Améliorer l'analyse de la performance des algorithmes numériques

David Parello, Bernard Goossens, Philippe Langlois

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

Améliorer l'analyse de la performance des algorithmes numériques

David Parello[1], Bernard Goossens[1], Philippe Langlois[1] et Kathy Porada[2]

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

Résumé
Cet article traite de la fiabilité des mesures de performances d'algorithmes numériques. Nous expliquons en particulier à quel point Rump a raison de dire que "*Mesurer le temps d'exécution d'une implémentation d'un algorithme ... sur les architectures d'aujourd'hui relève plus de l'aléa que de recherches scientifiques.*" Ni le décompte des opérations flottantes, ni les mesures basées sur les compteurs de performance ne sont satisfaisants en l'occurrence.
Nous proposons une analyse du parallélisme d'instructions (ILP) des algorithmes pour évaluer leur potentiel de performance plutôt que leur performance instantanée sur une exécution. Nous utilisons l'outil PerPI que nous avons développé pour automatiser l'analyse d'ILP. Nous montrons que PerPI donne une analyse de performance plus fiable que les mesures de temps et quasiment indépendante des machines. Nous montrons aussi ses faiblesses résiduelles.

Mots-clés : Parallélisme d'instructions, évaluation de performance, PerPI, reproductibilité.

1. Introduction : comment mesurer la performance d'un algorithme ?

Cet article présente une mesure fiable et reproductible de la performance d'algorithmes [1]. Cette mesure est basée sur l'outil PerPI (Performance et Parallélisme d'Instruction), conçu pour l'analyse d'ILP de codes x86. La présente section présente les écueils de la mesure de temps d'exécutions. La section 2 montre comment une analyse de l'ILP donne une évaluation du potentiel de performance. La section 3 décrit le fonctionnement de l'outil PerPI pour calculer un ILP. La section 4 présente et explique les résultats de l'analyse automatique d'ILP par PerPI pour 7 algorithmes de somme de nombres flottants.
Considérons le cas d'un algorithme numérique. De façon classique, on évalue sa complexité en temps en 2 étapes. Dans un premier temps, on compte le nombre d'opérations flottantes (sans les distinguer du point de vue de leurs coûts élémentaires). On obtient une complexité qui dépend de la taille des entrées [2] et qui tient compte des constantes multiplicatives des termes

1. Les expériences et les résultats de l'article sont évalués dans le cadre de Réalis [7].
2. D'autres facteurs comme par exemple la précision ne sont pas considérés à cette étape.

de plus haut degré. Dans un second temps, ce calcul théorique est comparé à des mesures expérimentales dont l'unité est soit la seconde, soit le cycle machine.
Les conclusions issues de telles analyses sont loin d'être fiables [3].
Cela tient en particulier à ce que les mesures de temps d'exécution donnent des résultats non reproductibles. Le temps d'exécution d'un programme varie, même si le jeu de données et l'environnement d'exécution sont constants. Cette incertitude a des causes multiples. Des événements parasites comme des tâches concurrentes ou des interruptions du système sont des facteurs bien connus. Les processeurs modernes sont faits de composants au comportement peu prévisible et pas seulement lié à l'application à mesurer et à ses données : l'ordonnanceur d'instructions, le prédicteur de branchements, le contrôleur de caches. Des conditions externes telles que la température peuvent modifier la fréquence d'horloge, ce qui rend non reproductibles les mesures basées sur les secondes écoulées. Les mesures basées sur le nombre de cycles sont piégeuses à cause de la durée fixe des "cycles bus" et de la durée variable des cycles des cœurs. Les instructions de calcul et de contrôle ont une latence en "cycles cœurs" et les instructions d'accès à la mémoire ont une latence en "cycles bus", dont la durée est un multiple variable de celle du "cycle cœur". Le cycle est non seulement ambigu mais peut aussi difficilement être considéré comme une durée fixe et unitaire sur les processeurs modernes.
On suppose à tort que les compteurs de performance des processeurs permettent de masquer cette complexité et qu'ils fournissent des mesures précises et fiables. Des travaux récents d'experts en performance des logiciels et des systèmes montrent que c'est une illusion. Zaparanuks et al. [13] incitent les analystes de la performance à être suspicieux des comptes de cycles fournis par les compteurs de performance. Par exemple, ils montrent que le nombre de cycles dépend de la place du code en mémoire. PAPI [6] est une bibliothèque bien connue pour exploiter les compteurs matériels. Les concepteurs de PAPI montrent que les compteurs de l'architecture x86_64 varient d'une exécution à l'autre. De plus, il n'y a pas de façon standard de compter le nombre d'instructions flottantes ou SSE. Même les compteurs d'instructions retirées, qui ne dépendent que de l'ISA (Instruction Set Architecture ou jeu d'instructions) donnent des résultats variables alors qu'ils devraient avoir un résultat constant d'une exécution à l'autre et même d'une machine à l'autre basées sur le même ISA. Dans [11] ils concluent qu'il est difficile de déterminer un "bon" compte pouvant servir de référence.

Le nombre de cycles et le nombre d'instructions sont les mesures clés de l'évaluation de performance. Le nombre de cycles est instable sur les machines actuelles. Même le nombre d'instructions, qui devrait être stable, est difficile à mesurer aujourd'hui. Il n'est pas étonnant que les mesures de temps publiées soient difficiles à reproduire.

2. L'ILP et le potentiel de performance d'un algorithme.

Une machine idéale [2] a une infinité de ressources : capacité de renommage infinie de la mémoire et des registres, prédicteur de saut parfait et infinité d'unités de calculs de latence 1 cycle. Exécuter un programme sur une machine idéale revient à disposer de sa trace d'exécution dès le départ et à en exécuter cycle après cycle toutes les instructions dont les sources sont calculées. L'exécution est un ordonnancement des instructions basé sur les seules dépendances producteurs/consommateurs. Sur une machine idéale chaque instruction s'exécute dès que possible, c'est-à-dire un cycle après l'exécution du dernier producteur dont elle dépend.
L'ILP est la quantité moyenne d'instructions exécutables en parallèle sur une machine idéale. Tous les processeurs actuels quand ils exécutent un programme, exploitent une partie de l'ILP grâce à des techniques connues de parallélisation : le pipeline, l'exécution superscalaire spécu-

lative, le renommage, la prédiction dynamique de sauts ou la spéculation des adresses, ...
Notre principale mesure est le nombre C de cycles d'une exécution idéale. Cette valeur est indicatrice de la meilleure performance possible de l'exécution quand celle-ci n'est contrainte que par les dépendances producteurs-consommateurs.
Cette mesure ne dépend plus ni de la machine réelle ni des conditions de l'exécution. Néanmoins, ce modèle dépend encore du jeu d'instructions (de l'architecture ou ISA). Par exemple, un ISA qui incluerait une instruction FMA (Fused Multiplication and Add) compterait un cycle de moins pour exécuter l'enchaînement d'une multiplication et d'une addition flottantes (tel que $a \times b + c$) qu'un ISA sans instruction FMA.
Nous comptons aussi le nombre I d'instructions exécutées. Le nombre moyen d'instructions exécutées par cycle I/C est l'ILP de l'exécution. D'une certaine façon, l'ILP élimine des dépendances architecturales. Dans l'exemple précédent sur le FMA, l'ILP est le même pour les deux architectures. L'ILP peut servir à comparer des programmes plutôt que des exécutions quand le jeu de données n'a d'influence ni sur le nombre d'instructions ni sur le nombre de cycles.

3. L'outil PerPI automatise l'analyse d'ILP.

Dans [1], Goossens et al. présentent l'outil logiciel PerPI qui automatise l'analyse d'ILP. PerPI mesure et permet de visualiser l'ILP d'algorithmes codés en x86.

3.1. Comment PerPI calcule l'ILP ?

PerPI est un outil Pin [8] développé par les auteurs de cet article. Pin [4] est un outil gratuit et programmable d'Intel (R).
Pin est un moteur d'instrumentation de code à l'exécution (JIT, Just In Time Compiler). L'instrumentation ajoute au code initial des routines définies par l'utilisateur.

```
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 1 – Reproductibilité : une exécution suffit

PerPI est une routine d'analyse calculant l'ILP du code exécuté pendant que le code est réellement exécuté. La routine effectue un saut vers le code analysé pour en exécuter une seule instruction puis reprend le contrôle pour mettre à jour ses mesures. Cette exécution en va-et-vient se poursuit jusqu'à ce que le code à mesurer ait été totalement exécuté. A chaque étape, PerPI calcule le cycle d'exécution de l'instruction courante, incrémente le nombre total d'instructions exécutées et met à jour le nombre de cycles de l'exécution (si le numéro de cycle de l'instruction courante est supérieur au nombre de cycles). PerPI calcule le rapport I/C, où I est le nombre d'instructions exécutées et C est le nombre de cycles de l'exécution.

N'importe quel exécutable x86 peut être mesuré par PerPI. PerPI peut aussi afficher un histogramme (ILP par cycle) et un graphe de dépendances. On peut trouver des détails dans [1].
Il suffit d'une exécution pour obtenir une mesure de performance (idéale) fiable et reproductible. C'est la contribution principale et significative de cet outil.

3.2. La mesure de PerPI pour les algorithmes de sommes de flottants.

La figure 1 est une copie d'écran des résultats de PerPI. Elle présente 7 exécutions de 3 algorithmes de sommes de flottants Sum , iFastSum et OnLineExact appliqués au même vecteur de 10000 sommandes. Les 3 premières exécutions (lignes 2 à 4) du même algorithme Sum présentent les mêmes nombres d'instructions I et de cycles C, resp. $I = 13781$ et $C = 10000$.
Les 3 exécutions suivantes (lignes 5 à 7) sont celles d'iFastSum. De façon surprenante, le nombre d'instructions exécutées n'est pas constant, variant de 696076 à 696088. Nous n'avons pas d'explication d'une telle variation. Cependant, elle est toujours très faible (0.0012% ici). Cela ne vient ni de PerPI ni de Pin. Le programme appliqué au même jeu de données a vraiment exécuté 12 instructions x86 supplémentaires lors de la première exécution. Cette variation impacterait bien entendu une mesure de temps comme elle impacte aussi très faiblement l'ILP.
Les lignes 9 et 10 de la figure 1 montrent deux mesures imbriquées : OnLineExact appelle iFastSum. D'une façon générale, PerPI fournit l'ILP de toutes les fonctions d'un programme à une profondeur bornée par un paramètre de la simulation ($depth = k$, non bornée si $k = -1$).

4. L'analyse de l'ILP de 7 algorithmes.

Nous étudions l'ILP de 7 algorithmes récents de somme de flottants. L'algorithme classique d'accumulation itérative Sum est pris comme référence. Sum2 et DDSum sont de la classe des algorithmes précis : la somme calculée est aussi précise que si elle avait été calculée avec une précision double. DDSum exécute Sum en arithmétique double-double comme dans [12]. Pour Sum2 , les calculs sont faits en précision courante avec en parallèle un cumul des erreurs d'arrondi qui compense le résultat final (algorithme de somme compensée de Rump et al.[5]).

```
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 2 – Environnement des expériences avec PerPI

Les 5 autres algorithmes sont iFastSum [14], AccSum [10], FastAccSum [9], HybridSum [14] et OnLineExact [15]. Ils sont de la classe des algorithmes fidèles : la somme calculée est l'un des 2 flottants consécutifs encadrant le vrai résultat, ou ce dernier s'il est lui-même un flottant.
Nous avons mesuré le nombre minimum de cycles C pour exécuter, sur une machine idéale, le code binaire généré par 3 versions récentes de gcc décrites en figure 2. Les figures suivantes présentent nos mesures exprimées comme des ratios comparés à Sum. En d'autres termes, nous mesurons le surcoût de l'amélioration de la précision (doublée ou fidèle) comme un facteur multiplicatif du nombre de cycles de la somme classique Sum, a priori imprécise.
Sum2 et DDSum sont présentés en figure 3 à gauche. Les 5 autres algorithmes de somme fidèle sont présentés en figure 3 à droite.

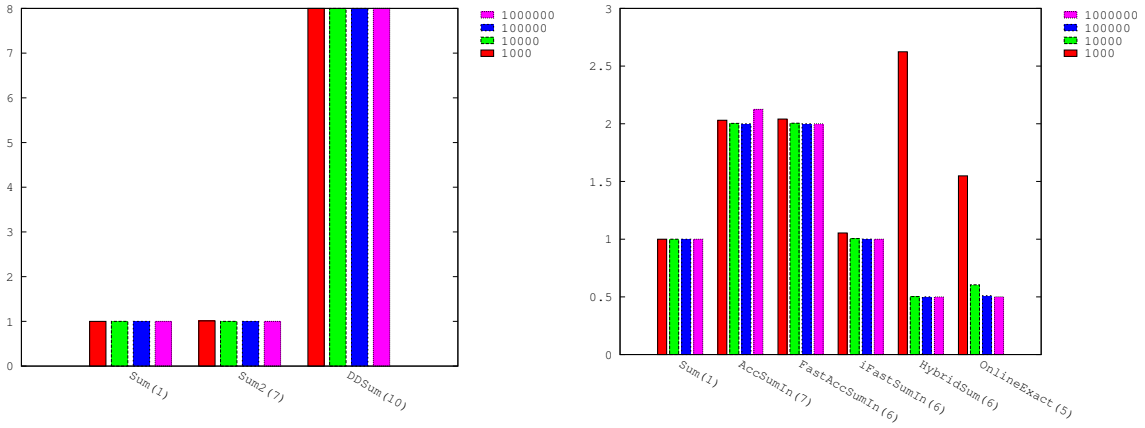

FIGURE 3 – Nombre de cycles : ratios vs. Sum pour Sum2 et DDSum (à gauche) et AccSum , FastAccSum iFastSum , HybridSum et OnLineExact (à droite) pour $n = 10^3, 10^4, 10^5, 10^6$ – la valeur entre parenthèses sur l'axe des x est le nombre de *flops* divisé par n.

HybridSum et OnLineExact ont un comportement impressionnant et intéressant. Pour n grand, leur exécution idéale produit une somme fidèle en à peu près la moitié du nombre de cycles de Sum (qui en l'occurrence renvoie un résultat sans aucun chiffre significatif). Les mesures de PerPI font bien apparaître cette propriété mais comment peut-on la justifier ?
Les algorithmes HybridSum et OnLineExact fonctionnent en 2 étapes. Une première étape accumule sans erreur les sommandes de même exposant dans un *petit* vecteur indicé par l'exposant (sa taille est liée à l'amplitude des exposants). Une seconde phase réalise la somme fidèle du petit vecteur. La première phase représente le coût principal de la somme pour n grand.
D'abord, Sum s'exécute idéalement en environ $C = n$ cycles. Le compilateur déroule la boucle interne de Sum. Gcc a déroulé 8 fois, ce qui réduit le code dynamique de contrôle. Le nombre d'instructions de contrôle exécutées est en gros divisé par 8. Cependant, ce déroulement ne raccourcit pas la plus longue chaîne de dépendances formée par la suite des accumulations de flottants. Pour cela il aurait fallu vectoriser ces calculs avec les registres SSE. Le compilateur ne le fait pas pour respecter la norme sur les calculs flottants qui interdit d'en changer l'ordre [3].
Ceci fait que l'exécution idéale de Sum pour n sommandes est bloquée à n cycles. Par contre, HybridSum et OnLineExact ne subissent pas cette contrainte puisqu'ils ne somment fidèlement que le vecteur issu de l'extraction des exposants. L'extraction en elle-même est un parcours du vecteur de longueur n (pour extraire l'exposant de chaque sommande et mettre à jour l'entrée correspondante du petit vecteur). Cette phase d'extraction bénéficie quant à elle du déroulement. En l'occurrence, gcc déroule 2 fois, ce qui divise par 2 le chemin critique du parcours. C'est ce qui explique que PerPI mesure environ 2 fois moins de cycles pour HybridSum et OnLineExact que pour Sum (pour n grand devant l'amplitude des exposants).

Zhu et Hayes désignent par δ l'amplitude des exposants. Pour HybridSum et OnLineExact, ils indiquent qu'elle "n'influence pas beaucoup le temps d'exécution, sauf … pour le cas d'une somme nulle" [15, p.7] — bien qu'amplitude et taille du petit vecteur augmentent conjointement. La figure 4 montre que les temps des exécutions idéales de ces algorithmes (normalisés

3. Pour une somme d'entiers, le déroulement est vraiment bénéfique à l'ILP car le code est vectorisé et la longueur du chemin critique est divisé par le facteur de déroulement.

par rapport à n) sont constants quand δ varie.

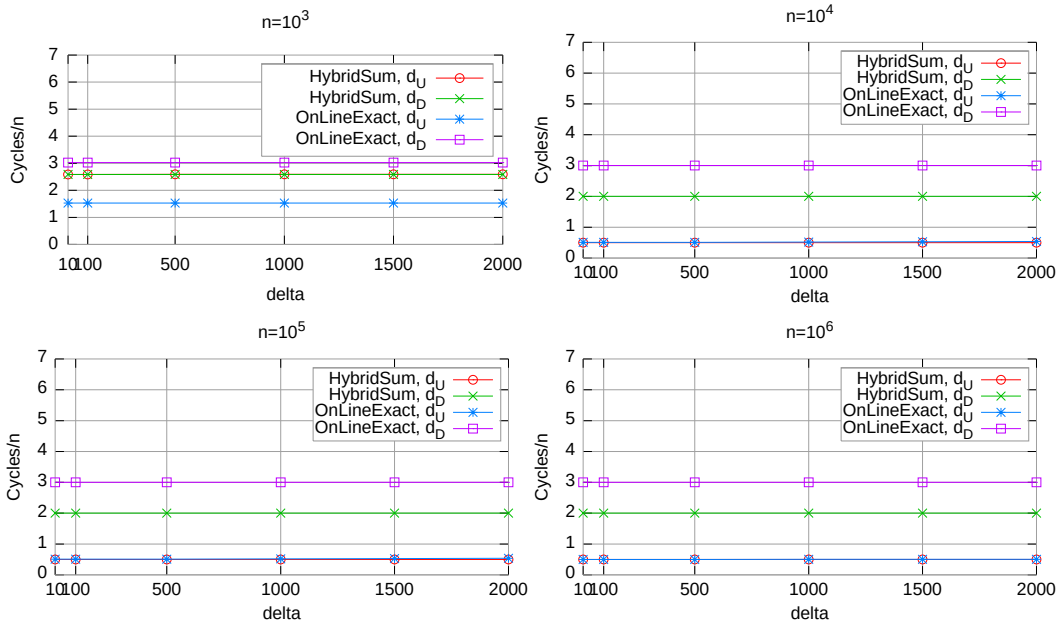

FIGURE 4 – La distribution d_D des exposants impacte plus OnLineExact que HybridSum alors que l'impact est identique pour d_U et $n \geq 10^4$.

Cependant, nous montrons maintenant que ces nombres de cycles idéaux varient avec des facteurs 4 et 6 (resp.) pour deux jeux de données d_U et d_D, de somme non nulle et ayant la même amplitude d'exposants δ.

Les exposants du vecteur de n entrées d_U —U pour uniforme— sont uniformément distribués dans l'intervalle $[-\delta/2, \delta/2]$. Le jeu d_D —D pour Dirac— se compose de $n-1$ entrées avec le même exposant $-\delta/2$, et la dernière entrée avec l'exposant $\delta/2$. La distribution des exposants modifie l'ILP de HybridSum et OnLineExact. Les 2 algorithmes ont des exécutions de longueurs égales pour une distribution uniforme alors qu'OnLineExact pâtit plus que HybridSum d'exposants fortement répétitifs. Ceci est d'autant plus surprenant qu'OnLineExact comprend une phase de concaténation qui, dans le cas d'exposants identiques, condense le petit vecteur (à environ 2 fois le nombre d'exposants différents). Cette optimisation réduit la phase finale d'OnLineExact à la somme fidèle (par iFastSum) d'un vecteur de taille minimale : par exemple 4 pour d_D. En comparaison, la même somme pour HybridSum traite un vecteur dont la taille est l'amplitude des exposants indépendamment de δ, c'est-à-dire 2048 ici. La mesure de PerPi (voir la figure 5) fait bien apparaître cette différence.

Pourtant, ces 2 versions de la phase finale du calcul contribuent de façon similaire à l'exécution idéale complète : seul le traitement des termes non nuls augmente la longueur du chemin critique. En effet dans la machine idéale les instructions traitant les valeurs nulles de la distribution d_D sortent de la chaîne des calculs pour s'exécuter dès le cycle 1.

La phase d'extraction explique la différence constatée. Etudions d'abord le corps de la boucle d'extraction de HybridSum présenté en figure 6. Il y a 8 instructions `(a)`,...,`(h)` exécutées en 6 cycles. Deux itérations successives ne sont liées que si elles ont le même indice j_h ou j_l : l'ac-

```
start : <HybridSum>
  start : <iFastSumIn>
  stop  : <iFastSumIn>     I[62719]::C[2580]::ILP[24.3097]
stop  : <HybridSum>      I[267980]::C[20020]::ILP[13.3856]
start : <OnlineExact>
  start : <iFastSumIn>
  stop  : <iFastSumIn>     I[334]::C[32]::ILP[10.4375]
stop  : <OnlineExact>    I[229263]::C[30026]::ILP[7.63548]
```

FIGURE 5 – L'appel final à iFastSum s'exécute en plus de cycles dans HybridSum que dans OnLineExact (2580 vs. 32) pour une distribution d_D mais le rapport s'inverse pour le calcul total (20020 vs. 30026) —ici $n = 10^4$ et $\delta = 500$.

cumulation est séquentielle dans l'entrée correspondante $a[\,]$. Ces itérations sont très parallèles en général (grâce à l'indépendance vis-à-vis de l'indice de boucle k) et pour d_U en particulier. Dans ce cas le chemin critique est le contrôle de boucle lui-même. La phase d'extraction d_U prend environ $n/2$ cycles, la boucle étant déroulée 2 fois —voir la figure 4. A l'inverse pour d_D la boucle d'extraction reste séquentielle : l'exposant j_h est partagé par $n-1$ itérations. En démarrant une nouvelle itération à chaque cycle, la phase d'extraction de d_D devrait prendre environ n cycles. Il faut regarder l'assembleur pour comprendre pourquoi la mesure est de $2n$ cycles —voir la figure 4 ou la figure 5. En fait, 2 itérations successives sont séparées par 2 cycles. Chaque accumulation `(f)` et `(h)` ($a[j_z] = a[j_z] + z$, avec z valant h ou l resp.) est traduite en 2 instructions assembleurs dépendantes : *i)* effectue une première addition registre/mémoire (h ou l en registre, le $a[\,]$ correspondant en mémoire) en registre, puis *ii)* copie le registre en mémoire (dans $a[\,]$). En disposant d'une instruction assembleur d'addition registre/mémoire vers la mémoire, on produirait un code qui éliminerait le cycle d'attente entre 2 itérations.

Cycle				
1	(a)	$a_s = x[k] \times$constante_separative		
2	(b)	$t = a_s - x[k]$		
3	(c)	$h = a_s - t$		
4	(d)	$j_h =$exposant(h)	(e)	$l = x[k] - h$
5	(f)	$a[j_h] = a[j_h] + h$	(g)	$j_l =$exposant(l)
6			(h)	$a[j_l] = a[j_l] + l$

FIGURE 6 – Boucle de la phase d'extraction de HybridSum : un seul cycle d'écart entre deux itérations successives par la dépendance entre $a[j_h]$ et $a[j_l]$.

C'est ce point de détail de l'ISA x86 qui explique le facteur 4 entre les cycles d'exécutions idéales de d_U ($n/2$) et d_D ($2n$) pour une même amplitude δ des exposants des sommandes additionnés par HybridSum. Seule l'analyse de l'ILP au niveau du code machine permet de comprendre un phénomène qui peut impacter le temps d'exécution brut. Dans une mesure de temps, cet éventuel impact se noie dans une quantité de parasites.
Etudions maintenant OnLineExact. La figure 7 montre qu'il y a plus de dépendances par les accès liés à la table a_1 produite par sa phase d'extraction. Les instructions `(c-c')` effectuent la mise à jour de la table $a_1[j] = a_1[j] + x[k]$ de telle façon que sa valeur reste dans c_u —ce qui évite un nouvel accès à la table en mémoire, dans les instructions `(d-e)`. Cela minimise le délai de la lecture-mise à jour-écriture de $a_1[j] = a_1[j] + x[k]$. Les dépendances des instructions `(b-c-c')` ajoutent 3 cycles de délai au démarrage de l'itération suivante si elle partage le même $a_1[j]$. La

Cycle				
1	(a)	j =exposant(x[k])		
2	(b)	$c = a_1[j]$		
3	(c)	$c_u = c + x[k]$		
4	(d)	$h = c_u - c$	(c')	$a_1[j] = c_u$
5	(e)	$t_1 = c_u - h$	(f)	$t_3 = x[k] - h$
6	(g)	$t_2 = c - t_1$		
7	(h)	$l = t_2 + t_3$		
8	(i)	$a_2[j] = a_2[j] + l$		

FIGURE 7 – Boucle de la phase d'extraction de OnLineExact : la chaîne d'instructions (b-c-c'), c'est-à-dire $a_1[j] = a_1[j] + x[k]$, ajoute un délai de 3 cycles entre deux itérations successives qui se partagent le même $a_1[j]$.

chaîne de dépendances du calcul complet pour d_U n'est pas significativement impactée par ce délai : le fort parallélisme (en regard de la chaîne d'incrémentations de l'indice de boucle k) de la phase d'extraction toute entière domine les quelques conflits sur $a_1[j]$. La phase d'extraction pour d_U par OnLineExact prend environ $n/2$ cycles pour les mêmes raisons que HybridSum. A l'inverse, la chaîne critique est impactée significativement par le délai de 3 cycles quand on somme une distribution d_D avec OnLineExact. La phase d'extraction pour d_D prend environ $3n$ cycles. Ce sont les contraintes algorithmiques d'OnLineExact qui expliquent le facteur 6 entre les sommes d_U $(n/2)$ et d_D $(3n)$ de même amplitude δ. Là encore, c'est l'analyse d'ILP au niveau du code machine qui permet d'expliquer les longueurs des dépendances induites par l'ISA.

La figure 8 montre les histogrammes des exécutions idéales de HybridSum et OnLineExact calculés par PerPI. Le nombre d'instructions exécutées est en ordonnée, les abscisses représentant les cycles ; les couleurs distinguent les types d'instructions, par exemple les instructions SSE sont en orange alors que les transferts de données sont en violet. Ces histogrammes confirment l'effet de la distribution des exposants sur les sommes d_U et d_D. Dans le cas de d_D (à droite) on peut séparer l'exécution en deux phases. D'abord, HybridSum et OnLineExact effectuent l'extraction des exposants, resp. environ les 5000 et 6000 premiers cycles, soit $n/2$ cycles : les instructions de décalages (en bleu clair) et logiques (en bleu) qui servent à identifier les exposants disparaissent après 5000 cycles. Cette phase correspond à peu près à la totalité de l'exécution pour d_U (à gauche). HybridSum (en haut) prend quelques cycles de plus dans le cas d_U (à droite) : chaque somme du vecteur des exposants commence dès que les extractions et les accumulations dans l'entrée correspondante sont achevées, sans attendre la terminaison du traitement du vecteur en entier. Etant donné que les exposants de d_U sont uniformément distribués, la phase finale de l'addition fidèle (par iFastSum) s'exécute en recouvrement parallèle avec la phase d'extraction. Pour d_D, la seconde phase de l'histogramme de HybridSum , c'est-à-dire après le cycle 5000, est la suite de la longue phase d'extraction de longueur $2n$ que nous avons identifiée précédemment. Elle ne se compose que d'instructions SSE et de transferts. De façon similaire, le dernier histogramme (en bas à droite) présente une étape d'extraction longue de $3n$ cycles pour OnLineExact appliqué à la distribution d_D.

Cette discussion montre à quel point l'analyse détaillée de la performance est délicate, même dans le modèle idéalisé de PerPI. Ce niveau de détail (l'ILP du code machine) est cependant nécessaire pour mettre en lumière l'impact du compilateur et de l'ISA sur les mesures.

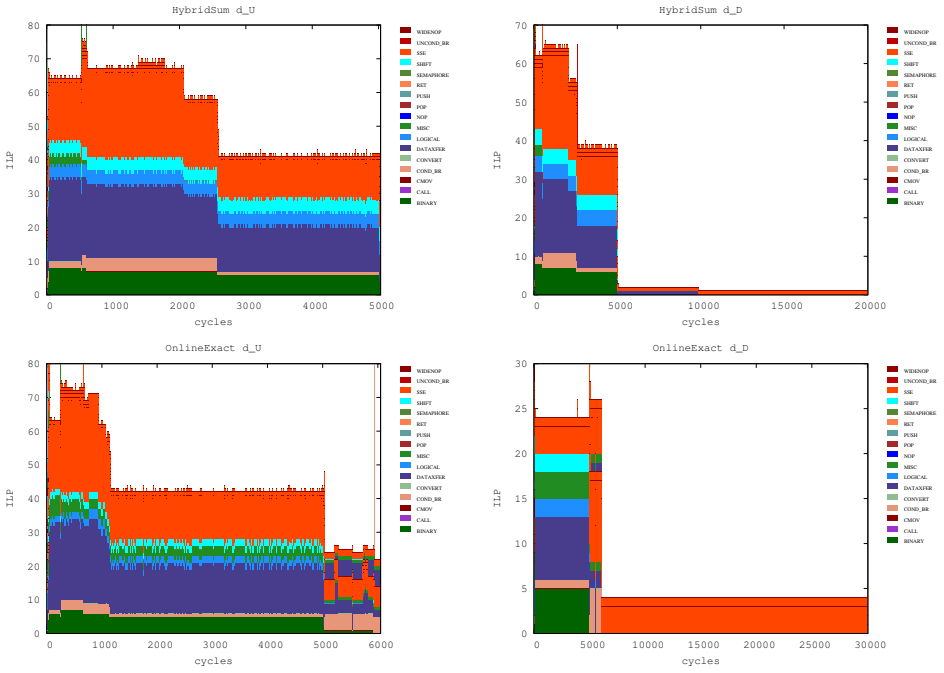

FIGURE 8 – Histogrammes des exécutions idéales de HybridSum (en haut) et OnLineExact (en bas) pour d_U (à gauche) et d_D (à droite), $n = 10^4$.

5. Conclusion

La comparaison d'algorithmes de même complexité requiert une évaluation précise de leur potentiel de performance. Les mesures de performances basées sur les compteurs matériels sont entâchées d'incertitudes et sont non reproductibles.
Nous avons montré dans cet article en quoi l'analyse de l'ILP des algorithmes fournit une mesure fiable de leur potentiel de performance. Nous avons présenté l'outil PerPI qui est une plate-forme logicielle pour analyser et visualiser l'ILP d'une exécution. Nous avons montré que les analyses automatiques de l'ILP fournies par PerPI sont reproductibles. L'analyse d'ILP permet de mieux comprendre le comportement intrinsèque des algorithmes et le parallélisme qu'ils contiennent. Elle permet aussi de prévoir le bénéfice que l'exécution réelle peut tirer de la microarchitecture. Elle peut enfin suggérer des idées d'optimisations, soit de l'algorithme pour en augmenter l'ILP, soit du programme pour tirer mieux partie du matériel comme cela est fait pour AccSum et FastAccSum dans [1].

Cet article représente une première étape vers plus de science et moins d'aléa dans l'analyse des temps d'exécution et la comparaison d'algorithmes. Pour autant les résultats fournis par PerPI ne sont pas encore assez abstraits. Nous avons montré qu'ils dépendent à la fois du jeu d'instructions et du compilateur. Trouver le bon niveau d'abstraction n'est pas chose facile. Faut-il se placer au niveau de l'expression algorithmique ou bien au niveau du langage d'assemblage ? Le premier est plus abstrait et générique et le second est plus concret et réaliste. Notre expérience nous laisse penser que les 2 niveaux ont leur utilité.
Au-delà de cet article, notre objectif est de garantir la reproductibilité pour améliorer la fiabilité des mesures pour les algorithmes de sommes précises de nombres flottants. Les résultats de

PerPI présentés ici fournissent une référence à l'évaluation de futures contributions. La prochaine étape est la mise à disposition d'un dépot logiciel ayant les 2 caractéristiques suivantes. D'abord, il permet de récolter et partager toutes les ressources nécessaires à la construction d'un résultat : fichiers de tests, de résultats, programmes sources, constructeurs, fichiers et générateurs de données, mesures réelles et idéales, ... Ensuite, le site doit permettre une interaction : déposez votre programme, vos données, exécutez le tout et comparez vos résultats (mesures réelles et idéales) à l'état de l'art présent sur le site.
Ainsi, nous contribuerons de façon fiable.

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. Hennessy (J. L.) et Patterson (D. A.). – *Computer Architecture – A Quantitative Approach*. – Morgan Kaufmann, 2003, 2nd édition.
3. Langlois (P.), Parello (D.), Goossens (B.) et Porada (K.). – *Less Hazardous and More Scientific Research for Summation Algorithm Computing Times*. – Rapport technique, septembre 2012. hal-lirmm-00737617. Soumis.
4. 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.
5. Ogita (T.), Rump (S. M.) et Oishi (S.). – Accurate sum and dot product. *SIAM J. Sci. Comput.*, vol. 26, n6, 2005, pp. 1955–1988.
6. URL = `http://icl.cs.utk.edu/papi`.
7. Parello (D.), Langlois (P.) et Goossens (B.). – *Sur la reproductibilité des mesures des performances d'algorithmes numériques avec PerPI*. – Rapport technique, décembre 2012. hal-lirmm-00762024, Soumis.
8. URL = `http:www.pintool.org`.
9. Rump (S. M.). – Ultimately fast accurate summation. *SIAM J. Sci. Comput.*, vol. 31, n5, 2009, pp. 3466–3502.
10. Rump (S. M.), Ogita (T.) et Oishi (S.). – Accurate floating-point summation – part I : Faithful rounding. *SIAM J. Sci. Comput.*, vol. 31, n1, 2008, pp. 189–224.
11. Weaver (V.) et Dongarra (J.). – Can hardware performance counters produce expected, deterministic results ? *In : 3rd Workshop on Functionality of Hardware Performance Monitoring, 2010*, pp. 1–11. – Atlanta, USA, 2010.
12. A reference implementation for extended and mixed precision BLAS. – `http://crd.lbl.gov/~xiaoye/XBLAS, .`
13. Zaparanuks (D.), Jovic (M.) et Hauswirth (M.). – Accuracy of performance counter measurements. *In : IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA*, pp. 23–32.
14. Zhu (Y.-K.) et Hayes (W. B.). – Correct rounding and hybrid approach to exact floating-point summation. *SIAM J. Sci. Comput.*, vol. 31, n4, 2009, pp. 2981–3001.
15. 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.