Outils d'analyse de systèmes parallèles

Module 9

INF8601 Systèmes informatiques parallèles

Michel Dagenais

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

Sommaire

Outils d'analyse de systèmes parallèles

Exemple: liste des outils du LLNL

- Débogage et vérification: DDT, FLiT, FPChecker, Intel Inspector, Valgrind, TotalView...
- Utilisation de la mémoire: Intel Inspector, Valgrind, memP, TotalView.
- Profilage: gprof, HPCToolkit, Intel Advisor, Intel VTune Amplifier, memP, mpiP, TAU...
- Traçage: HPCToolkit, Intel Advisor, Intel VTune Amplifier, TAU, Vampir, Intel Trace Analyzer and Collector...
- Analyse de performance: HPCToolkit, Intel VTune Amplifier, TAU, Vampir/VampireServer, Intel Trace Analyzer and Collector...
- L'environnement au laboratoire Lawrence Livermore, https://hpc.llnl.gov/software/development-environment-software

Outils d'analyse de systèmes parallèles

Insertion des capteurs de données

- Enoncés insérés par le programmeur, e.g. TRACE_EVENT de LTTng, et sujet à compilation conditionnelle. Activation dynamique du capteur.
- Capteurs insérés automatiquement par le compilateur ou un outil source à source, e.g. options `-finstrument-functions`, `-pg`, `-fprofile-arcs`.
- Capteurs insérés dans l'exécutable par décompilation, insertion, recompilation, e.g. DynInst.
- Capteurs insérés dans un programme en exécution par insertion de point d'arrêt (e.g. GDB, SystemTap), remplacement d'instruction (e.g. GDB fast tracepoint) ou décompilation et recompilation dynamique (e.g. DynInst).
- Echantillonnage sur interruption basée sur le temps ou les compteurs de performance (e.g. Oprofile, perf).
- Support matériel (e.g. ARM CoreSight, Intel PT).

Prise de trace

- Ecriture texte ou binaire dans un tampon.
- Tampon global à l'application ou au noyau, ou tampon par CPU ou par thread.
- Verrou, opérations atomiques globales ou locales à un CPU.
- Tampon circulaire, double ou multiples tampons.
- Ecriture sur disque en continu ou sur commande explicite (cliché lorsqu'une condition est détectée).

Outils d'analyse de systèmes parallèles

GCOV

- Instrumentation par gcc pour compter chaque entrée dans un bloc de base (séquence d'instruction sans saut à l'intérieur).
- A la fin de l'exécution du programme, un fichier est créé contenant le décompte pour chaque bloc.
- L'outil gcov permet de montrer pour chaque ligne de code source le nombre de fois qu'elle a été exécutée (importance, couverture de test).
- Surcoût d'environ 10 à 15%

GCOV

```
# Run the program compiled with options -ftest-coverage and -fprofile-arcs
[gzip-1.2.4a]$ gzip </tmp/evlogout >/dev/null
# Files .bb and .bbg are produced at compilation time, files .da at program exi
# Gcov reads the .bb .bbg .da and .c files and produces .c.gcov
[gzip-1.2.4a]$ gcov -b -f deflate.c
[gzip-1.2.4a]$ less deflate.c.gcov
     5  while (lookahead != 0) {
...branch 0 taken = 0%
  6933680 INSERT_STRING(strstart, hash_head);
  6933680 prev_length = match_length, prev_match = match_start;
  6933680 match_length = MIN_MATCH-1;
  6933680 if (hash_head != NIL && prev_length < max_lazy_match &&
branch 0 taken = 8%
branch 1 taken = 47%
branch 2 taken = 1%
              strstart - hash_head <= MAX_DIST) {
  3367555   match_length = longest_match (hash_head);
  3367555   if (match_length > lookahead) match_length = lookahead;
branch 0 taken = 0%
branch 1 taken = 100%...
```

GProf

- Instrumentation par gcc de chaque entrée dans une fonction.
- A l'exécution, pour chaque fonction, une table de hachage des fonctions appelantes avec le nombre d'appels est calculée. Surcoût d'environ 5%.
- A chaque 1ms de temps CPU, une interruption est levée et l'adresse de l'instruction courante est échantillonnée et la case correspondate d'un histogramme est incrémentée. Surcoût d'environ 0.5%.
- A la fin de l'exécution, les tables du nombre d'appel et l'histogramme sont écrits dans un fichier.
- L'outil Gprof fournit un profil de l'exécution temps self + childs par fonction. Excellent pour caractériser le temps CPU pris par chaque fonction ou ligne de code.
- Fait l'hypothèse que tous les appels prennent le même temps pour estimer les valeurs de self + childs.

GProf

```
# Run the program compiled and linked with option -pg
[gzip-1.2.4a]$ gzip </tmp/evlogout >/dev/null
# File gmon.out is produced when the program exits, used by gprof
[gzip-1.2.4a]$ gprof gzip >gprof.out
[gzip-1.2.4a]$ less gprof.out
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
27.13      7.53     7.53     1957     3.85     5.95  fill_window
23.20     13.97     6.44        1  6440.00 19205.66  deflate
14.84     18.09     4.12     1958     2.10     2.10  updcrc
12.16     21.46     3.38                             do_scan
 8.38     23.79     2.33                             short_loop
 3.78     24.84     1.05                             __mcount_internal
...
------------------------------------------------
                0.00    0.00       1/1958        zip [3]
                4.12    0.00    1957/1958        file_read [8]
[7]     15.6    4.12    0.00    1958         updcrc [7]
------------------------------------------------
                0.00    0.00       1/1958        lm_init [27]
                0.00    4.12    1957/1958        fill_window [6]
```

Oprofile

- La plupart des processeurs offrent des compteurs de performance pouvant compter les cycles d'exécution, les fautes de cache, les sauts pris / non pris, les rangements et chargements... et peuvent générer une interruption lorsque la valeur atteint un seuil (e.g. 100K ou 1M).
- Lors de l'interruption, l'adresse de l'instruction ou de la donnée courante est échantillonnée et la case correspondante d'un histogramme est incrémentée.
- A la fin de l'exécution, l'histogramme est écrit dans un fichier.
- L'outil Oprofile fournit un profil de l'exécution (temps, rangement, chargement, faute de cache...) par fonction ou par ligne de code. Excellent pour voir les problèmes subtils d'utilisation de cache ou blocages divers. De plus en plus remplacé par perf sur Linux.
- Surcoût inférieur à 0.5%.

Problèmes de cache avec Oprofile

```
Counted CPU_CLK_UNHALTED events, count 50000
vma      samples  %           image name
00000000 450385   48.1828     cc1plus
00000000 11617     1.2428     as
4207d260 7206      0.7709     libc-2.3.2.so (cc1plus)
00000000 7046      0.7538     bash
080c0f70 6966      0.7452     XFree86
00000000 6397      0.6844     vim
00032150 5843      0.6251     libkonsolepart.so (kdeinit)
081ec974 5016      0.5366     lyx
420745e0 4928      0.5272     libc-2.3.2.so (cc1plus)
0804c5a0 4462      0.4774     oprofiled
00009f30 4154      0.4444     libpthread-0.10.so (lyx)
0003b990 4143      0.4432     libkonsolepart.so (kdeinit)
c01163d0 3865      0.4135     vmlinux (cc1plus)
c0155ef0 3800      0.4065 vmlinux (bash)
```

Problèmes de cache avec Oprofile

```
$ opannotate --source --assembly `which oprofiled`
              :   index = hash->hash_base[odb_do_hash(hash,
              :   while (index) {
 1455 14.2689:     test %eax,%eax
   13  0.1275:     je 804c5ef
              :   if (index <= 0 || index >= hash->descr...) {
   12  0.1177:     mov 0x8(%esi),%ecx
    2  0.0196:     lea 0x0(%esi,1),%esi
   19  0.1863:     cmp 0x4(%ecx),%eax
   78  0.7649:     jae 804c638 <odb_insert+0x98>
              :   char * err_msg;
              :   asprintf(&err_msg, "invalid %u\n",index);
              :   odb_set_error(hash, err_msg);
              :   return EXIT_FAILURE;
              : }
              : node = &hash->node_base[index];
   12  0.1177:   lea (%eax,%eax,2),%edx
   41  0.4021: mov (%esi),%eax
```

Outils d'analyse de systèmes parallèles

L'environnement d'analyse Valgrind

- Valgrind permet de décompiler chaque section de code avant qu'elle ne soit exécutée afin d'insérer de l'instrumentation, la recompiler et l'exécuter.
- Une cache permet de prendre la version recompilée les fois suivantes.
- Le travail d'instrumentation est flexible et entièrement programmable (tracer les accès mémoire, les sauts pris / non pris...).
- Plusieurs outils sont construits par-dessus cet environnement: Memcheck, Cachegrind, Callgrind, Massif, Helgrind
- Avec un certain effort, l'usager peut programmer ses propres analyses.

Valgrind: Memcheck

- Chaque lecture et écriture de donnée en mémoire, de même que malloc/free et new/delete sont instrumentés.
- Un espace pour noter son état est alloué pour chaque bit de donnée en mémoire (shadow memory).
- Malloc et New: la mémoire devient disponible, Free et Delete, elle cesse de l'être.
- Ecriture: vérifier si la mémoire est disponible, la mémoire devient initialisée.
- Lecture, vérifier que la mémoire est disponible et initialisée.

Valgrind: Memcheck

- Détecte:
  - les accès invalides (i.e. autre que la pile, les variables globales et les régions allouées dynamiquement);
  - l'utilisation de bits ou octets non initialisés;
  - les fuites de mémoire;
  - les free redondants ou incorrects;
  - les recoupements d'adresse source et destination pour les fonctions de la famille de memcpy.
- Le programme roule entre 10 et 30 fois plus lentement.

Address Sanitizer

- Outil développé surtout par Google.
- Instrumente à la compilation toutes les écritures de données en mémoire, ainsi que les malloc / free.
- Pour chaque bloc de 8 octets, on note si la mémoire est allouée ou non.
- A chaque écriture, on vérifie si la mémoire est allouée.
- Un espace est ajouté entre les allocations pour détecter les dépassements de tampons.
- Un espace libéré est conservé en quarantaine un certain temps avant d'être réalloué.
- Le programme roule moins de 2 fois plus lentement.

Valgrind: Helgrind

- Toutes les lectures et écritures de données en mémoire partagée sont instrumentés, de même que les fonctions de synchronisation. Ralentissement par facteur d'environ 100.
- Chaque variable (case mémoire) à sa première écriture est dans l'état exclusif, possédée par le segment de fil qui l'a accédée.
- Si un autre segment de fil l'accède (en mode exclusif) et que ce segment est disjoint (synchronisation entre les deux qui les sépare), ce nouveau segment devient le propriétaire.
- Si un autre segment non disjoint l'accède en lecture, l'état (en mode exclusif ou partagé lecture) devient partagé lecture.
- Si un autre segment l'accède en écriture, les verrous associés à cette variable sont calculés comme $V = V \cap v(t)$, l'intersection des verrous pris les dernière fois et ceux détenus par le fil courant. Si V devient vide, un verrou a été oublié et c'est une erreur.

Helgrind: Détection des courses

```
#include <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
  var++; /*Unprotected vs parent*/
  return NULL;
}

int main ( void ) {
  pthread_t child;
  pthread_create(&child, NULL,
      child_fn, NULL);
  var++; /*Unprotected vs child*/
  pthread_join(child, NULL);
  return 0;
}
```

```
Thread #1 is the program's root thread
Thread #2 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: do_clone (in libpthread-2.8.so)
   by 0x4E33A30: pthread_create (libpthread-2.8.s
   by 0x4C299D4: pthread_create@* (hg_intercepts.
   by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at
0x601038 by thread #1
   at 0x400606: main (simple_race.c:13)
This conflicts with a previous write of size 4 b
   at 0x4005DC: child_fn (simple_race.c:6)
   by 0x4C29AFF: mythread_wrapper (hg_intercepts.
   by 0x4E3403F: start_thread (in libpthread-2.8.
   by 0x511C0CC: clone (in libc-2.8.so)
Location 0x601038 is 0 bytes inside global var "
declared at simple_race.c:3
```

Helgrind: ordre des verrous

- A chaque acquisition de verrou, ajouter s'il n'est pas déjà existant un lien dirigé de ce verrou vers le précédent acquis par le même fil d'exécution.
- A la fin de l'exécution, vérifier si des cycles existent dans le graphe formé par les liens entre les verrous.

```
Thread #1: lock order "0x7FEFFFAB0 before 0x7FEFFFA80" violated
  at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
  by 0x40081F: main (tc13_laog1.c:24)
Required order was established by acquisition of lock at 0x7FEFFFAB0
  at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
  by 0x400748: main (tc13_laog1.c:17)
followed by a later acquisition of lock at 0x7FEFFFA80
  at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
  by 0x400773: main (tc13_laog1.c:18)
```

Vérification des verrous: Lockdep

- En Linux avec LockDep toutes les acquisitions de verrous dans le noyau sont instrumentées.
- Graphe d'ordre de prise des classes (e.g. inode, page...) de verrous et des profondeurs (parent, enfant dans une arborescence) par un même fil d'exécution, absence de cycle (fermeture transitive).
- Vérification du niveau/contexte dans lequel est pris une classe de verrous: normal, irq, irq actifs ou non. Un verrou pris en mode normal avec irq actifs et aussi en mode irq peut mener à un bloquage.

Thread Sanitizer

- Outil développé surtout par Google.
- L'application prend 5 à 10 fois plus de mémoire et 2 à 20 fois plus de temps.
- Instrumente à la compilation toutes les lectures et écritures de données en mémoire partagée. Intercepte les fonctions qui synchronisent les fils, prennent des verrous ou accèdent la mémoire.
- A chaque accès, si par plus d'un fil, vérifier si la relation *strictement antérieur* est vérifiée.
- Quelques cases (2, 4, 8) par mot de 8 octets pour stocker l'information (thread, clock, r/w...) sur les accès précédents (peut se faire en parallèle avec accès atomiques).
- Conserver un tampon des derniers appels et retours pour chaque fil de manière à retrouver le chemin d'appel d'un accès antérieur en cas de conflit détecté avec l'accès courant.

Valgrind: Cachegrind

- Instrumente tous les accès en mémoire, instructions et données.
- L'application prend environ 50 fois plus de temps.
- Les caractéristiques de l'ordinateur utilisé peuvent être estimées.
- Le comportement de la mémoire cache est simulé de manière à calculer le nombre d'accès et de fautes en cache L1 ou LL.
- Il peut aussi compter le nombre de sauts pris et non pris.
- Le programme peut être annoté avec ces informations par fonction, par ligne ou par instruction.

Valgrind: Callgrind

- Extension à Cachegrind.
- Instrumente les entrées et sorties de fonction, échantillonne le compteur de programme (et 1 à n adresses d'appelants), et optionnellement les sauts conditionnels.
- Calcule le nombre d'appel pour chaque fonction et le temps d'exécution, par appelant.
- Pour chaque fonction donne le temps self et self + childs.
- Peut montrer le graphe d'appel et le temps passé dans chaque noeud, avec plus ou moins de précision selon le nombre d'appelants pris avec chaque échantillon.
- Peut calculer pour chaque saut conditionnel les statistiques de saut pris / non pris.

Google Perftools: CPU profile

- La librairie libprofiler.so est liée à l'application ou chargée avec LD_PRELOAD, et les informations sur la position des symboles est disponible.
- A chaque période (durée configurable, souvent 1 - 10ms) basée sur ITIMER_PROF (en option ITIMER_REAL), le contenu de la pile (chemin d'appel) est échantillonné.
- Les informations (chemin d'appel : nombre d'échantillons) sont écrites de temps en temps pendant l'exécution ou à la fin.
- Les outils d'analyse permettent de voir en post traitement le graphe d'appel avec le temps passé dans chaque fonction.
- Il n'y a pas un décompte exact du nombre d'appels comme avec gprof.
- Le surcoût est inférieur à celui de gprof qui utilise mcount mais supérieur à celui de gprof qui ne prend que l'échantillon du compteur de programme courant, sans celui des appelants.

Google Perftools: Heap profile

- L'application est liée avec libtcmalloc.so, ou cette librairie est chargée avec LD_PRELOAD, et les informations sur la position des symboles est disponible.
- Chaque allocation ou libération est prise en compte par la librairie et le contenu de la pile (chemin d'appel) est échantillonné.
- L'analyse permet de montrer un arbre d'appel avec la mémoire allouée ou utilisée (allocation - libérations) en octets ou en nombre d'objets pour chaque noeud, self et self + childs.

Valgrind: Massif

- L'utilisation de malloc / free et new / delete est instrumentée.
- Un décompte de la mémoire (utile et surcoût) utilisée est préservé à intervalle régulier.
- De temps en temps, pour un intervalle, l'information du chemin d'appel pour chaque allocation est préservée.
- Un arbre d'appel avec les allocations pour chaque noeud (self et self + childs) est présenté pour ces intervalles détaillés.
- Il est possible d'interagir avec Valgrind / Massif pendant l'exécution via le protocole de GDBserver, par exemple pour demander à un instant donné de prendre un décompte détaillé ou non.

Outils d'analyse de systèmes parallèles

Traces d'exécution sur multi-coeurs

- Outils spécialisés spécifiques aux fabricants sur AIX, IRIX...
- Linux Trace Toolkit (LTTng et UST), infrastructure de prise de trace à faible coût pour Linux, développé à Polytechnique avec plusieurs entreprises.
- Sun (Oracle) DTrace, outil basé sur les trappes pour associer des fonctions de rappel à des points de trace sur Solaris.
- Red Hat SystemTap, similaire à Dtrace mais pour Linux.
- Linux Ftrace et Perf, outils de bas niveau pour le traçage et le profilage qui reprennent et partagent certains éléments (points de trace) contribués par LTTng au noyau de Linux.
- GDB Tracepoints, permettant de placer efficacement des points de trace dans les applications à partir du débogueur.
- DynInst, outil pour l'insertion dynamique de points de trace dans des exécutables avant ou pendant leur exécution.

La chaîne complète

- Les développeurs insèrent (dans le code du noyau du système d'exploitation, dans les librairies et dans les applications) des points de trace statiques pendant le développement.
- Les développeurs, administrateurs système ou les utilisateurs peuvent activer des points de trace statiques et peuvent même ajouter des points de trace dynamiques.
- Les fonctions de rappel associées aux points de trace écrivent un événement dans un tampon en mémoire.
- Le contenu des tampons en mémoire est consommé sur place (par mémoire partagée), copié sur disque, envoyé sur le réseau ou accumulé dans un tampon circulaire pour être copié au besoin si un problème est détecté (flight-recorder mode).
- Le contenu des traces peut être examiné en ligne ou a posteriori.

L'écriture des événements

- Une condition peut être associée à un point de trace (e.g. lors d'une écriture dans un fichier de configuration).
- La fonction de rappel peut recevoir quelques arguments (e.g. nom du fichier ouvert, numéro de descripteur) et extraire certaines informations additionnelles (e.g. TID du fil d'exécution courant, chemin d'appel).
- L'événement est écrit dans le tampon: type de l'événement, estampille de temps, arguments de la fonction de rappel, autres informations extraites...
- Il y a un compromis entre la taille des événements et la facilité d'analyse de la trace (ajouter le PID/TID de la tâche courante dans chaque événement versus déduire cette information à partir des événements d'ordonnancement du système d'exploitation).

Exemple de définition de point de trace

```
TRACEPOINT_EVENT(
  sample_tracepoint,
  message,
  TP_ARGS(char *, text),
  TP_FIELDS(
    ctf_string(message, text)
  )
)
```

Le fichier .h est généré par lttng-gen-tp

C:

```
#include "sample_tracepoint.h"
tracepoint(sample_tracepoint, message, "Hello World\n");
```

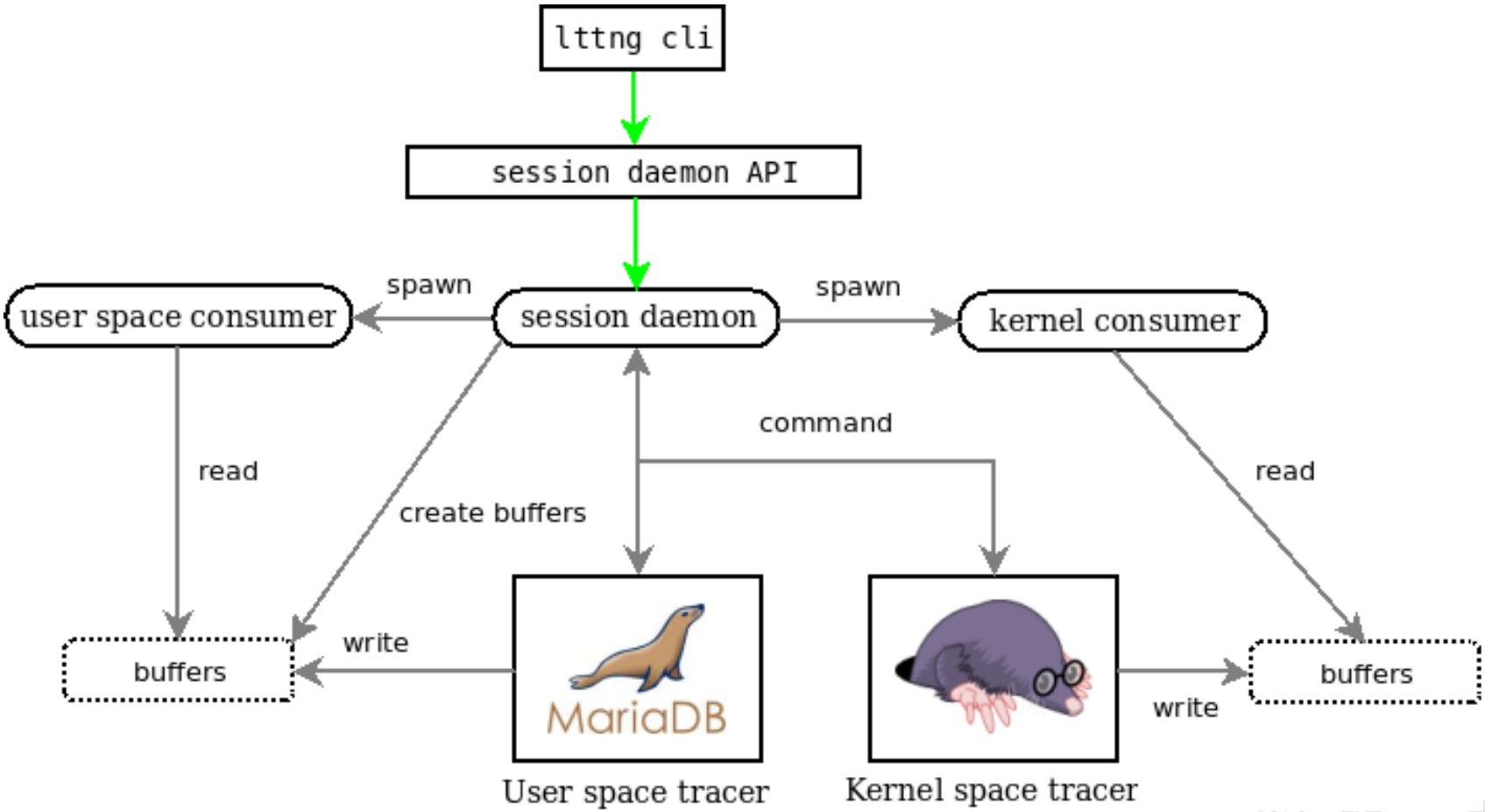
Python:

```
import sample_tracepoint
sample_tracepoint.message("Hello World")
```

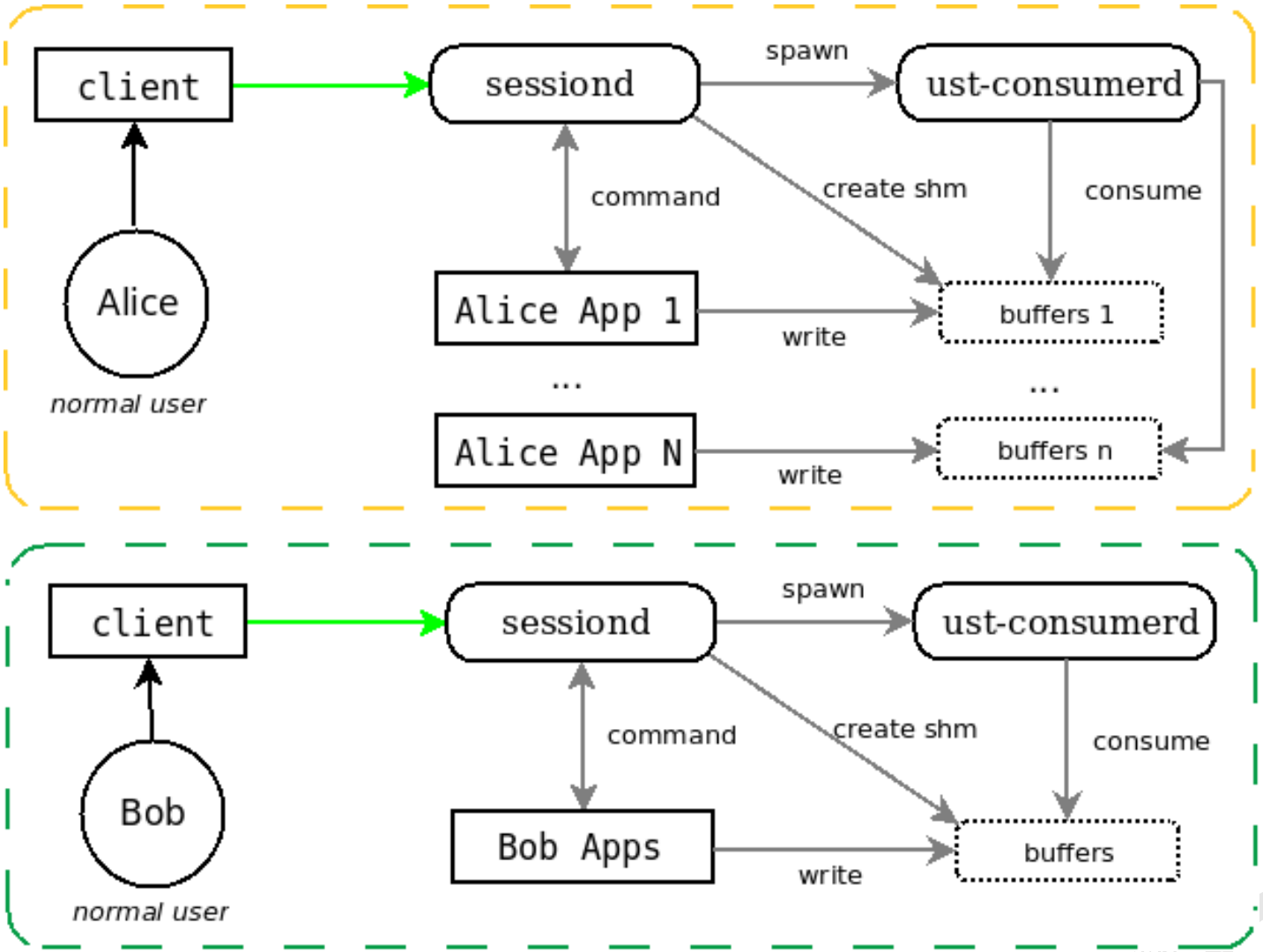
Tracer un système avec LTTng

- Le noyau Linux et chaque application qui utilise UST est une source de trace, qui peut lister les points de trace statiques qu'elle offre et en insérer dynamiquement.
- L'utilisateur root ou les membres du groupe tracing ont le privilège de tracer le noyau, les autres peuvent seulement tracer leurs propres applications.
- Créer une session, définir des canaux, activer des points de trace dans chaque canal, démarrer la trace, laisser l'exécution à tracer avoir lieu, arrêter la trace, puis l'analyser.
- Un démon de session est démarré, au besoin, par usager ou pour le système complet (root ou tracing group).
- Le démon de session relaie les commandes (activer un point de trace, démarrer, arrêter...) entre l'utilitaire de ligne de commande lttng et les applications tracées, et il démarre le démon consommateur qui écrit les traces sur disque.

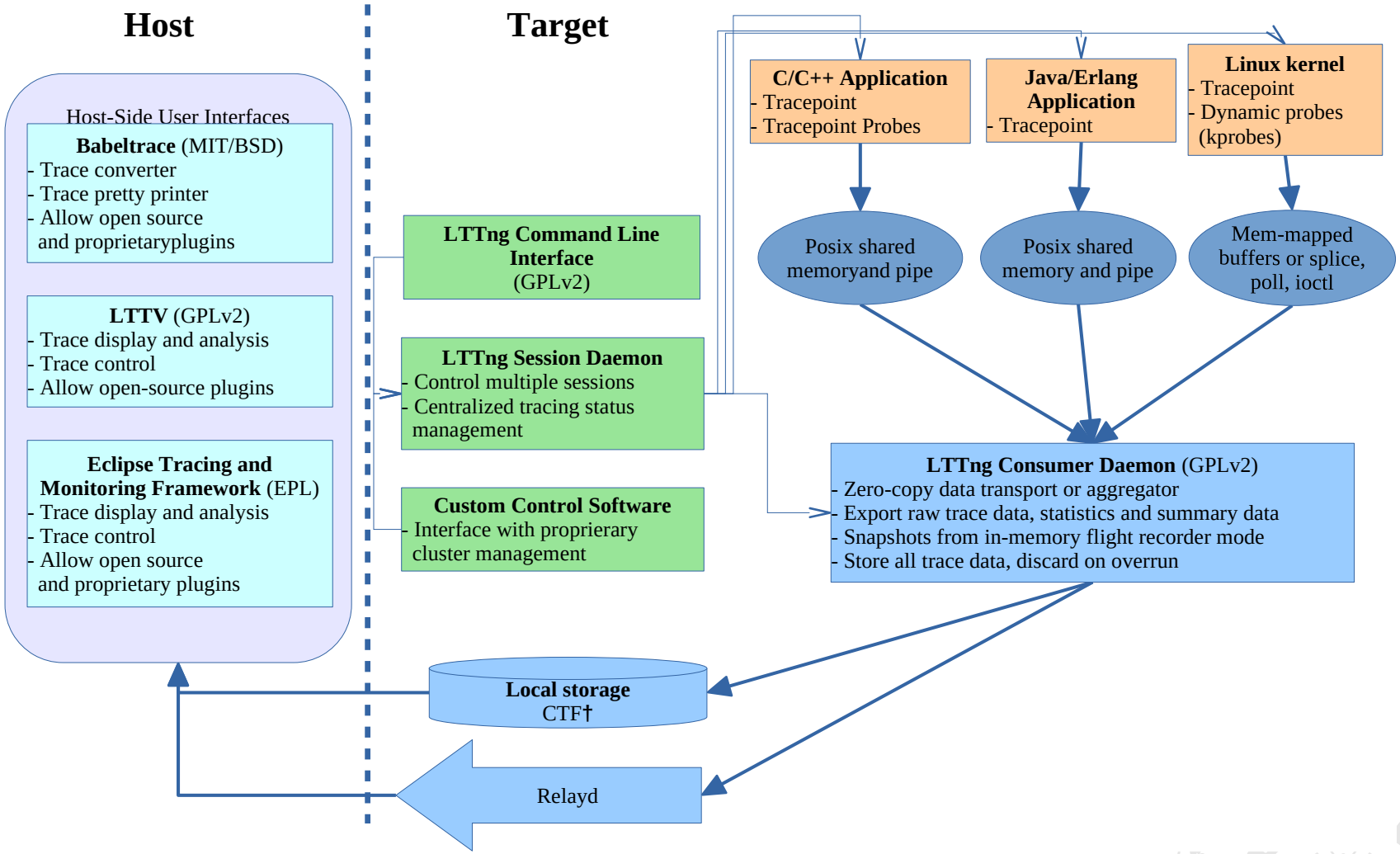
Trace simultanée noyau et application



Sessions d'usagers différents



LTTng 2.x Low-Overhead Tracing Architecture



Pourquoi une trace avec surcoût minimal?

```
if( sigaction(SIGUSR1, &new_action,NULL) <0) perror("Erreur\n");

if( sigaction(SIGUSR2, &new_action,NULL) <0)perror("Erreur\n");

if((pid = fork()) == 0) {
  kill(getppid(), SIGUSR1);
  pause(); // Mise en attente d'un signal
} else {
  kill(pid, SIGUSR2); // Envoyer un signal à l'enfant
  printf("Parent : terminaison du fils\n");
  kill(pid, SIGTERM); // Signal de terminaison à l'enfant
  pid = wait(&etat); // attendre la fin de l'enfant
  printf("Parent: fils a termine %d : %d : %d : %d\n",
      pid, WIFSIGNALED(etat), WTERMSIG(etat), SIGTERM);
}
```

Le bogue disparaît avec strace

```
root@ventoux# ./signal1
Parent : terminaison du fils
Parent: fils a termine 3824 : 1 : 15 : 15

root@ventoux# strace -o trace ./signal1
Signal SIGUSR2 reçu
Signal SIGUSR1 reçu
Parent : terminaison du fils
Parent: fils a termine 3827 : 1 : 15 : 15
root@ventoux#
```

Qu'arrive-t-il avec un traceur efficace?

```
root@ventoux# lttng create
Session auto-20120324-082956 created.
Traces will be written in /root/lttng-traces/auto-20120324-082956
root@ventoux# lttng enable-event -k -a
All kernel events are enabled in channel channel0
root@ventoux# lttng add-context -k -t pid
kernel context pid added to all channels

root@ventoux# lttng start
Tracing started for session auto-20120324-082956

root@ventoux# ./signal1
Parent : terminaison du fils
Parent: fils a termine 3851 : 1 : 15 : 15

root@ventoux# lttng stop
Tracing stopped for session auto-20120324-082956

root@ventoux# lttng view | less
root@ventoux# lttng destroy
Session auto-20120324-082956 destroyed at /root
```

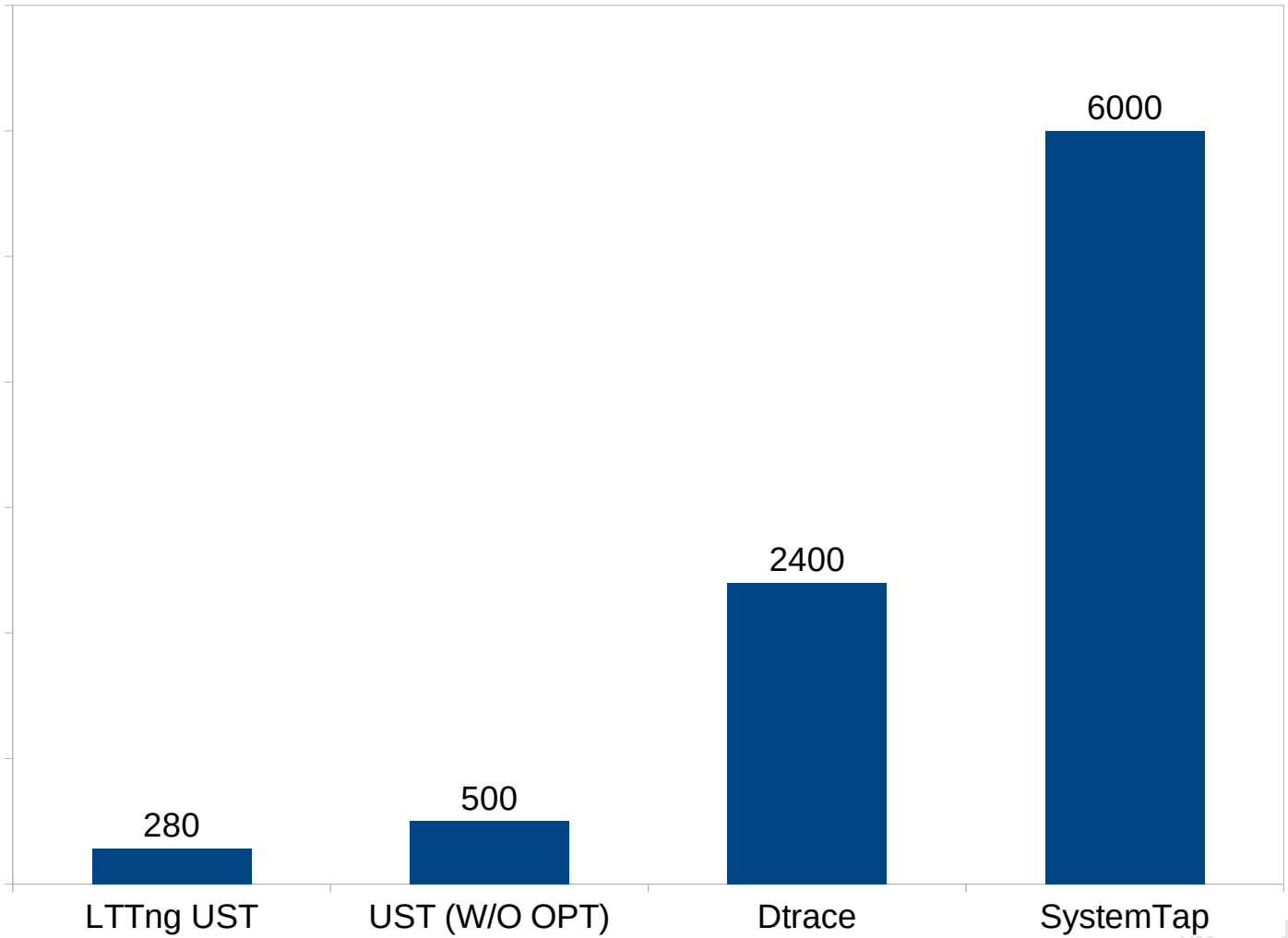
La trace des syscall et ordonnancements

```
[04.959234910] sched_process_fork: { 1 }, 3850, { parent_comm =
    "signal1", parent_tid = 3850, child_comm = "signal1",
    child_tid = 3851 }
[04.959237757] sched_migrate_task: { 1 }, 3850, { comm = "signal1",
    tid = 3851, prio = 20, orig_cpu = 1, dest_cpu = 3 }
[04.959240377] sched_wakeup_new: { 1 }, 3850, { comm = "signal1",
    tid = 3851, prio = 120, success = 1, target_cpu = 3 }
[04.959241790] exit_syscall: { 1 }, 3850, { ret = 3851 }
[04.959267207] sys_kill: { 1 }, 3850, { pid = 3851, sig = 12 }
[04.959271053] exit_syscall: { 1 }, 3850, { ret = 0 }
[04.959271925] sched_stat_wait: { 3 }, 0, { comm = "signal1",
    tid = 3851, delay = 0 }
[04.959273444] sched_switch: { 3 }, 0, { prev_comm = "kworker/0:1",
    prev_tid = 0, prev_prio = 20, prev_state = 0, next_comm = "signal1",
    next_tid = 3851, next_prio = 20 }
[04.959280598] exit_syscall: { 3 }, 3851, { ret = 0 }
```

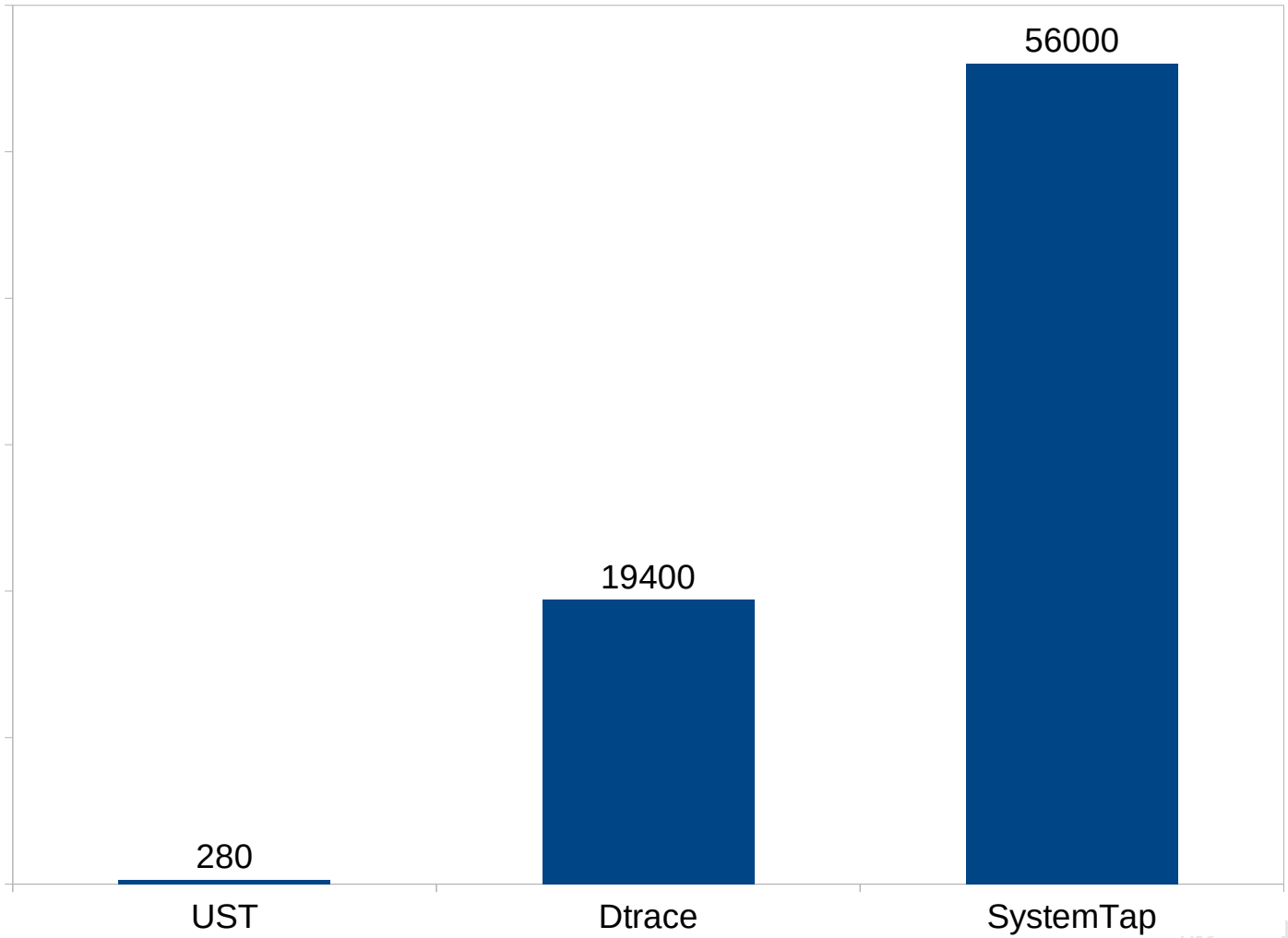
La trace des syscall et ordonnancements(2)

```
[04.959289045] sys_fstat64: { 1 }, 3850, { fd = 1, statbuf = 0xBFE03C14 }
[04.959290870] exit_syscall: { 1 }, 3850, { ret = 0 }
[04.959294082] sys_mmap_pgoff: { 1 }, 3850, { addr = 0x0, len = 4096, prot = 3,
    flags = 34, fd = 4294967295, pgoff = 0 }
[04.959297152] exit_syscall: { 1 }, 3850, { ret = -1216471040 }
[04.959307500] sys_write: { 1 }, 3850, { fd = 1, buf = 0xB77E2000, count = 29 }
[04.959320858] exit_syscall: { 1 }, 3850, { ret = 29 }
[04.959322118] sys_kill: { 1 }, 3850, { pid = 3851, sig = 15 }
[04.959324527] exit_syscall: { 1 }, 3850, { ret = 0 }
[04.959329465] sys_wait4: { 1 }, 3850, { upid = -1, stat_addr = 0xBFE03E50,
    options = 0, ru = 0x0 }
[04.959330523] sched_process_wait: { 1 }, 3850, { comm = "signal1", tid = 0,
    prio = 20 }
[04.959336128] sched_switch: { 1 }, 3850, { prev_comm = "signal1",
    prev_tid = 3850, prev_prio = 20, prev_state = 1,
    next_comm = "kworker/1:0", next_tid = 3022, next_prio = 20 }
[04.959348533] sched_switch: { 1 }, 3022, { prev_comm = "kworker/1:0",
    prev_tid = 3022, prev_prio = 20, prev_state = 1,
    next_comm = "kworker/0:0", next_tid = 0, next_prio = 20 }
[04.959366489] sched_process_exit: { 3 }, 3851, { comm = "signal1",
    tid = 3851, prio = 20 }
```

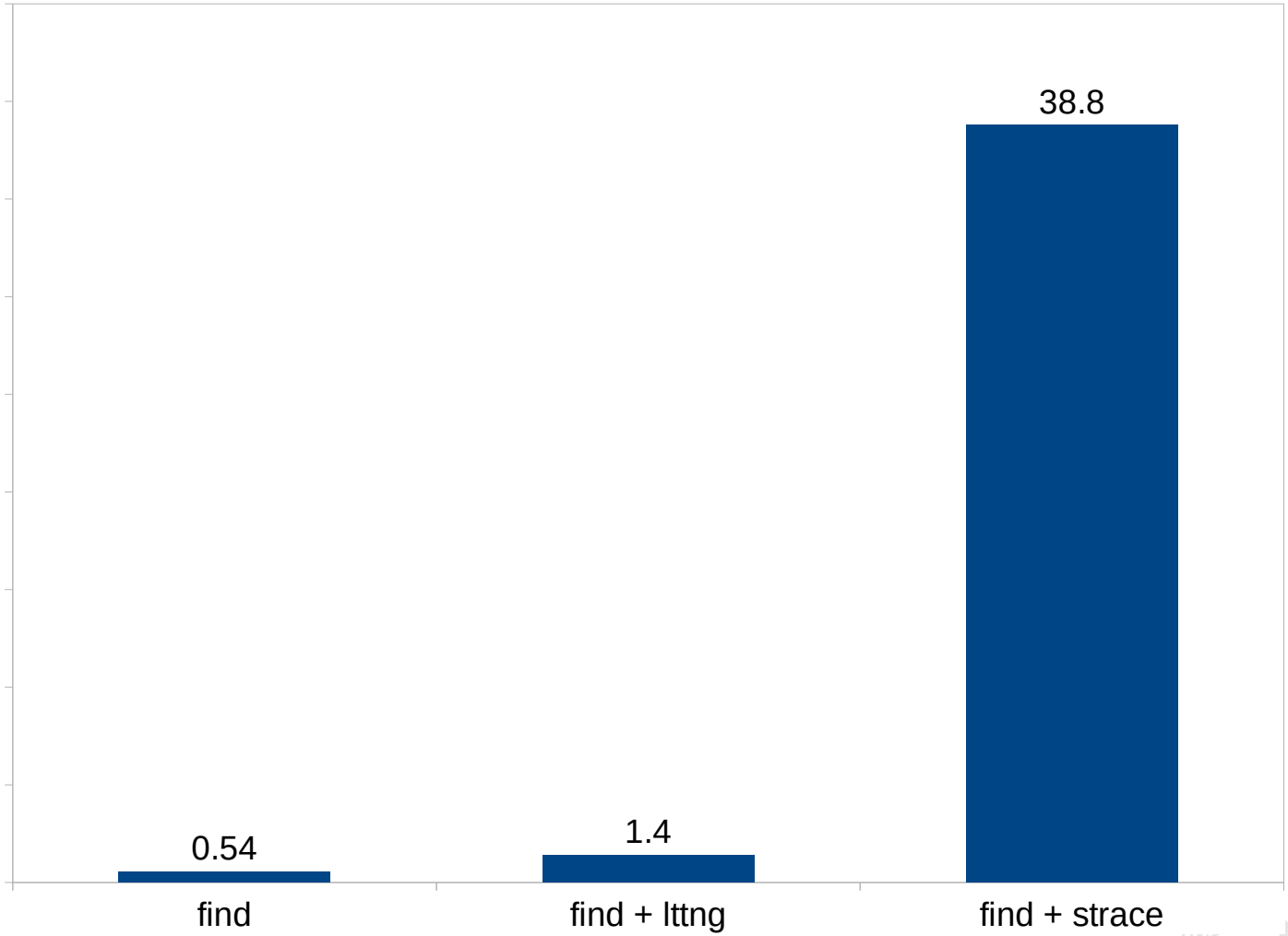
Trace en mode usager, ns / événement, 1 thread



Trace en mode usager, ns / événement, 8 threads



Strace VS LTTng, temps pour find sur 100 000 fichiers



Exemples d'optimisations utilisées

- Evénement tracé en espace usager sans aucune interaction avec le système d'exploitation.
- L'activation de point de trace basée sur if non probable.
- Format binaire natif. Normalisé en tant que Multi-Core Association Common Trace Format.
- Tampons par CPU avec opérations sans verrou, atomiques localement au CPU.
- Technique de synchronisation entre la lecture des informations de configuration (activation des points de trace, sessions...) et leur modification par la technique Read Copy Update (RCU).
- Aucune copie mémoire à mémoire pour envoyer le contenu vers le disque (Zero-copy).
- Points de trace insérés même dans un contexte NMI.
- Calcul efficace des estampilles de temps (rdtsc) même dans les machines virtuelles.

Outils d'analyse de systèmes parallèles

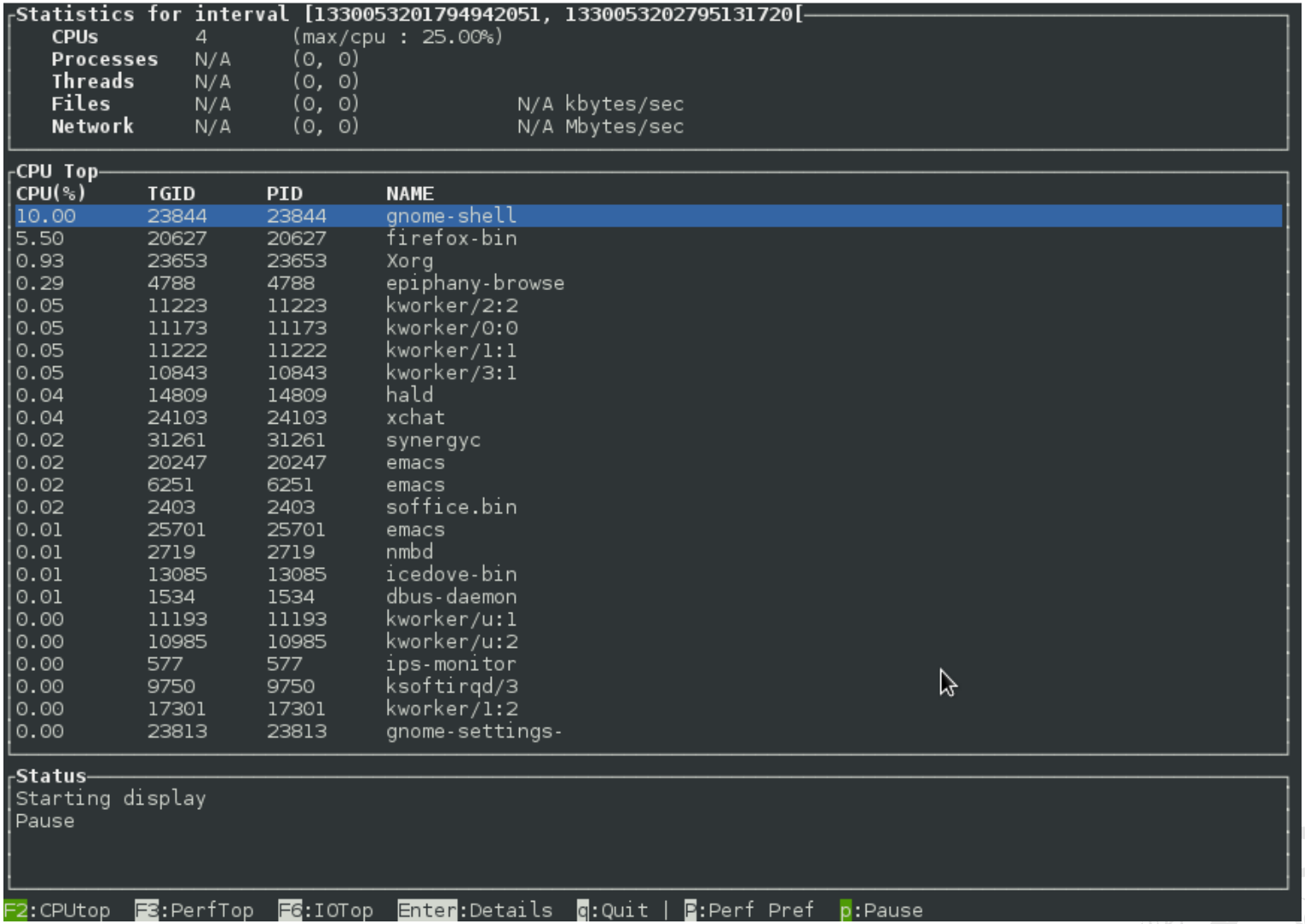
Analyse de trace

- Lister les événements de différentes traces en utilisant une référence de temps commune (synchronisation a posteriori)
- Construire un modèle du système tracé pour fournir plus d'information à l'usager (processus et fil courant sur chaque CPU, table des processus, descripteurs de fichiers ouverts...).
- Navigation efficace interactive même dans de très grandes traces (e.g. 500GiO).
- Présenter des statistiques et histogrammes, des lignes de temps montrant l'évolution de l'état de différentes ressources (état des processus, processus courant sur chaque CPU...).
- Aider à retracer les interactions entre les événements et les états afin de montrer le chemin critique d'un groupe d'applications.

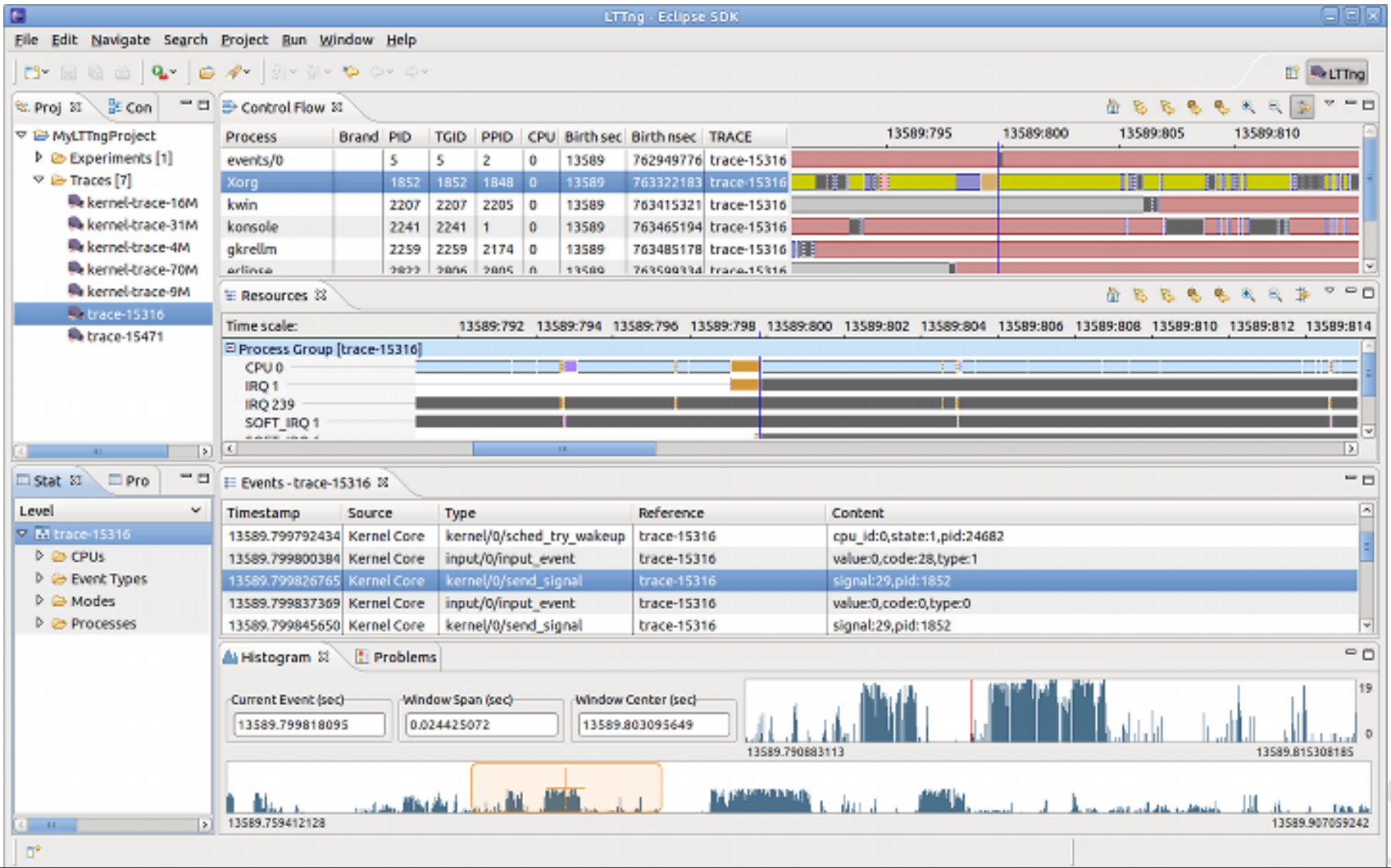
Babeltrace

```
[13:58:29.128909723] (+0.000002475) sys_read: { 0 }, { "firefox-bin", 3363 }, { fd = 5, buf =
 count = 16 }
[13:58:29.128911513] (+0.000001790) exit_syscall: { 0 }, { "firefox-bin", 3363 }, { ret = -11
[13:58:29.128919672] (+0.000008159) sys_write: { 0 }, { "firefox-bin", 3363 }, { fd = 5, buf
, count = 8 }
[13:58:29.128921404] (+0.000001732) exit_syscall: { 0 }, { "firefox-bin", 3363 }, { ret = 8 }
[13:58:29.128922884] (+0.000001480) sys_read: { 0 }, { "firefox-bin", 3363 }, { fd = 19, buf
, count = 1 }
[13:58:29.128925765] (+0.000002881) exit_syscall: { 0 }, { "firefox-bin", 3363 }, { ret = 1 }
[13:58:29.128928120] (+0.000002355) sys_write: { 0 }, { "firefox-bin", 3363 }, { fd = 5, buf
, count = 8 }
[13:58:29.128929552] (+0.000001432) exit_syscall: { 0 }, { "firefox-bin", 3363 }, { ret = 8 }
[13:58:29.129020005] (+0.000090453) exit_syscall: { 0 }, { "acpid", 1536 }, { ret = 1 }
[13:58:29.129025587] (+0.000005582) sys_rt_sigprocmask: { 0 }, { "acpid", 1536 }, { how = 0,
 oset = 0x0, sigsetsize = 8 }
[13:58:29.129027993] (+0.000002406) exit_syscall: { 0 }, { "acpid", 1536 }, { ret = 0 }
[13:58:29.129030188] (+0.000002195) sys_poll: { 0 }, { "acpid", 1536 }, { ufds = 0x7FFF2A055D
meout_msecs = 0 }
[13:58:29.129032570] (+0.000002382) exit_syscall: { 0 }, { "acpid", 1536 }, { ret = 0 }
[13:58:29.129033929] (+0.000001359) sys_rt_sigprocmask: { 0 }, { "acpid", 1536 }, { how = 1,
 oset = 0x0, sigsetsize = 8 }
[13:58:29.129035144] (+0.000001215) exit_syscall: { 0 }, { "acpid", 1536 }, { ret = 0 }
[13:58:29.129037520] (+0.000002376) sys_read: { 0 }, { "acpid", 1536 }, { fd = 4, buf = 0x7FF
 = 24 }
```

lttngtop


Statistics for interval [1330053201794942051, 1330053202795131720[
CPUs 4 (max/cpu : 25.00%)
Processes N/A (0, 0)
Threads N/A (0, 0)
Files N/A (0, 0) N/A kbytes/sec
Network N/A (0, 0) N/A Mbytes/sec
CPU Top
CPU(%) TGID PID NAME
10.00 23844 23844 gnome-shell
5.50 20627 20627 firefox-bin
0.93 23653 23653 Xorg
0.29 4788 4788 epiphany-browse
0.05 11223 11223 kworker/2:2
0.05 11173 11173 kworker/0:0
0.05 11222 11222 kworker/1:1
0.05 10843 10843 kworker/3:1
0.04 14809 14809 hald
0.04 24103 24103 xchat
0.02 31261 31261 synergyc
0.02 20247 20247 emacs
0.02 6251 6251 emacs
0.02 2403 2403 soffice.bin
0.01 25701 25701 emacs
0.01 2719 2719 nmbd
0.01 13085 13085 icedove-bin
0.01 1534 1534 dbus-daemon
0.00 11193 11193 kworker/u:1
0.00 10985 10985 kworker/u:2
0.00 577 577 ips-monitor
0.00 9750 9750 ksoftirqd/3
0.00 17301 17301 kworker/1:2
0.00 23813 23813 gnome-settings-
Status
Starting display
Pause
F2:CPUtop F3:PerfTop F6:IOTop Enter:Details q:Quit | P:Perf Pref p:Pause

Eclipse Linux Tools

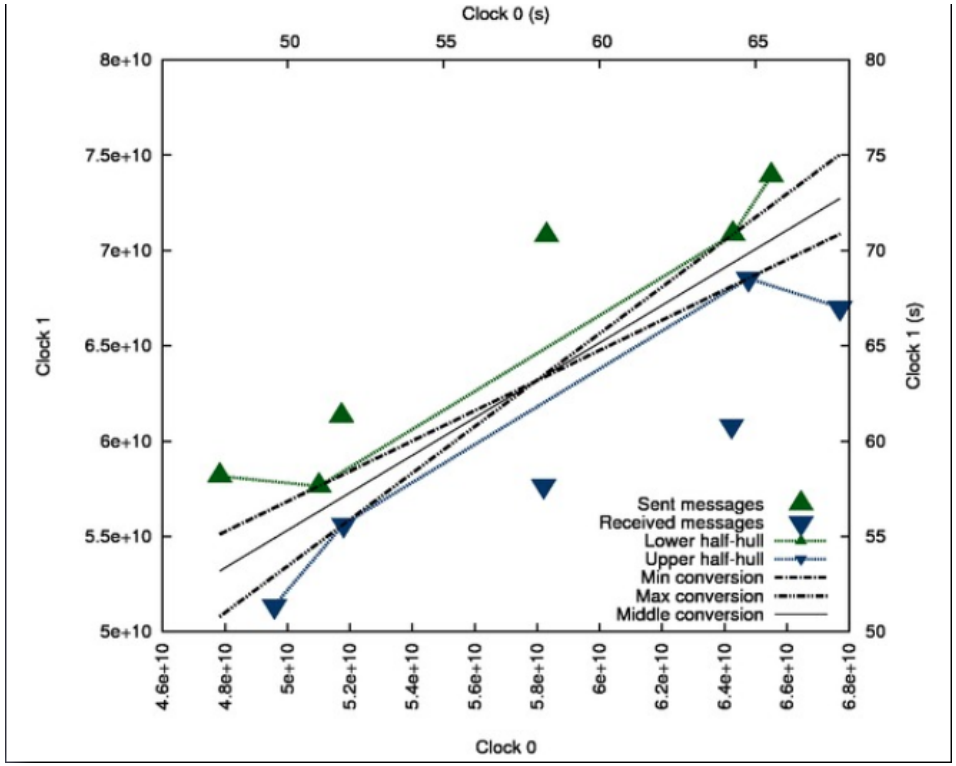


Synchronisation de traces

- Chaque noeud a une horloge indépendante.
- Différence de premier ordre: valeur initiale et fréquence différente.
- Effets de second ordre: variabilité de l'horloge, latence de lecture de l'horloge, dérive de la fréquence dans le temps (fonction par exemple de la température, l'humidité, la tension d'Hydro-Québec...).
- Identifier les envois et réceptions correspondants pour les paquets envoyés sur le réseau et utiliser cette information pour estimer les coefficients de la droite horloge A versus horloge B.

Synchronisation par enveloppe convexe

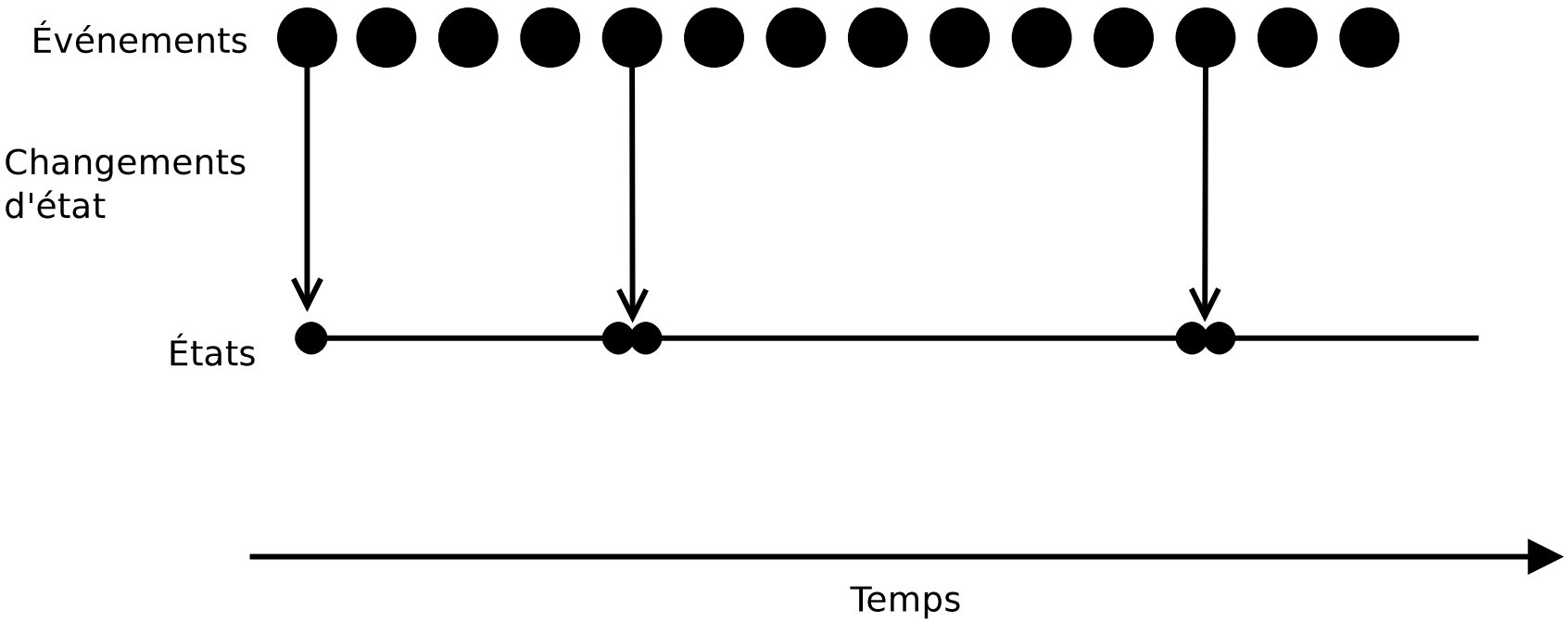
- Algorithme efficace incrémental pour estimer la droite de correspondance.
- Les envois et réceptions de paquets sont appariés par leur numéro de séquence TCP en utilisant une table de hachage.



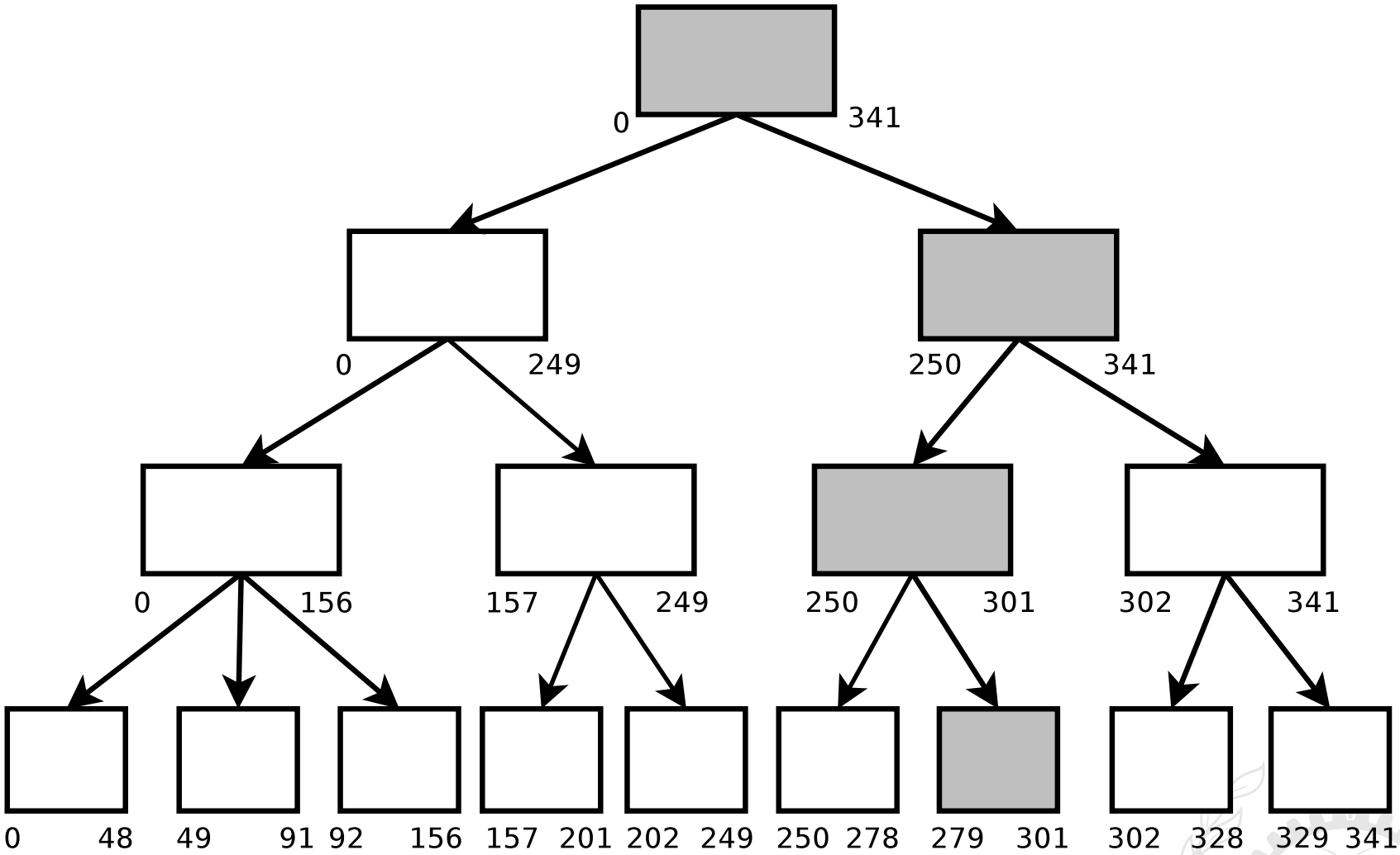
Modélisation de l'état des ressources

- Description en XML des différents événements (type, valeurs des champs...) et des changements d'état qu'ils causent
- Arbre d'attributs représentant les ressources (CPU, processus, descripteurs...) et leur état courant.
- Base de donnée spécialisée de l'historique de l'arbre des attributs d'état.

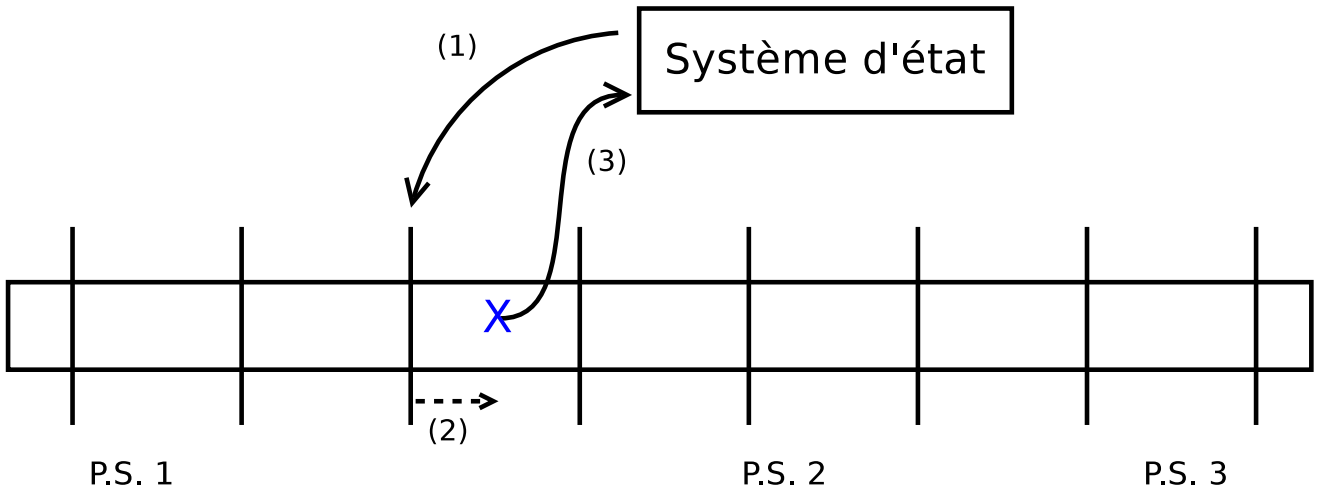
Modélisation de l'état

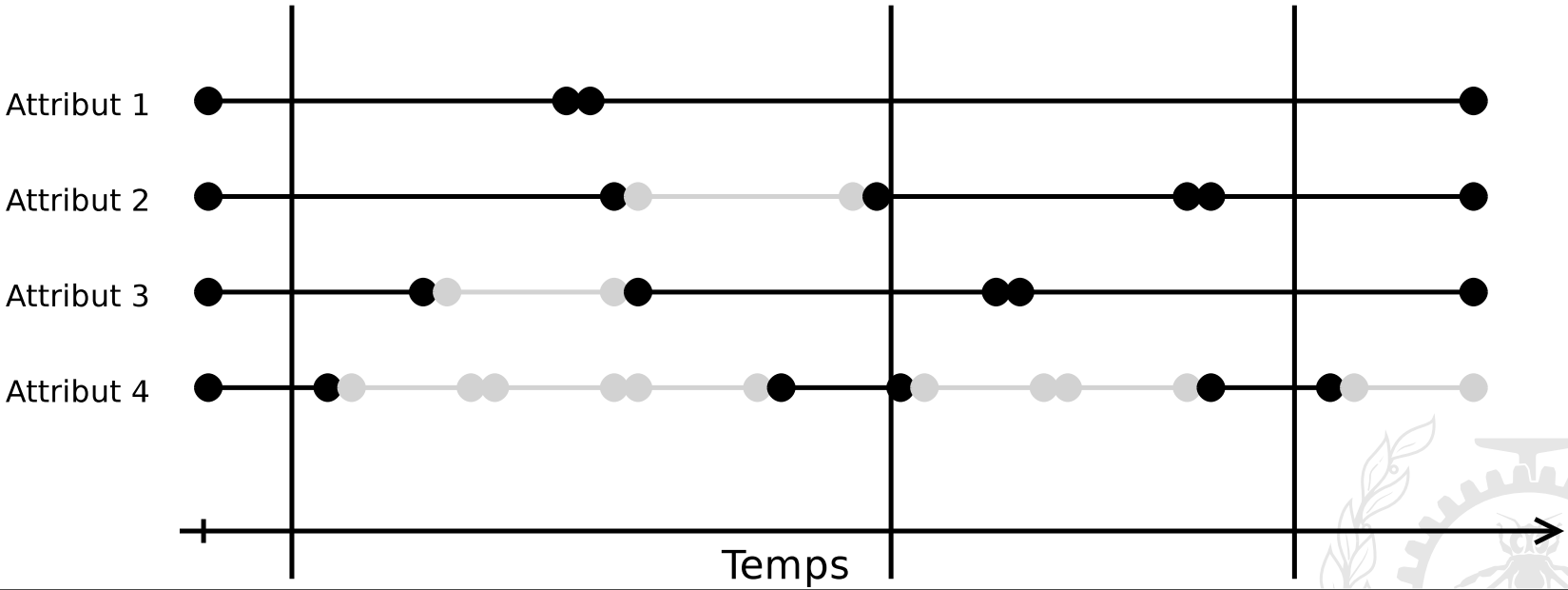


Historique de l'état

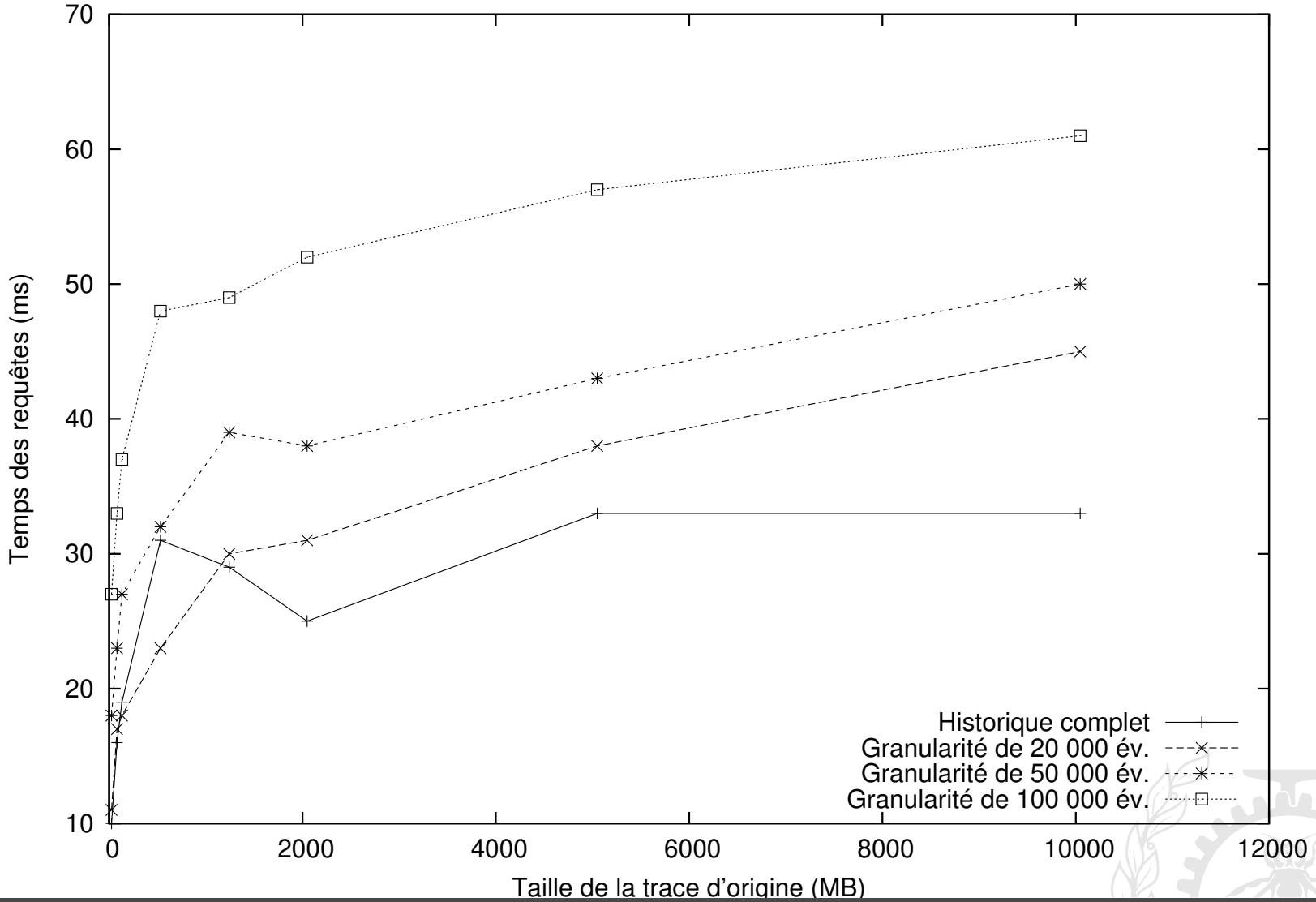


Etat modélisé





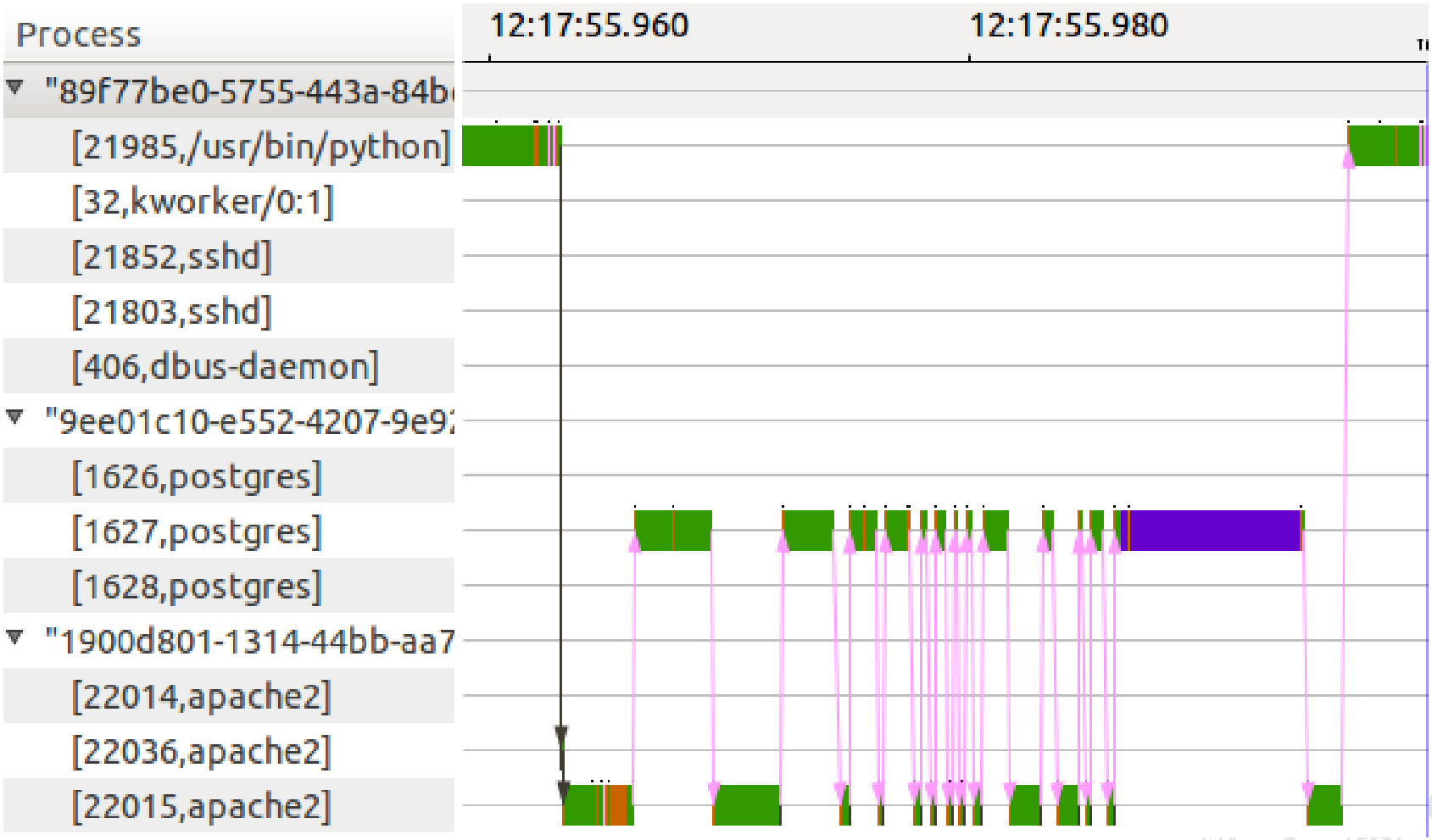
Requête sur l'historique d'état



Analyse du chemin critique

- Pour chaque processus, le temps est décomposé par état: en exécution, en attente de ressource (CPU, fichier, tube, socket, minuterie...).
- Le chemin critique pour atteindre un point donné dans un processus est la chaîne des états reliés par des "réveils" permettant débloquer un processus.
- Montrer comment le chemin critique se décompose par processus et par état (e.g. une requête HTTP a pris 3s de Firefox, 2s de X11, 5s de Apache, 3s de MySQL).

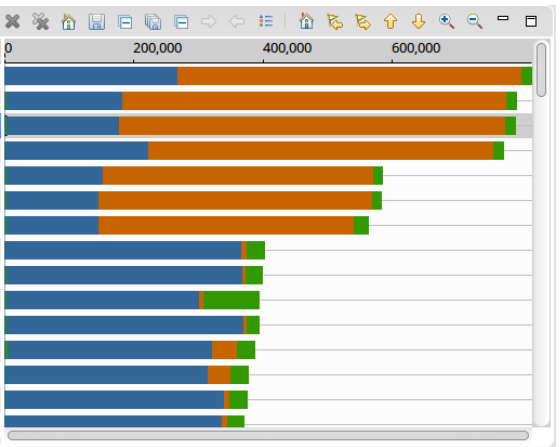
Chemin critique

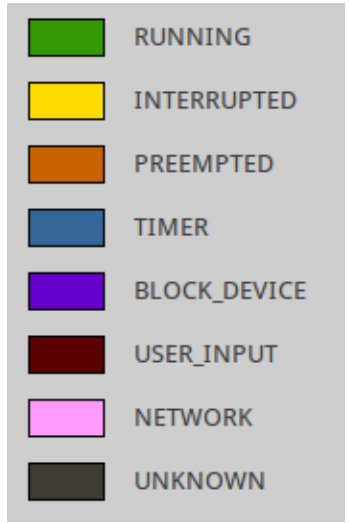


Comparaison de tâches temps réel

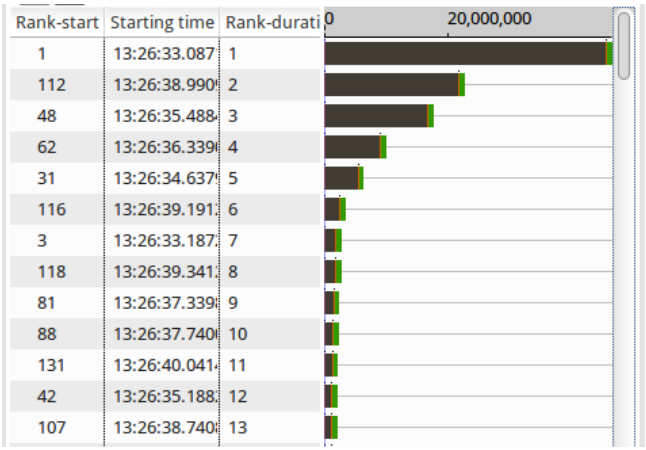
Stackbars

Rank-start	Starting time	Rank-duration	Elapsed	Tid	Name	Running	Preempted
6512	17:44:23.626622785	1	814966	22015	timer	17092	530678
5407	17:44:23.346201997	2	791927	22015	timer	20258	593460
4326	17:44:23.074673317	3	790409	22015	timer	19904	597648
6094	17:44:23.518375271	4	772687	22015	timer	18452	533306
307	17:44:22.066446552	5	584651	22015	timer	18796	417095
1227	17:44:22.297204106	6	583135	22015	timer	18651	423377
2330	17:44:22.573953293	7	563930	22015	timer	27012	394978
852	17:44:22.203374386	8	401778	22015	timer	28081	8022
92	17:44:22.012650591	9	398794	22015	timer	29505	5893
2842	17:44:22.702171764	10	394549	22015	timer	90284	7888
7852	17:44:23.966398830	11	393609	22015	timer	21984	5338
953	17:44:22.228628008	12	386999	22015	timer	32593	37716
889	17:44:22.212617985	13	377255	22015	timer	28344	35306
7152	17:44:23.788113115	14	374956	22015	timer	28409	8458
1676	17:44:22.409627769	15	370939	22015	timer	28334	7955

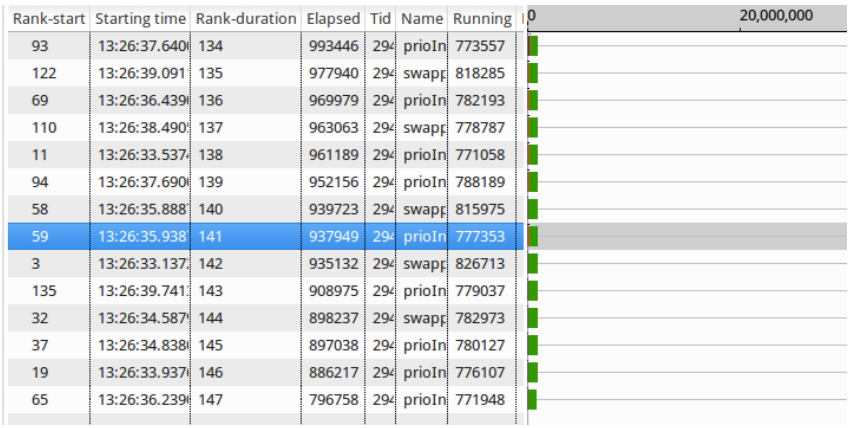




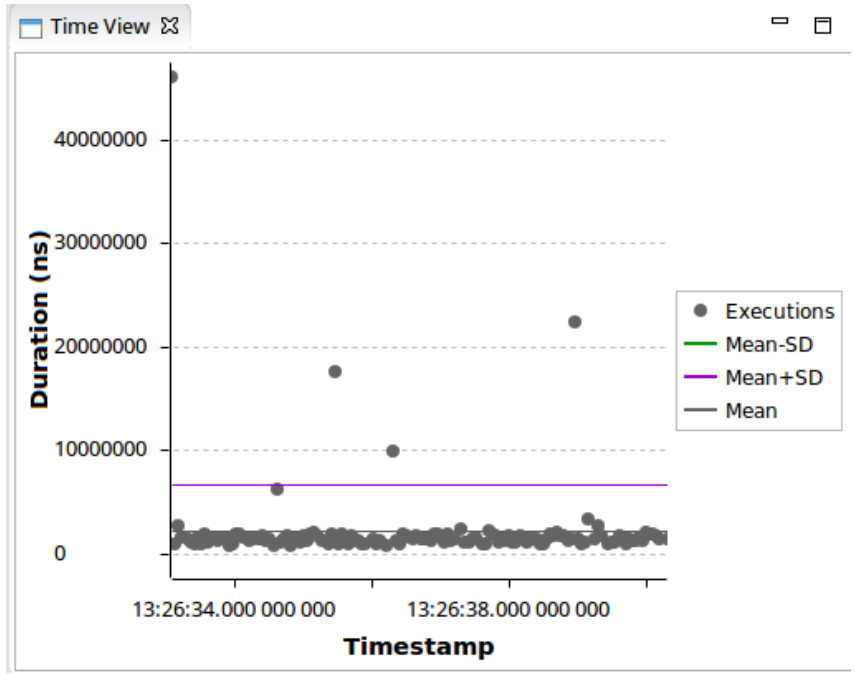
Distribution des tâches temps réel



Problematic executions

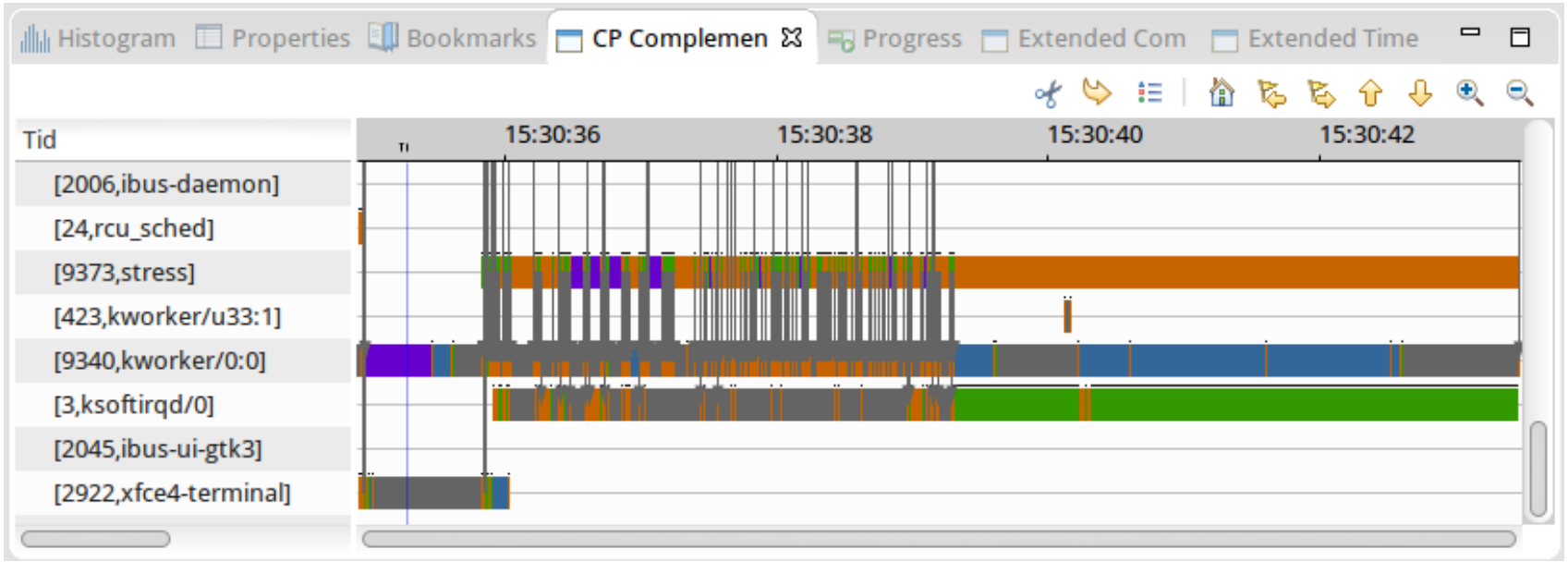


Normal executions



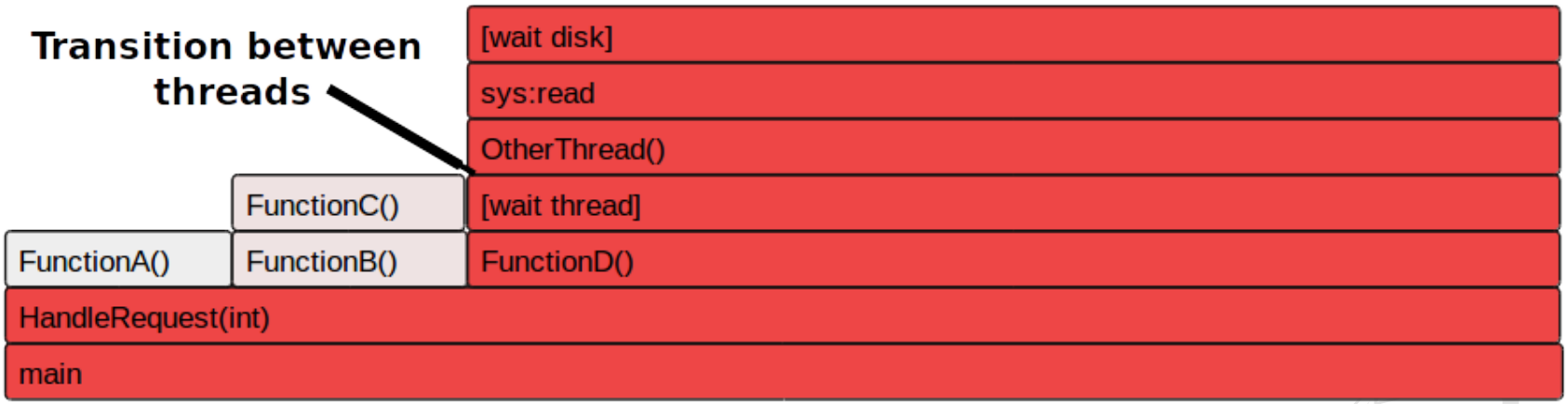
Time View

Chemin critique pour une tâche



Visualiser les différences entre groupes

- Vue de la pile d'appel en temps cumulé selon deux groupes différents d'exécutions (Flame Graph).
- Les différences de temps moyen d'exécution entre les deux groupes sont affichées (gris semblable, vert plus rapide et rouge plus lent).



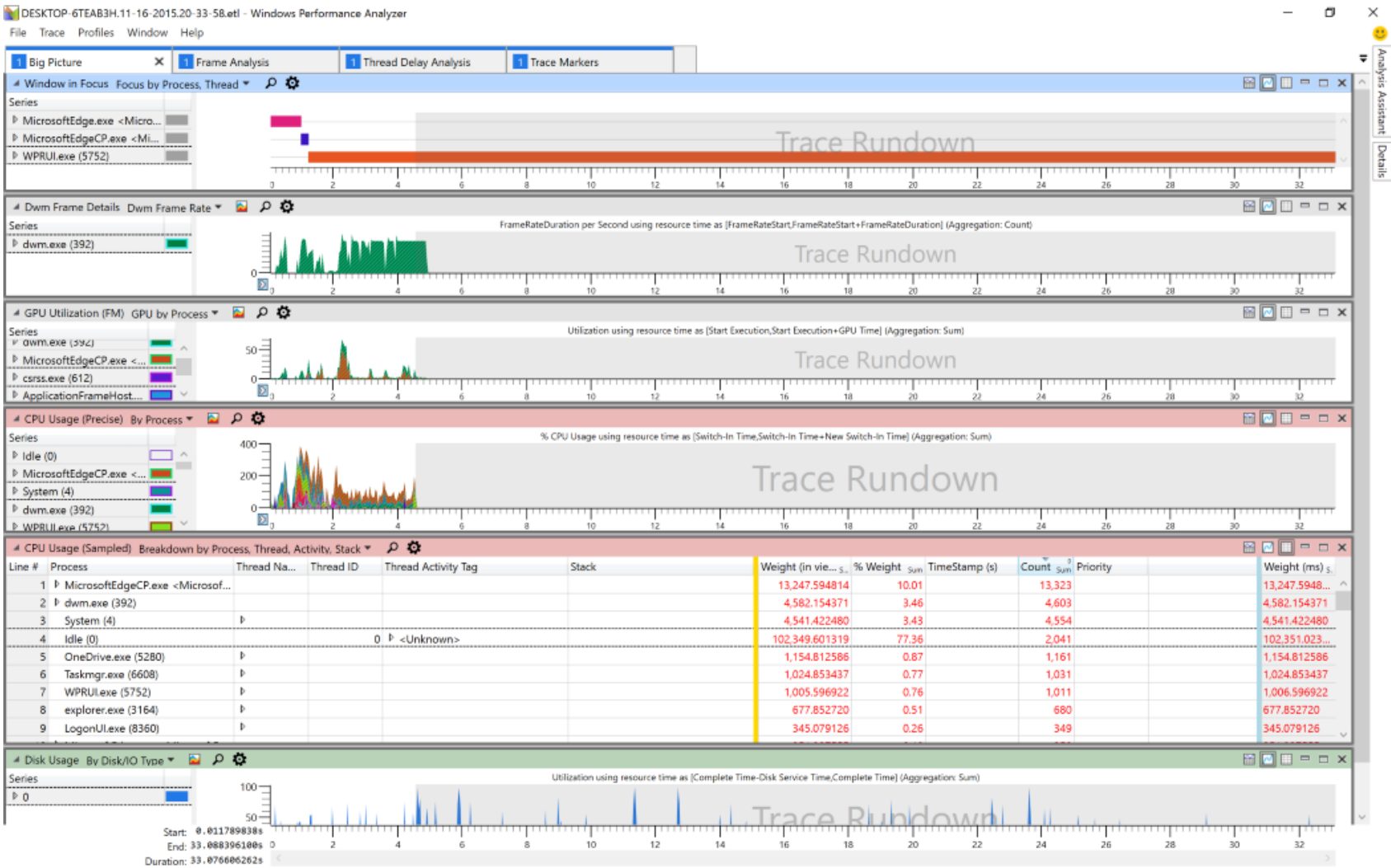
Intel Vtune

- Offert sur Linux et Windows.
- Profilage par minuterie ou compteur de performance. Temps par fonction (self, appelants, appelés...). Différents paramètres (instructions, fautes de cache...) par fonction ou par instruction.
- Vue de l'état des fils d'exécution en fonction du temps.
- Vue de l'attente active et passive, par verrou ou section critique, avec les appelants.
- API pour définir les tâches internes d'un programme (start/stop task) ou les primitives de synchronisation sur mesure.

Windows Performance Toolkit

- Instrumentation du noyau et des librairies système avec Event Tracing for Windows (ETW).
- Possibilité d'ajouter des points de trace dans ses propres applications.
- Peut activer dynamiquement le traçage avec un surcoût relativement faible.
- Les traces peuvent être examinées avec un outil graphique.
- La trace doit être examinée sur l'ordinateur où elle a été générée.

Windows Performance Analyzer



Vampir

- Développé en Allemagne par le centre de recherche de Jülich et l'université de Dresden pour étudier les programmes parallèles (MPI). Distribué commercialement par Pallas, compagnie achetée par Intel. Depuis 2005, la coopération avec Intel est terminée et Dresden continue le développement.
- Vue des événements ou états en fonction du temps.
- Vue du temps ou autres métriques par fonction. Arbre d'appel.
- Histogrammes de différentes métriques.
- Matrice des communications
- Possibilité de définir des métriques dérivées, des seuils, couleurs...
- Open Trace Format (format textuel et API pour lire les traces).

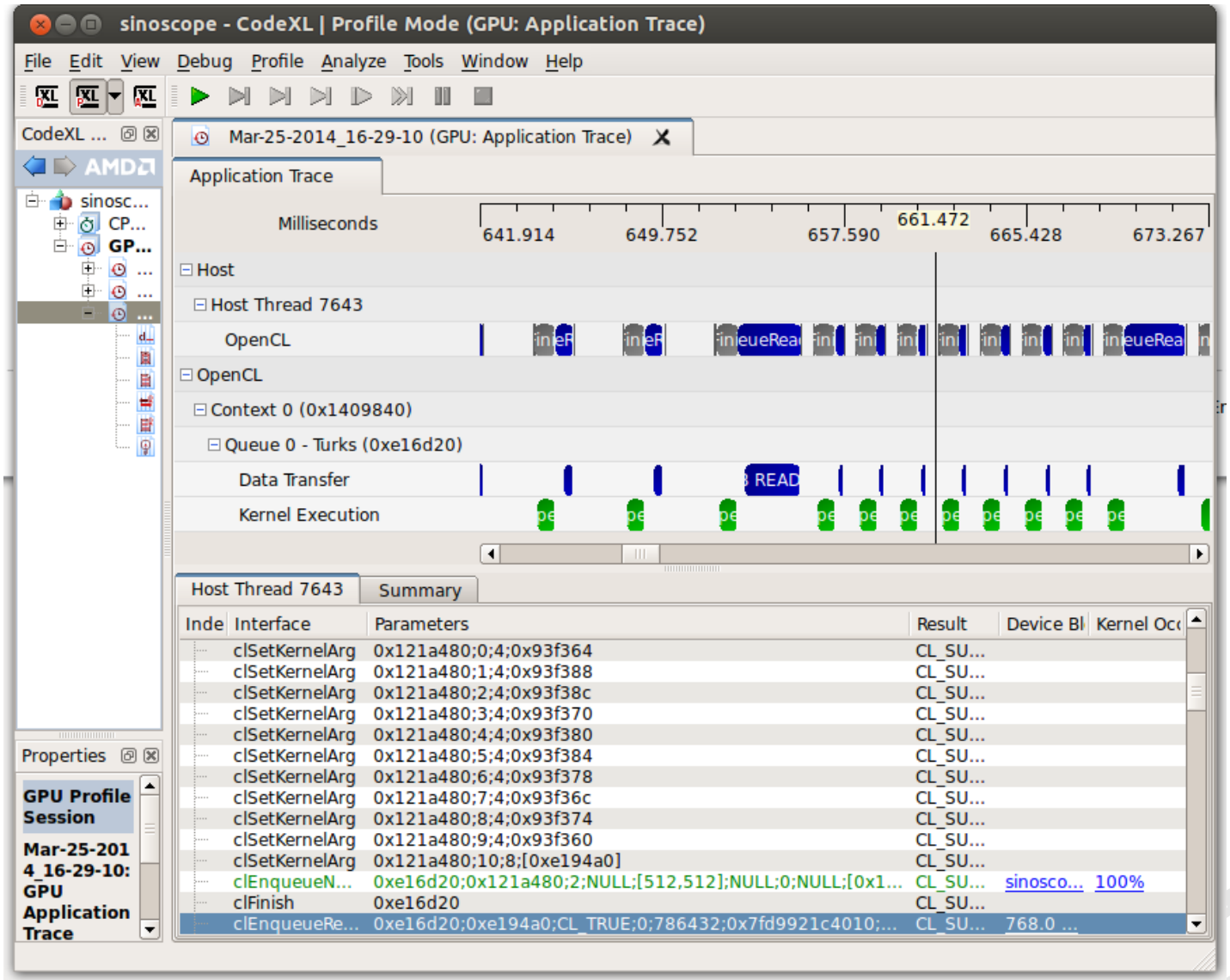
PARAVER

- PARAllel Visualization and Events Representation: développé à Barcelone au Centro Nacional de Supercomputación, initialement pour PVM.
- Outil de visualisation générique, entièrement adaptable à différents types de traces à quelques hypothèses près.
- Vues principales: lignes de temps et statistiques.
- Possibilité d'ajouter des modules d'analyse supplémentaires.
- Possibilité de comparer des traces.

Tuning Analysis Utilities (TAU)

- TAU est un projet conjoint de l'université d'Oregon, Los Alamos (LANL) et du centre de recherche de Jülich.
- Instrumentation par interposition de librairie, instrumentation par le compilateur, ou analyse statique de code source pour l'insertion automatique d'instrumentation (début et fin de fonction, de bloc de base...).
- Activation sélective des points de trace.
- Utilisation de minuterie ou de compteurs matériels pour obtenir un profil. Option de mémoriser le chemin d'appel de profondeur n.
- Paraprof pour examiner les profils.
- Jumpshot pour voir l'état des fils d'exécution en fonction du temps.

CodeXL de AMD



APItrace

- Trace les appels OpenGL (ES), Direct3D et DirectDraw
- Peut rejouer les appels pris dans une trace tout en examinant l'état:
  - voir les appels;
  - voir le contexte graphique;
  - regarder les images et textures.
- Permet d'éditer le contenu d'une trace.
- Ligne de temps des appels dans la trace.

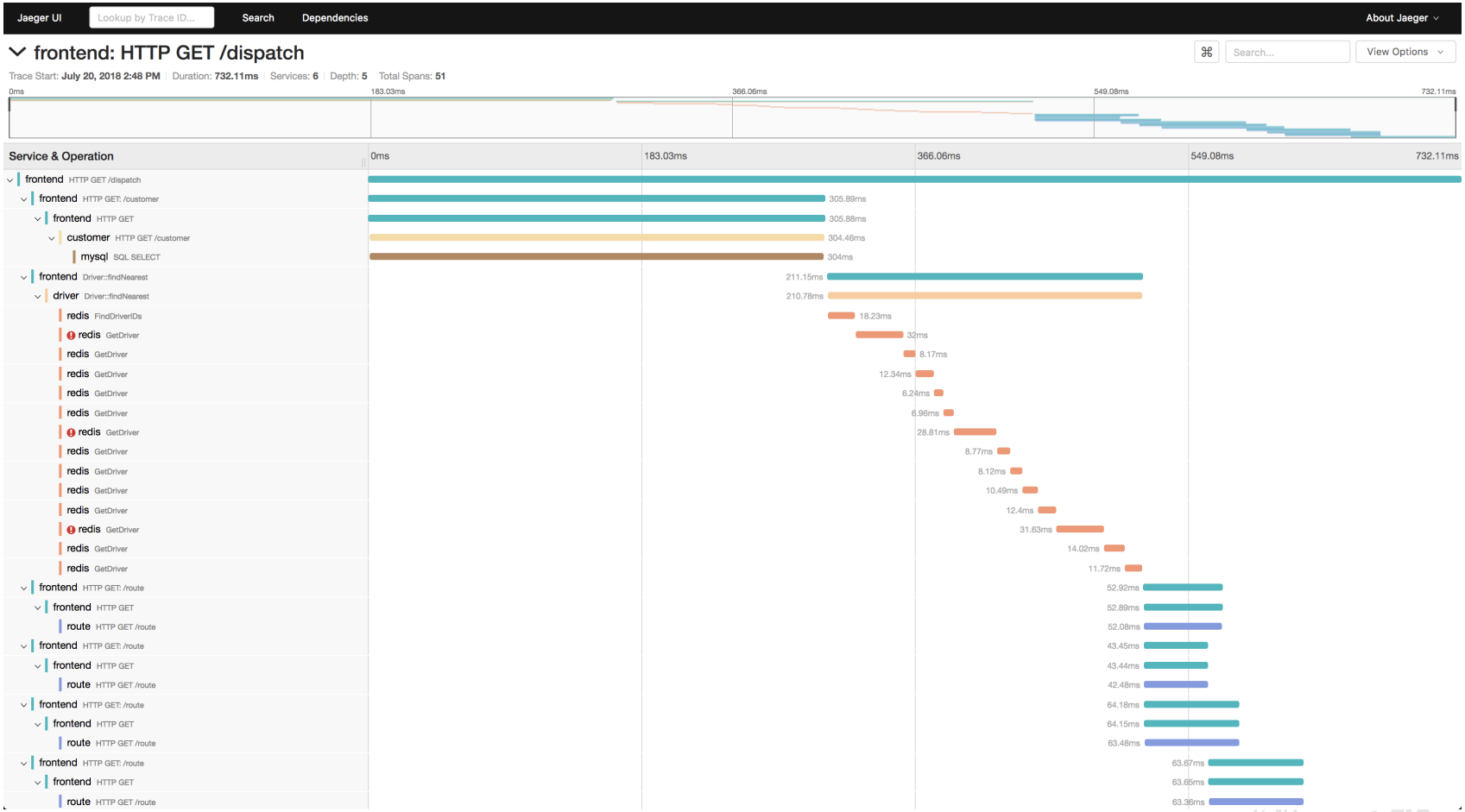
GPUView

- Trace des événements noyau (ordonnancements) et en particulier de toutes les commandes du sous-système de graphisme et des pilotes de cartes graphiques.
- Ligne de temps des processus et des queues du sous-système graphique et des cartes graphiques (avec tailles de ces queues).
- Possibilité de superposer la position de certains événements comme la synchronisation verticale (nouveau rafraichissement de l'image).

OpenTelemetry

- API pour insérer des points de trace et environnement pour la collecte de données.
- Journal / Log: structuré est comme des fichiers de trace de LTTng, non structuré est comme des journaux avec syslog.
- Métriques / Metrics: échantillons pris à intervalle.
- Trace: trace répartie pour des requêtes distribuées. Arbre de requêtes impriquées avec pour chacune le début, la fin et la requête parent.

Jaeger


Jaeger UI
Lookup by Trace ID...
Search
Dependencies
About Jaeger
frontend: HTTP GET /dispatch
Search...
View Options
Trace Start: July 20, 2018 2:48 PM | Duration: 732.11ms | Services: 6 | Depth: 5 | Total Spans: 51
0ms
183.03ms
366.06ms
549.08ms
732.11ms
Service & Operation
frontend HTTP GET /dispatch
frontend HTTP GET: /customer
frontend HTTP GET
customer HTTP GET /customer
mysql SQL SELECT
frontend Driver::findNearest
driver Driver::findNearest
redis FindDriverIDs
redis GetDriver
route HTTP GET /route
305.89ms
305.88ms
304.46ms
304ms
211.15ms
210.78ms
18.23ms
32ms
8.17ms
12.34ms
6.24ms
6.96ms
28.81ms
8.77ms
8.12ms
10.49ms
12.4ms
31.63ms
14.02ms
11.72ms
52.92ms
52.89ms
52.08ms
43.45ms
43.44ms
42.48ms
64.18ms
64.15ms
63.48ms
63.67ms
63.65ms
63.36ms

Jaeger et Prometheus

- Jaeger est un outil graphique d'analyse pour les traces réparties de OpenTelemetry.
- Prometheus est un outil de collecte et d'investigation de métriques (séries temporelles).
- Grafana est un outil de visualisation de flots de données comme les séries temporelles venant de Prometheus.
- Il existe plusieurs base de données spécialisées pour les séries temporelles comme TSDB ou InfluxDB.

Promela / SPIN

- Modèle en Promela qui ressemble au C.
- Ensemble de processus concurrents.
- Chaque énoncé ou bloc Atomic dans un processus est ordonnancé avant ou après les autres processus.
- Bloc do qui contient plusieurs branches; une de celles dont la condition est vraie est choisie.
- Possibilité d'ajouter des assertions et des printf.
- Le modèle peut être simulé en choisissant aléatoirement entre les alternatives et imprimant avec printf.
- Le modèle peut être validé. Il essaie tous les ordonnancements/alternatives possibles jusqu'à une assertion fausse et fournit la trace qui l'invalide.

Prise de verrou

```
byte A = 1; bool A_done = 0;
byte B = 2; bool B_done = 0;
byte x = 0; bool x_done = 0;
byte y = 0; bool y_done = 0;
proctype writer()
{ do
  :: ! A_done -> A = 3; A_done = true;
  :: ! B_done -> B = 4; B_done = true;
  :: A_done && B_done -> break;
  od; }
proctype reader()
{ do
  :: ! x_done -> x = A; x_done = true;
  :: ! y_done -> y = B; y_done = true;
  :: x_done && y_done -> break;
  od; }
Init { atomic {
  run reader(); run writer();
  do :: x_done && y_done -> break; od;
  printf("x = %d, y = %d\n", x, y); } }
```

```
x = 3, y = 4
x = 3, y = 4
...
x = 3, y = 2
x = 3, y = 2
...
x = 1, y = 2
x = 1, y = 2
...
x = 1, y = 4
x = 1, y = 4
...
```

Outils d'analyse de systèmes parallèles

Discussion

- Comment trouver les problèmes lorsque 1024 noeuds de 64 processeurs travaillent chacun à 4 milliards d'instructions par seconde?!
- Regarder les profils et métriques de haut niveau pour voir les tendances et trouver où sont consommées les ressources.
- Regarder les traces détaillées pour comprendre les problèmes complexes au plus bas niveau.