Programmation C avancée

Concepts & Outils
pour le développement

maj 01/2023

Tests

- Il existe plusieurs types de tests.
- Les plus importants sont :
  - Les tests unitaires
  - Les tests fonctionnels
  - Les tests d'intégration
  - Les tests de recette

Les tests unitaires

- Les tests unitaires ont pour objet de valider le **fonctionnement** d'une fonction.
- Pour qu'un test unitaire soit correct, il faut tester le fonctionnement « normal » ainsi qu'aux limites (cas NULL, domaine de valeur).
- Le test unitaire repose sur la **spécification** de la fonction et non son implémentation.

Les tests unitaires

- Ils doivent mettre en œuvre l'ensemble des fonctionnalités décrites dans les **spécifications**, et explorer le fonctionnement du module dans des conditions non spécifiées
  - Doivent également tester le code dédié à la gestion des erreurs
  - Définition de jeux de tests représentatifs
  - Comparaison à des résultats attendus

Les tests unitaires

- Le code dédié aux tests unitaires doit faire partie du code du module
  - Permet la ré-utilisabilité des tests en même temps que du code du module proprement dit
  - Facilite l'extensibilité des tests

Les tests unitaires

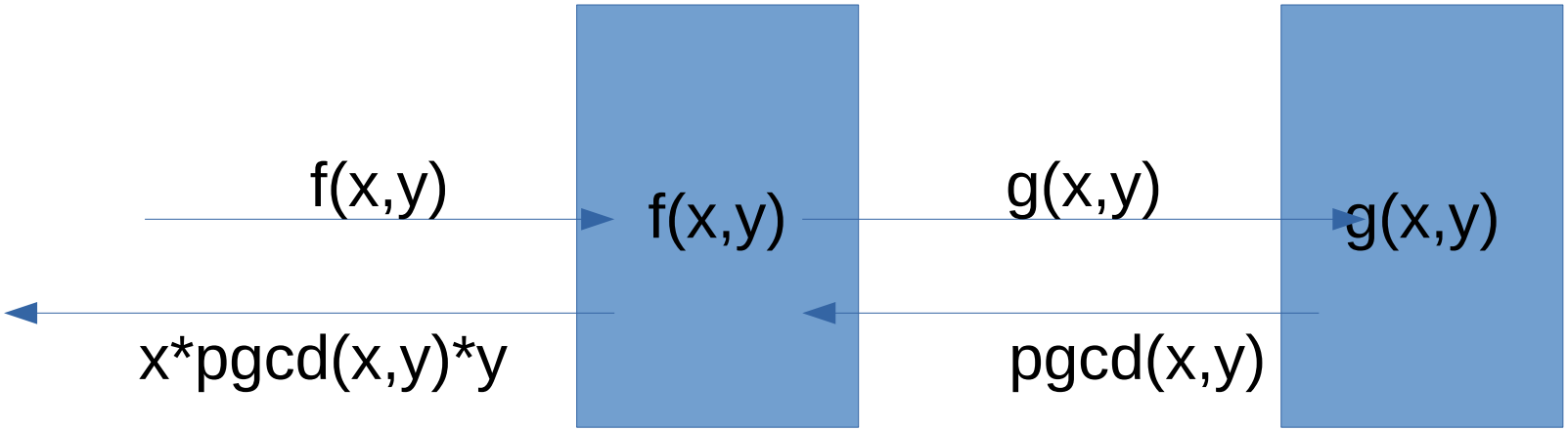
- Tout module `module.c` doit disposer d'au moins un fichier `module_test_main.c` contenant la procédure de test unitaire
  - Contient une fonction `main()`
  - Construit par la commande « `make test` »
  - Renvoie un code de succès « `exit (0)` » ou d'échec
    - Permet la conduite automatique des tests au moyen de scripts shell
- Procédure documentée dans le manuel de maintenance

Les tests unitaires : faussaires

- Dans le cas des tests unitaires, on souhaite que le tests concerne uniquement une fonction.
- Si le test plante, alors cela doit indiquer un bug dans la fonction correspondante.
- Que faire si une fonction ou un module repose sur un autre code ?

Tests inter module: faussaires

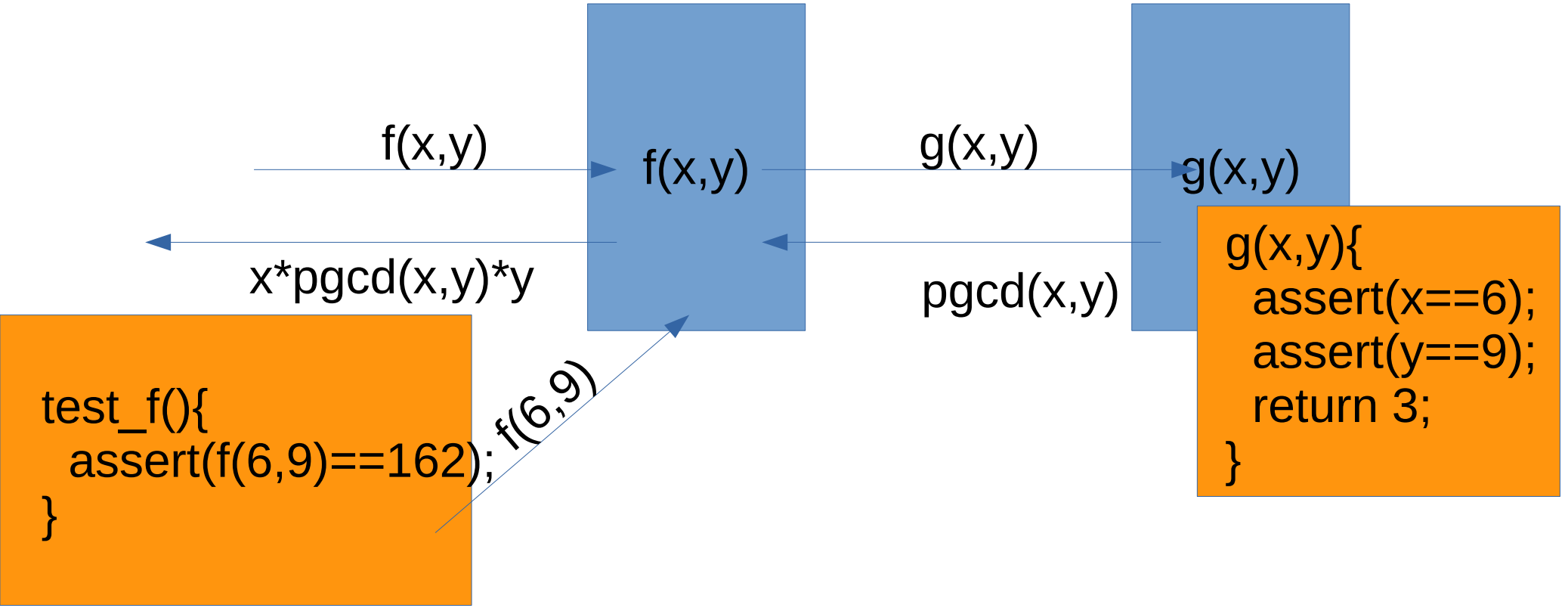
- Pour tester un module indépendamment d'un autre module



Pour tester f(), on écrit une fausse fonction g qui va tester
la valeur de ses paramètres et renvoyer une valeur pré-calculée

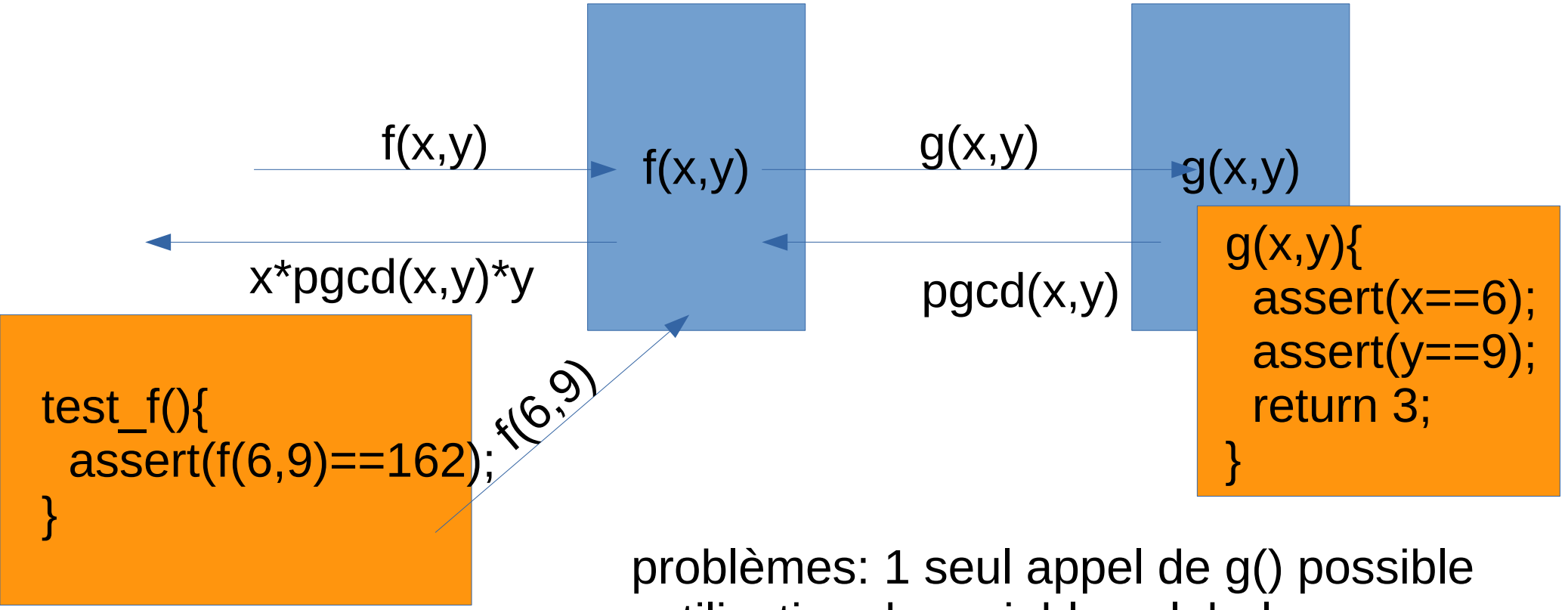
Tests inter module: faussaires

- Pour tester un module indépendamment d'un autre module



Tests inter module: faussaires

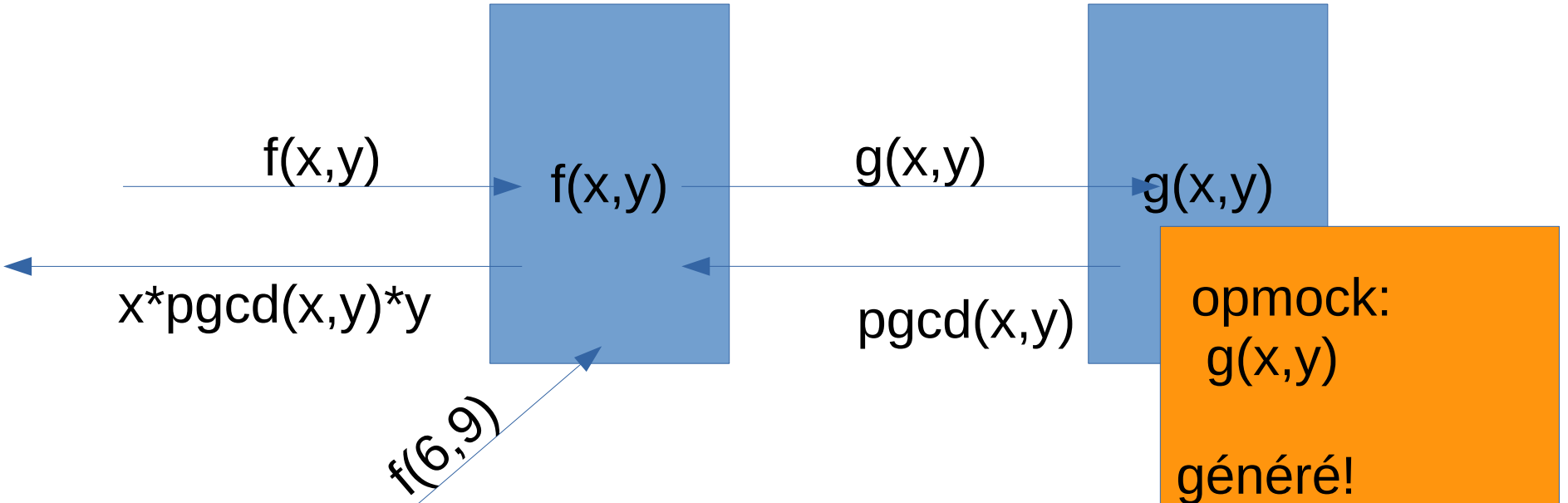
- Pour tester un module indépendamment d'un autre module



problèmes: 1 seul appel de g() possible
➔utilisation de variables globales
➔utilisation de générateur de mock (ex: opmock)

Tests inter module: faussaires

- Pour tester un module indépendamment d'un autre module



opmock:
g(x,y)

généré!

```
test_f(){
  g_ExpectAndReturn(6,9,3,comp_int,comp_int);
  assert(f(6,9)==162);
}
```

Les faussaires

- L’utilisation des faussaires (mock) en C n’est pas trivial, vous verrez que c’est beaucoup plus « naturel » dans d’autres langages supportant l’introspection (Java/python par ex…)
- Les faussaires sont aussi utiles pour paralléliser le développement : un développeur doit développer le module A et un autre le module B reposant sur A :
- Dans ce cas, on écrit une version faussaire du module A permettant d’entamer le développement du module B et parallèle du développement de A.

Tests d'implémentation

- Tout module `module.c` gérant un type `Module` doit contenir une méthode `moduleVerifie` destinée à vérifier la cohérence de l'instance de `Module` qui lui est passée en paramètre
  - Utile seulement s'il existe des conditions vérifiables
  - Sert à vérifier la cohérence des objets de type `Module` calculés par les méthodes du module
    - Assertion ou test en mode « debug »
    - Utilisation d'un drapeau « `MODULE_DEBUG` »
  - Mise à la disposition des tiers désireux d'étendre les fonctionnalités du module

Tests d'implémentation

```
int
matriceFaitQqch (
Matrice *     source,
Matrice *     destination,
int           paramètre)
{
...
#ifdef MATRICE_DEBUG                    /* Test de pré-condition     */
  if (matriceVerifie (source) != 0) { /* Test avec retour d'erreur */
    ...
    return (1);
  }
#endif /* MATRICE_DEBUG */
...
#ifdef MATRICE_DEBUG                      /* Test de post-condition */
  assert (matriceVerifie (destination) == 0);  /* Assertion (exit) */
#endif /* MATRICE_DEBUG */

  return (0);                                  /* On y est arrivé */
}
```

Tests d’implémentation

- Ces tests ne sont valables que pour des structures de données complexes
- Attention à ce que l’appel à ces tests n’induit pas d’effet de bord (modification de l’état de la structure).

Les tests d’intégration

- Les tests d’intégrations valident le fonctionnement global d’un module :
  - Cohérence des fonctions entre elles
  - Exécution de série d’opérations types
- Les tests d’intégrations peuvent également valider le fonctionnement de plusieurs modules entre eux
  - par exemple : couplage d’un module de graphe avec un module d’arbre

Tests d’acceptation / recette

- Ont pour but d'attester la validité du projet dans son ensemble
  - Mettent en œuvre des jeux de tests de taille réelle
  - Utilisés comme éléments contractuels pour la phase de recette du logiciel

Tests d’acceptation / recette

- Tout projet doit disposer d'un ou plusieurs fichiers contenant la procédure de tests de recette
  - Programmes ou scripts shell
- Procédure documentée dans le manuel de maintenance

Couverture, performance, non-régression

- Certaines caractéristiques peuvent être observées sur les tests (indépendamment de leurs types) :
  - Couverture : correspond au pourcentage du code effectivement exécuté par les tests. Permet de mettre en avant du code mort ou du code non testé
  - Non-Régression : correspond au fait qu'au fur et à mesure du développement, les tests passés continuent d'être validés
  - Performance : les tests permettent également de faire de la veille sur les ressources utilisées (CPU, mémoire…). Le suivi dans le temps permet de mettre en avant des dysfonctionnements

Couverture

- L'option –-coverage de gcc permet de générer des traces d’exécution

main.c:

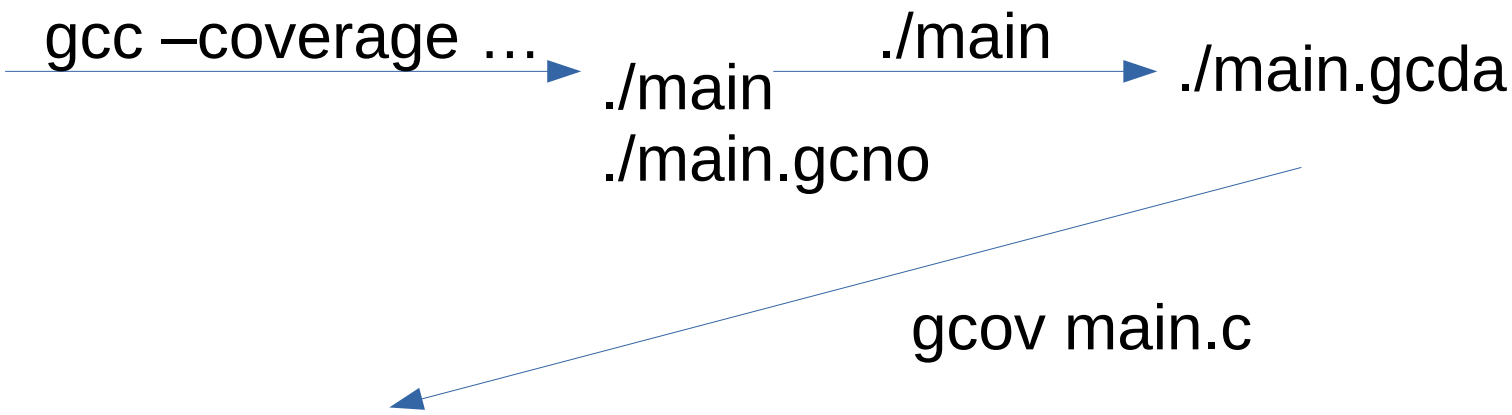
```
int main(int argc,char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```

Couverture

- L'option –-coverage de gcc permet de générer des traces d’exécution

main.c:

```
int main(int argc,char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```



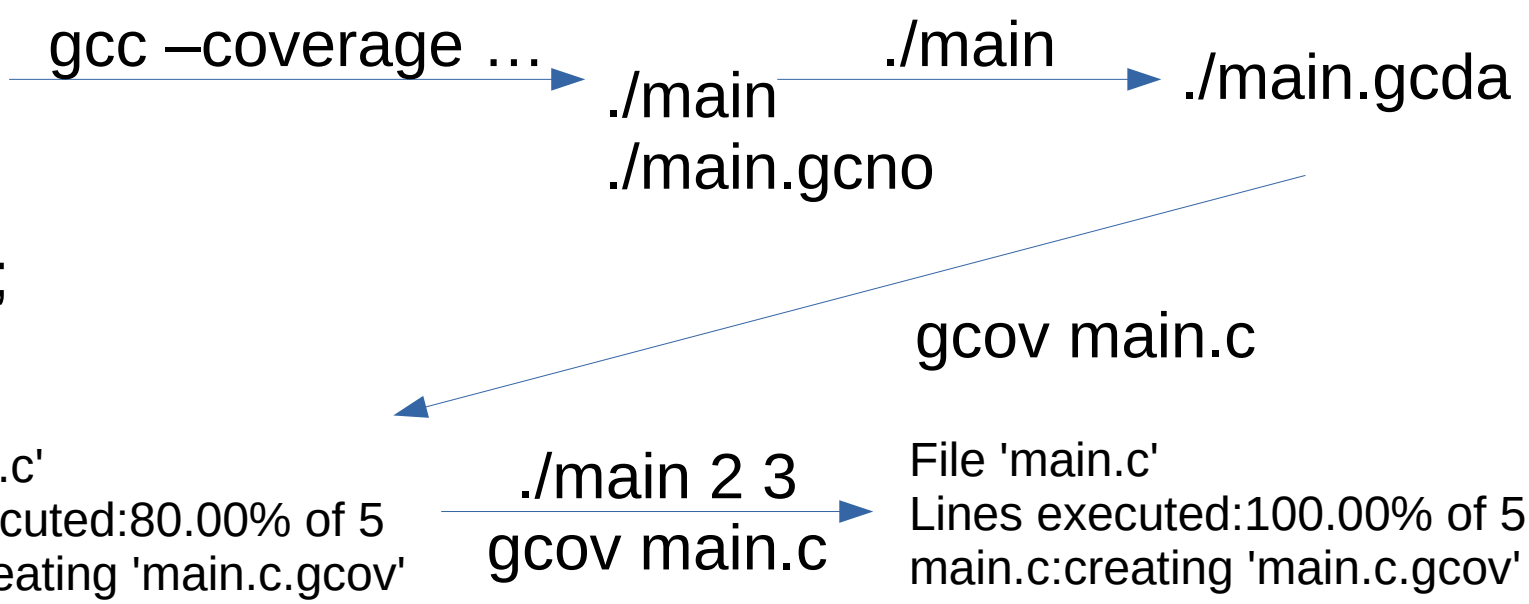

```
File 'main.c'
Lines executed:80.00% of 5
main.c:creating 'main.c.gcov'
```

Couverture

- L'option –-coverage de gcc permet de générer des traces d’exécution

main.c:

```
int main(int argc,char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```



gcc --coverage

- L’option --coverage est une « meta » option de gcc, elle implique les options :
  - -fprofile-arcs et -ftest-coverage lors de la compilation
  - -lgcov lors de l’édition de lien
- Il est possible de compléter avec l’option

  -fprofile-abs-path pour permettre à gcov de retrouver plus facilement les sources associées au binaire.

Couverture

- La couverture est une propriété très intéressante à observer
- Pour les tests, elle permet de mettre à jour du code non testé ou inutile
- Permet également lors de tests de recette ou de bêta de voir simplement quelles parties sont les plus utilisées, quelles erreurs n’arrivent « jamais » ou « souvent »...

Programmation C avancée

Performance

Analyse de performance (1)

- La performance d'un logiciel est évaluée en fonction des ressources nécessaires à l'obtention du résultat demandé
  - Temps mis
  - Ressources consommées (mémoire centrale, espace disque, bande passante réseau, ...)

Analyse de performance (2)

- Lorsque la performance est insuffisante, il faut déterminer l'origine de cette insuffisance afin d'essayer d'y porter remède
  - Problèmes d'implémentation, pas de spécification
  - Évaluation sur des cas réels, après maquettage
- Les remèdes pourront être :
  - Matériels : ajout ou remplacement de ressources
    - Plus le temps passe, plus les matériels sont puissants !
  - Logiciels : recodage de routines critiques ou bien modifications profondes de la structure du logiciel

Analyse de performance (3)

- Lorsque le manque de performance concerne le temps, les problèmes d'accès à la mémoire en sont les causes principales
  - En moyenne, près de la moitié des cycles consommés par les processeurs sont des cycles d'attente de la mémoire
- Il est donc essentiel de concevoir les algorithmes afin de minimiser les attentes mémoire

Principes de localité

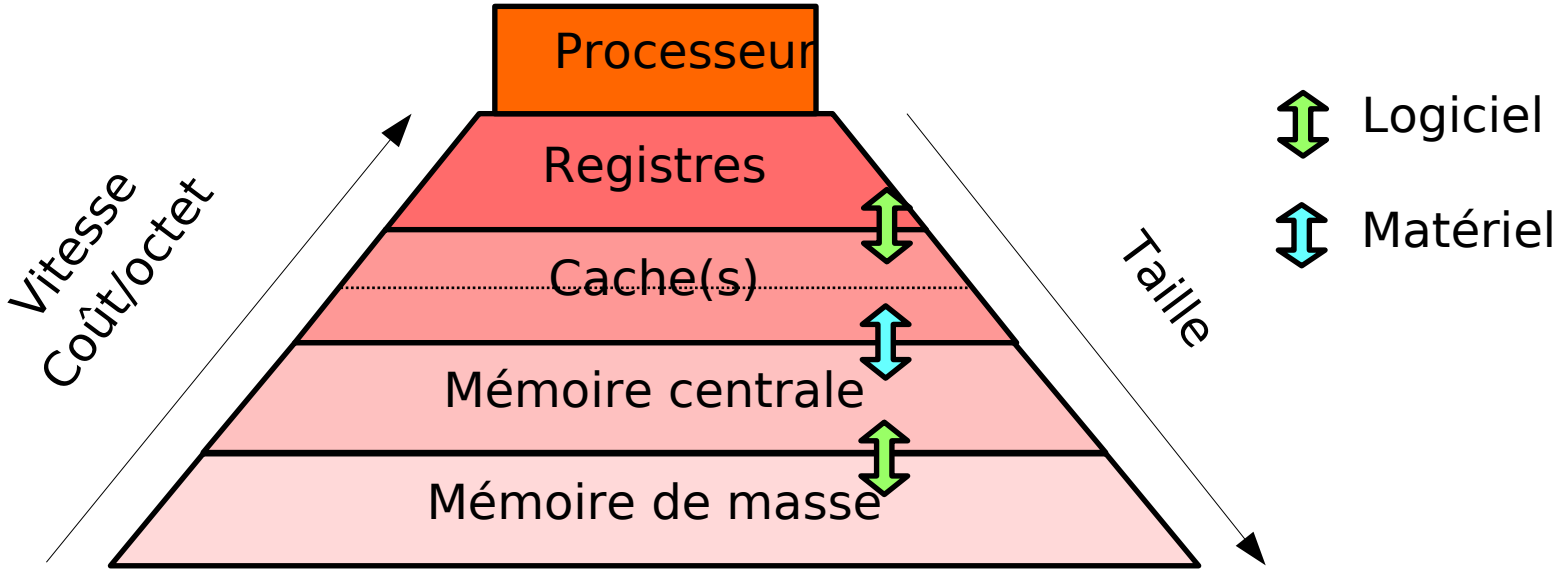
- Dans tout programme, il est possible de mettre en évidence un phénomène de localité des accès mémoire
  - Localité temporelle : plus une zone mémoire a été accédée récemment, plus sa probabilité de ré-accès est élevée
    - Lecture des instructions par le processeur (boucles), ...
  - Localité spatiale : plus une zone mémoire est proche de la dernière zone mémoire accédée, et plus la probabilité qu'elle soit à son tour accédée est importante
    - Parcours de tableaux, ...

Hiérarchie mémoire (1)

- Pour minimiser l'attente de la mémoire, il faut que les informations les plus fréquemment utilisées soient disponibles le plus rapidement possible
- On s'appuie sur les principes de localité pour mettre en place une hiérarchie de la mémoire
  - Mémoires rapides de faible capacité, proches du processeur
  - Mémoires de grande capacité aux temps d'accès plus longs, situées plus à distance

Hiérarchie mémoire (2)

- Met en œuvre les principes de localité
- Rend disponibles plus rapidement les données les plus fréquemment utilisées



- Les algorithmes doivent s'appuyer dessus !

Mémoires cache (1)

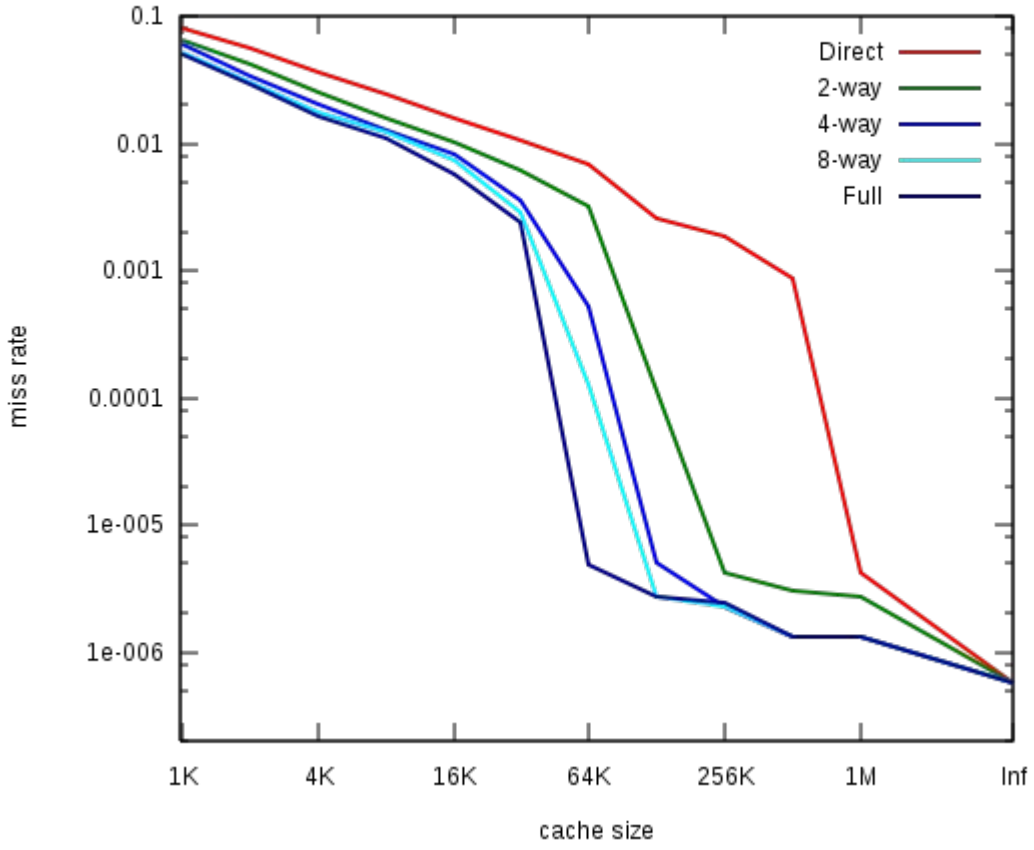
- Mémoire(s) rapide(s) située(s) entre le processeur et la mémoire centrale
- On trouve habituellement plusieurs niveaux de cache sur les architectures modernes
  - Cache de premier niveau
    - Sur le chip du processeur lui-même
    - Quelques dizaines de Ko seulement
    - Caches dissociés pour les instructions et les données
  - Caches de second/troisième niveau
    - Dans le boîtier du processeur ou à proximité
    - De quelques centaines de Ko à quelques Mo ; 2008: intel i7 8Mo; 2017: Intel Core i9 9900K 512K – 2Mo – 16Mo

      2020 : AMD Ryzen 9 5900X: 12 x (64k - 512K) - 64Mo

Mémoires cache (2)

- Les caches ne manipulent pas des octets ou des mots, mais des lignes (« *cache lines* »)
  - Lorsque le processeur demande un mot mémoire, qui n'est pas déjà présent dans le cache, le cache charge toute la ligne contenant le mot voulu à partir de la mémoire (de 8 à 512 octets)
  - Mise en œuvre du principe de localité spatiale
- Lorsque le cache est plein, la ligne la plus ancienne est effacée pour faire de la place à la nouvelle
  - Politique de remplacement de type LRU

Cache: Taille vs. Fautes



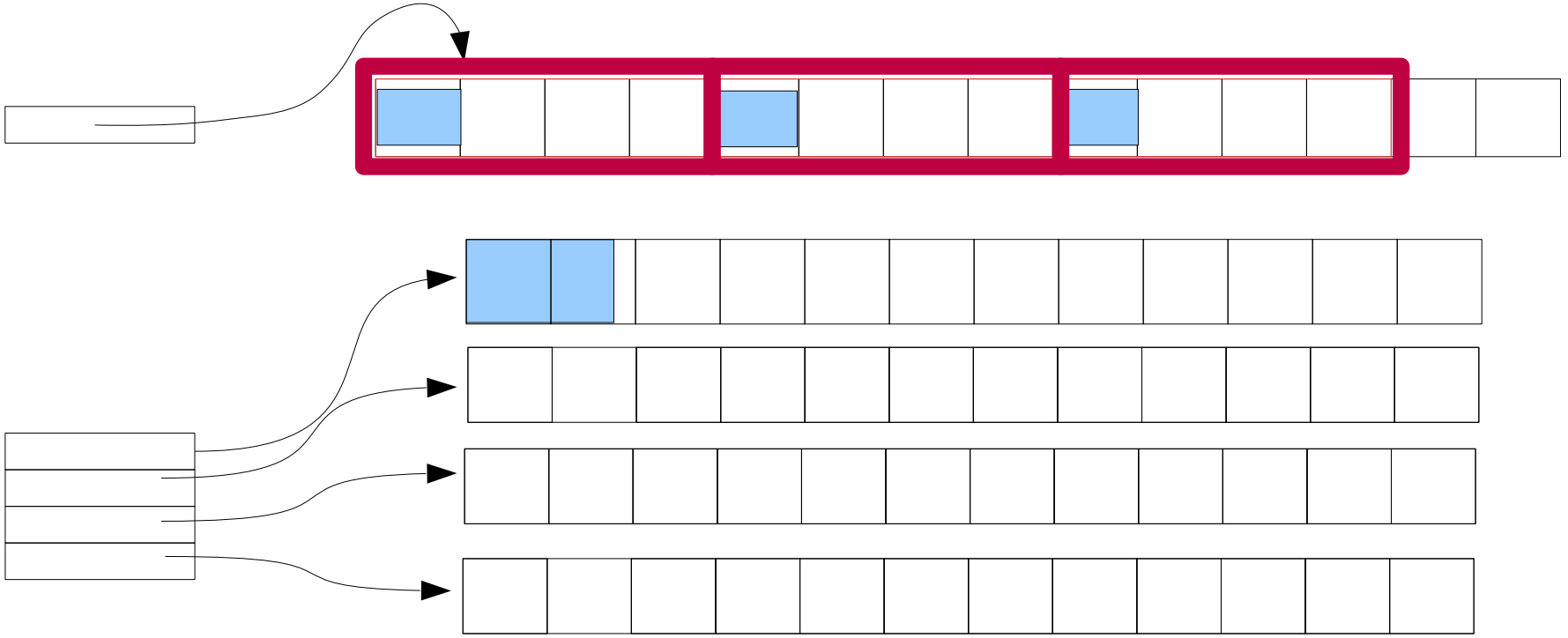
benchmark: SPEC CPU2000

Programmation « *cache friendly* »

- Pour qu'un algorithme soit efficace sur une architecture disposant d'une hiérarchie mémoire, il faut :
  - Effectuer la majorité des parcours de façon continue
    - Croissante ou décroissante
    - Maximise l'utilisation des données des lignes nouvellement chargées
  - Effectuer les parcours en écriture de façon continue
    - Les écritures sont plus chères car on doit répercuter les modifications en mémoire centrale

Programmation « *cache friendly* »

- Manipuler plusieurs tableaux de sous-structures plutôt qu'un unique tableau de structures de grandes tailles
  - Les lignes de cache ne stockent que des données utiles



Mesure de la performance

- La mesure de la performance d'un logiciel est complexe, car la prise de mesures doit perturber le moins possible le fonctionnement du système exécutant le logiciel
- Les mesures de performances peuvent être effectuées au niveau :
  - Du processeur
  - Du système d'exploitation
  - Du programme

Code profiling

- Le profilage de code est l'action d'instrumenter le code source afin d'en obtenir des mesures d'usage lors de l'exécution de jeux de tests
  - Ajout de routines de comptage de passages en différents points du code source
  - Activation de registres matériels du processeur dédiés à cette activité
    - Comptage du nombre de cycles consommés, des accès mémoire, des défauts de cache...

time

- Commande Unix classique donnant un résumé des ressources consommées par le programme passé en paramètre
  - Temps CPU consommé en mode utilisateur
  - Temps CPU consommé en mode système
  - Temps réel écoulé depuis le lancement
  - Taille mémoire utilisée par le code et les données
  - Nombre de défauts de page, etc...

```
% time gcc -O3 brol.c -o brol
0.05user 0.02system 0:00.18elapsed 43%CPU (0+0)k 0in+0out
(9major+2815minor)pagefaults 0swaps
```

Routines de mesure

- La routine `clock()`, appartenant à la bibliothèque standard, renvoie le temps CPU écoulé depuis une date d'origine
  - On calcule le temps consommé dans une routine par soustraction entre la valeur à la sortie et la valeur à l'entrée
- La routine `getrusage()` permet d'obtenir, au niveau du processus, les informations affichées par la commande **time**
  - Temps CPU utilisateur et système, mémoire, ...

getrusage

```
int getrusage(int who, struct rusage *usage);


 struct rusage {
        struct timeval ru_utime; /* user CPU time used */
        struct timeval ru_stime; /* system CPU time used */
        long   ru_maxrss;        /* maximum resident set size */
        long   ru_ixrss;         /* integral shared memory size */
        long   ru_idrss;         /* integral unshared data size */
        long   ru_isrss;         /* integral unshared stack size */
        long   ru_minflt;        /* page reclaims (soft page faults) */
        long   ru_majflt;        /* page faults (hard page faults) */
        long   ru_nswap;         /* swaps */
        long   ru_inblock;       /* block input operations */
        long   ru_oublock;       /* block output operations */
        long   ru_msgsnd;        /* IPC messages sent */
        long   ru_msgrcv;        /* IPC messages received */
        long   ru_nsignals;      /* signals received */
        long   ru_nvcsw;         /* voluntary context switches */
        long   ru_nivcsw;        /* involuntary context switches */
    };
```

Compteurs matériels (1)

- Les processeurs modernes disposent tous de circuits destinés à la mesure de performance
- Compteurs paramétrables d'événements internes au processeur, tels que nombres de cycles consommés, de lectures ou écritures, d'opérations à virgule flottante, de défauts de cache de premier ou deuxième niveau, ...
  - Deux registres compteurs de 40 bits disponibles sur le Pentium II, quatre de 48 bits sur l'Athlon

http://download.intel.com/design/PentiumII/manuals/24319202.pdf, table A.1

Compteurs matériels (2)

- Différentes bibliothèques permettent de sélectionner le type d'événements à compter et de lire la valeur des compteurs
  - Dépendantes du processeur et du système d'exploitation
- Tentatives d'offrir une interface unifiée pour plusieurs processeurs et systèmes
  - ***papi***, de l'Université du Tennessee

http://icl.cs.utk.edu/papi/
http://user.it.uu.se/~mikpe/linux/perfctr/

gprof (1)

- **gprof** est un outil de profilage, permettant de savoir dans quelles routines un programme passe le plus de temps, et quel est l'arbre d'appel du programme
- **gprof** analyse a posteriori les traces générées par l'exécution d'un programme, et produit un relevé statistique du temps passé dans chaque routine

gprof (2)

- Pour que l'exécution du programme génère des traces exploitables, il faut compiler avec l'option « `-pg` » de `gcc`
  - Réalise l'édition de liens avec les bibliothèques adaptées
- À la fin de l'exécution, les traces sont collectées dans le fichier « `gmon.out` »

```
% gcc -pg brol.c -o brol
% ./brol
% ls
brol
brol.c
gmon.out
```

gprof (3)

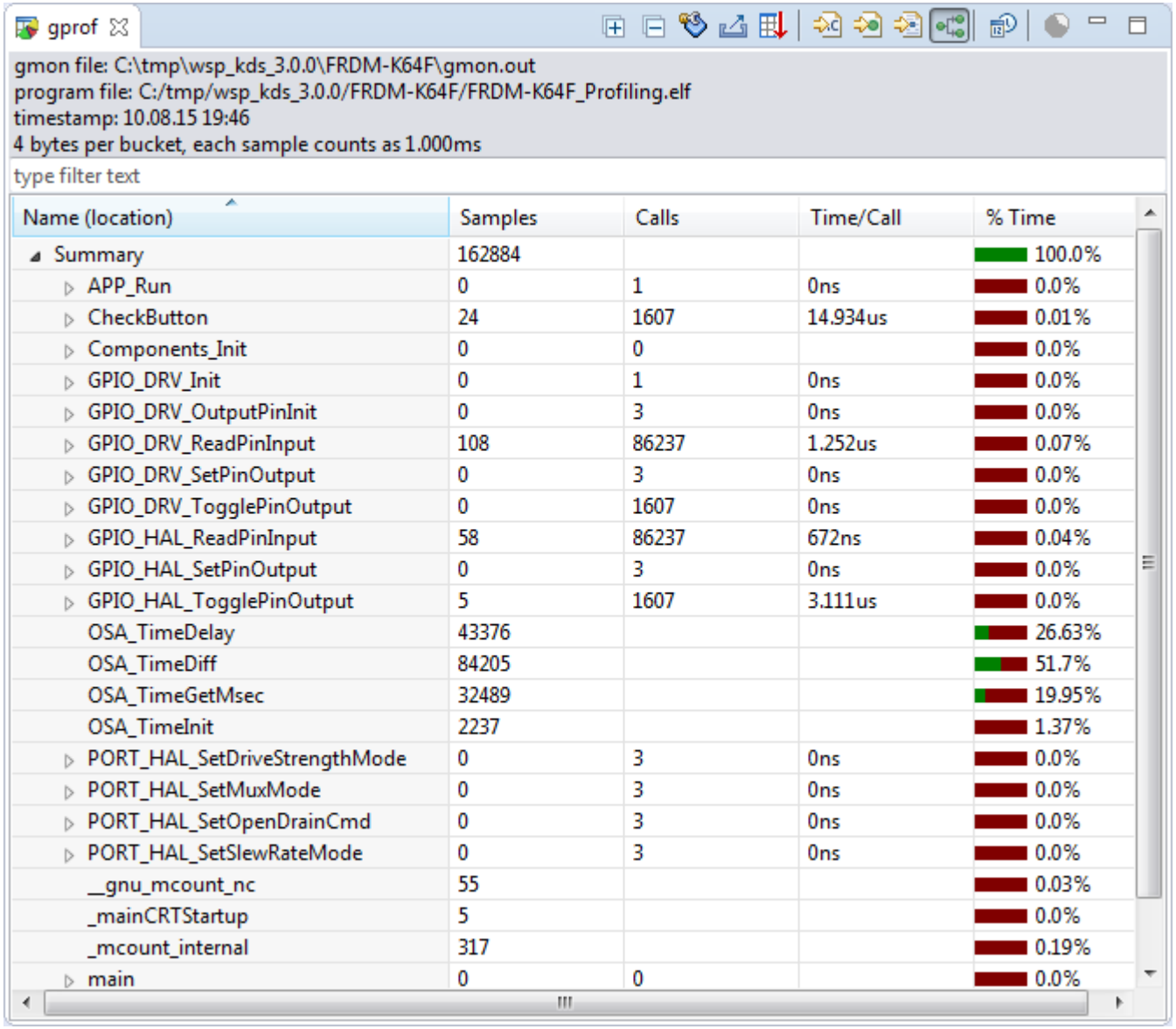
- Le rapport d'exécution est créé par la commande `gprof` proprement dite
  - Rapport d'exécution en format texte
- Classe les fonctions par ordre décroissant de temps consommé dans la fonction et dans les sous-fonctions qu'elle a appelées

```
% gprof brol gmon.out > rapport.txt
% more rapport.txt
```

gprof (4)

```
index % time    self  children    called     name
...
[6]     87.7    0.04   35.95    1099+5870    <cycle 2 as a whole> [6]
                0.03   19.27    4773+1647        vgraphSeparateSt [7]
                0.00    0.00    1098             vgraphSeparateMl[42]
-----------------------------------------------
                                 1647             vgraphSeparateSt [7]
                                 3674             vgraphSeparateMl2[9]
                0.04   35.95    1099/1099        hgraphOrderNd [5]
[7]     47.0    0.03   19.27    4773+1647    vgraphSeparateSt [7]
               15.06    1.72    2576/2576        vgraphSeparateFm [8]
                2.10    0.39    1098/1098        vgraphSeparateGg[14]
                0.00    0.00    1099/1099        stratTestEval [41]
                0.00    0.00    1098/1098        vgraphStoreInit [44]
                0.00    0.00    1098/1098        vgraphStoreSave [45]
                0.00    0.00    1098/1098        vgraphStoreExit [43]
                0.00    0.00     714/714         vgraphStoreUpdt [47]
                                 1098             vgraphSeparateMl[42]
                                 1647             vgraphSeparateSt [7]
```

gprof via un ide (eclipse)



Name (location)	Samples	Calls	Time/Call	% Time
Summary	162884			100.0%
APP_Run	0	1	0ns	0.0%
CheckButton	24	1607	14.934us	0.01%
Components_Init	0	0		0.0%
GPIO_DRV_Init	0	1	0ns	0.0%
GPIO_DRV_OutputPinInit	0	3	0ns	0.0%
GPIO_DRV_ReadPinInput	108	86237	1.252us	0.07%
GPIO_DRV_SetPinOutput	0	3	0ns	0.0%
GPIO_DRV_TogglePinOutput	0	1607	0ns	0.0%
GPIO_HAL_ReadPinInput	58	86237	672ns	0.04%
GPIO_HAL_SetPinOutput	0	3	0ns	0.0%
GPIO_HAL_TogglePinOutput	5	1607	3.111us	0.0%
OSA_TimeDelay	43376			26.63%
OSA_TimeDiff	84205			51.7%
OSA_TimeGetMsec	32489			19.95%
OSA_TimeInit	2237			1.37%
PORT_HAL_SetDriveStrengthMode	0	3	0ns	0.0%
PORT_HAL_SetMuxMode	0	3	0ns	0.0%
PORT_HAL_SetOpenDrainCmd	0	3	0ns	0.0%
PORT_HAL_SetSlewRateMode	0	3	0ns	0.0%
__gnu_mcount_nc	55			0.03%
_mainCRTStartup	5			0.0%
_mcount_internal	317			0.19%
main	0	0		0.0%

valgrind

- En plus de memcheck, valgrind dispose d’autres « outils » (valgrind est une plateforme)
  - cachegrind : analyse au niveau des caches processeur
  - callgrind : cachegrind + graphe d’appel, à utiliser avec kcachegrind (visualisateur)
  - massif : analyse de l’évolution du tas
  - dhat : analyse des accès au tas

kcachegrind