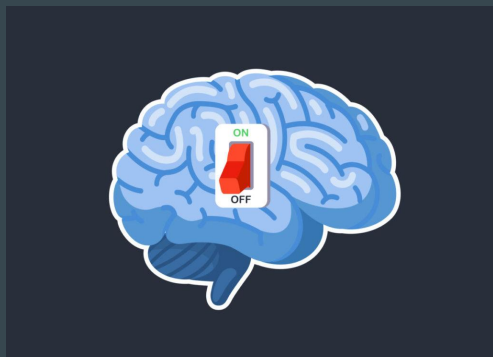
- ou bien htop
- ou bien btop



Principe du profilage

Pourquoi mon code est si lent ?

→ Que fait mon code, au juste ?



Avec cette modification, la fonction sera bien moins coûteuse !

→ Je compare chaque implémentation en conditions de test

“...premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%” — Donald Knuth

Méthodes de profilage

profilage \neq débogage ... mais méthodes similaires

- instrumentation :

- statique
- compilée
- machine virtuelle
- compteurs hardware

} analyse
post-mortem



Compréhension des rapports

Compréhension des rapports — types de profils

Flat profile

Toutes les fonctions avec leur coût propre

$$= t(\text{fonction}) - t(\text{sous-fonctions})$$

Unité temps, cycles, instructions (*Ir*), ...

-	f2	50%
-	f1	35%
-	fx	10%
-	main	5%
-	main	5%
-	f2	50%
-	fx	8%
-	f1	35%
-	fx	2%
-	fx	10%
-	f2	?
-	main	?
-	f1	?
-	main	?

Top-down

Similaire mais comme un call graph (descente dans les *callees*)

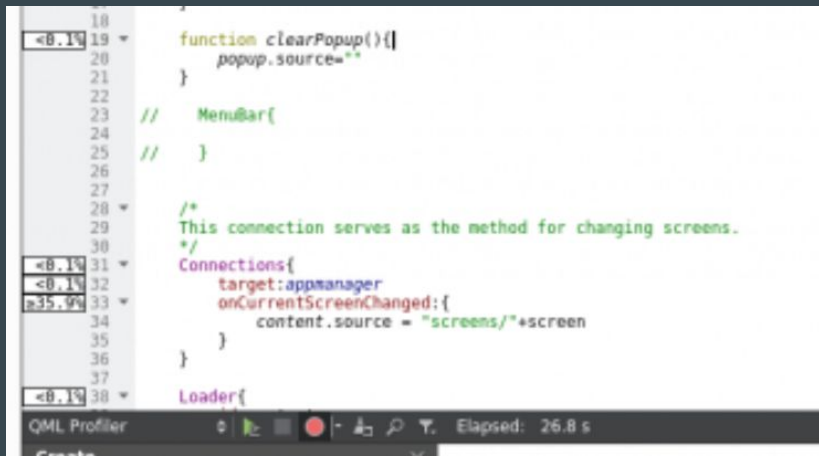
Bottom-up

Similaire, avec du contexte des appelants (*callers*)

Compréhension des rapports — annotation de code

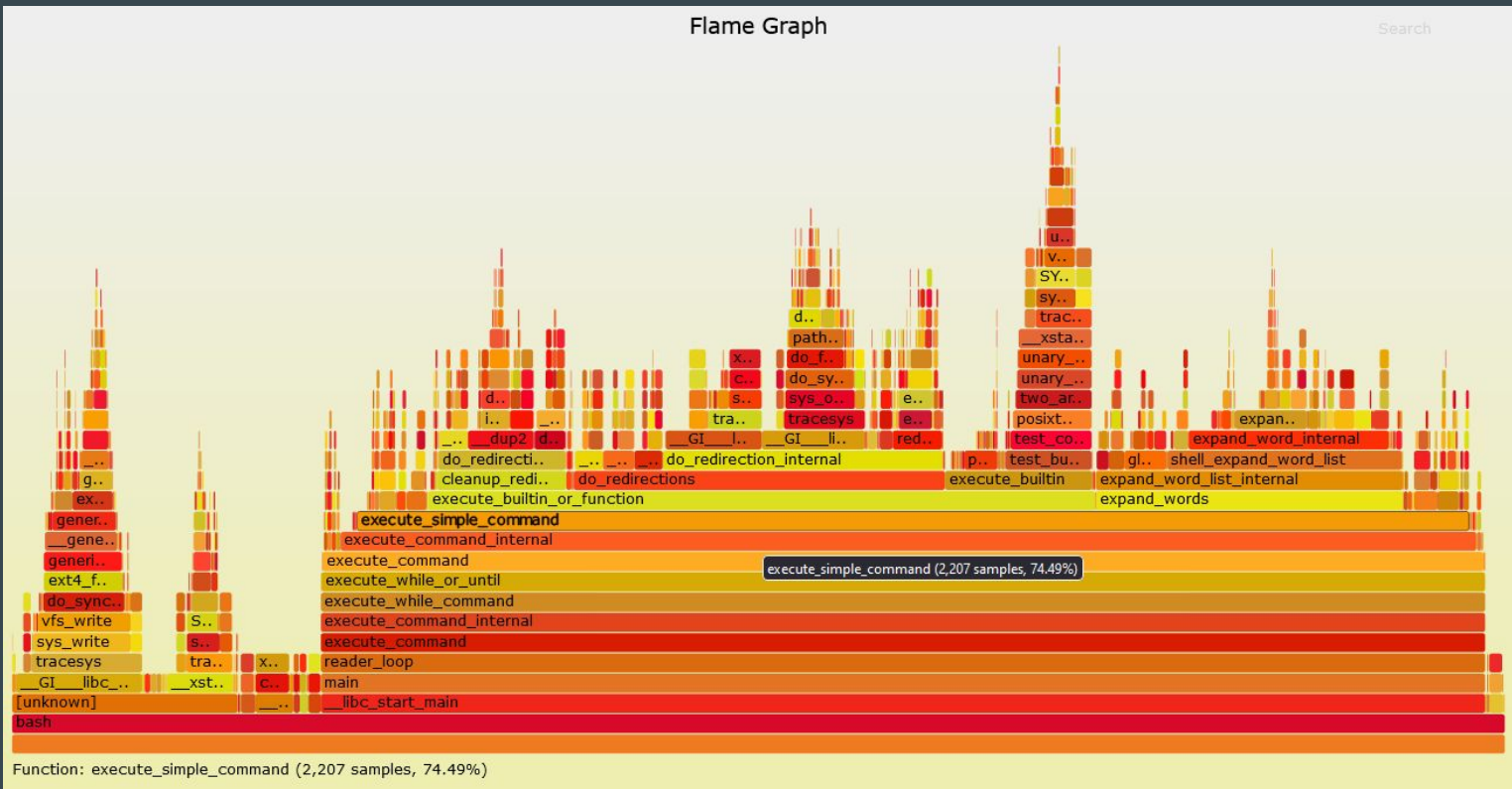
Correspondance entre mesures et code

- Mesures par instruction machine
 - désassemblage
- Souvent approximatives
 - Faire preuve de bon sens



Percent		
0.01	↓ js 71	cvtsi2sd %rax,%xmm0
2.09	↓ jmp 86	
71:	mov %rax,%rdx	
	shr %rdx	
	and \$0x1,%eax	
	or %rax,%rdx	
	cvtsi2sd %rdx,%xmm0	
	addsd %xmm0,%xmm0	
86:	mov -0x28(%rbp),%rax	
0.00	test %rax,%rax	
2.08	js 96	
	cvtsi2sd %rax,%xmm1	
	↓ jmp ab	
96:	mov %rax,%rdx	
	shr %rdx	
	and \$0x1,%eax	
	or %rax,%rdx	
	cvtsi2sd %rdx,%xmm1	
	addsd %xmm1,%xmm1	
0.01	divsd %xmm1,%xmm0	
26.22	movsd %xmm0,-0x8(%rbp)	
	pi += dt / (1.0 + x * x);	
2.11	movsd -0x8(%rbp),%xmm0	
8.52	movapd %xmm0,%xmm1	
0.00	mulsd -0x8(%rbp),%xmm1	
8.76	movsd 0x109(%rip),%xmm0	# 888 <_IO_stdin_used+0x18>
0.00	addsd %xmm0,%xmm1	
6.94	movsd -0x10(%rbp),%xmm0	
	divsd %xmm1,%xmm0	
32.34	movsd -0x20(%rbp),%xmm1	
0.01	addsd %xmm1,%xmm0	
6.55	movsd %xmm0,-0x20(%rbp)	
2.19	for (size_t i = 0; i < N; i++) {	
	addq \$0x1,-0x18(%rbp)	
	↑ jmpq 53	
	return pi * 4.0;	
ef:	movsd -0x20(%rbp),%xmm1	
	0xdf(%rip),%xmm0	# 890 <_IO_stdin_used+0x20>
	mulsd %xmm1,%xmm0	
	pop %rbp	
	← retq	

Compréhension des rapports — flame graphs



Outils de profilage

Outils de profilage — statique

call-graph

Outils de profilage — instrumentation compilée

Gprof

```
$ gcc ... -pg
```

```
...
```

```
$ ./a.out
```

```
...
```

```
$ ls
```

```
main.c  a.out
```

```
$ gprof a.out gmon.out
```



Flat profile:

counts as 0.01 seconds.

	self	total	
seconds	calls	s/call	s/call name
15.52	1	15.52	15.52 func2
15.50	1	15.50	15.50 new_func1
15.26	1	15.26	30.75 func1
0.03			main

(explanation follows)

each sample hit covers 2 byte(s) for 0.02% of
s

self	children	called	name
0.03	46.27		main [1]
15.26	15.50	1/1	func1 [2]
15.52	0.00	1/1	func2 [3]

15.26	15.50	1/1	main [1]
[2]	66.4	15.26 15.50	1 func1 [2]
		15.50 0.00	1/1 new_func1 [4]

Outils

Machine virtuelle

Callgrind + KCacheGrind

Outils — machine virtuelle



This little maneuver is gonna cost us 51 years

```
$ valgrind --tool=callgrind ./a.out
==68835== Callgrind, a call-graph generating cache profiler
...
==68835==
==68835== Events   : Ir
==68835== Collected : 104570619
==68835==
==68835== I   refs:      104,570,619
$ ls
main.c  a.out  callgrind.out.68835
```

Program under test

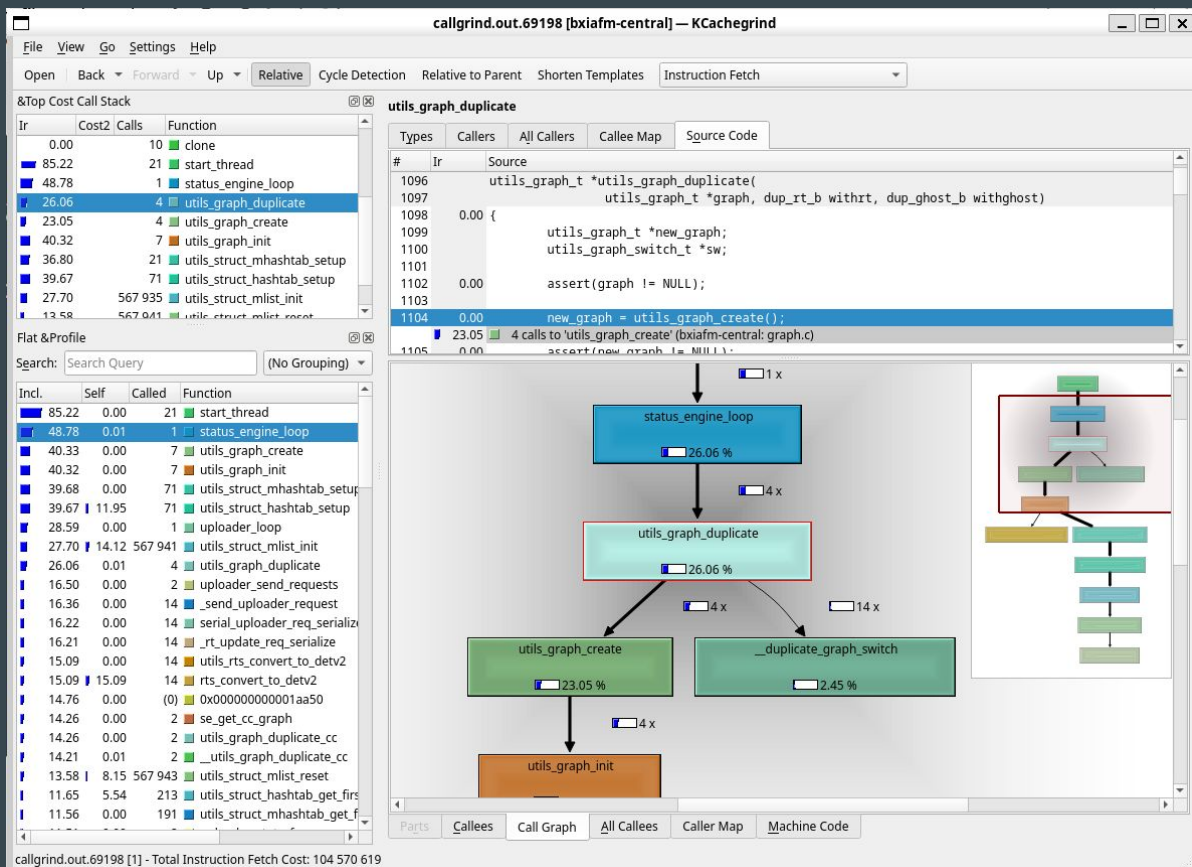
3

CPU

... et ?

Outils — machine virtuelle

\$ **kcachegrind** callgrind.out.68835



Outils — machine virtuelle

Facile à utiliser

Mesure toutes les instructions user

mais


Ne mesure que les instructions user

Pas le temps kernel ni off-CPU

Digression — échantillonnage

Le profileur du pauvre

```
$ gdb ./a.out  
(gdb) r  
...  
^C  
(gdb) bt  
(gdb) c  
...
```



- + Échantillonnage temps réel
 - + (user + kernel) + off-CPU
- Équivalent flat profile seulement
- Uniquement si suffisamment lent
- Très fastidieux

parfait mais inutilisable



Outils

Compteurs hardware

Linux perf / perf_events

Outils — perf

Commande `perf(1)`

- Interface Linux `perf_events`
 - sondes dynamiques noyau
 - compteurs hardware
 - ex : Intel PT
 - échantillonnage hardware
 - ex : AMD IBS, Intel TPEBS

- + Pas de recompilation
- + Très peu d'overhead
- + Mesures précises
- + Outillage complet
 - + Analyse kernel, inter-process
- Un peu dur à utiliser
- Bric-à-brac

```
$ cd /usr/share/man/man1
```

```
$ ls perf*
```

<code>perf.1.gz</code>	<code>perf-config.1.gz</code>	<code>perf-inject.1.gz</code>	<code>perf-lock.1.gz</code>	<code>perf-script-python.1.gz</code>
<code>perf-annotate.1.gz</code>	<code>perf-daemon.1.gz</code>	<code>perf-intel-pt.1.gz</code>	<code>perf-mem.1.gz</code>	<code>perf-stat.1.gz</code>
<code>perf-archive.1.gz</code>	<code>perf-data.1.gz</code>	<code>perf-iostat.1.gz</code>	<code>perf-probe.1.gz</code>	<code>perf-test.1.gz</code>
<code>perf-arm-spe.1.gz</code>	<code>perf-diff.1.gz</code>	<code>perf-kallsyms.1.gz</code>	<code>perf-record.1.gz</code>	<code>perf-timechart.1.gz</code>
<code>perf-bench.1.gz</code>	<code>perf-dlfilter.1.gz</code>	<code>perf-kmem.1.gz</code>	<code>perf-report.1.gz</code>	<code>perf-top.1.gz</code>
<code>perf-buildid-cache.1.gz</code>	<code>perf-evlist.1.gz</code>	<code>perf-kvm.1.gz</code>	<code>perf-sched.1.gz</code>	<code>perf-trace.1.gz</code>
<code>perf-buildid-list.1.gz</code>	<code>perf-ftrace.1.gz</code>	<code>perf-kwork.1.gz</code>	<code>perf-script.1.gz</code>	<code>perf-version.1.gz</code>
<code>perf-c2c.1.gz</code>	<code>perf-help.1.gz</code>	<code>perf-list.1.gz</code>	<code>perf-script-perl.1.gz</code>	

Outils — perf

```
$ perf record -g -- <cmd> [...]
```

```
...
```

```
$ ls
```

```
... perf.data
```

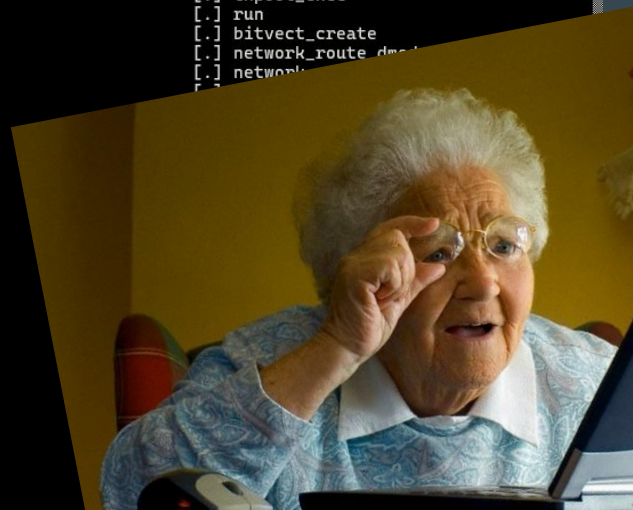
```
$ perf report
```

```
Samples: 273K of event 'cycles:u', 4000 Hz, Event count (approx.): 171952952194  
dmodc_build_available_group /home/trosh/bxiafm/rttoolkit/fmpoc [Percent: local period]
```

```
0.10      movslq %ecx,%rcx  
0.74      mov    %rax,(%r11,%rcx,8)  
0.28      mov    %esi,%r8d  
          for (int i = 0; i < sw->port_group_count; i++) {  
6.34      36:      add    $0x1,%edx  
0.49      39:      cmp    %edx,0x12c(%rdi)  
          ↓ jle    79  
          struct sw_port_group *port_group = &sw->port_group[i];  
2.35      movslq %edx,%rax  
0.30      shl    $0x5,%rax  
5.72      add    0xc0(%rdi),%rax  
          const struct sw *remote = port_group->remote;  
0.59      mov    (%rax),%rcx  
          if (target_cost != remote->cost[leaf_id])  
4.76      mov    0xb0(%rcx),%rsi  
12.93     movslq %r10d,%rcx  
4.85      movzbl (%rsi,%rcx,1),%ecx  
52.77     cmp    %r9d,%ecx  
          ↑ jne    36  
          if (port_group->direction != current_direction) {  
2.11      mov    0x1c(%rax),%esi  
0.31      cmp    %r8d,%esi  
0.01      ↑ je     24  
          if (port_group->direction < current_direction) {  
0.06      cmp    %r8d,%esi  
          ↑ jl     36  
          size = 0;  
0.06      mov    $0x0,%ecx
```

```
Samples: 273K of event 'cycles:u', Event count (approx.): 171952952194  
Children      Self      Command      Shared Object      Symbol  
+ 78.82%      0.00%      fmpoc        libc.so.6          [.] start_thread  
+ 78.82%      0.00%      fmpoc        fmpoc              [.] th_run  
+ 55.48%      0.00%      fmpoc        fmpoc              [.] route_dmodc_thread  
+ 55.48%      0.59%      fmpoc        fmpoc              [.] dmodc_compute_port_rt  
- 46.51%      20.67%      fmpoc        fmpoc              [.] dmodc_compute_route  
- 25.84%      dmodc_compute_route  
+ 9.90%      bitvect_create  
5.78%      dmodc_build_available_group  
4.35%      bitvect_set  
+ 2.38%      sw_alt_v3_add  
2.17%      bitvect_unpack  
+ 0.72%      sw_delete_alt_tuple  
+ 18.15%      start_thread  
+ 2.52%      __libc_start_call_main
```

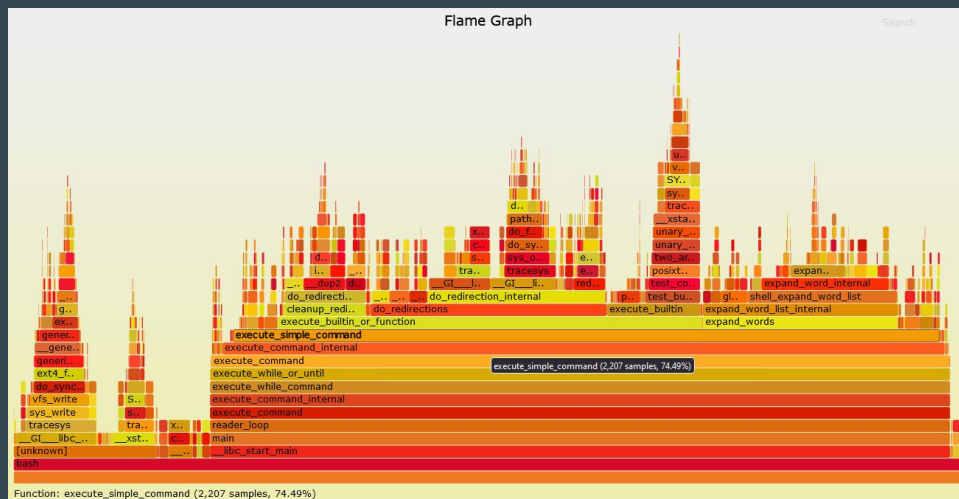
```
+ 20.35%      10.37%      fmpoc        fmpoc              [.] reset_thread  
+ 19.74%      0.17%      fmpoc        fmpoc              [.] xmalloc  
+ 19.57%      0.27%      fmpoc        fmpoc              [.] memrnd  
+ 19.49%      0.00%      fmpoc        libc.so.6          [.] __libc_start_call_main  
+ 19.49%      0.00%      fmpoc        fmpoc              [.] main  
+ 19.49%      0.00%      fmpoc        fmpoc              [.] context_run  
[.] memrnd_randbuf  
[.] memrnd_rand32  
[.] rank_thread  
[.] update_cost  
[.] thpool_exec  
[.] run  
[.] bitvect_create  
[.] network_route_dmodc  
[.] network
```



Outils — perf + Flame Graph

<https://brendangregg.com/flamegraphs.html> (2013)

1. Capturer les traces avec `perf record ... -g -- <cmd> ...`
2. Replier les traces avec un script Perl
3. Générer un SVG interactif avec un autre script Perl
4. Afficher le SVG dans un navigateur



Outils — flamegraph-rs

<https://github.com/flamegraph-rs/flamegraph>

Dépend de linux-perf, mais plus pratique si on ne veut que le flamegraph

```
$ flamegraph ... -- <cmd> ...
```

```
...
```

```
[ perf record: Woken up x times to write data ]
```

```
[ perf record: Captured and wrote xMB perf.data (x samples) ]
```

```
$ ls
```

```
a.out  perf.data  perf.svg
```

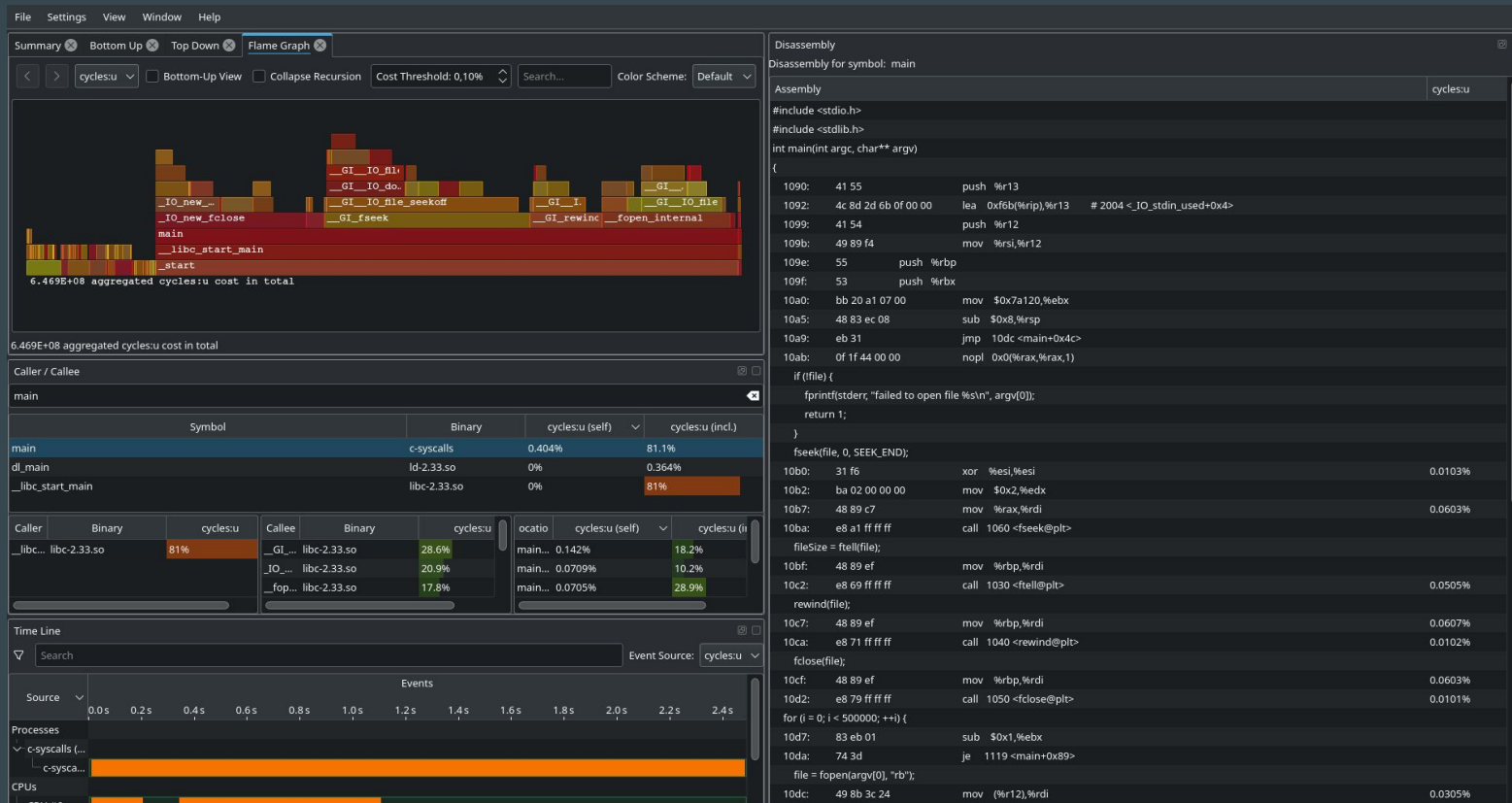
Outils — Hotspot

<https://github.com/KDAB/hotspot>

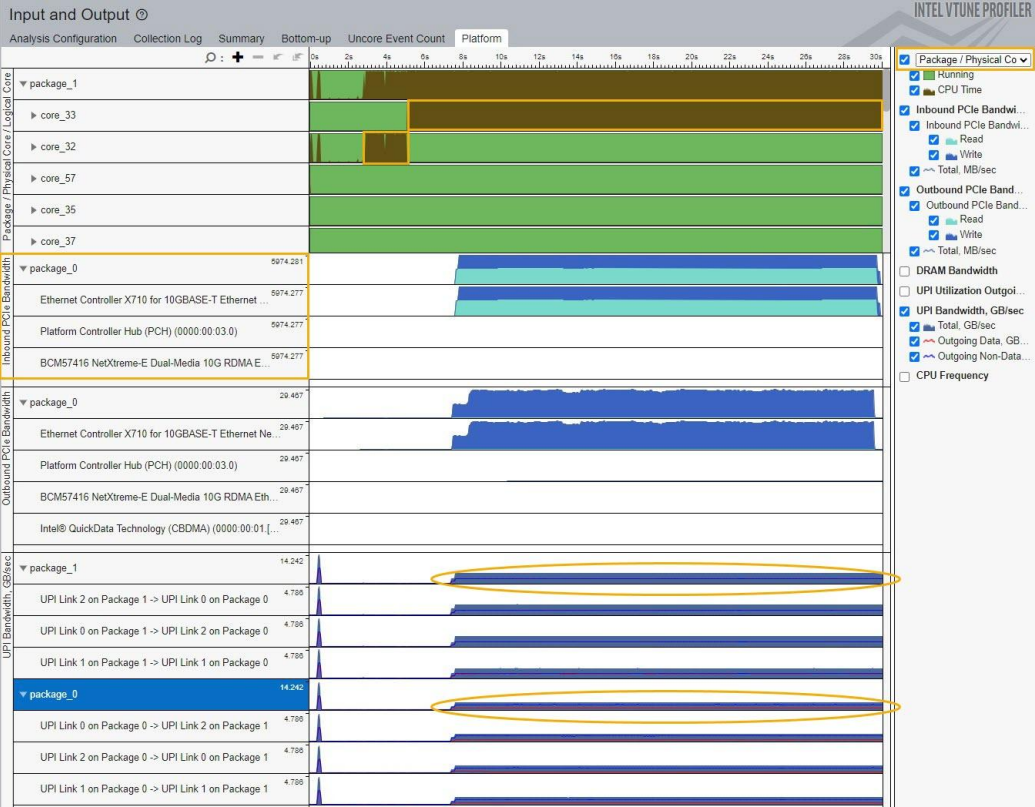
Interface intégrée pour linux-perf

- Enregistrement possible depuis Hotspot
- Visualisation temporelle par processus+thread
- Flamegraph intégré
- Désassembleur
- Liste de lignes de code par coût

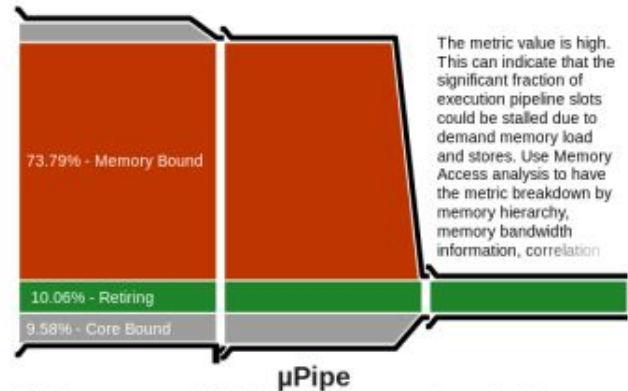
Outils — Hotspot



Outils propriétaires



Clocks:	349,198,500,000
Instructions Retired:	69,671,000,000
CPI Rate:	5.012
Retiring:	10.1% of Pipeline Slots
Front-End Bound:	6.0% of Pipeline Slots
Bad Speculation:	0.5% of Pipeline Slots
Back-End Bound:	83.4% of Pipeline Slots
Memory Bound:	73.8% of Pipeline Slots
	1.3% of Clocks
	0.0% of Clocks
	9.2% of Clocks
	71.3% of Clocks
	74.6% of Clocks
	22.9% of Clocks
	100.0% of Clocks
	0.0% of Clocks
	9.6% of Pipeline Slots
	10
	0s



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

There is no data to calculate the metric.

Intel® VTune™ Profiler
Visual Studio
&c.

Autres types de profilage

Utilisation de la mémoire

- `valgrind --tool=massif`

Prédiction de branchement

- `valgrind --tool=callgrind --branch-sim=yes`

Défauts de cache, de pages, &c.

- `valgrind --tool=callgrind --cache-sim=yes`
- `Linux-perf`
- `Intel V-Tune`

&c.

Limitations des profileurs

Limitations des profileurs

Limitations techniques

- Quasiment impossible de mesurer précisément sans impacter le comportement
- Chaque enregistrement n'est valable que :
 - pour une machine,
 - pour un cas d'usage,
 - pour un environnement spécifique

Limitations conceptuelles

- Beaucoup d'infos, pourquoi ?
- Attention aux œillères
 - Pas de profileur d'algorithme / de structures de données
- Attention aux optimisations superficielles

Questions ?