Sur la reproductibilité des mesures des performances d'algorithmes numériques avec PerPI

David Parello, Philippe Langlois, Bernard Goossens

ComPAS'2013 : RenPar'21/ SympA'15 / CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Sur la reproductibilité des mesures des performances d'algorithmes numériques avec PerPI

David Parello and Philippe Langlois and Bernard Goossens

Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan.
Univ. Montpellier II, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier.
CNRS, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier.
prenom.nom@univ-perp.fr

Résumé
Nous décrivons les étapes qui permettent de reproduire les résultats présentés dans l'article compagnon *Améliorer l'analyse de la performance des algorithmes numériques*, des mêmes auteurs et soumis à SympA'15.

Mots-clés : sommation précise, arithmétique flottante, évaluation de la performance, parallélisme d'instruction, PerPI

---

1. Introduction.

Le point de départ de l'article compagnon *Améliorer l'analyse de la performance des algorithmes numériques* est le constat des limites de la confiance que l'on peut avoir en certaines performances mesurées et publiées, en particulier pour ce qui concerne le temps de calcul de récents algorithmes de sommation de n nombre flottants. Ni le classique décompte du nombre d'opérations flottantes au niveau algorithmique, ni les mesures expérimentales effectuées à partir des compteurs matériels intégrés aux unités de calcul ne sont entièrement satisfaisants. Cette incertitude est d'autant plus fâcheuse que ces nouveaux algorithmes de sommation ne se distinguent plus par la qualité numérique de la solution qu'ils calculent : tous les bits fournis sont exacts (à la précision du format flottant utilisé).

Nous proposons d'évaluer le *potentiel de performance* de ces algorithmes en les exécutant sur la machine idéale de Hennessy-Patterson : chaque instruction est exécutée au cycle suivant celui qui produit les sources dont elle a besoin. Le nombre de cycles de cette *exécution idéale* est une mesure significative du potentiel de performance de l'algorithme ainsi analysé.

Nous simulons cette exécution idéale et automatisons les mesures de ces caractéristiques grâce à l'outil PerPI [1] sur lequel nous revenons à la section 6.

Intel(R) Core(TM)2 Duo CPU P8800 @ 2.66GHz, x86_64
Linux version 3.2.0-3-amd64 (Debian 3.2.23-1)
gcc-4.7 (Debian 4.7.1-7) 4.7.1
gcc-4.6 (Debian 4.6.3-8) 4.6.3
gcc-4.5 (Debian 4.5.3-12) 4.5.3
gcc -std=c99 -march=core2 -msse2 -mfpmath=sse -O3 -funroll-all-loops

FIGURE 1 – Environnements pour les mesures avec PerPI

Dans l'article compagnon, nous détaillons l'analyse et les résultats de performance de huit algorithmes de sommation présentés au tableau 2.1. La figure 1 précise l'environnement utilisé qui comporte en particulier trois versions successives de gcc. En effet, bien que fiables et reproductibles, les mesures obtenues restent toujours dépendantes des codes générés par les compilateurs.

L'objectif de cet article est d'illustrer la reproductibilité de l'approche présentée. Nous décrirons donc les différentes étapes qui conduisent aux résultats présentés dans les trois principales figures de l'article compagnon.
En section 2, nous précisons les paramètres de la performance des algorithmes de sommation. Les trois sections suivantes permettront d'obtenir les figures annoncées : en section 3, les deux diagrammes-bâtons qui résument les performances générales des algorithmes étudiés, puis en section 4 l'illustration de l'influence de la distribution des exposants des opérandes sur les performances de deux algorithmes très récents : HybridSum et OnLineExact. Au delà de ces mesures, PerPI fournit des histogrammes de l'exécution idéale : la section 4 permettra aussi d'obtenir de tels tracés pour les deux algorithmes précédents. En section 5, nous mentionnons rapidement un aspect de non-reproductibilité à la fois très rarement rencontré et sans conséquence sur l'analyse présentée. Nous terminons par les principaux éléments de l'installation de PerPI, tous les détails étant accessibles à l'URL suivant.

`http://perso.univ-perp.fr/david.parello/perpi/`

Les fichiers nécessaires à cet article sont accessibles à l'URL suivante.

`http://perso.univ-perp.fr/david.parello/perpi/realis01`

2. Les paramètres de la performance des algorithmes de sommation

Nous mesurons avec PerPI le nombre de cycles de l'exécution idéale d'algorithmes de sommation précis et fidèles.

2.1. Les paramètres de la performance mesurée et leurs valeurs

Les paramètres de la performance de ces algorithmes varient comme présenté dans le tableau 2.1. Ces paramètres sont
- la longueur de la somme ou du vecteur des entrées x_i, soit n pour $\sum_{i=1}^{n} x_i$;
- le nombre de conditionnement de la somme, soit $\mathrm{cond} = \sum_i |x_i| / |\sum_i x_i|$;
- l'étendue des exposants (binaires) des valeurs à sommer, soit $\delta = e_{max} - e_{min}$ où $e_{max} = \max_e\{2^e \leq |x_i| < 2^{e+1}, i = 1, \cdots, n\}$ et $e_{min} = \min_e\{2^e \leq |x_i| < 2^{e+1}, i = 1, \cdots, n\}$. En pratique, δ varie dans l'intervalle $[0, 2046]$ pour les binary64 de l'IEEE-754.

Algorithme	Paramètres	Précision
Sum	n	Sommation classique
Sum2, DDSum	n, $cond$	Sommation (deux fois plus) précise
AccSum, FastAccSum, iFastSum	n, $cond$	Sommation fidèle
HybridSum, OnLineExact	n, $cond$, δ	Sommation fidèle

TABLE 1 – Paramètres du temps d'exécution des algorithmes de sommation. n : longueur de la somme, $cond$: nombre de conditionnement, δ : étendue de l'exposant

Le tableau 2.1 fournit les valeurs de paramètres testés. Pour notre étude, les paramètres principaux sont n et $cond$. En effet l'étendue des exposants δ est fortement liée aux valeurs de $cond$ choisies : δ est en particulier une fonction croissante de $cond$ à n fixé. Cependant nous reviendrons en fin de cette section sur l'influence particulière de la distribution de ces exposants à n et δ fixés.

Paramètres	valeurs testées
n	$10^3, 10^4, 10^5, 10^6$
$cond$	$[10^8, 10^{40}] \approx [\mathbf{u}^{-1/2}, \mathbf{u}^{-2.5}]$
δ	$10, 100, 500, 1000, 1500, 2000$

TABLE 2 – Ensemble des valeurs de paramètres testés.

Avec les figures 2 et 3, nous présentons les mesures du nombre de cycles de l'exécution idéale des algorithmes de sommation pour différentes longueurs de somme $n = 10^3, 10^4, 10^5, 10^6$. Ces sommes ont des conditionnements du même ordre de grandeur, ici $cond = 10^{32}$.
La section 3 est consacrée à la reproduction de cet effet.

2.2. Influence de la distribution des exposants à δ fixé

Nous exhibons l'influence de la distribution des exposants à δ (et n) fixé(s) sur les algorithmes HybridSum et OnLineExact. Pour cela, nous construisons les vecteurs d'entrée spécifiques d_U et d_D.

Ces vecteurs d_U et d_D partagent les mêmes longueurs (n), conditionnement ($cond$) et étendues d'exposants (δ) — ces paramètres sont arbitrairement choisis selon le tableau 2.1. En revanche, ces vecteurs diffèrent par la distribution des exposants de leurs entrées.

- les exposants du vecteur d_U sont uniformément répartis sur l'intervalle $[-\delta/2, \delta/2]$ — U pour uniforme.
- les $n-1$ premières composantes du vecteur d_D sont de même exposant $-\delta/2$, tandis que celui de sa dernière composante vaut $\delta/2$ — D pour Dirac.

La figure 4 exhibe que cette distribution des exposants modifie différemment le parallélisme d'instruction de HybridSum et OnLineExact.
La section 4 est consacrée à la reproduction de cet effet.

2.3. Préalable : génération des exécutables

2.3.1. `configuration.make` et Makefile

Ce préalable commence par la mise à jour du fichier de configuration `configuration.make` selon les lieux d'installation de PerPI (PIN, Pilp5 et résultats, voir section 6). Si besoin, il s'agit aussi de paramétrer les différents compilateurs (ou versions) utilisés. Dans ce cas, des répertoires dédiés à chaque compilateur doivent être créés (dans `obj` et `bin`). Pour cela, il suffit modifier le fichier `configuration.make` fourni en mettant à jour la définitions de ces versions. Nous avons par exemple utilisé les trois versions suivantes de gcc comme annoncé en figure 1.

```
COMPILER_VERSION_1 = 4.5
COMPILER_VERSION_2 = 4.6
COMPILER_VERSION_3 = 4.7
```

Si il est utilisé seul, le compilateur gcc par défaut ne demande aucune modification particulière.

Ce fichier permet enfin l'initialisation des paramètres des algorithmes testés, soit ici les valeurs de n, $cond$ et δ. Certains grandes longueurs de vecteurs à sommer nécessitent un temps de calcul important, en particulier pour $n \geq 10^6$ (nous y reviendrons plus loin). Pour une première utilisation, `configuration.make` limite n aux valeurs 10^3 et 10^4.

```
VECT_SIZES = 1000 10000 # 100000 1000000
COND_SIZES = 32
DELTA_SIZES = 10 100 500 1000 1500 2000
```

Générons maintenant les exécutables nécessaires aux sections suivantes

2.3.2. Cas de la mesure d'un seul exécutable

Les exécutables des générateurs sont obtenus par
`> make all_gen`
Les exécutables des sommations et des tests sont obtenus par
`> make all_sum`
Ces commandes mettent à jour les répertoires `obj/` et `bin/`. Ce dernier doit maintenant contenir les 7 exécutables suivants.

```
generator4 generator5dirac generator5unif
pingcctestperpicond5 pingcctest_HS_OLE
gcctestperpicond5 gcctest_HS_OLE
```

Les trois premiers contruisent les vecteurs d'entrée à sommer. Les exécutions idéales s'effectueront avec les deux exécutables `pingcc*` pour les différents vecteurs d'entrée générés. Les deux derniers `gcc*` sont des versions instrumentées des `pingcc*` pour vérifier de façon classique le comportement des algorithmes testés : précision des solutions calculées, mesures expérimentales des temps d'exécutions, ... Ces versions instrumentables sont fournies à titre indicatif : seules les exécutions idéales des `pingcc*` seront simulées et analysées avec PerPI.

2.3.3. Cas de la mesure de plusieurs exécutables

Dans l'article compagnon, nous avons souligné l'influence du compilateur, des ses versions et de ses options, sur l'exécution idéale et sa mesure. Il est instructif de générer et mesurer

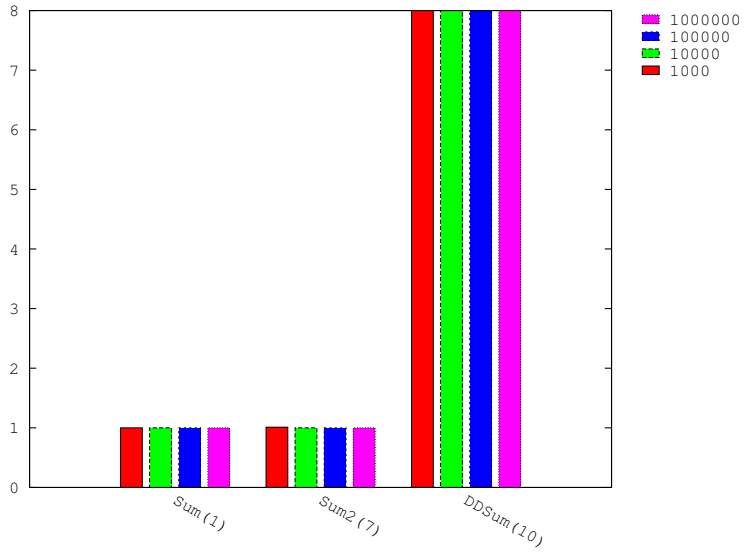

FIGURE 2 – Nombre de cycles de la sommation précise : ratios vs. Sum pour $cond = 10^{32}$ et $n = 10^3, 10^4, 10^5, 10^6$. (Axe des x : la valeur entre parenthèses est le nombre de flop sur n)

plusieurs exécutables issus du même source en faisant varier, principalement ici, les versions du compilateur gcc.
Rappelons qu'il a fallu précédemment mettre à jour `configuration.make` ainsi que les répertoires `obj` et `bin`. Ces derniers accueilleront automatiquement les différentes à versions analyser.
En ce qui nous concerne, trois versions des exécutables `*testperpicond5` ont été générés dans `bin/4.5, bin/4.6, bin/4.7`. Les résultats de PerPI correspondants sont générés dans des sous-répertoires équivalents de `perpires/`.

Nous ne ferons pas de différences selon les versions (répertoire vs. répertoire/sous-répertoire) dans ce qui suit.

3. Mesure du nombre de cycles des algorithmes de sommation.

Les figures 2 et 3 présentent ces mesures sous forme d'un ratio par rapport au nombre de cycles de l'exécution idéale de la sommation itérative classique Sum.
Quatre étapes conduisent à ces figures.

1. La génération des vecteurs à sommer et ses paramètres.
2. Le calcul des sommes précises et fidèles.
3. La mesure de l'exécution idéale et ses paramètres.
4. L'extraction des résultats et la génération des figures 2 et 3.

3.1. La génération des vecteurs à sommer et ses paramètres

Commande :

```
> make gen_data
```

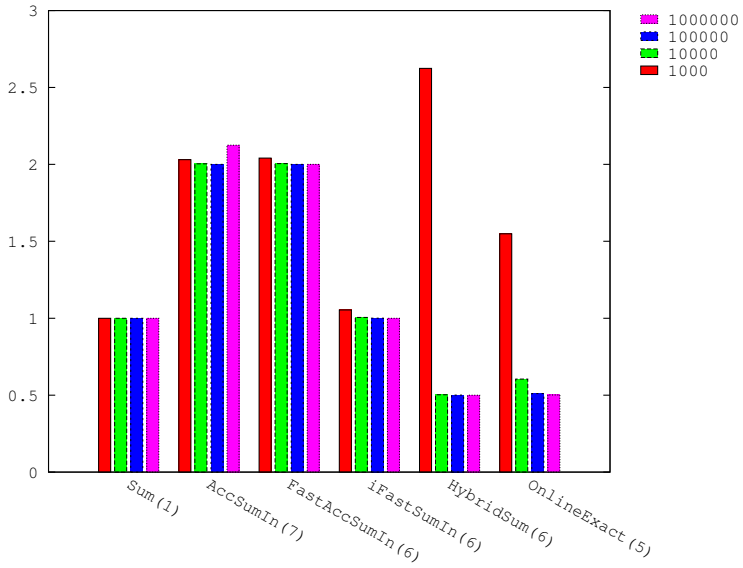

FIGURE 3 – Nombre de cycles de la sommation fidèle : ratios vs. Sum pour cond $= 10^{32}$ et $n = 10^3, 10^4, 10^5, 10^6$. (Axe des x : la valeur entre parenthèses est le nombre de flop sur n)

Fichier utilisé :
`generator4` dans `bin/`
Fichiers générés :
`n_cond.dat` dans `data/`

Il s'agit ici de générer des vecteurs d'entrée de longueurs variables $n = 10^3, 10^4, 10^5, 10^6$, et qui partagent des conditionnements du même ordre de grandeur, ici cond $= 10^{32}$. Rappelons que la reproductibilité de l'analyse d'ILP avec PerPI permet de se contenter d'un exemplaire de chacun de ces vecteurs.

Nous utilisons l'algorithme de génération de produits scalaires arbitrairement mal conditionnés proposé sous la forme d'un code Matlab dans [3]. Nous simplifions cet algorithme au cas de la sommation et proposons un codage en C décrit dans `gensum.h`.

```
// Conditionnning oriented : Ogita-Rump-Oishi's (2005)
double GenSum(double *x, double *C, unsigned int n, double c);
double GenSum2(double *x, double *C, unsigned int n, double exp_c);
```

Seul le dernier paramètre diffère selon ces deux versions : `c` est la valeur de cond souhaitée tandis que `exp_c` est son exposant (en base 10) ; *i.e.* `c` = $10^{\texttt{exp\_c}}$. Le générateur retourne la valeur effective `C` du conditionnement du vecteur généré qui peut être supérieure à la valeur souhaitée. Ensuite la commande

```
> make gen_data
```

génère un vecteur d'entrée pour chacune des valeurs de n et (de l'exposant) de cond indiqués dans le `configure.make` [1]

1. Rappelons que pour les premières utilisation, n a été limité aux valeurs 1000 et 10000.

Remarques.
- Ce générateur fait appel à une sommation très précise : AccSum dans [3]. D'autre part, générer de grands vecteurs peut prendre du temps sur une machine "ordinaire" : plusieurs heures pour $n = 10^6$ par exemple. Dans la version proposée, nous avons remplacé AccSum par OnLineExact.
- L'étendue des exposants des vecteurs générés par `GenSum` ne permet pas de couvrir les grandes valeurs de δ du tableau 2.1. Nous avons modifié `GenSum` en `GenSumDelta` de façon à obtenir de telles valeurs pour des longueurs n multiples de 4. `GenSum` est suffisant dans cette section qui ne dépend pas fondamentalement de l'étendue des exposants — les performances de HybridSum et OnLineExact sont annoncées indépendantes de δ dans [5].

```
// a la GenSum with a larger exponent range
double GenSumDelta(double *x, double *C, unsigned int n,
                   double exp_c, int delta);
```

3.2. Le calcul des sommes précises et fidèles

Commande :

```
> make rungcc_data
```

Fichiers utilisés :

`gcctestperpicond5` dans `bin/`
`n_cond.dat` dans `data/`

Cette commande exécute les différents algorithmes de sommations sur les vecteurs d'entrée précédemment générés. Cette exécution inclus quelques mesures de cycles "réels" basées sur le compteur matériel de performance (timestamp counter), et donc très indicatives. Les sommes et les mesures "réelles" sont affichées à l'écran — on remarquera que les résultats affichés diffèrent selon que les algorithmes soient mesurés ou non (les boucles de mesures accumulent les sommes calculées).

3.3. La mesure de l'exécution idéale et ses paramètres

Commande :

```
> make perpigcc_data
```

Fichiers utilisés :

`pingcctestperpicond5` dans `bin/`
`n_cond.dat` dans `data/`

Fichiers générés :

`n_cond_ilptool.profile, n_cond_ilptool_profile.xml` dans `perpires/`

Cette étape conduit à la mesure de l'exécution idéale des algorithmes de sommation sur l'ensemble des vecteurs d'entrée générés. La commande indiquée lance l'exécution des sommations dans PerPI. Les fichiers générés contiennent (entre autres) les mesures qui vont maintenant être exploitées.

L'exécution idéale est accompagnée des affichages déjà mentionnés mais qui cette fois doivent fournir les mêmes sommes calculées pour les algorithmes fidèles ($\mathtt{cond} \approx \mathbf{u}^2$).

3.4. L'extraction des résultats et la génération des figures 2 et 3

Commande : `> ./diag-batons-cycles.py --all|extract|graph = True`

Fichiers utilisés :
`n_cond_ilptool.profile, n_cond_ilptool_profile.xml` dans `perpires/`
Fichiers générés :
`*.eps` dans `fig/`

La génération des figures 2 et 3 est obtenue grâce au script Python `diag-batons-cycles.py` qui automatise les étapes suivantes — option `all`.

1. Extraction des caractéristiques de l'exécution idéale à partir des fichiers `.profile` précédemment générés — option `extract`.
2. Si plusieurs exécutables ont été générés et mesurés (c'est le cas ici), extraction des nombres minimal de cycles pour chaque exécution idéale à vecteur d'entrée constant — option `extract`.
3. Création des diagrammes-bâtons comme ceux des figures 2 et 3 — option `extract`.
4. Création des histogrammes de l'exécution idéale comme ceux de la figure 5 — option `graph`.

Remarque de configuration. Il est nécessaire de mettre à jour `diag-batons-cycles.py` avec les longueurs n et les versions de compilateurs effectivement testés. Par défaut ici, `sizes=[1000, 10000]` et `versions=["4.5","4.6","4.7"]` .

4. Influence de la distribution des exposants à δ fixé sur HybridSum et OnLineExact

Nous exhibons maintenant l'influence de la distribution des exposants à δ (et n) fixé(s) sur les algorithmes HybridSum et OnLineExact. Les figures 4 et 5 illustrent cette l'influence.

Comme précédemment, voyons comment les 4 étapes suivantes permettent d'obtenir ces deux figures.

1. La génération des vecteurs d_U et d_D.
2. Le calcul des sommes fidèles HybridSum et OnLineExact.
3. Les mesures des exécutions idéales.
4. L'extraction des résultats et la génération des figures 4 et 5.

4.1. Génération des vecteurs d_U et d_D

Commande :
`> make gen_du_dd`
Fichiers utilisés :
`generator5unif, generator5dirac` dans `bin/`
Fichiers générés :
`n_delta_unif.dat, n_delta_dirac.dat` dans `data/`

Les vecteurs d_U et d_D ont été introduits en section 2.2. Ils partagent les mêmes longueurs (n), conditionnement (cond) et étendues d'exposants (δ) — ces paramètres sont arbitrairement choisis selon le tableau 2.1.

4.2. Le calcul des sommes fidèles HybridSum et OnLineExact

Commande :

```
> make rungcc_du_dd
```

Fichiers utilisés :

`test_HS_OLE` dans `bin/`
`n_delta_unif.dat, n_delta_dirac.dat` dans `data/`

Cette commande exécute les sommes fidèles des vecteurs d_U et d_D par HybridSum et OnLineExact. Les sommes ainsi calculées sont égales.

4.3. Les mesures des exécutions idéales de HybridSum et OnLineExact

Commande :

```
> make perpi_du_dd
```

Fichiers utilisés :

`test_HS_OLE` dans `bin/`
`n_delta_unif.dat, n_delta_dirac.dat` dans `data/`

Fichiers générés :

`n_delta_unif_ilptool.profile, n_delta_unif_ilptool_profile.xml`
`n_delta_dirac_ilptool.profile, n_delta_dirac_ilptool_profile.xml`
dans `perpires/`

Pour chacun des paramètres n et δ, cette étape mesure les caractéristiques des exécutions idéales de HybridSum et OnLineExact sur les vecteurs d_U et d_D. Les fichiers générés contiennent (entre autres) les mesures qui vont maintenant être exploitées.

4.4. L'extraction des résultats et la génération de la figure 4

Commande :

```
> ./diag-du-dd.py
```

Fichiers utilisés :

`n_delta_unif_ilptool.profile, n_delta_unif_ilptool_profile.xml`
`n_delta_dirac_ilptool.profile, n_delta_dirac_ilptool_profile.xml`
dans `perpires/`

Fichiers générés :

`*.pdf` dans `fig/`

La figure 4 est obtenue grâce au script Python `diag-du-dd.py`.Comme précédemment, ce script extrait les caractéristiques des différentes exécutions idéales et produit les 4 tracés des ratios Cycles/n qui composent la figure 4.

Remarque de configuration. Il est nécessaire de mettre à jour `diag-du-dd.py` avec les longueurs n effectivement testées Pour la première utilisation, `sizes=[1000, 10000]`.

4.5. Histogrammes des exécutions idéales de HybridSum et OnLineExact

Les scripts Python jusque-là utilisés sont composés d'appels en ligne de commande. PerPI dispose aussi d'une interface graphique. Cette dernière permet une utilisation interactive bien adaptée aux tracés et à la manipulation (zooms par exemple) des histogrammes ou des graphes des instructions de l'exécution idéale. La figure 5 est un exemple d'histogrammes pour HybridSum et OnLineExact appliqués aux vecteurs d'entrée d_U et d_D de longueur $n = 10^4$.

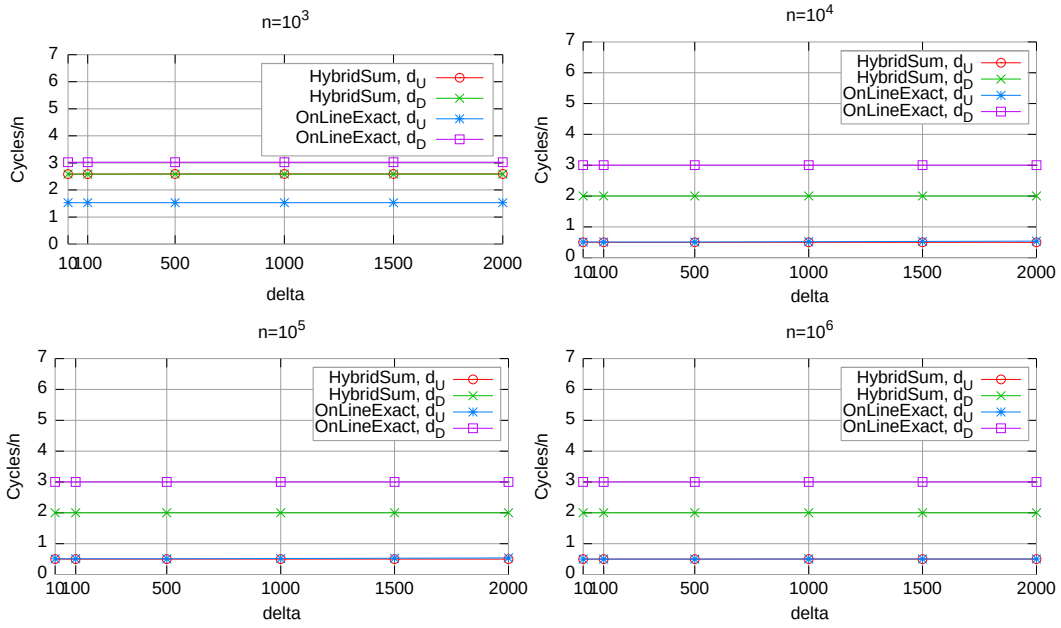

FIGURE 4 – OnLineExact souffre plus que HybridSum de la distribution de l'exposant exhibée par d_D ; cet effet disparaît pour d_U et $n \geq 10^4$.

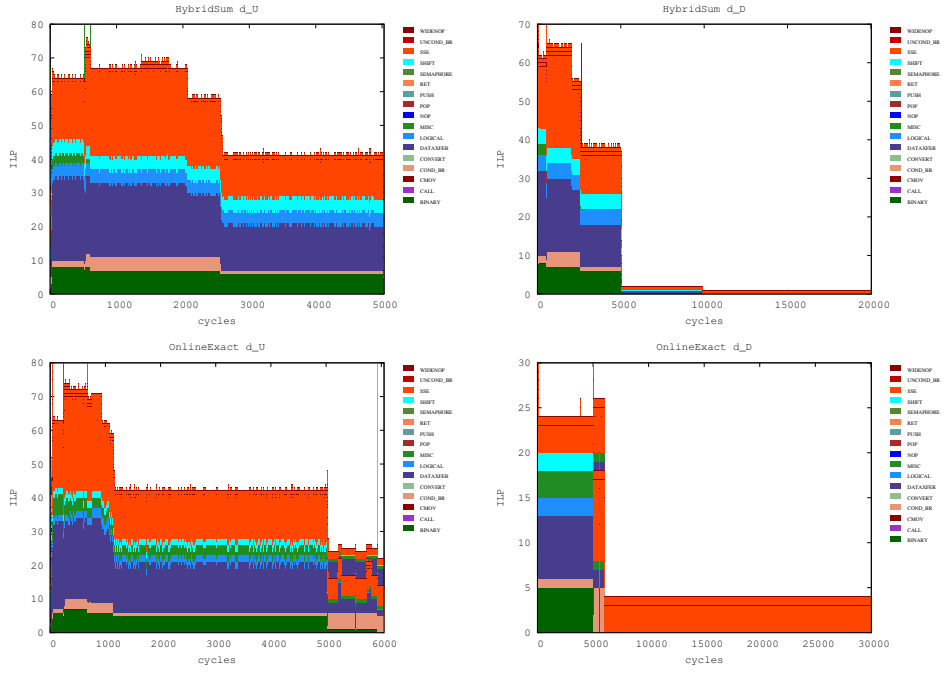

FIGURE 5 – Histogramme des exécutions idéales de HybridSum (haut) et OnLineExact (bas) pour d_U (gauche) et d_D (droite), $n = 10^4$.

La prise en main de l'interface graphique de PerPI (`ILPManager.py` dans le répertoire `Ilptool`)

```
start : <main>   (depth: 2, lcid: 104)
  stop  : <Sum>   (depth: 3, lcid: 10201)(cid: 10201)    I[13781]::C[10000]::ILP[1.3781]
  stop  : <Sum>   (depth: 3, lcid: 10203)(cid: 10203)    I[13781]::C[10000]::ILP[1.3781]
  stop  : <Sum>   (depth: 3, lcid: 10205)(cid: 10205)    I[13781]::C[10000]::ILP[1.3781]
  stop  : <iFastSumIn>   (depth: 3, lcid: 10207)(cid: 10207)    I[696088]::C[18043]::ILP[38.5794]
  stop  : <iFastSumIn>   (depth: 3, lcid: 10241)(cid: 10241)    I[696076]::C[18043]::ILP[38.5787]
  stop  : <iFastSumIn>   (depth: 3, lcid: 10275)(cid: 10275)    I[696076]::C[18043]::ILP[38.5787]
  start : <OnlineExactSum>   (depth: 3, lcid: 10309)
    stop  : <iFastSumIn>   (depth: 4, lcid: 10320)(cid: 10320)    I[29704]::C[611]::ILP[48.6154]
  stop  : <OnlineExactSum>   (depth: 3, lcid: 10309)(cid: 10309)    I[301467]::C[10607]::ILP[28.4215]
 stop  : <main>   (depth: 2, lcid: 104)(cid: 104)    I[2884900]::C[49320]::ILP[58.4935]
Global ILP (cid: 0)    I[2895541]::C[49572]::ILP[58.4108]
```

FIGURE 6 – Reproductibilité : un run suffit . . . en général

est décrite à l'URL

`http://perso.univ-perp.fr/david.parello/perpi/`

Nous renvoyons le lecteur à cette page Web qui détaille comment obtenir de tels tracés.

5. Sur les limites de la reproductibilité de PerPI.

La figure 6 aussi présentée dans l'article compagnon illustre un des principaux intérêts de l'approche proposée : une exécution suffit à mesurer de façon fiable et reproductible les caractéristiques de l'exécution idéale. Cette copie d'écran permet aussi d'exhiber un phénomène de non reproductibilité, heureusement rencontré très rarement mais qui reste pour nous inexpliqué à ce jour.

Les trois premières exécutions de Sum affichent le même nombre d'instructions I et de cycles C, resp. ici $I = 13781$ et $C = 10000$. C'est un exemple typique du caractère reproductif des mesures que nous obtenons avec PerPI, ce qui en pratique allège et fiabilise considérablement ces analyses.
Les trois exécutions suivantes concernent iFastSum, un autre algorithme de sommation fidèle. De façon surprenante, le nombre d'instructions de ces exécutions idéales n'est pas constant : il varie de 696076 à 696088 instructions exécutées. Cette variation est très faible (0.0012% ici) et n'est jamais apparu sur la mesure du nombre de cycles C. Ni PerPI, ni Pin (sur lequel nous reviendrons en section 6) ne sont à l'origine de cette variation. Le programme d'iFastSum appliqué à une même donnée exécute effectivement 12 instructions x86 supplémentaires dans le premier calcul par rapport aux deux suivants.
De notre point de vue, cette observation ne remet en question ni la fiabilité, ni la reproductibilité des mesures obtenues. Les variations introduites par les compilateurs, leurs versions et leurs options sont parfois bien plus déstabilisantes et réduisent ainsi l'abstraction de l'analyse proposée. En revanche, PerPI fournit une trace qu'il nous faut maintenant analyser pour identifier ces mystérieuses instructions supplémentaires.

6. Installer PerPI.

Le téléchargement de PerPI se fait à partir de l'onglet *Download* de la page Web d'URL déjà mentionné.

Le logiciel PerPI se compose de deux parties. L'une, *Pilp5*, est le calcul et l'autre, *Ilptool* est la présentation des résultats.
La partie de calcul est un outil Pin [2] [4]. Son installation nécessite celle d'une version récente de Pin. Nos expériences ont été faites avec la version 2.11, dites 49306, du 11 avril 2012. Une version plus récente peut être téléchargée (version 2.12, dites 54730, du 10 octobre 2012).
Selon la version de noyau Linux utilisée, il peut être nécessaire (c'est le cas avec la distribution 12.04 d'Ubuntu) pour installer Pin, puis pour utiliser PerPI, de modifier une variable du système. Cela se fait en exécutant sous *root* la commande :

```
# echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

L'installation de Pin se fait en deux commandes shell : (elle est expliquée dans le fichier téléchargeable *Release notes* à l'URL [4] ; l'installation décrite dans le manuel de l'utilisateur n'est plus à jour)

```
//On suppose que le répertoire courant est celui où Pin est placé
$
$ cd source/tools/SimpleExamples
$ make dir obj-intel64/opcodemix.so
```

On vérifie que l'installation de Pin est réussie en exécutant un exemple de Pintool qui écrit dans le fichier *opcodemix.out* le nombre d'instructions exécutées pour chaque *opcode*.

```
//On suppose que le répertoire courant est celui où Pin est placé.
//On effectue le test à partir de la commande /bin/ls appliquée
//au répertoire courant.
$ ../../../pin -t obj-intel64/opcodemix.so -- /bin/ls
...
//Le fichier opcodemix.out contient le nombre d'instructions
//exécutées par /bin/ls.
$ cat opcodemix.out
...
```

On installe ensuite Pilp5 (installation décrite dans l'onglet *Tutorial*, champ *Install* de la page web PerPI).
La partie de présentation des résultats (*Ilptool*) de PerPI s'appuie sur *libxml2* (Libxml++2.6 version 2.7.8 ou postérieure), *Gnuplot* et *Python*. Il est nécessaire d'installer des versions récentes de Gnuplot (version 4.4 ou postérieure) et Python (version 2.7.3 ou postérieure) ainsi que les librairies Python *matplotlib* (version 1.1.1 ou postérieure), *networkx* (version 1.6-2 ou postérieure) et Gnuplot.py (version 1.8 ou postérieure) (téléchargement à partir de la page Web PerPI). Cette partie de présentation est utile pour l'automatisation des exécutions et indispensable pour la visualisation de résultats graphiques (histogrammes et graphe de dépendance).
L'installation de *Ilptool* est décrite au champ *Getting started* de la page Web PerPI.

Bibliographie

1. Goossens (B.), Langlois (P.), Parello (D.) et Petit (E.). – PerPI : A tool to measure instruction level parallelism. *In : Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part I*, éd. par Jónasson (K.). pp. 270–281. – Springer.

2. Luk (C.), Cohn (R.), Muth (R.), Patil (H.), Klauser (A.), Lowney (G.), Wallace (S.), Reddi (V.) et Hazelwood. (K.). – Pin : Building customized program analysis tools with dynamic instrumentation. *In : PLDI '05 : Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation,*. ACM, pp. 190–200.
3. Ogita (T.), Rump (S. M.) et Oishi (S.). – Accurate sum and dot product. *SIAM J. Sci. Comput.*, vol. 26, n6, 2005, pp. 1955–1988.
4. URL = `http:www.pintool.org`.
5. Zhu (Y.-K.) et Hayes (W. B.). – Algorithm 908 : Online exact summation of floating-point streams. *ACM Transactions on Mathematical Software*, vol. 37, n3, septembre 2010, pp. 37 :1–37 :13.