HAL
open science

Outils de caractérisation du comportement mémoire et d'estimation de la consommation énergétique

Maha Idrissi Aouad, Olivier Zendra

To cite this version:

Maha Idrissi Aouad, Olivier Zendra. Outils de caractérisation du comportement mémoire et d'estimation de la consommation énergétique. [Rapport Technique] RT-0350, INRIA. 2008, pp.53. inria-00256860v3

HAL Id: inria-00256860

https://inria.hal.science/inria-00256860v3

Submitted on 4 Mar 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Outils de caractérisation du comportement mémoire et d'estimation de la consommation énergétique

Maha IDRISSI AOUAD

Olivier ZENDRA

N° RT-0350

Février 2008

COM

Rapport technique

ISSN 0249-6399

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche NANCY - GRAND EST

Outils de caractérisation du comportement mémoire et d'estimation de la consommation énergétique

Maha Idrissi Aouad[1], Olivier Zendra[2]

Thème COM - Systèmes communicants
Équipe - Projet TRIO

Rapport de recherche n° RT-0350 – Février 2008 - 53 pages

Résumé : Ce rapport présente différents outils de caractérisation du comportement mémoire et d'estimation de la consommation énergétique ; il décrit toutes les étapes de compilation et d'installation de chacun de ces outils ainsi que la manière de les utiliser en détaillant leurs entrées/sorties et en expliquant leurs résultats obtenus.

Mots clés : comportement mémoire, estimation, consommation énergétique

[1] UHP - Nancy 1, INRIA Nancy - Grand Est – LORIA, TRIO – Maha.IdrissiAouad@loria.fr
[2] INRIA Nancy - Grand Est – LORIA, TRIO – Olivier.Zendra@inria.fr

615, rue du Jardin Botanique
B.P. 101
54602 Villers-lès-Nancy Cedex (France)
Téléphone : +33 03 83 59 30 00 — Télécopie : +33 03 83 27 83 19

Tools for caracterizing memory's behavior and estimating energy consumption

Abstract: This report presents different tools for characterizing memory's behavior and estimating energy consumption ; it describes all the compilation and installation steps of each tool, the way to use them and gives details on inputs/outputs and explanation of the results obtained.

Keywords: memory's behavior, estimation, energy consumption

RT n° RT-0350

Financement : Ce travail a été réalisé dans le cadre du projet ANR MORE (*Multicriteria Optimizations for Real-time Embedded systems*), financé par l'Agence Nationale de la Recherche sur le programme “Architectures du Futur” 2006 (numéro ANR-06-ARFU-002).

Financing: This work was realized for the MORE project (*Multicriteria Optimizations for Real-time Embedded systems*), financed by the ANR (Agence Nationale de la Recherche) in the program “Architectures du Futur” 2006 (number ANR-06-ARFU-002).

Table des matières

I. Outils de caractérisation mémoire....4
Gprof....5
Gcov....11
Valgrind....15
Insure++....20
TotalView....27
MemoryScape....32
II. Outils d'estimation de consommation énergétique....36
PTCMP....37
SimpleScalar....40
Wattch....46
CACTI....50

I. Outils de caractérisation mémoire

Gprof

Présentation :

Le *profiling* d'un programme permet d'identifier les endroits où celui-ci passe le plus de temps, mais également quelles sont les fonctions qui sont exécutées et combien de fois. Un *profiler* est un programme capable d'analyser un exécutable. Le résultat de l'analyse est appelé un *profile*. Le profiler standard dans le monde *GNU* est le programme *gprof*. Il fait partie du *package binutils* du projet *GNU* et est capable d'analyser des programmes écrits en *C*, *C++*, *Pascal* ou *Fortran77*.

Note :

- Avant de télécharger le profiler g*prof*, vérifiez qu'il n'est pas déjà installé sur votre machine ou s'il existe parmi vos paquetages, auquel cas installez-le directement en utilisant votre gestionnaire de paquetages.
- Fonctionne sur une *Red Hat Entreprise Linux* 64 bits Version *5*.

Compilation d'un programme pour le *profiling* :

Compilez votre programme avec l'option *-pg* qui permet d'altérer à la fois le comportement du compilateur et celui de l'éditeur de liens.

1. Exécutez la commande ci-dessous :
   (1) gcc `prog.c` -o `prog -`pg

Entrée :

1. Lancer le programme normalement avec ses paramètres éventuels :
   (1) `prog arg1 arg2`

Si le programme se termine normalement, soit par une terminaison normale de la fonction *main()*, soit par un appel à *exit(0)*, un fichier de données appelé *gmon.out* sera généré et sera placé dans le répertoire courant. Si ce fichier

existe déjà dans le répertoire, il sera écrasé. Un programme dont l'exécution a été interrompue (*CTRL-C* ou faute de segmentation) ne pourra pas être analysé.

2. Exécution de *gprof* :

(1) gprof `prog` gmon.out

Si l'on omet le nom de l'exécutable, le fichier *"a.out"* est utilisé par défaut. Si l'on omet le nom du fichier de données, le fichier *"gmon.out"* est utilisé par défaut.

Sortie :

Les résultats de l'analyse sont représentés par deux tableaux : le profil plat (*flat profile*) et le graphe d'appels (*call graph*).

Le profil plat (*flat profile*) :

Il montre combien de temps le programme passe dans chacune des fonctions et combien de fois chacune d'entre elles est appelée. Ce tableau permet donc de visualiser facilement quelles sont les fonctions qui consomment le plus de ressources. A titre d'exemple, le profil plat du *benchmark Bitcount* est donné en figure 1.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
16.73	0.01	0.01	1125000	8.92	8.92	AR_btbl_bitcount
16.73	0.02	0.01	1125000	8.92	8.92	BW_btbl_bitcount
16.73	0.03	0.01	1125000	8.92	8.92	bit_count
16.73	0.04	0.01	1125000	8.92	8.92	bit_shifter
16.73	0.05	0.01	1125000	8.92	8.92	ntbl_bitcount
16.73	0.06	0.01				main
0.00	0.06	0.00	1125000	0.00	0.00	bitcount
0.00	0.06	0.00	1125000	0.00	0.00	ntbl_bitcnt

Fig. 1 : Profil plat du *benchmark Bitcount*

Les fonctions sont classées par ordre décroissant de temps de calcul. Deux fonctions spéciales, *mcount* et *profil* sont des fonctions de *profiling* et apparaissent dans chaque profil plat. Leurs temps donnent une estimation de la surcharge de calcul induite par le *profiling*. Les différentes colonnes ont les significations suivantes :

- ***% time*** : pourcentage du temps d'exécution total passé à exécuter cette fonction.
- ***cumulative seconds*** : temps total cumulé que le processeur a passé à exécuter cette fonction, ajouté au temps passé à exécuter toutes les autres fonctions précédentes dans le tableau.
- ***self seconds*** : nombre de secondes effectivement passé à exécuter cette seule fonction.
- ***calls*** : nombre de fois où la fonction a été appelée. Si la fonction n'a jamais été appelée ou si son nombre d'appels n'a pas pu être déterminé, le champ est laissé vide.
- ***self ms/calls*** : estimation du nombre de millisecondes passées dans cette fonction par appel.
- ***total ms/calls*** : estimation du nombre de millisecondes passées dans cette fonction et dans ses enfants par appel.
- ***name*** : nom de la fonction.

Le graphe d'appels (*call graph*) :

Il montre, pour chacune des fonctions, quelles sont les fonctions qui l'appellent, quelles autres fonctions elle appelle et combien de fois elle le fait. Il présente également une estimation du temps passé dans chacune des sous-routines de chacune des fonctions. A titre d'exemple, le graphe d'appels du *benchmark Bitcount* est donné en figure 2.

```
.       .             Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 16.61% of 0.06 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01    0.05                 main [1]
                0.01    0.00 1125000/1125000     bit_shifter [5]
                0.01    0.00 1125000/1125000     AR_btbl_bitcount [2]
                0.01    0.00 1125000/1125000     BW_btbl_bitcount [3]
                0.01    0.00 1125000/1125000     ntbl_bitcount [6]
                0.01    0.00 1125000/1125000     bit_count [4]
                0.00    0.00 1125000/1125000     ntbl_bitcnt [8]
                0.00    0.00 1125000/1125000     bitcount [7]
-----------------------------------------------
                0.01    0.00 1125000/1125000     main [1]
[2]     16.7    0.01    0.00 1125000         AR_btbl_bitcount [2]
-----------------------------------------------
                0.01    0.00 1125000/1125000     main [1]
[3]     16.7    0.01    0.00 1125000         BW_btbl_bitcount [3]
-----------------------------------------------
                0.01    0.00 1125000/1125000     main [1]
[4]     16.7    0.01    0.00 1125000         bit_count [4]
-----------------------------------------------
                0.01    0.00 1125000/1125000     main [1]
[5]     16.7    0.01    0.00 1125000         bit_shifter [5]
-----------------------------------------------
                0.01    0.00 1125000/1125000     main [1]
[6]     16.7    0.01    0.00 1125000         ntbl_bitcount [6]
-----------------------------------------------
                0.00    0.00 1125000/1125000     main [1]
[7]      0.0    0.00    0.00 1125000         bitcount [7]
-----------------------------------------------
                0.00    0.00 1125000/1125000     main [1]
[8]      0.0    0.00    0.00 1125000         ntbl_bitcnt [8]
-----------------------------------------------
```

Fig. 2 : Graphe d'appels du *benchmark Bitcount*

Les lignes composées de tirets séparent le tableau en différentes entrées composées d'une ou plusieurs lignes. Dans chacune des entrées, la ligne primaire est celle qui débute par un nombre entre crochets. La fin de cette ligne spécifie la fonction concernée. Les lignes précédentes décrivent les fonctions appelantes de cette fonction et les lignes suivantes décrivent les fonctions appelées par cette fonction. Ces fonctions sont appelées les fonctions *enfants*. Toutes les entrées sont classées par temps passé dans la fonction et ses enfants.

1. La ligne primaire :

La ligne primaire est la ligne qui désigne la fonction étudiée et donne les statistiques globales correspondantes. Les différentes colonnes ont les significations suivantes :

- ***index*** : les entrées sont numérotées par des entiers consécutifs. Chaque fonction possède donc un index qui apparaît au début de la ligne primaire. Chacune des références à une fonction, en tant que fonction appelante ou fonction enfant, présente son index ainsi que son nom.
- ***% time*** : pourcentage du temps total passé dans la fonction, en incluant le temps passé dans chacune des fonctions enfants.
- ***self*** : temps total passé dans cette fonction.
- ***Children*** : temps total passé dans les appels aux fonctions enfants.
- ***called*** : nombre total de fois où la fonction a été appelée. Si la fonction est appelée récursivement, il apparaît deux nombres séparés par un +. Le premier nombre désigne le nombre d'appels non-récursifs et le second le nombre d'appels récursifs.
- ***name*** : nom de la fonction. L'index est répété juste après ce nom. Si la fonction fait partie d'un cycle de récursion, le numéro du cycle est inscrit entre le nom de la fonction et son index.

2. Les lignes des fonctions parentes (fonctions appelantes de la fonction) :

Une entrée de fonction présente une ligne pour chacune de ses fonctions parentes, c'est-à-dire pour chacune des fonctions qui l'ont appelée. Les colonnes correspondent à celles présentes pour la ligne primaire, mais certaines de leurs significations ne sont pas les mêmes :

- ***self*** : estimation du temps passé dans la fonction enfant considérée lorsqu'elle a été appelée par sa fonction parente.
- ***children*** : estimation du temps passé dans chacune des fonctions enfants de la fonction enfant considérée lorsqu'elle a été appelée par sa fonction parente.
- ***called*** : le premier nombre indique le nombre de fois où la fonction enfant considérée a été appelée par sa fonction parente, tandis que le second nombre indique le nombre d'appels non-récursifs total.
- ***name*** et ***index*** : Nom de la fonction parente et son index.

3. Les lignes des fonctions enfant :

Une entrée de fonction présente une ligne pour chacune de ses fonctions enfants. De la même manière que précédemment, les colonnes correspondent à celles présentes pour la ligne primaire, mais certaines de leurs significations ne sont pas les mêmes :

- ***self*** : estimation du temps passé dans la fonction enfant considérée lorsqu'elle a été appelée par sa fonction parente.
- ***children*** : Estimation du temps passé dans chacune des fonctions enfants de la fonction enfant considérée lorsqu'elle a été appelée par sa fonction parente.
- ***called*** : le premier nombre indique le nombre de fois où la fonction enfant considérée a été appelée par sa fonction parente, tandis que le second nombre indique le nombre d'appels non-récursifs total.

Bibliographie :

http://www.gnu.org/software/binutils/manual/gprof-2.9.1/text/gprof.txt

http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html

Gcov

Présentation :

Gcov permet de savoir précisément quelles lignes de codes sont effectivement exécutées, combien de fois une ligne de code donnée est exécutée, et en conjonction avec g*prof* combien de temps est nécessaire au déroulement d'une partie du code. Une fois ces informations connues, il est possible de regarder si l'on peut optimiser les parties les plus coûteuses.

Note :

- Avant de télécharger le profiler g*cov*, vérifiez qu'il n'est pas déjà installé sur votre machine ou s'il existe parmi vos paquetages, auquel cas installez-le directement en utilisant votre gestionnaire de paquetages.
- Il est souhaitable de placer peu d'expressions sur la même ligne, car l'information fournie par *gcov* est donnée ligne par ligne. Les macros complexes cacheront forcément de l'information, aussi est-il préférable d'utiliser des fonctions (et éventuellement de les qualifier d'*inline*) pour pouvoir acquérir un maximum d'informations.
- Attention, *gcov* ne fonctionne qu'avec les programmes compilés avec *gcc*.
- Fonctionne sur une *Red Hat Entreprise Linux* 64 bits Version *5*.

Compilation d'un programme pour le *profiling* :

Compilez les sources de votre programme avec les options *-fprofile-arcs -ftest-coverage*. Ces options sont nécessaires afin de demander au compilateur d'ajouter de l'information pour *gcov*, typiquement du code servant à générer les fichiers pour *gcov* lors de l'exécution.

1. Exécutez la commande ci-dessous :

(1) gcc - fprofile-arcs -ftest-coverage -o `source source.c`

La compilation génère un fichier du type *"source.gcno"* dans le répertoire des sources.

Entrée :

1. Lancez le programme normalement avec ses paramètres éventuels :
   (1) `source arg1 arg2`

L'exécution génère des informations de *profile*. Pour chaque fichier compilé avec l'option *-fprofile-arcs*, un fichier du type *"source.gcda"* est émis dans le répertoire des sources. Il faut ensuite lancer gcov à proprement parler, en lui donnant comme argument la liste des fichiers sources ayant été compilé avec *-fprofile-arcs*.

2. Exécution de *gcov* :
   (1) gcov [-b] [-l] [-f] [-o répertoire] `source.c`

- ***-b*** : demande d'écrire les fréquences de prise de branchement et émet un résumé sur la sortie standard.
- ***-l*** : demande la création de nom à rallonge pour les fichiers d'entête, si du code exécutable s'y trouve.
- ***-f*** : émet des statistiques par fonction et non simplement pour le fichier.
- ***-o répertoire*** : indique le répertoire dans lequel se trouve les fichiers résultant de l'exécution du programme.

Gcov ré-écrit le programme source en préfixant chaque ligne par le nombre de fois ou elle a été exécutée. Les résultats sont clairement dépendant des données, ce qui signifie que des exécutions avec d'autres données donneront des résultats différents.

Pour un branchement, *gcov* écrit un pourcentage représentant le nombre de fois où la branche a été prise divisé par le nombre de fois où le test a été exécuté. Si la branche n'a jamais été exécutée, le message *"never executed"* est émis.

Pour un appel de fonction, un pourcentage indiquant le nombre de fois que l'on est revenu de l'appel sur le nombre d'appels est affiché. C'est souvent 100%, sauf si l'on utilise des *exit* ou des *longjmp*.

Les traces d'exécutions (contenues dans les fichiers *.gcda*) s'accumulent : cela permet de mener des campagnes de statistiques sur un grand nombre de données, afin d'obtenir des informations plus fiables. Il suffit d'effacer le

fichier *.gcda* pour remettre les compteurs à zéro. De même, si *gcov* se rend compte que le *.gcda* est obsolète par rapport au programme en cours d'exécution, il l'écrase.

Sortie :

Gcov génère un fichier *"source.c.gcov"* à partir d'un fichier *source.c* dans lequel chaque ligne est précédée du nombre de fois où elle a été exécutée. Avec l'aide de *gprof*, on peut déterminer le temps absolu pris par un bout de code spécifique qui n'est pas une fonction. A titre d'exemple, la sortie de *gcov* sur le programme *"hello.c"*.

1. Les commandes :
   (1) gcc -fprofile-arcs -ftest-coverage -o `hello hello.c`
   (2) `hello`
   (3) gcov -b -f `hello.c`

Function 'main'
Lignes exécutées: 100.00% de 3
Pas de branchement
Appels exécutés: 100.00% de 1

File 'hello.c'
Lignes exécutées: 100.00% de 3
Pas de branchement
Appels exécutés: 100.00% de 1
hello.c:creating 'hello.c.gcov'

2. Le résultat dans *"hello.c.gcov"* :

```
-:    0:Source:hello.c
-:    0:Graph:hello.gcno
-:    0:Data:hello.gcda
-:    0:Runs:1
-:    0:Programs:1
-:    1:#include <stdio.h>
-:    2:
```

```
        -:    3: int main(void)
function main called 1 returned 100% blocks executed 100%
        1:    4: {
        1:    5:     printf("Hello world!\n");
appel   0 a retourné 100%
        -:    6:
        1:    7:     return 0;
        -:    8: }
```

Bibliographie :

http://www.linuxcommand.org/man_pages/gcov1.html

www.foo.be/docs-free/cours-prog-unix-c.ps

Valgrind

Présentation :

Valgrind est un outil d'aide pour trouver des problèmes de gestion de mémoire dans les programmes. Lorsqu'un programme est exécuté sous la supervision de *Valgrind*, toutes les lectures et écritures de mémoire sont vérifiées et les appels à *malloc/new/free/delete* sont interceptés. *Valgrind* peut donc détecter de nombreux problèmes qui seraient autrement très durs à trouver ou à diagnostiquer. *Valgrind* est un logiciel composé de plusieurs outils (chacun ayant un rôle particulier) comme expliqué ci-dessous :

- *Memcheck* : outil de contrôle mémoire,
- *Addrcheck* : similaire à M*emcheck* mais avec une granularité moindre pour une exécution plus rapide,
- *Cachegrind* : affiche des statistiques concernant le cache du processeur simulé,
- *Massif* : outil de profil mémoire.

Notre étude sera basée sur les résultats de l'outil de profil mémoire *Massif*. Celui-ci donne des informations sur :

- la taille de la pile,
- la taille d'administration de tas,
- la taille du tas.

Note :

- Avant de télécharger le logiciel *Valgrind*, vérifiez qu'il n'est pas déjà installé sur votre machine ou s'il existe parmi vos paquetages, auquel cas installez-le directement en utilisant votre gestionnaire de paquetages.
- Fonctionne sur une *Red Hat Entreprise Linux* 64 bits Version *5*.

Compilation et installation :

Pour installer à partir du *Subversion repository* :

1. Téléchargez le code source de *Valgrind* à partir du SVN en suivant les instructions sur http://www.valgrind.org/downloads/repository.html

2. Exécutez les commandes ci-dessous :
   (1) cd `valgrind`
   (2) ./autogen.sh

3. Continuez en suivant les instructions.

Pour installer à partir d'une distribution *tar.bz2* :

4. Exécutez les commandes ci-dessous :
   (1) ./configure --prefix=`/là/où/vous/voulez/l'installer`
   (2) make
   (3) make install /* nécessite éventuellement les privilèges *root* */

5. Vérification. Tapez la commande :
   (1) valgrind `ls -l`

N.B : Ne déplacez pas l'installation de *Valgrind* à un endroit différent de celui que vous avez spécifié via la commande *--prefix* au moment de la compilation. Cela risque de causer des problèmes, spécialement lorsque *Valgrind* manipule les appels de *fork*/*exec*.

Entrée :

Compilez votre programme avec l'option *-g* afin d'inclure les informations de *debugging*.

1. Exécution de l'outil *massif* de *Valgrind* :
   (1) valgrind --tool=massif `prog arg1 arg2`

Sortie :

Les résultats fournis par m*assif* sont : un résumé, un graphique et un fichier texte.

Le résumé :

Il est imprimé en sortie standard. Le résumé de la commande *ls -l* est donné en figure 3.

```
==3567== Total spacetime:   10,732,791 ms.B
==3567== heap:              86.2%
==3567== heap admin:         1.4%
==3567== stack(s):          12.3%
```

Fig. 3 : Résumé de l'analyse de l'outil *Massif*

Toutes les mesures sont données en valeur espace-temps, c'est à dire l'espace (en *octets*) multiplié par le temps (en *ms*). Il faut noter que *Massif* diminue la vitesse d'exécution d'un programme, seules les valeurs relatives sont donc intéressantes. L'exemple présenté figure 3, mesure toutes les parties de la mémoire :

- *heap* : nombre d'octets alloués dans le tas, via *malloc()*, *new*, *new[]*.
- *heap admin* : chaque bloc alloué demande des données d'administration qui laissent des traces des blocs. Cette valeur n'est qu'une estimation par *massif* qui utilise par défaut 8 octets par bloc alloué.
- *stack(s)* : l'espace utilisé par la pile.

Le graphique (*massif.pid.ps*) :

Il représente l'évolution de la mémoire utilisée au cours de l'exécution du programme, autrement dit, le profil de l'utilisation mémoire. A titre d'exemple, le profil mémoire de la commande *ls -l* est donné en figure 4. Celui-ci peut être décomposé en bandes dont chacune représente une ligne de programme contenant une allocation dans le tas. Notons que *pid* dans le nom du graphique représente le numéro d'identification du programme analysé.

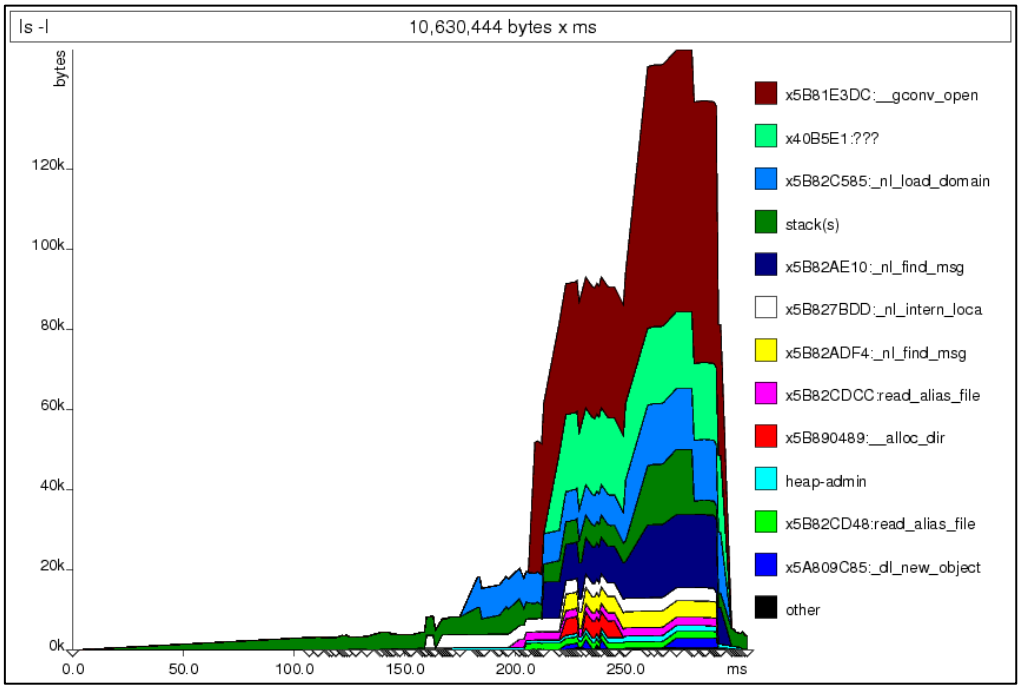

Fig. 4 : Profil mémoire de la commande *ls-l*

Les triangles sur l'axe des abscisses représentent chaque point de mesure. Ces points apparaissent donc lorsqu'une allocation ou une dés-allocation a lieu ce qui pose des problèmes quant à la validité de la taille maximale de la pile. En effet, les variations de la pile peuvent être très importantes entre deux échantillonnages. Notons que, si les variations du tas sont importantes, alors les prises de mesures seront plus rapprochées et la précision sur la taille de la pile sera plus grande.

De plus, le temps d'exécution pour deux mesures successives de l'outil ne sera pas toujours le même à cause du délai aléatoire de l'*OS*, il pourra donc y avoir des différences entre les graphes de plusieurs mesures. Il apparaîtra donc une dilatation ou un rétrécissement en temps du graphe, mais les informations sur la mémoire seront inchangées.

Le fichier texte (*massif.pid.txt*) :

Le fichier texte, quant à lui, précise les allocations en pile et donne plus d'informations sur les allocations en précisant les fonctions qui les ont appelées. Notons que *pid* dans le nom du fichier texte représente le numéro d'identification du programme analysé.

Bibliographie :

www.hlrs.de/organization/amt/services/tools/debugger/valgrind/doc/valgrind_manual_3.2.1.pdf

www.kdab.net/Presentations/LinuxForum2004-Software-Development/paper/paper-10179-en.pdf

http://www.valgrind.org/

http://www.valgrind.org/downloads/repository.html

Insure++

Présentation :

Insure++ est un produit de la société *Parasoft* permettant d'optimiser le développement de programmes *C* et *C++* sur les architectures *Linux*, *Windows* et *Solaris*. Son utilisation est payante. *Insure++* est disponible au *LORIA* grâce à un système à jetons, pour *Linux* et *Solaris*. *Insure++* dispose de plusieurs outils parmi lesquels *Inuse* et *Chaperon*.

Chaperon est un outil qui permet de tester les accès à la mémoire. Il fonctionne comme un *debugger* en mode observateur : il n'est donc pas nécessaire de recompiler les *benchmarks* avec des options spéciales.

Inuse est un outil graphique conçu pour aider les développeurs à éviter les problèmes liés à la mémoire en affichant l'évolution de la mémoire allouée par un programme donné. Il permet de :

- Visualiser les fuites mémoire.
- Voir combien de mémoire votre programme utilise en réponse à des événements utilisateur particuliers.
- Vérifier l'utilisation totale de la mémoire de votre programme.
- Détecter la fragmentation mémoire pour savoir si des stratégies d'allocation différentes pourraient améliorer les performances.

Note :

- Licence au *LORIA* valable pour une *Linux* 32 bits.

- Fonctionne sur une *Red Hat Entreprise Linux* 64 bits Version *5* (possibilité d'installer et de compiler la version 32 bits d'*Insure++* en mode de compatibilité. Pour plus de détails, voir l'annexe).

Compilation et installation :

1. Sélectionnez la version du logiciel *Insure++* correspondant à votre architecture et téléchargez la (http://sedre.loria.fr/LogSci/insure).

2. Placez l'archive *insure++ "ins++.linux2.tar.gz"* dans le répertoire où vous désirez l'installer.

3. Décompressez l'archive *insure++* avec la commande ci-dessous :
   (1) tar xvfz `ins++.linux2.tar.gz`

4. Lancez le script :
   (1) ./install
   N'effacez **surtout pas** le fichier *"ins++.linux2.tar.gz"*.

5. Répondez aux questions jusqu'à ce que vous arriviez à la question suivante :
   Do you use Parasoft License Server? (yes/no) [no]

   S'il ne vous pose pas cette question c'est qu'il a détecté un serveur de licence et l'installation est finie, dans le cas contraire :
   (1) Répondez **yes**
   (2) Puis entrez le nom du serveur de licence : **insure.inria.fr**
   (3) Enfin entrez le numéro de port : **2002**

6. Effacez les fichiers et répertoires suivants : ins++.linux2.tar.gz, install et .install.linux2 à l'aide de la commande :
   (1) rm -rf `ins++.linux2.tar.gz install .install.linux2`

7. Vérifiez le fichier *".psrc"*. Il doit contenir au moins les lignes suivantes :
   LicenseServer.host insure.inria.fr
   LicenseServer.port 2002

8. Modifiez votre *PATH* pour y inclure les binaires de *Insure* contenus dans **répertoire installation insure++/**bin.linux2

Entrée :

1. Compilation et édition de liens pour *Inuse* :
   (1) gcc `-c hello.c`
   (2) insure `-o hello hello.o`

2. Permettre l'affichage de l'exécution d'*Inuse* :

(1) `insure++.inuse on`

Assurez-vous que le fichier *".psrc"* contient cette ligne, sinon ajoutez y la.

3. Exécution du programme :

(1) `hello`

Exécutez votre programme normalement avec ses arguments éventuels. *Inuse* sera démarré automatiquement et à la fin de l'exécution du programme, il générera certains graphiques.

Sortie :

Inuse génère différents rapports sous forme de graphiques, comme expliqué ci-dessous :

- **Historique du tas (*Heap History*)** : affiche la quantité de mémoire allouée en tas et en processus utilisateur en fonction du temps.
- **Fréquence de bloc *(Block Frequency)*** : affiche un histogramme montrant le nombre de blocs de chaque taille qui ont été alloués.
- **Organisation du tas (*Heap Layout*)** : montre la disposition de la mémoire dans les blocs alloués dynamiquement en incluant les espaces libres entre eux. On peut utiliser ce rapport pour voir la fragmentation ainsi que les fuites mémoire (*memory leaks*).
- **Organisation mémoire au cours du temps (*Time Layout*)** : montre le séquencement des blocs alloués.
- **Résumé d'utilisation (*Usage Summary*)** : montre combien de fois chaque type d'appel mémoire a été fait. Il montre également la taille courante du tas ainsi que la quantité de mémoire activement utilisée.
- **Comparaison d'utilisation (*Usage Comparison*)** : compare graphiquement la mémoire à partir de différentes exécutions d'un programme ou parmi les exécutions de différents programmes.
- **Requête (*Query*)** : permet de lister les blocs mémoire alloués par le programme en fonction de leurs *id*, leurs tailles, et/ou leurs traces de pile.

Voici à titre d'exemple, les rapports de l'historique du tas ainsi que du résumé d'utilisation générés par *Inuse* pour le *benchmark gsm*. HWM veut dire *High Water Mark*, il donne l'utilisation maximale de la mémoire durant tout le temps d'exécution du programme.

```
** TCA log data will be merged with tca.log **
************************* INSURE SUMMARY ************************ v7.0.8 **
*   Program      : toast                                                 *
*   Arguments    : -fps -c data/large.au                                 *
*   Directory    : /home/Maha/Desktop/MORE/Bench_linux/Insure/gsm        *
*   Compiled on  : Not available                                         *
*   Run on       : Nov 12, 2007  16:38:30                                *
*   Elapsed time : 00:00:11                                              *
*   Malloc HWM   : 1014 bytes                                            *
************************************************************************

No leaks were found.
```

Fig. 5 : Sortie du terminal du *benchmark Gsm*

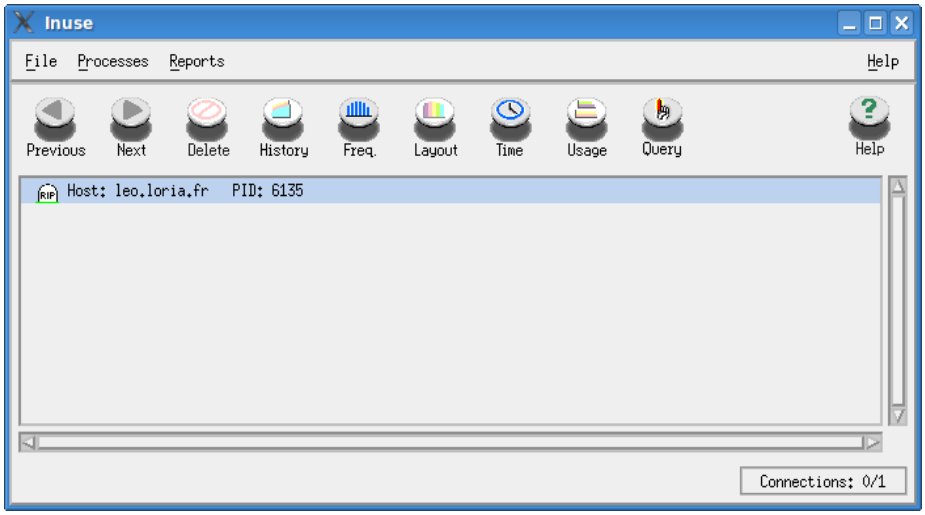

Fig. 6 : Sortie *Inuse* du *benchmark Gsm*

Une fois que la fenêtre présentée en figure 6 est affichée, vous pouvez cliquez sur les différents boutons de la barre d'outils afin de générer le rapport souhaité.

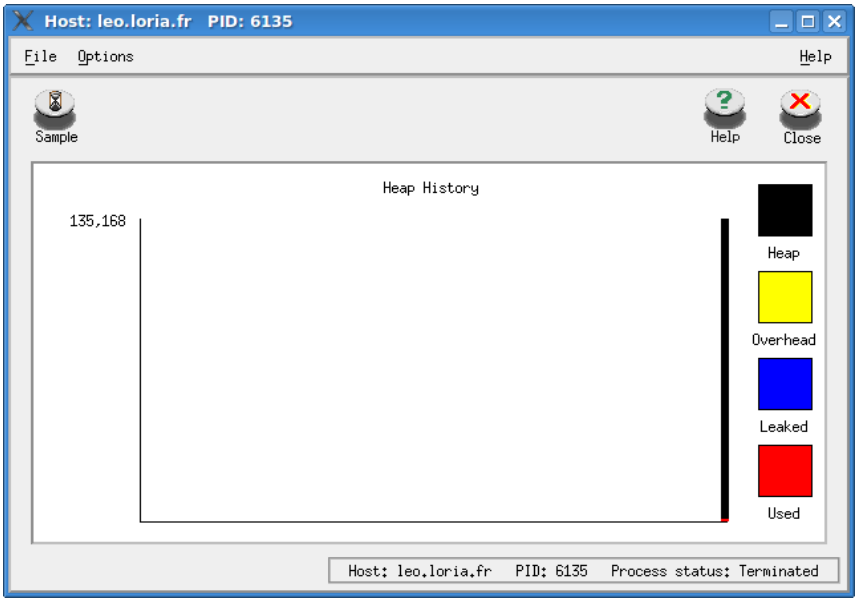

Fig. 7 : Rapport de l'historique du tas du *benchmark Gsm*

Les codes de couleurs d'*Inuse* sont comme suit :

- Noir : mémoire totale allouée au tas.
- Bleu : quantité de fuite mémoire.
- Rouge : mémoire allouée par le programme.
- Jaune : surcoût (*overhead)* associé à la mémoire.
- Vert : espace libre disponible pour l'allocation.

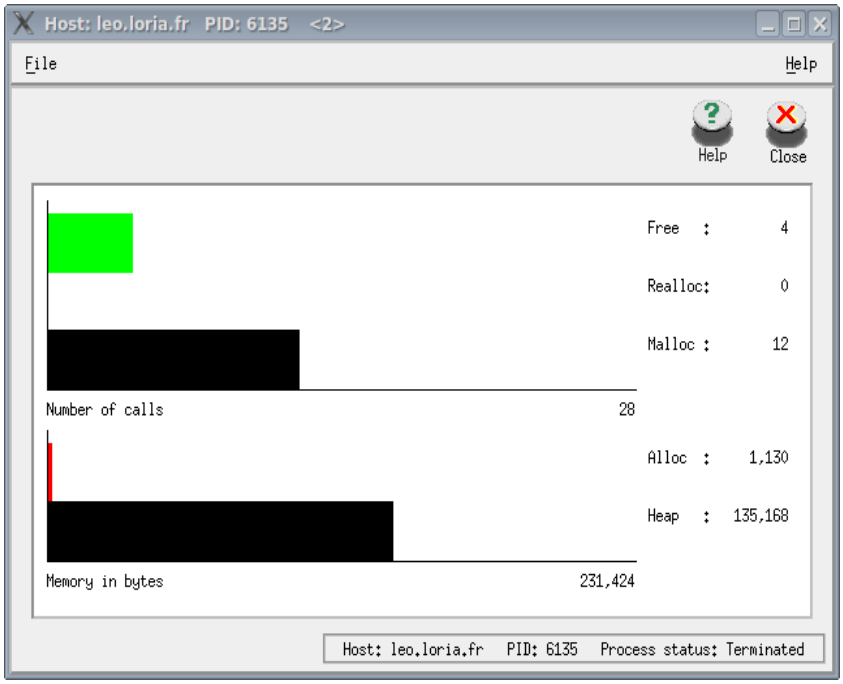

Fig. 8 : Rapport du résumé d'utilisation du *benchmark Gsm*

- *"Alloc"* représente le nombre total de bytes alloués dynamiquement par le *benchmark Gsm*.
- *"Heap"* représente le nombre total de bytes actuellement alloués par le système au tas du *benchmark Gsm*.
- Le nombre donné par *"Numbers of calls"* et *"Memory in bytes"* représente la valeur extrême de l'axe des abscisses pour chaque graphique.

Annexe : Installation et compilation de la version 32 bits en mode de compatibilité

(1) Lancer une fois le script *install* (pour avoir tous les autres scripts extraits dans *.install/*)

(2) Fichier : install
Ligne : début du script
Action : changer EXTRACTED= en EXTRACTED=1

(3) Fichier : .install/ins++install.pl

Ligne : &Arch_Verify("linux2");
Action : commenter la ligne qui devient : #&Arch_Verify("linux2");

(4) Relancer le script *install*.

(5) Continuer l'installation jusqu'à ce qu'elle se bloque puis passer à l'étape suivante.

(6) Fichier caché : .psrc
Action : rajouter les lignes suivantes :
(1) linsure++.compiler gcc
(2) LicenseServer.host insure.inria.fr
(3) LicenseServer.port 2002
(4) insure++.inuse on
(5) insure++.summarize leaks outstanding

(7) Compilation et édition de liens pour *Inuse* (forcer la compilation en mode 32 bits avec -m32) :
(1) gcc -m32 -c `hello.c`
(2) insure -m32 -o `hello hello.o`

(8) Aller dans le répertoire *lib.linux2* et créer les liens suivants :
(1) ln -s libinsure.so libinsure_32.so
(2) ln -s libinsure_st.so libinsure_st_32.so
(3) ln -s libinsure_mt.so libinsure_mt_32.so
(4) ln -s libinsure_mt_nptl.so libinsure_mt_nptl_32.so

Bibliographie :

http://sedre.loria.fr/LogSci/insure/

http://www.loria.fr/service/dst/logiciels/insure

Il existe une documentation avec l'installation de *Insure++* dans, **répertoire installation insure++**/manuals/, qui contient une documentation en *html* ***"index.htm"*** et le manuel en *PDF*.

TotalView

Présentation :

TotalView est un *debugger* (graphique et ligne de commande), particulièrement orienté vers le déboguage de code distribué (mais utilisable pour tout type de code).

Note :

- Fonctionne sur une *Red Hat Entreprise Linux* 64 bits Version *5*.
- 32 licences pour linux sont disponibles au LORIA.

Compilation et installation :

Les binaires sont disponibles sur le site de l'éditeur : http://www.totalviewtech.-com/download.htm
Récupérez la version adaptée (linux 32 ou 64 bits) et suivez les instructions de l'éditeur pour l'installation. En résumé :

1. Décompressez l'archive :
   (1) tar xvf `totalview.8.3.0-0-linux-x86-64.tar`

2. Lancez le script d'installation et suivez les instructions.
   (1) cd `totalview.8.3.0-0/`
   (2) ./Install

3. Le script demande confirmation pour l'installation de plusieurs composants :
   (1) Installez au minimum "`totalview-common.tar.Z`" et "`totalview-linux-x86-64.tar.Z`" (ou "`totalview-linux-x86.tar.Z`").
   Le serveur de licence n'est pas nécessaire sur un poste client.

4. TotalView a besoin d'un fichier de licence, par défaut "$PREFIX/toolworks/flexlm-10.8.0-3/license.dat" où "$PREFIX" est le répertoire d'installation.

Créez un fichier de licence sous le nom "license.dat" contenant les 2 lignes suivantes :

(1) SERVER moscou.loria.fr ANY 7127
(2) USE_SERVER

5. Placez ce fichier de licence dans "$PREFIX/toolworks/flexlm-10.8.0-3/" (créez ce répertoire s'il n'existe pas).
6. *TotalView* peut maintenant être utilisé, lancer par exemple :

(1) $PREFIX/toolworks/totalview.8.3.0-0/bin/totalview

7. Modifiez votre *PATH* pour inclure les binaires de *TotalView*.

Entrée :

Compilez votre programme avec l'option *-g* afin d'inclure les informations de *debugging*.

1. Exécution de l'outil *TotalView* :

```
(1)   totalview prog arg1 arg2
```

On peut aussi ne saisir que `totalview` en ligne de commande et saisir les arguments (dans l'onglet *Arguments)*, les noms des fichiers d'entrées et de sorties (dans l'onglet *Standard I/O*) directement au niveau de l'interface graphique de *TotalView* comme montré par la figure 9.

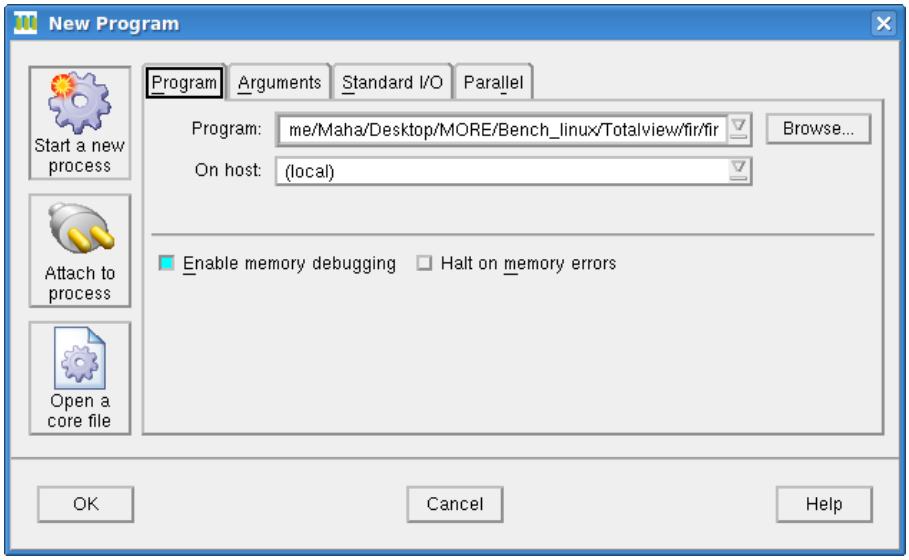

Fig. 9 : Interface de saisie des arguments du programme à exécuter

N.B : Il faut s'assurer que la case *Enable memory debugging* est cochée avant de lancer l'exécution de *TotalView* afin de pouvoir récupérer les informations pour la mémoire.

Sortie :

TotalView génère une interface de commande, montrée en figure 10, qui permet d'effectuer un certains nombre d'actions comme sélectionner un *break-point*, lancer l'exécution, l'arrêter, etc. En ce qui nous concerne, on se focalise sur l'outil *Memory debugging* auquel on peut accéder par le menu *Tools*.

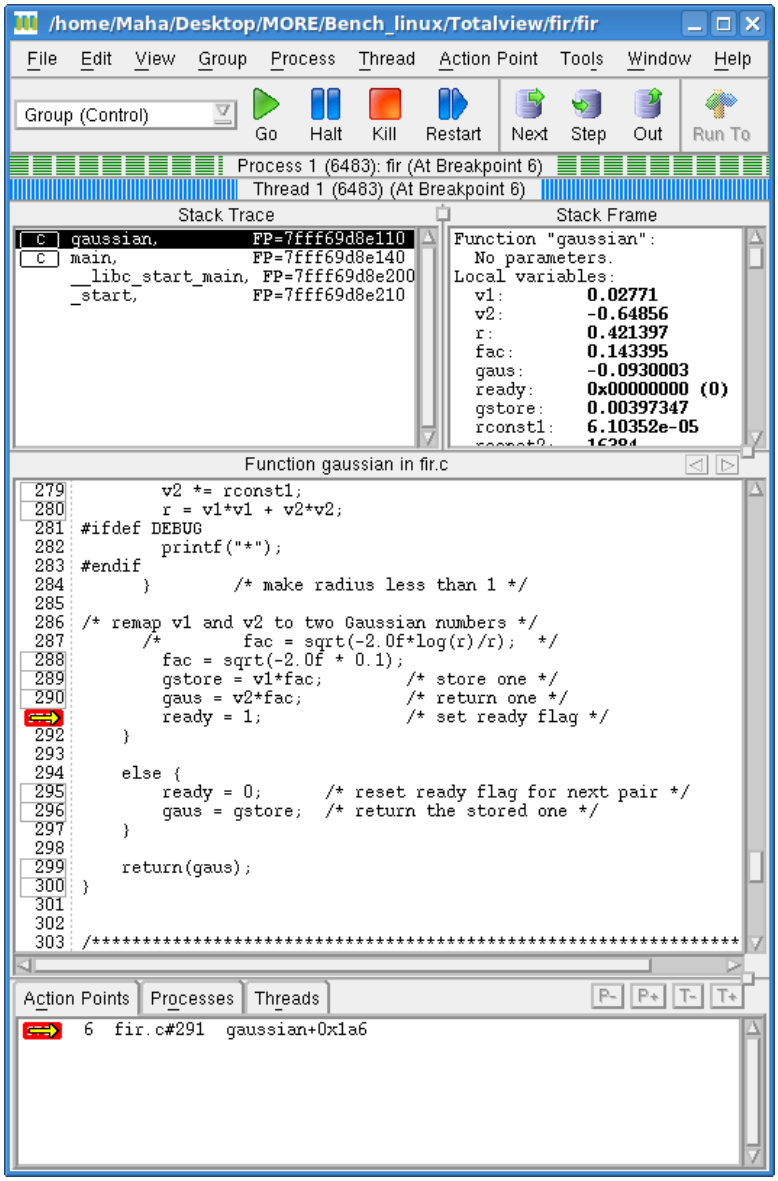

Fig. 10 : Interface de commande de *TotalView*

Cet outil est présenté en figure 11. Une fois l'exécution du programme lancée, on peut générer différents rapports pour la mémoire : détection de fuite mémoire, état du tas, utilisation de la mémoire ou comparaison de la mémoire entre différentes exécutions d'un même programme. Il suffit de choisir l'onglet correspondant au rapport que l'on souhaite générer et de cliquer sur le bouton *Generate View* en bas de la fenêtre.

Il est possible de choisir le type de vue que l'on souhaite avoir en la sélectionnant dans la liste déroulante parmi *Chart View*, *Library View* et *Process View*. Dans le cas d'une vue graphique de l'utilisation de la mémoire, comme montré en figure 11, on peut également choisir le type de graphique souhaité (barres, courbes ou camembert). On peut sauvegarder certains rapports sous format texte ou *html* en cliquant sur le bouton (symbolisé par une disquette) juste à côté de *Generate View*.

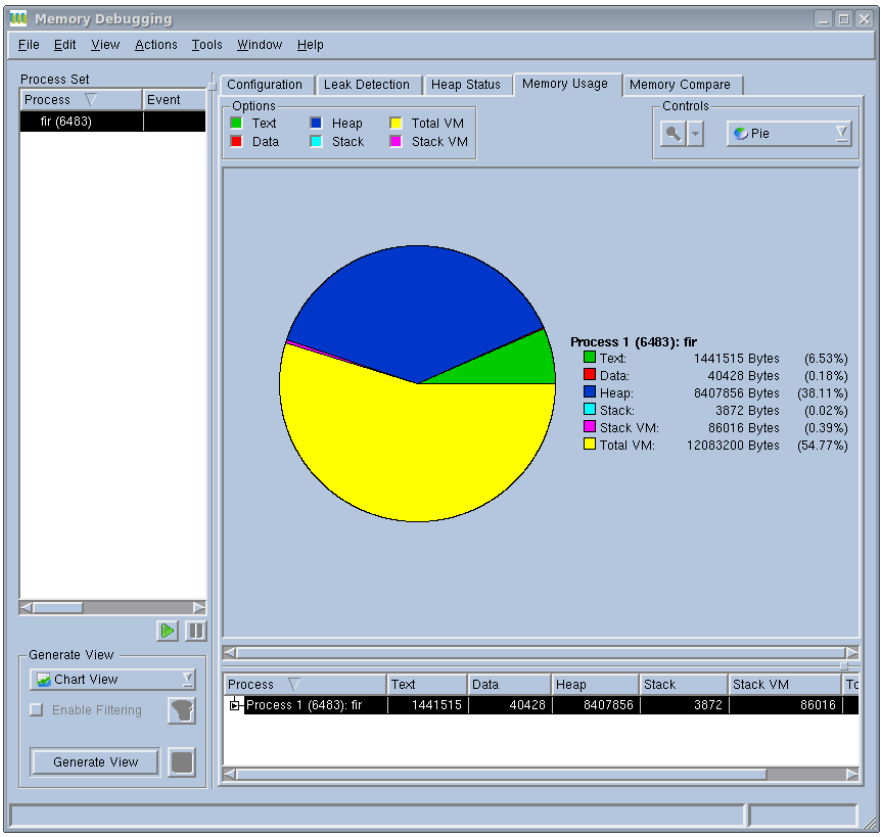

Fig. 11 : Rapport graphique d'utilisation de la mémoire avec *TotalView*

Bibliographie :

http://www.loria.fr/service/dst/logiciels/totalview

http://www.totalviewtech.com/Documentation/index.php

http://dps.c-s.fr/srchtml/produits/totalview.html

MemoryScape

Présentation :

MemoryScape est un débogeur mémoire qui permet d'identifier, d'étudier et de résoudre les problèmes mémoire des codes C/C++, Fortran, multi-processus et multi-threads. Basé sur la technologie de débogage de *TotalView Technologies*, *MemoryScape* possède une interface graphique intuitive qui simplifie le travail de débogage de la mémoire utilisée par vos codes.

Il permet de chercher les fuites de mémoire et de connaître le taux d'utilisation de la mémoire, de la pile d'allocation, lors de l'exécution d'une application.

MemoryScape permet de sauvegarder l'état de la pile et de le comparer à l'état courant de la mémoire ou à de précédents états. Il permet aussi de créer des rapports qui montrent les différences entre des états de la mémoire permettant de s'assurer que les modifications apportées au programme ont résolu le problème. Grâce à ces sauvegardes des états de la mémoire, on peut tracer l'évolution de l'usage de la mémoire.

Note :

- Licence non disponible au LORIA.

- La version d'évaluation du débogeur mémoire *MemoryScape* a été testée afin de déterminer si l'achat de l'outil était nécessaire. Il en ressort que *MemoryScape* fournit les mêmes rapports que ceux obtenus en utilisant *TotalView*, l'interface graphique est cependant mieux travaillée. Les seuls ajouts de *MemoryScape* par rapport à *TotalView*, sont :

- la possibilité d'utiliser le mode *scripting* afin de générer tous les rapports en évitant de passer par les interfaces graphiques,
- un suivi de l'évolution de l'utilisation du tas grâce au graphique généré par le *Heap usage monitor* comme montré en figure 12. Ce graphique ne peut être ni enregistré, ni modifié.

Compilation et installation :

Les binaires sont disponibles sur le site de l'éditeur : http://www.totalviewtech.-com/download.htm
Récupérez la version adaptée (linux 32 ou 64 bits) et suivez les instructions de l'éditeur pour l'installation. Les étapes d'installations étant les mêmes que pour TotalView.

N.B : Modifiez votre *PATH* pour inclure les binaires de *MemoryScape*.

Entrée :

Compilez votre programme avec l'option *-g* afin d'inclure les informations de *debugging*.

1. Exécution de l'outil *MemoryScape* :
   (1) `memscape prog arg1 arg2`

On peut aussi ne saisir que `memscape` en ligne de commande et saisir les différents arguments directement via l'interface graphique de *MemoryScape*.

Sortie :

MemoryScape génère une interface de commande, montrée en figure 12, qui permet d'effectuer un certains nombre d'actions comme générer un rapport, lancer l'exécution, l'arrêter, etc.

Comme expliqué auparavant, le graphique *Heap usage monitor* donne un suivi de l'évolution de l'utilisation du tas.

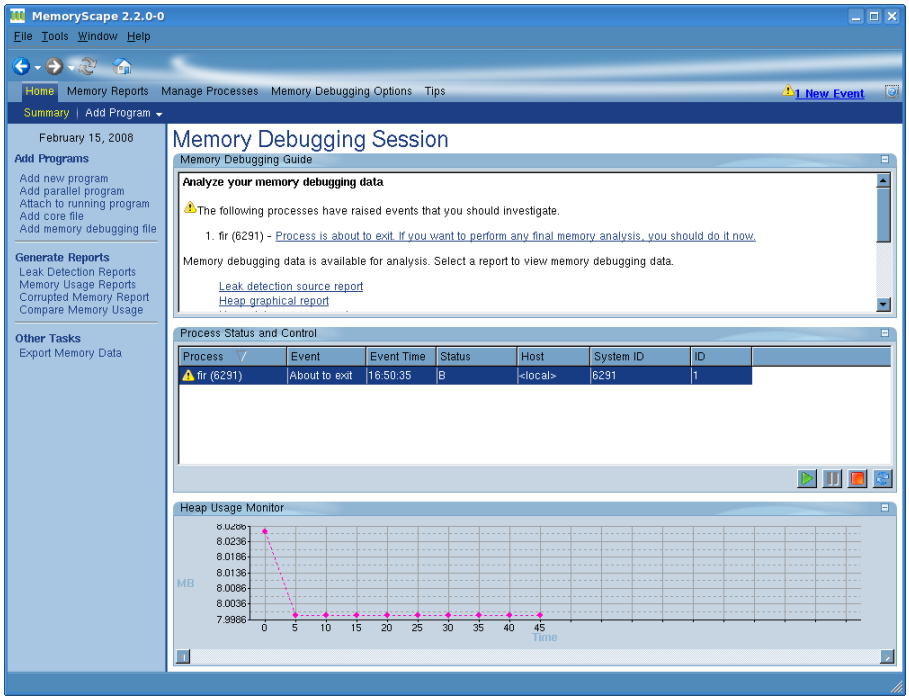

Fig. 12 : Interface de commande de *MemoryScape*

Une fois l'exécution du programme lancée, on peut générer différents rapports pour la mémoire : détection de fuite mémoire, état du tas, utilisation de la mémoire ou comparaison de la mémoire entre différentes exécutions d'un même programme. Exactement comme pour TotalView.

A titre d'exemple, la figure 13 donne le rapport graphique de l'utilisation de la mémoire pour le *benchmark Fir*. On remarque alors que ce graphique est identique à celui généré par *TotalView* pour le même *benchmark* en figure 11.

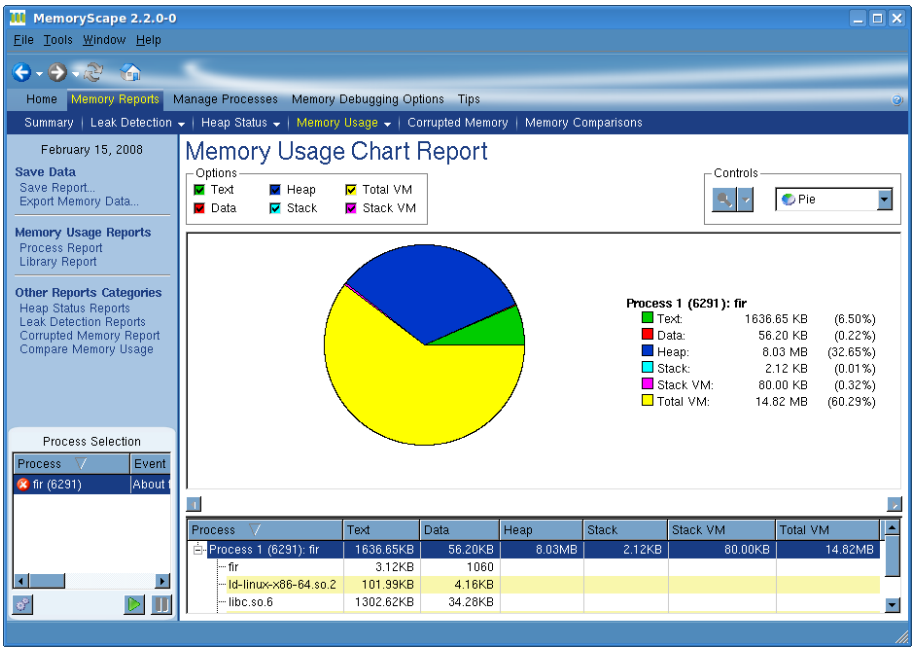

Fig. 13 : Rapport graphique d'utilisation de la mémoire avec *MemoryScape*

Bibliographie :

http://www.totalviewtech.com/Documentation/index.php

http://dps.c-s.fr/srchtml/produits/memeoryscape.html

II. Outils d'estimation de consommation énergétique

PTCMP

Présentation :

Parallel Turandot CMP (*PTCMP*), anciennement appelé *Power Timer*, est un simulateur qui donne des informations sur l'énergie et la température consommées par cycle et par coeur. Il contient un certain nombre d'améliorations par rapport à *Turandot* et à *Turandot CMP*, parmi lesquelles :

- simulation parallèle multicoeur.
- *parallel target benchmark modeling*.
- scripts de compilation propres, empreinte mémoire, assertions et avertissements.
- compatibilité multi plate-forme.

Note :

- Fonctionne sur une *Red Hat Entreprise Linux* 64 bits Version *5*.
- Outil propriétaire d'*IBM*.
- Documentation non disponible.

Compilation et installation :

1. Téléchargez le logiciel *PTCMP* (http://www.princeton.edu/~jdonald/research/ptcmp).

2. Placez l'archive "*ptcmp.tgz"* dans le répertoire où vous désirez l'installer.

3. Décompressez l'archive *PTCMP* avec la commande ci-dessous :
   (1) tar xvfz `ptcmp.tgz`

4. Exécutez :
   (1) ./configure
   (2) make

5. Vérification, dans le répertoire ptcmp/Sources/turandot/src vous devriez trouver l'exécutable :

gpffturandot

Entrée :

Les seuls formats connus par *PTCMP* sont ce qu'on appelle des *traces* (fichiers portant l'extension "*.fF"* ou *".tt6e"*). Il existe des *traces* pré-générées pour les *benchmarks* des suites *SPEC CPU2000*, *MediaBench*, *SPLASH2 et BioPerf*. Toutefois, dans notre cas nous avons besoin de générer nous-même ces *traces*. Pour ce faire, il faut utiliser un outil spécifique parmi les trois ci-dessous qui existent :

- *Aria* est un outil binaire de traduction plus rapide que la simulation. Aria nécessite une exécution directe des instructions *PowerPC* et s'exécute seulement sur les machines *AIX*. Un autre inconvénient d'*Aria* est qu'il ne fonctionne que sur les machines *AIX* basées sur les binaires *xcoff*. Les sorties générées par Aria ont l'extension *.fF*.
- *Amber* est plus lent qu'*Aria*. Il est disponible seulement sur les machines *PowerPC Macintosh*. *Amber* est un produit de niveau industriel moderne. Il compte moins de *bugs* qu'*Aria* et est écrit "pour les développeurs plutôt que pour les chercheurs". *Amber* peut *tracer* des *benchmarks multithread*. Comme *Aria*, *Amber* enregistre les instructions à partir de l'exécutable et des libraires partagées mais ne peut pas enregistrer le code de l'*OS*.
- *SimpleScalar-PPC* (*SS-PPC*) génère des *traces* avec l'extension *.tt6e* à partir des binaires *xcoff* comme *Aria*. Même si les binaires ont toujours besoin d'être compilés sur une machine *AIX*, le *traçage* peut être fait et refait sur des systèmes *Linux x86*. En revanche, *SimpleScalar-PPC* n'est pas très robuste et peut seulement exécuter quelques *benchmarks* de la suite *SPEC CPU2000*. Une autre restriction est que *SS-PPC* peut seulement utiliser les binaires compilés de façon statique et donc ne *trace* pas les bibliothèques partagées comme le fait *Aria*. Actuellement, *SS-PPC* ne supporte pas les formats binaires *ELF* ou *Mach-O* compilés dans d'autres systèmes. Il n'a aucune implémentation pour *SMT*, *chip multi-*

processing, instructions *PPC64*, modélisation de puissance, modélisation de température et *thread communication*.

1. Exécutez :

(1) ./gpffturandot -t `trace.fF` -stderr `test.out` -tempout `temp.out` \
-dumpstatus `dump.out` -tempinit `./power-models/init/1c1s/template.d1.init`

Sortie :

Bibliographie :

http://www.princeton.edu/~jdonald/research/ptcmp/

La principale documentation est le fichier *README*. Vous trouverez une copie de ce fichier ainsi que d'autres documents dans le sous-répertoire *doc* après avoir décompressé l'archive *"ptcmp.tgz"*.

Pour une meilleure compréhension de la structure de *PTCMP*, vous pouvez vous référez à cet article :
[James06] James Donald and Margaret Martonosi, "An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation." In Computer Architecture Letters, vol. 5, August 2006.

SimpleScalar

Présentation :

SimpleScalar est un simulateur de processeur au niveau architectural. C'est un programme gratuit et libre ce qui permet de le modifier pour créer d'autres simulateurs (*Wattch*, *SimplePower*, ...). En réalité, *SimpleScalar* est composé de plusieurs simulateurs présentés ci-dessous.

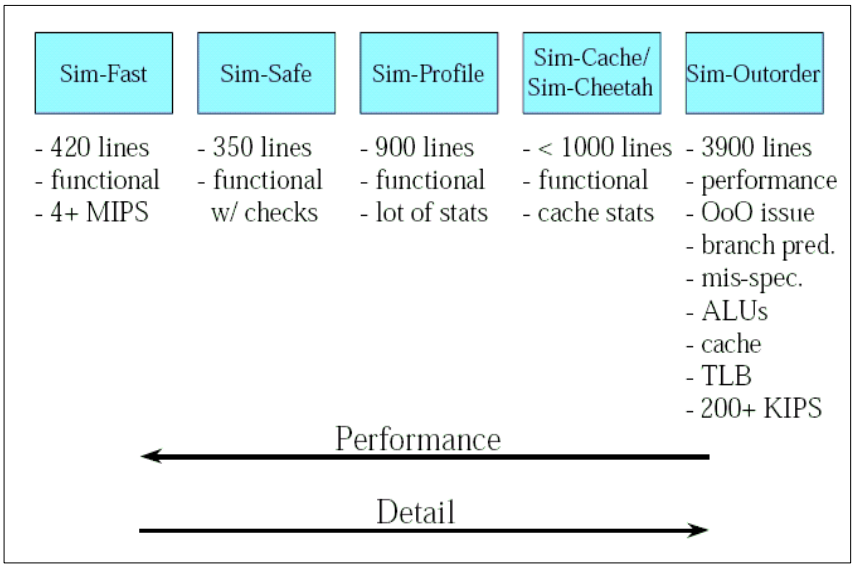

Fig. 14 : Les simulateurs de *SimpleScalar* d'après [Austin97]

Note :

- Problèmes d'installation dûs à l'ancienneté des librairies utilisées par *SimpleScalar*.

- Ne supporte pas les architectures 64 bits.

- N'est plus maintenu par les concepteurs.

- Pour fonctionner correctement, nécessite une vieille distribution *Linux* 32 bits.

- Si vous ne disposez pas des exécutables contenus dans /home/user/simplescalar/bin, et des exécutables crées lors de la compilation des *"binutils"*, vous devez utiliser une vieille distribution *Linux* 32 bits afin que la compilation se déroule correctement.

- Fonctionne sur une *Red Hat Entreprise Linux* 64 bits Version *5*.

Compilation et installation :

1. Créez un nouveau répertoire là où vous souhaitez installer *SimpleScalar*. Dans ce qui suit on notera ce dossier : /home/user/simplescalar.

2. Copiez ces trois fichiers dans /home/user/simplescalar (http://www.simplescalar.com) :

simplesim-3v0d.tgz
simpletools-2v0.tgz
simpleutils-2v0.tgz

3. Exécutez les commandes ci-dessous pour décompresser les fichiers :

tar xzvf `simplesim-3v0d.tgz`
tar xzvf `simpletools-2v0.tgz`
tar xzvf `simpleutils-2v0.tgz`

4. Vous obtenez alors ces répertoires :

`binutils-2.5.2`	*GNU binary utilities code*
`f2c-1994.09.27`	*Sources pour la traduction du format FORTRAN au C*
`gcc-2.6.3`	*Sources pour le compilateur GNU C*
`glibc-1.09`	*Sources pour la bibliothèque GNU libc*
`simplesim-3.0`	*Sources du simulateur SimpleScalar*
`ssbig-na-sstrix`	*Répertoire d'installation des outils de format big-endian*
`sslittle-na-sstrix`	*Répertoire d'installation des outils de format little-indian*

Pour la bonne installation de *SimpleScalar*, il faut respecter l'ordre de compilation des différents éléments :

1. Compilation de *binutils*
2. Compilation du simulateur
3. Compilation de *gcc*
4. Compilation de *glibc*, si nécessaire (librairie *libc* déjà compilée)
5. Compilation de l'outil Fortran, si nécessaire

Vous risquez de rencontrer certains problèmes de compilation, afin de les résoudre veuillez vous référer à l'annexe à la fin de cette fiche.

5. Détermination du type *endian* de votre machine :
   (1) cd `/home/user/simplescalar/simplesim-3.0`
   (2) make `sysprobe`
   (3) ./`sysprobe -s`

La dernière commande retourne *"big"* ou *"little"* indiquant que votre machine est *"big-endian"* ou *"little-endian"*, respectivement. Pour simplifier, les commandes suivantes supposerons une installation *little-endian*.

6. Compilation et installation des *binutils*, exécutez :
   (1) cd `/home/user/simplescalar/binutils-2.5.2`
   (2) sh ./`config.guess`

   Cette commande vous donnera le *host* de votre machine (du type : CPU-Compagnie-Système), si la valeur retournée n'est pas reconnue, utilisez le *host* suivant : i386-*-linux

   (3) ./configure --host=`i386-\*-linux\`
   --target=`sslittle-na-sstrix --with-gnu-as --with-gnu-ld\`
   --prefix=`/home/user/simplescalar`
   (4) make
   (5) make install

7. Compilation et installation du simulateur *SimpleScalar*, exécutez :
   (1) cd `/home/user/simplescalar/simplesim-3.0`
   (2) make config-pisa
   (3) make

8. Compilation et installation du compilateur *GNU C*, exécutez :
   (1) cd `/home/user/simplescalar/gcc-2.6.3`
   (2) ./configure --host=`i386-\*-linux\`
   --target=`sslittle-na-sstrix --with-gnu-as --with-gnu-ld\`
   --prefix=`/home/user/simplescalar`
   (3) make LANGUAGES=c
   (4) ../simplesim-3.0/sim-safe ./enquire -f >! float.h-cross
   (5) make install

9. Vérification, dans /home/user/simplescalar/bin vous devriez trouver :

sslittle-na-sstrix-ar
sslittle-na-sstrix-as
sslittle-na-sstrix-c++filt
sslittle-na-sstrix-gasp
sslittle-na-sstrix-gcc
sslittle-na-sstrix-ld
sslittle-na-sstrix-nm
sslittle-na-sstrix-objcopy
sslittle-na-sstrix-objdump
sslittle-na-sstrix-ranlib
sslittle-na-sstrix-size
sslittle-na-sstrix-strings
sslittle-na-sstrix-strip

C'est la liste complète des *"binary utilities"* et des compilateurs *C*.

Entrée :

Les seuls formats binaire connus par *SimpleScalar* sont le binaire *Alpha* et le *PISA* (Portable Instruction Set Architecture). Comme nous n'avons pas de machine *Alpha* à notre disposition, nous utilisons le format *PISA*. Pour pouvoir simuler un exécutable avec SimpleScalar, celui-ci doit être compilé avec un compilateur *gcc* spécial qui permet la création d'un fichier binaire au format *PISA*.

1. Génération des binaires *PISA* pour *SimpleScalar*, exécutez :
   (1) cd `/home/user/simplescalar/bin`
   (2) ./sslittle-na-sstrix-gcc -g -O -o `hello hello.c` -lm

   Cette commande va compiler le programme *"hello.c"* et générer un exécutable au format *PISA* nommé *hello*.

2. Exécution de *SimpleScalar* (simulateur *sim-safe*):
   (1) ../simplesim-3.0/sim-safe `hello`

   Exécute hello en utilisant le simulateur *sim-safe*.

Sortie :

Voici par exemple la sortie du simulateur *sim-safe* de *SimpleScalar* sur *hello* :

```
sim: ** simulation statistics **
sim_num_insn                  7803 # total number of instructions executed
sim_num_refs                  4258 # total number of loads and stores executed
sim_elapsed_time                 1 # total simulation time in seconds
sim_inst_rate            7803.0000 # simulation speed (in insts/sec)
ld_text_base            0x00400000 # program text (code) segment base
ld_text_size                 70080 # program text (code) size in bytes
ld_data_base            0x10000000 # program initialized data segment base
ld_data_size                  8192 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base           0x7fffc000 # program stack segment base (highest address in stack)
ld_stack_size                16384 # program initial stack size
ld_prog_entry           0x00400140 # program entry point (initial PC)
ld_environ_base         0x7fff8000 # program environment base address address
ld_target_big_endian             0 # target executable endian-ness, non-zero if big endian
mem.page_count                  26 # total number of pages allocated
mem.page_mem                  104k # total size of memory pages allocated
mem.ptab_misses                 26 # total first level page table misses
mem.ptab_accesses           475404 # total page table accesses
mem.ptab_miss_rate          0.0001 # first level page table miss rate
```

Fig. 15 : Sortie du simulateur *sim-safe* de *SimpleScalar* sur *hello*

Annexe : Fichiers à modifier

(1) Fichier : /home/user/simplescalar/binutils-2.5.2/configure
Ligne : 494
Action : remplacer toute la ligne par "rm -f ${tmpfile}.com \
${tmpfile}.tgt ${tmpfile}.hst ${tmpfile}.pos"
Ligne : 1004
Action : commenter

(2) Fichier : /home/user/simplescalar/binutils-2.5.2/libiberty/functions.def
Ligne : 36, 42, 43, 44, 56
Action : commenter

(3) Fichier : /home/user/simplescalar/binutils-2.5.2/bfd/sysdep.h
Ligne : 44, 45
Action : commenter

(4) Fichier : /home/user/simplescalar/binutils-2.5.2/ld/ldmisc.c
Ligne : 24
Action : remplacer toute la ligne par "#include <stdarg.h>"

Ligne : 344, 348, 357, 361, 389, 393, 402, 407
Action : commenter

(5) Fichier : /home/user/simplescalar/gcc-2.6.3/cccp.c
Ligne : 194
Action : commenter

(6) Fichier : /home/user/simplescalar/gcc-2.6.3/sdbout.c
Ligne : 56
Action : remplacer toute la ligne par "#if 0"

(7) Fichier : /home/user/simplescalar/gcc-2.6.3/insn-output.c
Ligne : 675, 750, 823
Action : ajouter "\" à la fin de la ligne

(8) Fichier : /home/user/simplescalar/gcc-2.6.3/gcc.c
Ligne : 172
Action : commenter

(9) Fichier : /home/user/simplescalar/gcc-2.6.3/cp/g++.c
Ligne : 90
Action : commenter

Bibliographie :

[Austin97] T. M. Austin: *A User's and Hacker's Guide to the SimpleScalar Architectural Research Tool Set, Version 2.0*

[Austin97a] T. M. Austin, D. Burger: *The SimpleScalar Tool Set, Version 2.0*

http://hpc5.cs.tamu.edu/docs/simplescalarinstallmanual.htm

www.capsl.udel.edu/courses/cpeg323/2007/projects/simplescalar_build_memo-.pdf

http://www.simplescalar.com

Wattch

Présentation :

Le logiciel *Wattch* est un outil d'estimation de consommation d'énergie basé sur *SimpleScalar* 3.0. Le logiciel *Wattch* permet trois types d'utilisation présentés en figure 16 :

- Configuration de l'architecture (Scénario A) : permet de choisir entre plusieurs architectures celle qui possède une consommation d'énergie moindre,
- Optimisations du compilateur (Scénario B) : permet de choisir des stratégies d'optimisation de compilation (notre sujet),
- Optimisations d'estimation de la consommation (Scénario C) : permet de compléter le modèle de *Wattch* pour obtenir une estimation plus précise.

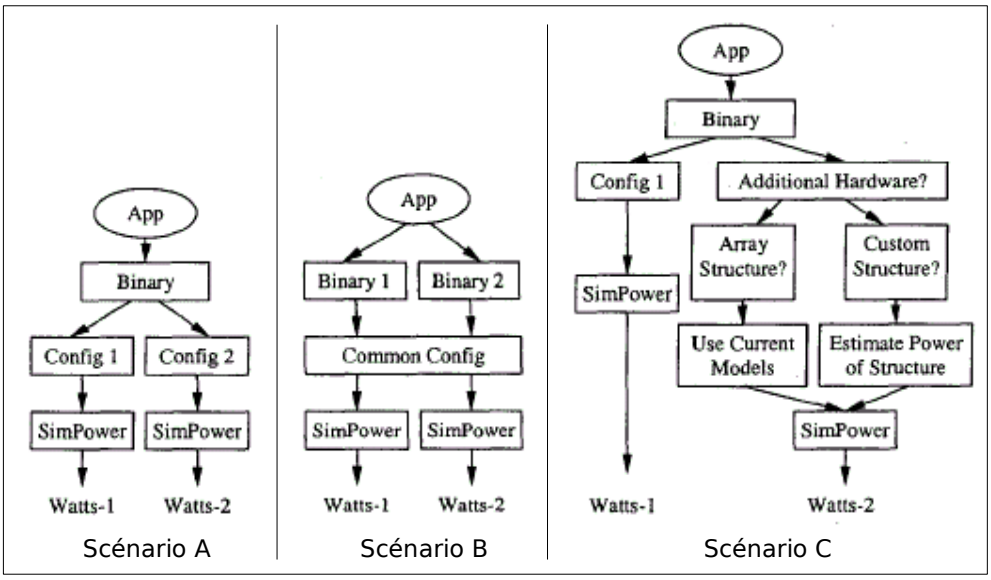

Fig. 16 : Les différents types d'utilisation de *Wattch* d'après [Brooks00]

Note :

- *Wattch* fonctionne sur toutes les plate-formes sur lesquelles *SimpleScalar* fonctionne.

- Nécessite la compilation des *binutils* et de *gcc* fournis avec *SimpleScalar* afin de pouvoir générer des exécutables au format *PISA*.

- La documentation de *Wattch* est assez légère voire inexistante et fait beaucoup appel à la documentation de *SimpleScalar*.

- Fonctionne sur une *Red Hat Entreprise Linux* 64 bits Version *5*.

Compilation et installation :

1. Créez un nouveau répertoire là où vous souhaitez installer *Wattch*. Dans ce qui suit on notera ce dossier : /home/user/wattch.

2. Copiez ce fichier dans /home/user/wattch (http://www.eecs.harvard.edu/~dbrooks/wattch-form.html)

   sim-wattch-1.02.tar.gz

3. Exécutez la commande ci-dessous pour décompresser le fichier :

   tar xzvf `sim-wattch-1.02.tar.gz`

4. Vous obtenez alors ce répertoire :

   `sim-wattch-1.02` *Sources du simulateur Wattch*

5. Compilation et installation du simulateur *Wattch*, exécutez :
   (1) cd `/home/user/wattch/sim-wattch-1.02`
   (2) make

Entrée :

Les seuls formats binaire connus par *Wattch* sont le binaire *Alpha* et le *PISA* comme pour le simulateur *SimpleScalar*. Pour pouvoir simuler un exécutable avec *Wattch*, il faut utiliser le compilateur *gcc* utilisé par *SimpleScalar* afin de créer un fichier binaire au format *PISA*. Pour cela, voir Génération des binaires *PISA* pour *SimpleScalar*.

1. Exécution de *Wattch* :
   (1) cd `/home/user/wattch/sim-wattch-1.02`
   (2) ./sim-outorder `hello`

Cette commande lance le simulateur *sim-outorder* de *Wattch* sur l'exécutable au format *PISA* nommé *hello*.

Sortie :

Wattch fournit trois valeurs différentes de mesure de puissance selon la politique d'extinction d'horloge (*clock gating*) utilisée :

- pas d'extinction (*rename_power*),
- extinction simple de l'horloge (*rename_power_cc1*),
- extinction agressive, idéale (*rename_power_cc2*) aucune puissance consommée lorsque l'horloge est éteinte,
- extinction agressive, non-idéale (*rename_power_cc3*) une fraction de la puissance est consommée quand le bloc est éteint.

Le manuel de *Wattch* conseille d'utiliser la dernière information qui semble la plus adaptée au monde actuel. Voici par exemple la sortie du simulateur *sim-outorder* de *Wattch* sur *hello* (seules les informations utiles sont présentées) :

```
rename_power_cc3           1241.6710# total power usage of rename unit_cc3
bpred_power_cc3    7364.5544# total power usage of bpred unit_cc3
window_power_cc3           7885.8383# total power usage of instruction window_cc3
lsq_power_cc3      1846.7088# total power usage of lsq_cc3
regfile_power_cc3          6627.4768# total power usage of arch. regfile_cc3
icache_power_cc3          11357.0543# total power usage of icache_cc3
dcache_power_cc3          18796.9806# total power usage of dcache_cc3
dcache2_power_cc3          6338.7410# total power usage of dcache2_cc3
alu_power_cc3     29629.1846# total power usage of alu_cc3
resultbus_power_cc3        6188.5825# total power usage of resultbus_cc3
clock_power_cc3   53515.3230# total power usage of clock_cc3
avg_rename_power_cc3          0.1069# avg power usage of rename unit_cc3
avg_bpred_power_cc3           0.6342# avg power usage of bpred unit_cc3
avg_window_power_cc3          0.6791# avg power usage of instruction window_cc3
avg_lsq_power_cc3             0.1590# avg power usage of instruction lsq_cc3
avg_regfile_power_cc3         0.5707# avg power usage of arch. regfile_cc3
avg_icache_power_cc3          0.9780# avg power usage of icache_cc3
avg_dcache_power_cc3          1.6188# avg power usage of dcache_cc3
avg_dcache2_power_cc3         0.5459# avg power usage of dcache2_cc3
avg_alu_power_cc3             2.5516# avg power usage of alu_cc3
avg_resultbus_power_cc3       0.5329# avg power usage of resultbus_cc3
avg_clock_power_cc3           4.6086# avg power usage of clock_cc3
fetch_stage_power_cc3     18721.6087# total power usage of fetch stage_cc3
dispatch_stage_power_cc3           1241.6710# total power usage of dispatch stage_cc3
issue_stage_power_cc3     70686.0358# total power usage of issue stage_cc3
avg_fetch_power_cc3           1.6123# average power of fetch unit per cycle_cc3
avg_dispatch_power_cc3        0.1069# average power of dispatch unit per cycle_cc3
avg_issue_power_cc3           6.0873# average power of issue unit per cycle_cc3
total_power_cycle_cc3    150792.1153# total power per cycle_cc3
avg_total_power_cycle_cc3            12.9859# average total power per cycle_cc3
avg_total_power_insn_cc3             17.8982# average total power per insn_cc3
```

Fig. 17 : Sortie du simulateur *sim-outorder* de *Wattch* sur *hello*

Les informations par bloc sont intéressantes car elles permettent de connaître plus précisément quelles unités consomment, ce qui peut aider dans la conception d'optimisations. Nous gardons donc ces données à la fois en terme de puissance et d'énergie (celles-ci devant être remaniées pour les traduire en *Joules*).

Bibliographie :

[Brooks00] D. Brooks, V. Tiwari, M. Martonosi: Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, ISCA, 2000

http://www.eecs.harvard.edu/~dbrooks/wattch-form.html

CACTI

Présentation :

CACTI, initialement développé par *Wilton* et *Jouppi*, est un modèle analytique intégré estimant les cycles, la surface, l'énergie, la puissance dynamique et les temps d'accès des mémoires caches. En intégrant tous ces modèles ensemble, les différences entre temps, puissance et surface sont toutes fondées sur les mêmes hypothèses et, par conséquent, sont mutuellement cohérentes.

Il existe plusieurs versions augmentées de *CACTI*. Parmi elles, on trouve *eCACTI* [Mamidipaka et Dutt, 2004] qui est un outil utilisé pour trouver une évaluation de la surface du cache, des temps d'accès et de la dissipation de puissance pour un ensemble donné de paramètres (taille du cache, taille du bloc, associativité et technologie). Les principales améliorations dans *eCACTI* sont :

estimation des courants de fuite,
calcul des largeurs des éléments architecturaux de la mémoire cache (par exemple le décodeur),
calcul de la dissipation de puissance dans chacun des sous-blocs du cache.

CACTI 4.2 est la version la plus récente de *CACTI*. Elle répare un petit nombre de bugs dans *CACTI 4.1* et *CACTI 4.0*, supporte un plus grand nombre de paramètres ainsi qu'un modèle des courants de fuite et reflète les récentes avancées dans la technologie des semi-conducteurs (prise en compte des technologies suivantes : 180, 130, 100, 90, 70, 65, 45 et 32 nm). Le lecteur intéressé trouvera plus de détails dans le rapport technique de *CACTI 4.0* [Tarjan et al., 2006]. Enfin, une interface graphique [CACTI 4.2, 2006] a été également ajoutée pour rendre *CACTI* facilement accessible via le Web à une plus grande communauté d'utilisateurs.

CACTI estime l'énergie des accès à la mémoire cache. Cependant, il peut aussi être utilisé dans le cas de la mémoire *Scratch-Pad* (SPM). Pour calculer l'énergie d'un accès à la mémoire SPM, *CACTI* est utilisé comme dans le cas de la mémoire cache mais sans prendre en considération la consommation des éléments suivants : tag memory array, tag column multiplexer, tag sense

amplifier et tag output drivers comme expliqué dans [Steinke et al., 2002b] . La version *CACTI 4.0* dispose d'une interface graphique dédiée à la mémoire SPM et une autre à la mémoire cache facilitant ainsi les calculs. Il existe même une version Bêta de *CACTI* [CACTI 5.0 Beta, 2006] également disponible via le Web qui prend en considération la mémoire DRAM.

Compilation et installation :

Il n'y a rien à installer, il suffit de se connecter sur l'interface graphique disponible à l'adresse Internet suivante : "http ://quid.hpl.hp.com :9081/cacti/".

Entrée :

La figure suivante donne un exemple des valeurs à saisir possibles.

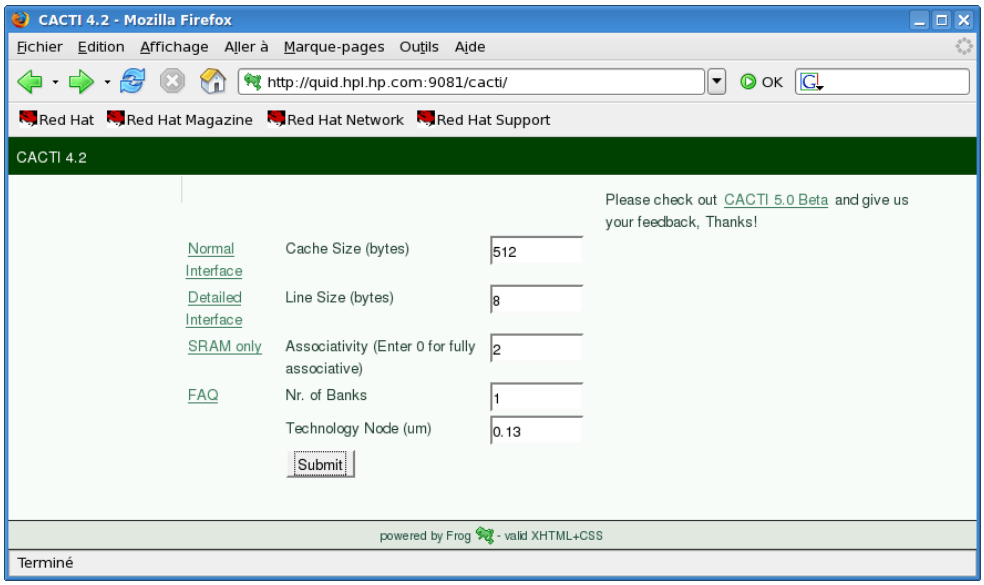

Fig. 18 : Interface graphique de saisie de l'outil *CACTI*

Sortie :

La figure 19 illustre les différentes valeurs obtenues en sortie.

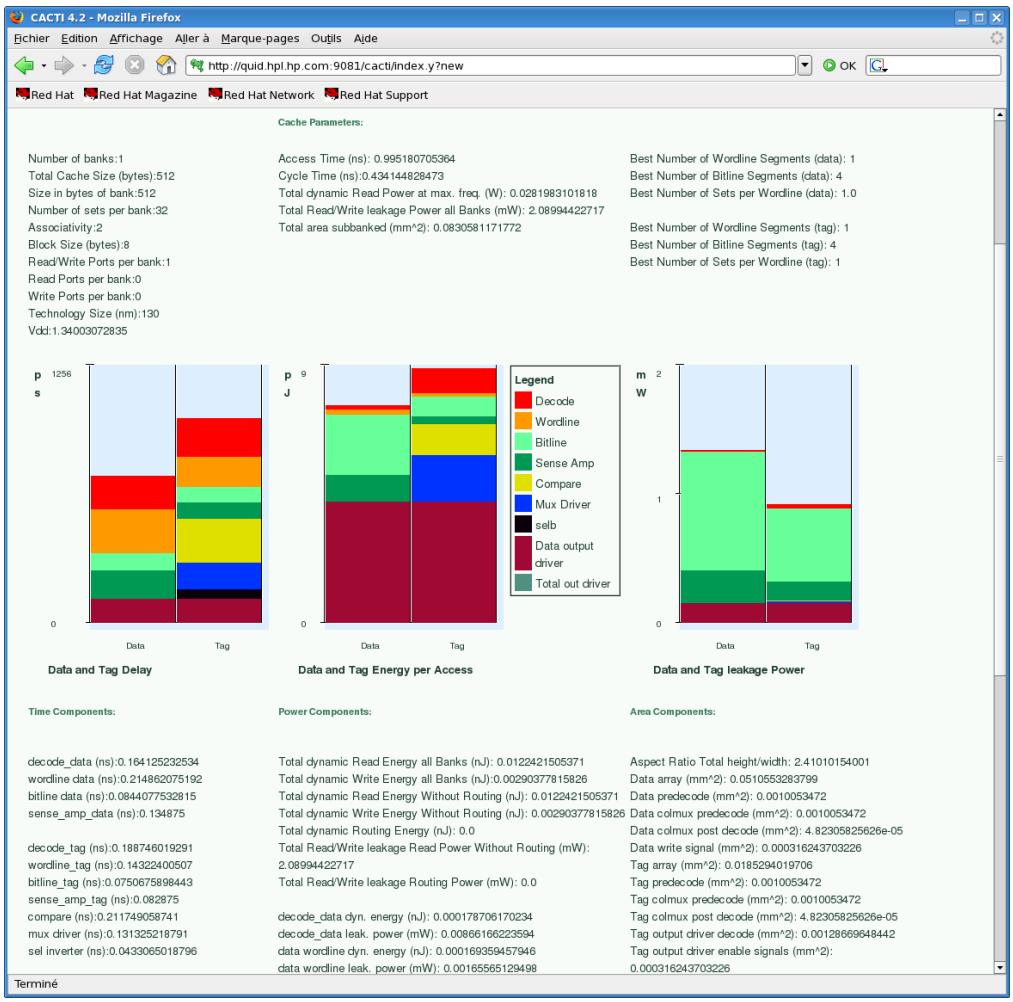

Fig. 19 : Interface graphique de sortie de l'outil *CACTI*

Bibliographie :

[CACTI 4.2, 2006] CACTI 4.2, 2006. http ://quid.hpl.hp.com :9081/cacti/.

[CACTI 5.0 Beta, 2006] CACTI 5.0 Beta. http ://quid.hpl.hp.com :9082/cacti/, 2006.

[Mamidipaka et Dutt, 2004] M. Mamidipaka et N. Dutt. eCACTI : An enhanced power estimation model for on-chip caches. Technical Report TR-04-28, CECS, UCI, 2004.

[Tarjan et al., 2006] D. Tarjan, S. Thoziyoor, et N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006- 86. HP Laboratories Palo Alto, June 2006.

[Wilton et Jouppi, 1996] S.J.E. Wilton et N.P. Jouppi. Cacti : An enhanced cache access and cycle time model. IEEE Journal of Solid-State Circuits, 1996.