ANNEXE C : Journal

Cette annexe contient le "fichier de log", dans lequel l'activité et les modifications du programme sont notées, tout comme les idées et les problèmes rencontrés, afin d'en garder une trace écrite pour ne pas refaire les mêmes erreurs ensuite et conserver les morceaux de codes valides hors du fichier source.

L'activité a été notée entre le 12 avril 1999 au 28 janvier 2000, le journal n'est donc pas exhaustif mais il a permis de garder la tête froide lors des phases les plus difficiles de la programmation. On y retrouve l'histoire et les détails absents du corps du mémoire pour ce qui concerne l'affichage, le codage, les listes de modifications, l'équation de détection...

```
vbe43.asm:
montre les bizarreries de la prédiction statistique de branchements.
premier minimum à strip=6 puis ensuite tous >=15.
les autres sont étonnament désordonnés.
Ces aléas de prédiction seront "cachés" par d'autres mécanismes
plus importants comme le calcul et l'accès à la mémoire.

t01.asm: montre que la vidéo est un goulet d'étranglement.
le meilleur strip count est 1, mais les strip > 15 convergent
vers cette valeur.

t02.asm: montre l'intérêt de l'affichage à la fin de la fenêtre:
décroissance monotone (1/x) du temps par cellule.

Un affichage utile et rapide est donc possible avec le strip mining.

rb07.asm: montre l'intérêt d'une fenêtre glissante. lorsque sa
taille est > à la L2, la courbe remonte. ma théorie est donc confirmée.
au niveau du speedup, on gagne d'un facteur 10 pour une taille de
950*730 environ (STRIP=19). peu d'opérations sont effectuées cependant,
et l'affichage est lourd (la version finale ne sera qu'une copie de bloc
vers la vidéo).

rb08.asm: idem mais sans affichage, même conclusion, mais speedup
moins important par rapport à STRIP=1 (1:5)

sp02.asm: on peut observer le comportement pour des tailles TRES importantes,
comme prévu le STRIP descend vite. time for a huge CRAY.
la solution: un processeur par "bande".

6 juin 99:

sp06.asm: l'image im000003.bmp est la première image valide
que j'ai réussi à générer.
l'essai porte sur une taille raisonnable, plus grosse que la L2.
un speedup de 6,7 (2327006/346362) est obtenu avec une boucle interne
assez raisonnable (translation horizontale) et affichage à taille normale.


Il faudra étudier l'influence de:
- la hiérarhie mémoire (PMMX contre PII: résultats étonnants)
- la variation de la courbe de speedup avec la taille, hauteur, largeur.
- l'affichage (influence du zoom)
- l'alignement (enlever les "align")
- etc


im000001.bmp: expérience sur PMMX, 3072*2880 (8,8MB).
la courbe montre le creux de la L1 à strip=5, puis une décroissance
plus timide pour la L2. Par rapport au PII, le goulot d'étranglement
de la mémoire est important car la L2 est externe. ccl:
un strip petit est suffisant pour un PMMX, ou du moins quand la L2
est externe. l'expérience sur PII montre que la cache interne, par sa
vitesse (bande passante), et son relais par la L1, favorise des strips
plus nombreux, jusqu'à atteindre la limite de la L2.

Dans tous les cas, le balayage linéaire (double boucle) est déconseillé.
STRIP=1 montre bien un pic important. De plus, l'affichage simultané
augmente la pénalité. Il faudra voir ce qui se passera quand le calcul sera
plus lourd.

Remarque: pour un gros tableau de 8,8MO, on arrive à
8847360/8184118=1.08104... cycles par cellule. à comparer avec
d'autres programmes. le calcul n'est pas effectué, mais l'affichage et
une partie de la translation est opérationnelle. on devrait "tuner" le
code (alignement, pipeline scheduling, etc) pour grapiller une ou deux
décimales mais on n'a pas le temps. ce sera pour une autre "rebuild".

MESURES ET CONDITIONS:
1 cycle=5ns (200MHz) sur PMMX avec 256Ko de L2 et SDRAM
fichier:        taille:    strip: temps/tableau temps/strip=1
t0001.bmp:  256*256 (1/16M)  32      33822         129203
            384*384          27      70797         273786
t0003.bmp:  512*512 (1/4M)   21     123735         464412
```

```
            768*768          15     301074        1365041
t0005.bmp: 1024*1024 (1M)    14     636469        3620042
           1536*1536         9     1596089        7996253
t0007.bmp: 2048*2048 (4M)    7     3309295       14186996
           3072*3072         5     8644870       31753869
           4096*4096         3    20248441       55846489
Les résultats sont faibles à +/- 1% (ordre de grandeur des variations)
Les mesures ont été faites par la boucle de calibration à zoom=4:1
(-> variation de la quantité de données à afficher entre les != tailles)

CALCULS:
 taille:  speedup larg*strip cycle/cel vitesse  v. sans SM
 256*256  3,82008    8192     0,51608  387Mc/s   101Mc/s
 384*384  3,86719   10368     0,48012  416Mc/s   107Mc/s
 512*512  3,67246   10752     0,47201  423Mc/s   112Mc/s
 768*768  4,55391   11520     0,51044  391Mc/s    86Mc/s
1024*1024 5,68769   14336     0,60698  329Mc/s    57Mc/s
1536*1536 5,00990   13824     0,67651  295Mc/s    59Mc/s
2048*2048 4,28701   14336     0,78899  253Mc/s    59Mc/s
3072*3072 3,67315   15360     0,91604  218Mc/s    59Mc/s
4096*4096 2,75806   12288     1,20690  165Mc/s    60Mc/s


Speedup:
--------
Le speedup n'est pas lié à d'autes facteurs, et n'est pas
un critère de performance absolu, mais relatif, permettant
de comparer les algorithmes. Il montre que l'algorithme de
trip mining est au moins deux fois supérieur au balayage
linéaire. Il est probablement influencé par le ratio
de fréquence entre le processeur et le bus: 200MHz par rapport à un bus
à 64 bits @ 66MHz (rapport triple de fréquence). Le taux de transfert
théorique de 66*8=528 MO/s est le double de la "vitesse" moyenne,
car avant de charger une ligne dans la cache, il faut libérer de la place
en écrivant une ligne vers la mémoire centrale. Mais comme une partie
des opérations peuvent s'effectuer dans la cache interne,
la vitesse "apparente" (avec le strip mining) est supérieure
à 528/2.

Le transfert maximum correspond à la taille 512*512=256KO,
c'est à dire: la taille de la L2. La "ventilation" avec la mémoire
centrale est donc minimale. Un algorithme de strip mining de
2ème "ordre" est donc envisageable mais des nombreuses contraintes
rendent le gain trop "cher", sauf dans le cas d'accès à
une mémoire de masse lente (disque dur par exemple). Le speedup
encore accessible à ce moment reste du côté du calcul.

La 'vitesse' sans strip mining est divisée par deux lorsque la
taille du tableau est supérieure à la L2: de 112 à 57.
l'algorithme de LRU n'est pas efficace pour gérer des tableaux
plus larges que sa taille. un algorithme de remplacement aléatoire
rendrait par contre le strip mining plus "statistique", moins
déterministe. un mécanisme de "gel" de lignes de caches serait
dans notre cas très efficace !. Q: la diminution de la vitesse
par deux est-il dû à deux transferts au lieu d'un, ou à la latence
de la mémoire ?

Q: comment faire pour augmenter la vitesse de calcul des très
gros tableaux sans complications ? cache warming ?
faut-il mettre un WBINVD à la fin d'un calcul de tableau ?

larg*strip: largeur * strip count
---------------------------------
la valeur converge vers une valeur de 87% de la taille de la cache L1.
cela veut dire que l'algorithme de strip mining fonctionne comme prévu.
Il serait intéressant de connaître quelles données sont présentes en
cache pour "pousser" le 87% vers le 100%. C'est très probablement les
données correspondant aux ligne supérieures et inférieures dans lesquelles
le programme écrit lors de la phase de déplacement. Il faudrait donc
considérer la formule 'larg*(strip+2) à la place. Avec la "vitesses de
traitement", cette grandeur larg*(strip+2) est probablement une bonne
manière de juger (en temps réel dans le programme) si le strip count est
adapté, et sinon de savoir dans quel régime le programme tourne
(<L1:sous-utilisation, +/-=L1:OK, >>L1:L2).
```

```
cycle/cel: nombre de cycles de processeur pour une cellule (un octet)
---------
la valeur est bonne, comparée à un algorithme par "octets". On pourrait
même comparer cette valeur au temps nécessaire pour consulter
la table des collisions. Pour que le programme reste compétitif,
il faut que le temps de calcul soit inférieur au temps de déplacement
dans l'algorithme par octets. chaque médaille a son revers.

vitesse: nombre de cellules (octets) transférés par seconde
-------
La vitesse minimale du tableau (253 millions de cellules par seconde)
est largement supérieure à mes espérances, et la pointe à 423 Mc/s
est une heureuse surprise. j'ai l'impression d'avoir un superordinateur
alors que ce n'est qu'un banal PC qu'on peut trouver dans un bureau.
Et encore, dans les entreprises, ils changent leurs ordinateurs assez
souvent.
Mais attention ! le calcul en lui-même n'est pas effectué. la "vitesse"
de calcul va chuter dramatiquement avec des lois de collisions "saturées".


v. sans SM: vitesse sans strip mining
----------
elle est assez faible comparée à celle avec strip mining, mais
elle correspond plus à la vitesse "normale" d'un tel programme.
il y a deux raisons qui pondèrent la fiabilité de ce nombre:
 l'affichage occupe une grande partie du temps de transfert vers l'extérieur,
et le mécanisme de balayage agit de manière plus complexe que
ce qu'il effectue vraiment (for(i=0;i<1;i++){(calcul)} revient à (calcul) ).
en fait, c'est un cas particulier de l'algorithme de strip mining,
qui correspond dans la tâche effectuée à un algorithme linéaire,
mais qui n'est pas fait pour cela.


J'estime qu'une partie du travail a été accomplie. la cible de ce type de
programme étant la famille Pentium sur socket (5ème génération de processeurs)
je ne m'avance pas plus dans ce domaine. Le PII avec sa cache rapide
et transactionnelle dans le même boîtier crée des effets plus complexes
qu'il sera intéressant d'analyser : l'influence de la largeur des lignes
reste à définir mais la L2 fait _presque_ office de L1 pour les très gros
tableaux. Il faudra donc explorer les deux méthodes de biprocessing
(vertical/horizontal) pour trouver la meilleure.

La méthodologie de "l'extérieur vers l'intérieur" (de la boucle de
calcul): on programme les opérations annexes pour qu'elles prennent
un temps minimal. par exemple: les boucles de balayage sont "tunées"
avant de mettre qqc dedans, qui sera lui aussi tuné avant d'être
complet. Ainsi, l'intérieur de la boucle est dans des conditions
autant optimales que possible:
-pmode32 -> mode FLAT 32 bits
-interface -> occupation mémoire
-boucle de balayage -> prédiction de branchement, alignement des données
-translation -> alignement, scheduling
-calcul -> scheduling et opérations minimum.

avec une méthode de conception "de l'intérieur vers l'extérieur",
l'extérieur a souvent tendance à "s'encrasser" d'opérations
peu importantes qui pouraient être effectuée dans des niveaux
extérieurs par exemple.

De toute manière, lorsqu'un problème se pose avec la première
méthode, comme un problème d'allocation des registres, il est
souvent très facile de réécrire la partie délicate afin de
"repartir sur des bonnes bases".

Vient maintenant la partie plus complexe de la translation verticale !



Mesures sur le PII:
on aurait du faire des mesures intermédiaiares sur le P200MMX.

PII biprocesseur (mode monoprocesseur) 266 MHz
```

```
MGA millenium AGP
64MO SDRAM

fichier:         taille:  strip: temps min. temps max (strip=1) remarques
tst\PII0000.bmp  256*256   32        38889        106120
rem: <L1, STRIP "saturé"
tst\PII0001.bmp  384*384   29        84737        237022
   efficacité maximum (strip<=32)
tst\PII0002.bmp  512*512   23       149876        411926
tst\PII0003.bmp  768*768   18       394849       2160390
tst\PII0004.bmp 1024*1024  14       790396       4868350
tst\PII0005.bmp 1536*1536  10      2072506      11023138
tst\PII0006.bmp 2048*2048  07      4473040      19606746
rem: La courbe redescend un peu
tst\PII0007.bmp 3072*3072  32     10688105      44295856
rem: L2, strip saturé, la courbe vient de dépasser le minimal local de la L1
tst\PII0008.bmp 4096*4096  32     18935142      77168892
rem: L2, strip saturé
tst\PII0009.bmp (idem pour la valeur maxi)

 taille*freqCPU/tps

calculs:
 taille:  MO(Ms)     speedup         larg*strip  temps/tableau     freq
          (x*y)  (strip max/strip=1)            tps min/freqCPU 1/precedent
 256*256  0,0625     2,7283            8192       0,0001458 s     6857,1/s
 384*384  0,1406     2,7971           11136       0,0003177 s     3146,9/s
 512*512  0,25       2,7484           11776       0,0005620 s     1779,2/s
 768*768  0,5625     5,4714           13824       0,0014806 s      675,3/s
1024*1024 1          6,1593           14336       0,0029639 s      333,3/s
1536*1536 2,25       5,3187           15360       0,0077718 s      128,6/s
2048*2048 4          4,3833           14336       0,0197739 s       59,6/s
3072*3072 9          4,1444           98304       0,0400803 s       24,9/s
4096*4096 16         4,0754          131072       0,0710067 s       14,1/s

(suite)
 taille:     cycle/cel   vitesse    v. sans SM
            tps/taille taille*freq
 256*256      0,5933    449Mc/s       164Mc/s
 384*384      0,5746    494Mc/s       165Mc/s
 512*512      0,5717    466Mc/s       169Mc/s
 768*768      0,6694    398Mc/s        72Mc/s
1024*1024     0,7537    353Mc/s        57Mc/s
1536*1536     0,8784    303Mc/s        57Mc/s
2048*2048     1,0664    250Mc/s        57Mc/s
3072*3072     1,1325    235Mc/s        56Mc/s
4096*4096     1,1286    236Mc/s        57Mc/s

Les données sont plus complexes à décortiquer mais on peut remarquer
certaines choses:

Vitesse: on peut penser que la L2 est à la moitié de la fréquence de la CPU:
décroissance de 500Mc/s à 250Mc/s (en gros)

fréquence du bus: 66MHz=57MB/s*1,16, or l'affichage nécessite 1/16 de la
bande passante (zoom 4:1, donc 4*4 moins de données à afficher)
On remarque ainsi que le calcul de l'affichage n'est pas encore un problème.


On peut considérer 3 phases lorsque la taille et la largeur croissent:
1-la taille est inférieure à la L1
2-La taille est supérieure à la L1 mais inférieure à la L2
3-la taille est supérieure à la L2

lorsque une des tailles (L1, L2) est dépassée, la performance
chute rapidement car le type de balayage ne respecte pas beaucoup le
mécanisme de LRU, de par le balayage en colonnes. Cela explique probablement
le pic du speedup à 6,1 lorsque le type de mémoire utilisée changeait.


Speedup:
le début à 2,7 environ fut une surprise, mais comme la mémoire utilisée
restait en L2 et que celle-ci est accédée plus vite que sur le PMMX,
et que les calculs "normaux" se font sur des dimensions plus grandes,
```

```
le gain de plus de la moitié reste satisfaisant.
à 1MO, le pic de 6,1 n'est pas encore bien compris. mais ce qui compte
reste la "vitesse" de calcul effective, pas l'accélération par rapport
à un code qui ne sera pas utilisé.
à partir de 4MO, le speedup est de 4 et semble se stabiliser. On tourne
alors entièrement avec la L2. Le speedup diminuera probablement si
la taille de la fenêtre dépasse la taille de la L2.

Largeur*Strip:
mêmes interprétation que sur le PMMX sauf que la L2 permet de
s'adapter lorsque le calcul ne tourne plus en L1. On imagine donc
que Larg*strip va converger vers 256Ko.


temps par tableau:
Il faut un temps très court pour accéder à tous les éléments du tableau
grâce au strip mining. La vitesse de rafraichissement est donc très
rapide. Attention cependant, car plus le tableau est grand, plus le
temps de circulation augmente. le temps d'établissement d'un fluide
peut être arbitrairement donné par 3*X*X*Y temps de cellule, donc une
accélération de 4 permet de traiter un tableau seulement sqrt(2)*plus
grand à même vitesse.


freq. d'un tableau:
attention, cette valeur est plus grande que la fréquence de rafraichissement
de l'affichage. il faut fixer STRIP=1 pour avoir un affichage à chaque
cycle, mais c'est environ 4*plus lent. un tableau de 512*512 est affiché
environ à la fréquence du moniteur: 70Hz*23 strip=1610 tableaux, proche
de la fréquence de 1779 calculée. Un tableau de 1024*1024 est affiché
à 333/14=23,78Hz, ce qui est proche de la fréquence d'une pellicule de cinéma.


Cycles CPU pour une cellule:
le temps augmente de manière presque monotone de 0,6 à 1,2.
cette augmentation correspondant à l'augmentation de la taille
du domaine de calcul, on peut conclure que nous sommes toujours
en "memory bound". Les calculs de vitesse le montrent clairement.
La CPU passe donc beaucoup de temps à attendre la mémoire centrale.

De plus, la fréquence d'horloge est plus grande et le processeur
a besoin de nombreux étages de pipeline, le CPI augmente par rapport
au PMMX avec ses 5 ou 6 étages. Un cache miss est donc plus pénalisant.
Heureusement la grande cache L2 recolle un peu les pots cassés.


Vitesse:

Les valeurs sont comparables en ordre de grandeur, ce qui est
surprenant car les architectures ne sont justement pas comparables.
Le point commun reste le goulet d'étranglement numéro un: le
bus d'accès à la mémoire large de 64 bits seulement à 66MHz.
Il est étrange de dire qu'un Pentium MMX à 200MHz est aussi
puissant qu'un Pentium II à 266 MHz et pourtant les chiffres
le prouvent: ils sont similaires. La seule explication est que
FHP EST MEMORY BOUND sur ce type de plateforme.
Je ne suis donc pas sûr que l'utilisation du deuxième processeur
soit utile si le code reste memory bound, car les deux compères
partagent le même bus. Il faudra donc que le code soit très lourd
en calcul pour que l'option du multiprocessing soit intéressante.


remarque: la légère remontée de la vitesse pour 4096*4096 vient
certainement du fait que l'affichage n'est pas complet, seuls
trois quarts sont affichés (zoom=4:1, donc 4096*3040 max.).
cela suffit à modifier de quelques % les résulats.



Conclusions:

Les chiffres ont parlé.
```

```
L'algorithme de strip mining avait été prévu pour manipuler des tableaux
de la taille de la mémoire centrale à la vitesse de la cache L1,
pour des processeurs de type Pentium. Les mesures montrent que le calcul
s'effectue à la vitesse prévue lorsque le tableau tient dans la L2.
Or le PII a une architecture différente qui permet effectivement d'atteindre
ce but : le pic local de la courbe de performance est oublié pour
un minimum plus important correspondant à la cache L2.
Le PII associe donc la capacité de la L2 avec la vitesse de la L1
grâce à des bus séparés. Mais la vitesse est alors limitée par celle
de la cache L2, deux fois plus lente. Il faudrait un Xeon, cinq fois
plus cher au moins, mais avec une L2 encore plus grande et à la vitesse
du processeur, pour éviter ce désagrément.

La performance atteinte avec le strip mining est atténuée pour l'utilisateur
par le fait qu'il faille attendre plusieurs secondes pour traiter le tableau
immense avec un nombre de strip maximal. Pour visualiser un événement
de dimension temporelle fine, l'utilisateur peut toutefois revenir
à un stip count plus faible mais avec une performance diminuée.
Son intension est alors d'observer, non de calculer, donc
cela n'est pas important.

Les mesures ont prouvé que les programmes FHP sont "memory bound",
et que le strip mining apporte une solution utile car le speedup
en bande passante est compris entre 2,7 et 6,1, avec une moyenne
de 4 environ sur les deux machines mesurées. Le goulet d'étranglement
de la mémoire centrale n'est pourtant pas levé, et les PC de bureau
ne vont pas être améliorés avant longtemps. La technologie RAMBUS
par exemple fonctionne à 800MHz mais s'accompagnera probablement
d'un bus moins marge.

FHP étant "memory bound", un calcul sans strip mining aurait donc pu
profiter de la latence de la mémoire pour effectuer les calculs
booléens complexes. Les calculs peuvent donc être plus complexes
et moins optimisés. Mais le strip mining change le problème en
"CPU bound" pour les petites simulations, lever cette difficulté
est le prochain but à atteindre.

FHP étant "mémory bound", il se pose aussi le problème de
savoir si l'usage d'un deuxième processeur peut encore améliorer
les performances dans l'architecture considérée ici. La réponse est
positive a priori, si le code devient "CPU bound". De plus, le
chargement des données se faisant par paquets qui sont ensuite traités
dans les mémoires caches, il est probable que les deux processeurs
"entrelacent" leurs requêtes vers la mémoire lorsqu'un processeur
est en phase de calcul.

La boucle de recherche exhaustive permet de trouver des solutions
imprévues, donc les meilleures dans n'importe quels cas, pour toutes les
plateformes. Un strip mining plus complexe ne devient intéressant
que pour des mémoires énormes et très lentes car cela nécessite
beaucoup plus de stockage temporaire. L'organisation de la mémoire
en niveaux de caches de plus en plus lents et de plus en plus larges
et de moins en moins chers est basée sur le postulat que les données
ont une localité temporelle et spaciale. Or les automates cellulaires
et le modèle FHP ne correspondent pas à ces critères et sont peu efficaces
sur les architectures classiques. le Strip mining permet de diminuer
l'impact du balayage sur les performances, en réorganisant les données
pour augmenter leur localité spatiale et temporelle. Il faut noter
que le cas des automates cellulaires n'est pas isolé, et que des
application courantes balaient linéairement des tableaux plus grands
que leur mémoire cache. Ainsi, l'inversion de matrices est un cas
important pour le calcul scientifique qui a fait l'objet d'optimisations,
en particulier dans le compilateur des processeurs PowerPC qui dispose
d'une option pour effectuer une transformation de la boucle de calcul.

Le but de ces mesures est principalement de trouver la "barrière"
supérieure des performances, afin de se donner pour objectif
de ne pas descendre torp en-dessous lorsque le calcul sera en place.
Une chute de 90% des performances par exemple montrera que le calcul
est trop lourd ou mal effectué. De plus, la boucle de calcul n'a pas
encore été optimisée. L'optimisation peut améliorer la performance
en crête mais pas la performance sur les tableaux immenses (16Mo),
on peut donc conclure que l'optimisation fait partie de la partie
"CPU bound".
```

```
voilà.


9 juin 1999:

WBINVD placé au début de la boucle de calcul:
sur un tableau de 4096*4096, on constate une amélioration
de 0,3%, alors que sur un petit tableau de 256*256, il y a un
ralentissement de 0,8%. Les avantages et inconvénients semblent
trop minces pour s'y intéresser maintenant.


Remarques pour la programmation sur x86: c'est vraiment
l'architecture la plus difficile à programmer et à optimiser,
car les règles de codage sont beaucoup trop nombreuses et
inutiles. non seulement les instructions sont à 2 opérandes
mais en plus il y a peu de registres. La pression sur l'allocation des
registres est très fortes. le processeur Alpha est bien supérieur, non
seulement pour son bus plus large et plus rapide, mais par son jeu
d'instruction plus cohérent à 3 opérandes, ses registres plus nombreux et
plus larges, et ses règles de codage simples. En plus, il peut
effectuer quatre opérations par cycle conte 3 pour le PII et 2 pour le PMMX.
(en crête...) A même fréquence d'horloge, le processeur ALPHA est donc bien
supérieur aux processeurs x86.


Optimisation de la boucle interne, déplacement horizontal:
la méthode est changée. il y avait un grand nombre de dépendances
de données. C'est dur aussi de coder avec les contraintes du PMMX et
du PII à la fois ! Les deux codes de déplacement ont été optimisés
et entrelacés. résultat :
taille  strip min.   max
512*512  22   116370 446978

il y a une légère différence de strip count.

speedup min: 123735/116370=1,0632
6% c'est déjà mieux. On atteint alors 450 Mc/s !
Ce sera la meilleure "vitesse" atteinte car les
modifications à venir la diminueront.

speedup max: 464412/446978=1,039

4% c'est pas trop mal, mais je me demande comment faire mieux.
la solution est problement un "cache warming" mais le PMMX
ne le supporte pas. Et surtout, j'ai passé du temps à optimiser
le code mais il n'y a pas autant de gain qu'escompté.


essayons avec une taille moins "memory bound":
taille:  strip: min:  max:
235*256  32     31795 127187

speedup min=1,0637521 : idem.
Le Pentium est vraiment plein de mécanismes lents à se mettre en branle.

speedup max=1,01585
1,5%: ça devient ridicule. mais c'est vraiment toujours memory bound.

_________

essayons sans accéder aux données:
8 instructions de (dé)chargement ont été supprimées.
256*256, strip=32, min=24679 et max=98330
29% du temps est donc utilisé pour les accès à la mémoire.

il y a 21+8=29 instructions dans la boucle, soit 38%.
la conclusion serait donc d'avoir moins de calculs.

_____________

essayons: 18 instructions ont été mises en commentaire,
les 3 restantes servent à contrôler la boucle. les instructions
```

```
qui accèdent à la mémoire ont été rétablies.

taille: 256*256, Strip=32, min=21857, max=124469

max normal/max sans calcul=127187/124469=1,021836
min normal/min sans calcul=31795/21857=1,45468

"max" est mémory bound sans l'ombre d'un doute.
lorsque le calcul s'effectue dans la L1, il occupe "seulement"
45 % du temps. Cela contredit les observations des mesures précédentes
(sans accès à la mémoire) mais il faut remarquer que la mémoire est
toujours accédée pour interpréter les listes de modification.

Conclusion: on peut rajouter des calculs dans la boucle sans trop
influencer les performances. Le mouvement des bits au moins ne semble
pas poser de problème, on verra ensuite pour le calcul des collisions.

--------------

Il faut aussi remarquer que dans la mesure précédente,
l'affichage était aussi effectué, ce qui explique le "saut"
de 2 à 45% lorsque le strip count diminue, donc que l'affichage
augmente. l'affichage non seulement accède aux données mais aussi à
la mémoire vidéo. réessayons en désactivant l'affichage et
la liste de modification:

taille=256*256, strip=32, min=14906, max=38421

cela semble plus raisonnable mais je ne comprends pas le rapport
de 2,5 entre les deux nombres, il ne devrait pas y avoir de différence
notable. Peut-être s'agit-il de l'utilisation de la pile ? Mais le calcul
prend 46% du temps en "CPU bound". Il reste donc de la place pour rajouter
du calcul. Le rapport avec max est encore plus optimiste : 127187/38421=3,31.
Le calcul ne prend plus qu'un tiers du temps en "memory bound".


---------------- fichier source renommé SP08.asm ---------

Intro: Les ordinateurs modernes sont des êtres furieusement non-linéaires.
Ils sont devenus tellement complexes qu'il est très difficile de
prévoir les performances d'un code même très simple.

Le strip mining est une méthode de réorganisation des données et du balayage
pour les très gros tableaux et pour des calculs qui possèdent une certaine
localité de traitement. Cependant la mise en place de balayages complexes
suppose une complication du code et demande de nouveaux buffers. Un
déplacement horizontal de bits a pu être mis en place car il est
perpendiculaire au sens de déplacement de la fenêtre de strip mining,
et n'apporte pas de complication notable. Le déplacement vertical
est par contre plus beaucoup plus compliqué car il nécessite un buffer.

Le déplacement vers le haut est encore assez simple puisqu'il est
dans le sens inverse de la fenêtre. le problème se résume alors à
une variable temporaire qui doit être initialisée et vidée correctement
hors de la boucle interne. le résultat est dans SP08.asm.

---------------- fichier source renommé SP09.asm ---------

Le buffer temporaire pour le balayage et l'affichage :

Le graphe des dépendences des données a montré que pour chaque colonne balayée
il faut un buffer aussi long que cette colonne. ce buffer est cependant accédé
assez rarement, deux mots de 64 bits à la fin de chaque balayage de colonne.
Le buffer nécessaire à cette fonction sera entrelacé avec un autre buffer
qui mémorisera les résultats à afficher (densité, vorticité...) constitué
par chaque calcul de cellule.

Grâce au changement du sens de balayage de la double boucle interne,
c'est à dire vertical à l'intérieur, le buffer horizontal (contenant une
ligne) des programmes des premières générations a été supprimé pour
la boucle interne, mais les dépendances de données existent encore
et il faut en tenir compte lors des balayages verticaux. On ne peut plus
utiliser la première garde horizontale pour le buffer circulaire de ligne.
```

```
Il faut remarquer que même si le buffer nécessaire est par nature
bidimensionnel, il est facilement linéarisable avec un buffer
circulaire simple. Son organisation a été choisie pour diminuer son
impact sur les performances, en occupation de mémoire cache et
en instructions nécessaires à son traitement. Ainsi, même s'il nécessite
plus de place mémoire, ce buffer n'est pas beaucoup plus complexe et gourmand
que les buffers temporaires des premiers programmes.

Chaque "cellule" de ce buffer comprend 10 mots de 64 bits: 2 pour le buffer
vertical (de colonne), et 8 pour les 64 octets à afficher, soit 80 octets.
Il faut d'abord trouver de l'espace pour contenir ((CEL+16)*XSIZE)*32
octets. On le place au-dessus du tableau de calcul, afin d'éviter
les problèmes d'allocation mémoire avec MSDOS. En plus, la mémoire
MSDOS ne peut pas en général contenir des tableaux de plus de 500Ko,
ce qui est pourtant probablement nécessaire pour des très grosses
simulations sur un PII notament. La formule de l'espace occupé est donc
modifiée: TAILLE= ((XSIZE+CEL)*(YSIZE+2))+(((XSIZE*4)+XSIZE)*512) octets.
Le buffer est géré comme un buffer circulaire classique dont la taille
est ((CEL+16)/CEL)*STRIP*XSIZE. Le pointeur de début est décrémenté (au besoin)
avant de lancer le calcul en double boucle, dans lequel un autre pointeur
augmente (modulo TAILLE) à partir du pointeur de début. Un buffer circulaire
est nécessaire à la fois pour réduire au minimum le déplacement des
blocs d'informations (on change le pointeur à la place) et l'occupation
en mémoire (et l'occupation en cache par la même occasion).

L'organisation des cellules est directement liée à la réduction du nombre de
pointeurs, car il ne reste plus qu'un seul registre disponible pour pointer
(ESI). Or, le buffer remplit deux fonctions qui sont différentes de par
l'occupation et le type de balayage qu'elles effectuent. Le buffer de balayage
vertical est accédé une fois à la fin du balayage d'une colonne, alors
que le buffer d'affichage est accédé pour chaque cellule. C'est donc cet usage
qui prime sur le premier d'un facteur 4/5. Le balayage doit donc s'effectuer
dans l'ordre privilégiant la localité maximale du buffer d'affichage.
La perte en bande apssante et en occupation utile de la mémoire cache est
d'un peu moins d'1/5 car le buffer vertical n'est utilisé qu'à la fin des
colonnes. Les données sont cependant convoyées avec celles du buffer
d'affichage, ce qui évite 4/5 des "cache miss" dans ce cas. Cela ressemble
à du "cache warming" mais est dû à une restriction forte du nombre de
registres pointeurs.


Une fois que la fonction et la gestion du buffer sont dédinies, ont peut
mettre au point le balayage dans ses détails. L'approche est de "suivre"
le cheminement des données le long des XSIZE colonnes de STRIP mots,
afin de profiter au maximum de la bande passante (car les blocs ne sont
pas alignés sur 32 octets).
Les colonnes ayant plus de localité que les lignes, elles sont ainsi
la structure de base du tableau. A l'image du balayage du tableau central,
le balayage du tableau temporaire est orienté colonnes.

Le tableau est géré comme un buffer circulaire, afin de ne pas déplacer
de données mais les pointeurs associés. Cela est aussi dû au fait que
le balayage du tableau principal a un démarrage progressif, il faut faire
attention pour que le balayage effectué avec un certain strip count
(au démarrage ou à l'arrivée) puisse fonctionner au cycle suivant lorsqu'il
augmente ou diminue. Ce sont des effets de bord importants qu'il faut
gérer correctement. La stratégie du pointeur sur la pile résout facilement
ce problème et est expliqué dans les paragraphes suivants.

Au cours du balayage, le tableau temporaire est parcouru linéairement,
en "suivant" le balayage du tableau principal. Mais au démarrage et
à l'arrivée, lorsque la taille balayée change, il faut "sauter" des cellules.

L'approche est "externe->interne" car on met en place la boucle externe
d'abord. Elle gère l'aspect du strip mining de "haut niveau", dans
son principe, du point de vue de la fenêtre de calcul glissante.
Ensuite, la boucle secondaire gère le buffer de balayage vertical.
Enfin, la boucle interne gère le buffer d'affichage.
Il faut noter aussi que la synchronisation du tampon avec la procédure
de balayage est importante. Elle est assurée par la boucle externe.

Le tampon est balayé par un seul pointeur, ESI, mais dans chaque imbrication
de boucle, il va être sauvegardé sur la pile et modifié de plus en plus
finement:
```

Pour la boucle extérieure, le pointeur est avancé de −80 octets (une cellule
de buffer) modulo la taille du buffer, afin de faire "glisser" la fenêtre
de calcul vers le bas.
Pour la boucle intermédiaire, le pointeur est sauvé sur la pile et avancé à
chaque colonne de STRIP*80/64, modulo la taille du buffer. La gestion du
buffer de mouvement vertical est ici assurée.
Pour la boucle interne, le pointeur est sauvé sur la pile et avancé de
80 octets pour "suivre" le balayage principal.
Il est à noter que, contrairement à la boucle principale, le besoin
en pointeurs en moindre car la boucle de contrôle est déjà en place.
Nous pouvons donc nous servir de la pile tranquillement pour gérer le pointeur.
Néanmoins, un DSP avec plusieurs générateurs d'adresses et un mécansime
de boucle câblé aurait été plus performant et plus simple à programmer
(comme quoi, ce n'est pas seulement un problème de langage de programmation).

Le mécanisme ici décrit permet de limiter au maximum le trafic sur les bus
mémoires, mais occupe au moins la moitié de la mémoire cache dans laquelle
tourne le calcul. Ainsi, le trafic mémoire est augmenté car il y a moins
de lignes de cache libres : on peut donc s'attendre à des performances
diminuées de moitié par rapport aux mesures déjà effectuées. Les mesures
diront si l'algorithme se comporte comme prévu.

0) vider temp

dans la colonne:
1) lecture C
2) temp C −> C
3) calcul
4) resultat−>tempC + 1
5) "decremente" ESI.

il y a besoin effectivement de 2 mots par cellule temporaire,
parce que les résultats n'appartiennent pas à la même génération
du calcul. la perte n'est plus de 1/5 mais le nombre d'opérations
augmente.

nuit du samedi 12 juin:

Aujourd'hui est historique, j'ai enfin réussi à comprendre et à programmer
(le premier étant en fait bien plus difficile que le second) le déplacement
vers le bas. c'est beaucoup plus vicelard que je pensais mais j'ai pris
mon stylo à deux mains et j'ai fait "tourner" les bits sur le papier
jusqu'à bien comprendre ce qu'ils faisaient et en déduire l'algorithme.
la gestion des pointeurs est beaucoup plus simple que je l'avais imaginé,
en remarquant certaines choses qui se dessinaient sur la feuille.

Je l'ai codé avec assez peu de peine, juste un bug idiot avec ESI
poussé sur la pile pour la liste de modification mais pas restoré après...
je n'ai pas mis longtemps à trouver le pot aux roses avec un peu de logique,
car j'ai analysé en partant de l'arrière (du code fautif) les dépendances
de données pour trouver qu'il ne correspondait pas à ce qu'on s'attendait.

A part cela, le déplacement vertical a fonctionné tout de suite ou presque.
il ne faut pas non plus oublier de "vider" le buffer avant de calculer une
fenêtre pour éviter de voir apparaître des particules imprévues, qui sont
en fait les bits qui devraient disparaître en bas et qui "bouclent".

J'ai ensuite mis en place les listes de modifications. avec un peu de peine
car elles commencent à devenir complexes. comme je l'ai pensé depuis des
années, il va falloir fabriquer un programme de génération de ces listes.
Les listes actuelles font la réflexion des particules horizontales et
verticales, mais ce n'est pas parfait car il manque quelques trucs dans
les coins. pour l'instant ce n'est pas trop grave mais il ne faut pas
l'oublier.

J'ai enfin calibré la boucle : comme prévu, les performances sont divisées
par 2. heureusement qu'il y a le strip mining ! un XEON serait aussi le
bienvenu. mais le speedup entre la version "strip idéal" et "strip=1"
est toujours environ 4.

Prochaine étape: les obliques, où on se met (enfin) sur un réseau hexagonal.

```
Il faudra en plus revoir l'organisation générale des tableaux. je pense
maintenant que les gardes horizontales du bas et du haut sont inutiles.
la garde verticale est cependant toujours importante.
<remarque: la direction F écrit vers le haut, donc il faut la garde du haut>

Les premières mesures avec le déplacement vertical indiquent en gros
des performances réduites de moitié. On remarque aussi une redescente plus
nette de la performance lorsque le best strip est dépassé, car il faut
"swapper" en plus le buffer temporaire.

PMMX 200MHz, L2=256Ko
taille   best strip   best cycle    cycles for strip=1
 256*256     20          43269          125163
 512*512     12         162093          450402
1024*1024     7         891808         3148301
2048*2048     3        5355502        13069373

calculs:
strip*largeur:  speedup(relatif):  vitesse crete: vitesse non crete:
   5120             2,89267              302          104
   6144             2,77866              323          116
   7168             3,53024              235           66
   6144             2,44036              156           64

la vitesse lorsque strip=1 n'est pas surprenante et n'a pas changé
par rapport aux autres versions plus "légères" du programme: on est encore
en régime "memory bound".

le strip*largeur ne dépassse pas 7168, soit moins de la moitié
de la cache L1. l'autre moitié est utilisée par le buffer temporaire
et par quelques variables temporaires, sans oublier les cellules
"voisines" accédées lors du déplacement des particules.

le speedup relatif est plus faible, indiquant que nous nous rapprochons
de la limite CPU/memory bound. 2,5-3 en moyenne. mais comme l'indique le
speedup, le strip mining permet toujours de calculer au moins deux
fois plus vite qu'un code classique.

La vitesse crête par contre n'est pas beaucoup altérée, la crête
se situant toujours pour 512*512=taille L2. la meilleure performance
atteinte sur le PMMX étant de 450Mc/s, le processeur ne fonctionne plus
qu'à 70% de la performance crête. On s'attendait plutôt à 50%,
ce qui est une bonne surprise. j'envisage donc une performance
moyenne de 200Mc/s pour les programmes FHP sur le PMMX.

En débutant ce projet sur un Pentium 100, j'envisageais plutôt une
performance de 20Mc/s :-) Il faut dire merci à la plus grande cache L1
(16Ko au lieu de 8)[x2] et au MMX qui rajoute des registres qui permettent
de travailler sur de plus grands nombres (64 bits)[x2]. le processeur
plus rapide et l'utilisation du mode protégé expliquent enfin le dernier
doublement de performance[2x2x2=8]. Par contre, la mémoire est toujours
à 66MHz et 64 bits de large... (supériorité de l'esprit sur la matière ;-D)



Notes: le fichier est maintenant SP10.asm.
NASM en mode 16 bits n'arrive plus à l'assembler, j'ai donc chargé
le nouveau nasm 0.98 en mode protégé pour digérer les 104Ko de fichier source
(bon d'accord, 58Ko quand il n'y a plus de commentaires). le binaire
fait 8Ko et le listing 234Ko. et j'ai fait tout çà à la main...
Je suis à quelques centaines de lignes du but, et ce sont vraiment les
plus difficiles !

La réunion de quelques variables que j'estime accédées le plus souvent,
dans un même endroit, ne donne que très peu d'amélioration environ 1/1000
dans le meilleur des cas. je garde toutefois cette configuration.

La réorganisation de la cellule (affectation de A, B, C,...)
semble plus prometteuse, de l'ordre du %. Il faudra vérifier cela
plus attentivement.


SP11.asm:
je mets en place le déplacement hexagonal en duplicant les déplacements
```

verticaux. ça fonctionne bien car c'est du copier−coller. l'assignation
des registres est un peu embêtante mais ça reste humain.
A et D ne sont pas à modifier, les horizontaux fonctionnent bien,
et des éléments de chacun vont être copiés pour ajouter de la complexité
au balayage.

nuit du lundi 14 juin:

Enfer et damnation ! bonne mère, je suis dans un sacré pétrin.
En mettant en place le déplacement en F (vers le haut à gauche)
je me suis aperçu de comportements très bizarres des particules.
*****
Après pas mal de questions j'ai fait un programme en Pascal afin de
profiter du débugger (qui ne m'a finalement pas beaucoup servi).
mais surtout Pascal est duffisament lent pour voir les choses se passer.

résultat:
− le déplacement vertical vers le bas est OK.
− le déplacement vers le haut est refait et il est plus simple
   que je ne l'avais programmé même si ça fonctionnait.
   Je n'avais pas correctement analysé les dépendenaces de données
   dans le temps.
− le gros problème cependant est la remise en question des
   listes des modifications. Elles ne permettent pas de modifier les
   particules allant vers le haut !
  La solution pour ce problème est d'abandonner la structure prévue et
de mettre en place un mécanisme plus fin, le plus simple étant un pointeur
dans la partie WALL de chaque cellule. le programme en pascal donne un
résultat positif. On peut aussi par la même occasion abandonner la garde
verticale :−)

bref la tempête a fait rage et a fait des dégâts.
Je savais bien que mon truc était possible mais je me suis trompé sur le
comment...

nuit du mardi au mercredi 15 juin:

j'ai remis de l'ordre dans le programme.
les balayages nord, sud, est et ouest fonctionnent.
le sud−ouest est opérationnel. le nord−ouest par contre est
instable pour les strips>1. je me demande vraiment quand je m'en
sortirai. je croyais pourtant que j'avais dompté l'espace−temps
mais chaque pas vers la solution pose de nouveaux problèmes.
je croyais avoir fait le plus difficile alors que le reste à venir est
inconnu parce que pas encore programmé, donc imprévisible. a‹ïe.

j'avais commencé à ébaucher une procédure de conversion
vectorielle−> liste de modification, mais devant la complexité
inutile pour les prototypes je suis revenu à la vieille méthode
du tout cousu main sur mesure. Après de longs cafouillages, les
listes de modification par cellule fonctionnent comme désiré.
c'est maintenant le déplacement hexagonal qui laisse à désirer !
donc, retour à Turbo Pascal pour voir ce qui se passe vraiment.

L'informatique est vraiment un monde étrange où ce qui est complexe
semble plus important que le simple. Il me semble paradoxal de passer
des années sur un code qui tournera quelques heures en tout...
Il me semble paradoxal que les algoritmes les plus complexes
soient les meilleurs. Peut−être parce que les idées, par leurs
abstractions, n'arrivent pas à saisir la complexité du monde.
lemonde est−il un assemblage d'idées ?

je continue ma descente aux enfers. je n'arrive plus à dormir.
c'est plus frustrant que la plupart des problèmes que j'ai eus,
sauf peut−être lorsque j'ai découvert le FORTH, mais
ici l'enjeu est différent. J'ai déjà réécouté tous mes CDs,
je recommence à écouter des cassettes. Je n'ai pas d'échappatoire
car je suis convaincu de pouvoir réussir. je croyais avoir résolu

la plupart des problèmes mais la réalité a rattrapé la théorie.
je reste optimiste car je pense avoir résolu la moitié des
problèmes. mais je reste sans voix devant les nouveaux problèmes
qui se posent. la pression est forte mais j'arrive à tenir car
j'ai peu d'obligations "annexes".

Yannick Sustrac m'a mailé les résultats du programme SP07.asm
(je crois que c'est celui-là, du moins, il n'y avait pas le
balayage horizontal optimisé, donc cela correspond aux
premières mesures de cette page.)
Son ordinateur est un Cyrix (IBM) avec (apparemment) 256 Ko
de cache L2, vidéo S3 sur bus PCI, SDRAM. je ne suis pas sûr du reste.
je ne connais même pas la fréquence réelle de la CPU.
néanmoins on peut déjà exploiter quelques résultats:

```
>> mes resultats (Les strips min ne correspondent pas du tout)
>> fichier:         taille:  strip: temps min. temps max (strip=1) remarques
>> tst\PII0000.bmp  256*256   32        49018        111305
>> tst\PII0001.bmp  384*384   32       101699        275009
>> tst\PII0002.bmp  512*512   32       171817        450830
>> tst\PII0003.bmp  768*768   32       373296       1253318
>> tst\PII0004.bmp 1024*1024  32       668435       3144734
>> tst\PII0005.bmp 1536*1536  31      1465001       6902381
>> tst\PII0006.bmp 2048*2048  30      2618118      12112979
>les deux colonnes de droite ont l'air OK.
>mais je suis scié par la colonne du strip count. il reste tout le temps à 32 !
>ensuite il redescend après 1MO. tu aurais 1MO de cache ?????
>il y a un truc que je pige pas. tu as bien marqué le numéro de la ligne sur
>laquelle le signe || s'est arrêté ? je sais pas trop comment prendre ce
>résultat.
>
>En tous cas, le speedup:
>256²:  2,27
>384²:  2.7
>512²:  2,62
>768²:  3,35
>1024²: 4,7
>1536²: 4,71
>2048²: 4,62
>
```

les nombres de cycles sont du même ordre de grandeur que sur les
autres ordinateurs testés.
Le speedup est similaire à ceux des Intel pour une plateforme a priori
différente à plusieurs niveaux. Le strip mining montre son efficacité sur
les très gros tableaux, qui sont le domaine choisi pour le projet.
Et même sur les petits tableaux, le gain est intéressant.

Après un reboot "forcé", le BIOS a eu un CRC error et je suis retourné
dans le BIOS pour la première fois depuis longtemps. j'ai découvert
qu'il est possible de configurer la SDRAM pour des facteurs dont j'ignorais
l'existence, comme l'entrelacement de banques. je me demande ce que
ça veut dire mais je me doute qu'il y a quelques % de performance à
grapiller de ce côté-là. Comme je ne me souviens pas des anciens
paramètres, j'en ai mis d'autres qui sont probablement bien différents.
les performances d'aujourd'hui ne peuvent donc plus être comparées avec
les anciennes, sauf en ce qui concerne les ordres de grandeur.
Le BIOS devrait être pris en compte dans la caractérisations des
mesures.

J'en suis à SP15.asm.

Nuit du mercredi au jeudi 16 juin:

La solution du problème est finalement venue au petit matin.
Un simple dessin, et je me suis aperçu de mon erreur. Cuisant
échec d'une erreur trop belle, d'une conviction qui a duré quelques
années. Parce que je n'étais pas allé jusqu'au bout du problème avant.

La conclusion finalement c'est que le balayage vertical à l'intérieur
de la boucle obligeait un "saut spatio−temporel" trop grand d'un pas de temps.
La solution est de réorganiser la boucle le plus simplement du monde,
c'est à dire avec la ligne à l'intérieur. Les choses redeviennent
alors presque naturelles, du déjà fait, du connu. juste encore la diagonale
qui est délicate (sud−est) mais le bon sens est efficace.

Je reviens de loin, et il n'y a plus qu'à programmer. Pascal m'aura permis
de tester les algorithmes rapidement, parce qu'il est lent, et parce que
je n'ai pas à attendre l'affichage d'une ligne puisque les données sont
en mémoire vidéo. Cette fois−ci je recommence à reprendre pied.
le reste à venir devrait aller comme du papier à musique, c'est déjà
préparé. Peut−être le passage en multiprocesseur sera plus délicat car
moins préparé, mais il n'y a pas 36 solutions.

Il faut donc réorganiser la boucle principale, puis implémenter le balayage
progressivement pour l'hexagone. au boulot.

SP17.asm: A et D sont en place. la boucle a été réorganisée sans problème,
ce n'était que de la réécriture comme j'en ai l'habitude.

sp18: mise en place des déplacements verticaux.

jeudi 17 juin, 4h15 du matin:
les 6 directions sont fonctionnelles et stables pour tous les strip counts.
il reste encore le problème des parois qui foirent un peu... mais le plus
dur est enfin fait !!!! sp18 est renommé sp19.asm.

512*512, strip=14, best=277675, s=1:528986
512*14=7168, best/s=1:1,905, vitesse max:188Mc/s, vitess s=1:99Mc/s

cela donne un nouvel ordre de grandeur de la vitesse finale.
le déplacement des particules n'est pas optimisé, il est juste posé et
il fonctionne. je ferai le ménage lorsque le calcul sera au point.
Je m'attends aussi à une autre baisse de performance lorsque la sommation
sera au point, car elle utilisera le reste du buffer principal qui, je crois,
n'est pas chargé complétement en mémoire cache (si on prend la granularité de
32 octets).

6h du mat. : corrigé un léger bug pour la direction E. je prends maintenant
le problème des parois à bras le corps, car les "trous" laminent mes beaux
blocs de particules...

vendredi 18, 5H30 du mat:
je me suis réveillé à 2H du mat, et j'ai tout de suite allumé
l'ordinateur. la fabrication des listes de modification n'est pas
évidente mais c'est du niveau d'un projet de licence de Jean Méhat.
C'est en le programmant qu'on comprend mieux le programme, mais
une avancée théorique a été effectuée hier (un peu de bon sens aide parfois)
Lorsque deux segments perpendiculaires se croisent, il faut mettre
un "point dur" (réflichissant total) à l'intersection pour éviter
que des particules entrent "à l'intérieur" du segment. le cas n'est pas
difficile à détecter mais un peu compliqué car il y a un grand nombre
de cas particuliers à prendre en compte.
Un point important n'est pas encore en place: la reconnaissance qu'une
cellule existe déjà autre part. cela permettra de tirer parti au maximum
de la méthode des pointeurs lors du calcul. Pour cela, la recherche de
cellules existantes doit se faire dans le sens inverse de croissance
du buffer de cellules.

10H du mat: je peux faire des points isolé et des lignes verticales
dans certaines conditions. il reste encore 3 cas particuliers à traiter,
ce qui prendra du temps supplémentaire, mais procurera un confort
d'utilisation incontestable pour le programme :−) par contre,
de 1: heureusement que j'ai Turbo Pascal, de 2: ça va être
un beau merdier de passer ça en assembleur, mais de 3: j'ai un algo
raisonnable. pour l'instant.

"Ce travail consiste à inventer et programmer un ensemble d'algorithmes
à applications scientifiques, industrielles, pédagogiques
et ludiques (joujou de luxe ;−D) avec des implications/répercussions
algorithmiques et architecturales (ordinateurs), pour la simulation
interactive de phénomènes de mécanique des fluides."

yannick sustrac wrote:
> >le modèle que j'utilise a un temps entre deux collisions entre particules
> >(le chemin que parcourt une particule entre deux collisions) assez grand,
> >donc le programme passe son temps à "bouger des particules". si on veut
> >optimiser le programme il faut donc commencer par là :−)
> Ca me repelle la simulation N−body c'est fait pour simuler l'evolution des
> galaxies en astrophysique a partir de particules de gaz animées de mouvement.
moi, en plus, je ne fais que des "intéractions locales". en gros,
N−body doit regarder tout le monde, alors que moi, je me contente seulement de
l'état d'une seule zone très restreinte. ce qui accélère vraiment les chauses.

> Il y a un article la dessus ds SCientific−Computing Wolrd (C'est gratis)
> http://www.iop.org/Mags/SCW
V voir... mais ça me dit qqc :−)

> > ensuite une fois que les particules bougent correctement et assez vite,
> Elle bougnet en 2D ou 3D tes particules?
bah en 2 D.
la 3D est vraiment trop chiante à comprendre, car c'est impossible
à faire directement, alors on fait un détour par la 4è dimension et
on "projette" le résultat en 3D. je te dis pas la mémoire que ça prend,
et le temps de calcul en monopraucesseur... sans parler du reste.

> > on rajoute le calcul
> De quoi?
des collisions.
les particules bougent, mais elles intéragissent assez peu.
si elles n'intéragissent pas, ça donne (pour la 1D horizontale)
le programme que tu as. une fois qu'elles bougent correctement,
on les fait intéragir (localement, càd site par site). là, ça devient
"intéressant" pour les simulations. mais tant que ça bouge pas correctement,
tu peux toujours courir, même si ça fait des trucs "marrants".

> >au milieu
> et pourquoi au milieu du calcul du chemin?
le "calcul" se fait en 2 "étapes" principales:
−déplacement
−collision
dans les boucles on effectue le déplacement. une fois que le déplacement
"roule", on peut mettre la collision dans son "cocon", au coeur de la boucle.
sinon, tu peux toujours essayer de commencer par le calcul de collision.
mais faire des collisions sur des particules qui bougent pas, ça
me ferait prendre encore plus pour un idiot qu'on le croit déjà...

> > et le tour est joué. il faudra faire des mesures sur le ratio
> > entre particules bougées et particules
> >qui font une collision avec réorganisation. en tous cas, ça devrait tourner
> >autour de 3 ou 5, donc si on bouge les particules plus vite, le programme
> >tourne plus vite. Et je sais pas si le PIII supporte les instructions
> >booléennes sur 128 bits mais dans ce cas ça ira simplement 2 fois plus vite
> >au moins (sans parler du bus à 133MHz et de la fréquence d'horloge plus
> >élevée :−D)
> Si c'est pour trouver que ca va + vite avec des processuer + recent ...
nan. c'est pour dire que ça va aussi vite que ce que le processeur permet.
avec une technique "classique" tu l'as dans le baba. si tu avais plus de cache,
ça servirait à rien. alors que tu as bien vu qu'avec mon "truc", tu vas
4* plus vite que normalement. sur PIII, tu ne bénéficierait que du bus 2* plus
rapide, alors que ma méthode permet de profiter des autres améliorations.

> >le programme que tu as permet de connaitre les caractéristiques de ton
> >système, et permet de valider ma théorie selon laquelle le strip mining
> >améliore la vitesse d'environ 3 fois par rapport à un balayage linéaire.
> >et un calcul 3 fois plus rapide, ça fait de mal à personne (sauf à ma
> >tête). c'est pour ça que je me casse le cul dessus.
> alors maintenant tu n'a plus qu'a me donner q explication sur ton strip
> mining... et les diff par rapport au balyage lin.

ahem, ahem. je m'éclaircis la voix car c'est un truc assez ... tu vas voir.

Lorsque la mémoire cache a été mise au point, son application a été basée
sur le principe de localité spatiale et temporaire des données.
en gros, il y avait de grandes chances pour que qu'en t+1, les données
accédées soient proches des données accédées en t. C'est valable pour de
nombreuses choses, comme les instructions: surtout les petites boucles.
avec les programmes générés par compilateur, ça marche aussi car toutes
les variables globales sont groupées. enfin il ne faut pas oublier la pile,
intensément utilisée. pour plus de détails, on peut par exemple lire
Patterson Hennessy, disponible dans la bibliothèque de tout informaticien
qui se respecte.

or, moi je veux traiter des tableaux immenses, gros comme la mémoire centrale.
donc si je balaie linéairement tout ce tableau, à chaque frontière de ligne de
cache se produit un cache miss, qui prend du temps à se résoudre, encore plus
si le système est en mémoire virtuelle et que la page n'est pas en mémoire,
et que l'adresse doit être traduite... le problème est que le postulat d'en
haut ne s'applique pas à mon cas.

Pour accélérer le processus, il faut augmenter la "localité spatiotemporelle".
je l'ai fait en réorganisant les boucles. la boucle externe balaie une fenêtre
de calcul qui "glisse" sur le tableau. cette fenêtre contient un nombre
variable de lignes, ce nombre de ligne je l'ai appelé "strip count". Et quand
tu appuies sur le pied à coulisse, on effectue le calcul pour chaque valeur
possible du strip count, de 1 à 32 (mais je vais probablement lever la limite
de 32 vers 64). la calibration consiste à tester toutes les tailles de
fenêtres pour voir laquelle est la plus rapide, objectivement, sans autre
considération, ce qui permet de trouver des résultats inattendus comme sur
ton ordinateur.

voilà, je sais pas si tu as compris mais je vais recopier ça dans mon mémoire
car ça semble assez bien dit.

mail à Yannick Sustrac:

> >> Les resultat que j'ai eu sont differents des tiens:
> >c'est normal puisque c'est un système différent ;−)
> ...tu me prends pour un con voir + loin
nan chte pran pa pour un con. c'est simplement
que tu comparerais jamais un Mac avec un PC, donc on peut pas comparer
"directement" un Intel avec un IBM, sauf pour voir la vitesse de calcul.
ce qui m'a mis sur le Q.

> >les deux colonnes de droite ont l'air OK.
> >mais je suis scié par la colonne du strip count. il reste tout le temps
> >à 32 !
> ... et c'est pour ca que je te disais que les resultats etaient differents!!
tu parles qu'ils sont différents, ils sont mieux que sur PII 266 !

> >ensuite il redescend après 1MO. tu aurais 1MO de cache ?????
> ...croa pô... 512ko
c'est déjà bien quand même !

> >il y a un truc que je pige pas. tu as bien marqué le numéro de la ligne
> >sur laquelle le signe || s'est arrêté ?
> OUI
j'ai vu, mais je ne te prends pas pour un con, c'est seulement que ça
me scie...

> >je sais pas trop comment prendre ce résultat.
> ... ma ca correspond a qoui le STRIP (ou SLIP)
la taille de la fenêtre de calcul.
pour voir sa taille en octets, tu multiplies le STRIP par la largeur du
tableau.

> >tu disais que tu as un clone IBM (cyrix) ? dans mes archives j'arrive pas à
> >trouver où tu m'en parles... la vitesse du bus (tu sais, les jumpers sur la
> >carte mère) la taille et le temps de réponse des RAM et de la L2...
> V−bus 66Mhz
> RAM 98Mo EDO/100ns
> L1 32ko (Le pentium MMX n'en a que 16. c'est peut être la la difference)

```
je sais que c'est 16Ko données + 16Ko instructions.
je ne sais pas si c'est la même chose pour toi.
> L2 512ko 7ns
je crois que c'est ce dernier paramètre qui fait toute la différence...

> >> C'est tres nul comme saisie des tailles de tableau avec la sourie.
> >? qu'est-ce que tu veux dire ?
> Faits une saisie avec un saisie clavier pour les tailles tableau. C'est la
> merde pour faire correspondre la sourie a un pixel!
ah.
pourtant en zoom 0 (1:1) les coordonnées sont affichées, et le résultat quand
tu cliques est arrondi sur 3 bits (valeur la plus proche) donc c'est assez
tolérant.
mais je n'ai pas eu le temps d'écrire toute une interface superjolie
cette année en assembleur... la fonction est disponible, un peu
délicate, mais au moins y'a plus à réassembler le programme à chaque fois :-)
enfin, j'essairai d'améliorer c'détail quand je pourrai.

> >> >j'en ai encore pour qqs semaines dessus, après je rédige le mémoire.
> >> T'as pas commencé ton memoire! mais tu vas ou là?
> >nan j'ai plein de trucs partout mais il va falloir passer pas mal de
> >temps pour tout organiser... j'en ai pour 100 apges facile ! sans parler
> >des annexes !
> ... tu l'auras jamais cette maitrise....au train ou tu vas tu va être
> obliger de rempiler.
je crois que l'année prochaine sera plus cool.
je compte bosser un peu et passer les UE que j'ai court-circuitées cette
année. Mais je passerai la soutenance absolument à la rentrée, car le
programme prend une bonne tournure. bientôt, on pourra même dessiner les
parois à la souris !

> >> j'le sent mal ton projet ;-((
> >je n'ai jamais été si près du but.
> >300000 autres trucs à faire pour le diplôme. résultat: 4 ans de recherches,
> >400% d'accélération :-) pour t'en convaincre, regarde les chiffres
> >au-dessus.
> Tu te fout de moi! il y a quatre ans les CPU etait a 133Mhz avec 8ko de
> cache, 4 ans de recherche pour s'appercevoir que les PC vont 4 X plus
> vite ... bravo!
nan. quelle que soit la technologie, la marque ou l'âge, si tu as de la mémoire
cache, l'algo que j'ai mis au point permet d'accélérer le calcul des gaz sur
réseaux. c'est une histoire de différence d'échelle, pas de valeur absolue.
et ça, c'est puissant.


Renseignements pris, la CPU est à 200MHz et le bus à 66MHz.


mercredi 22:
j'ai proposé une architecture pour le F1 du FCPU.
mais à 8H15 j'ai surtout réussi à faire fonctionner le générateur
de listes de modifications pour la première fois sans un accroc.
maintenant, il faut transformer l'algo en Pascal vers l'assembleur,
alors j'analyse l'algo sur le papier.
SP19, où les particules bougent correctement, est l'exemple des problèmes
causés par des parois défectueuses.
SP20 est le chantier pour mettre en place les parois automatiquement.
d'abord, afficher les parois :-) (copier,coller c'est fait).

ensuite mettre l'ossature.

d'abord la fonction qui scanne la liste de segments
(ça a déjà été fait mais cette fois-ci on compte moins sur les registres)
On compte sur le bit G pour afficher les points avant d'implémenter
la génération des listes elles-mêmes. cela permet de détecter les erreurs
au plus tôt.
Cette fois-ci, on peut se permettre de faire des coordonnées en virgule
fixe, afin que les dimensions s'adaptent à l'écran. 0:0 représente
le point (x=0,y=1) et 0FFFFh:0FFFFh représente (x=XSIZE,y=YSIZE-2).
c'est automatique. par contre, pour faire des objets plus délicat,
il faudra tenir compte du ratio XSIZE/YSIZE.
```

jeudi 24, 8h du mat:

la base pour les listes de modifications, c.à.d. les boucles pour
les lignes, est en place. il faut maintenant fabriquer les cellules
individuelles, ce qui est encore plus compliqué et plus difficile
à débeuguer car on ne peut pas tracer le code ni examiner les données.
Dans toute cette partie-là, je ne préoccupe ni du style ni de la vitesse
car j'y passerais trop de temps et j'ai simplement envie que ça fonctionne.
De plus, le temps d'exécution de la routine est très bref, donc
je m'occuperai des détails plus tard. Heureusement que j'ai déjà
débroussaillé l'algorithme en Pascal !!!
le source est maintenant SP21.asm

vendredi 2h du mat:

SP21.asm a du mal à retrouver des cellules déjà créées, ce qui
explique qu'un bon millier de paragraphes ne soit pas suffisant.
je passe à SP22. j'ai la crêve de la mort mais je suis vivant.
l'état importe peu tant que je peux encore taper sur le clavier.

bug idiot: j'ai oublié un $ devant base__. résultat, la liste
de cellules n'était pas balayée et des cellules étaient rajoutées
jusqu'à saturation. bête. j'ai perdu une heure dessus.
"tout àa pour un dollar". ça fait pas cher de l'heure.

Mon travail se résume finalement à 90% d'intuition
et 10% de travail. mais c'est le travail qui est 99% de
la difficulté ! tout est relatif. et j'aimerais bien prendre de
super vacances.

SP22 commence à fonctionner: on peut créer une liste pour chaque
cellule, et elles se chevauchent automatiquement. maintenant,
il faut arriver à rajouter des éléments à ces listes.
je passe à SP23.

6h: j'ai un bug hallucinant.
je lance le programme, il bugge au niveau des listes de modifications.
je retourne sous l'éditeur de Turbo Pascal, et le clavier envoie
des ordres dans tous les sens. je dois alors rebooter.
j'ai l'impression qu'il y a des pointeurs balladeurs quelque part.
où ?

6h30: cette fois-ci c'est vraiment un beau bug.
il affiche une ligne "fantome".
en l'écrivant, je crois avoir trouvé la raison,
puisque je fais le pushad/popad à l'extérieur
de la fonction make_cell...

vérification faite, c'est vraiment une histoire de puhasd/popad:
la multiplication modifiait eax, qui était réutilisé ensuite pour
l'appel suivant. wow...

8h: je passe à SP23.
10H: SP23 donne des premiers resultats satisfaisants
pour les cas généraux. Les grandes géométries sont bien traitées
mais les détails ne sont pas encore gérés.

Samedi: j'ai un nouveau stock d'amoxicilline.
j'ai aussi reçu plein de super emails hier.
j'ai même saturé la table de relocation, je ne peux plus utiliser $base__,
mais je peux utiliser $base1__ :-)
Il ne reste plus grand chose à faire sur le programme de génération de
listes. j'y travaille aujourd'hui.

21H30: SP23 fonctionne presque parfaitement.
SP24 tente d'éliminer des points durs indésirables.

Astuce: pour déterminer a priori la taille de la fenêtre

```
par calibration, utiliser un tunnel de même largeur mais de
hauteur plus réduite, afin de réduire le temps de calibration.

SP24 a corrigé un avant−dernier problème: mauvais registre utilisé
comme masque. SP25 va s'attaquer à la dernière difficulté: l'utilisation
abusive de cellules consécutives. il faut vérifier l'existence
d'"antécédents" identiques. il est déjà 3h du mat et je suis mort.
à demain.


Dimanche, 3H du mat:
j'essaie de revoir l'algorithme sans trop toucher au déjà fait.
pas très facile, vu la complexité, mais avec du travail, on y arrivera.
SP25 est en bordel, je passe à SP26
Peut−être que si je réussis, je pourrais mettre un "garbage collector" ?

6H40: l'algo fonctionne du feu de dieu, je l'envoie à Sustrac !
Je vais enfin pouvoir me reposer.



8H: quelques mesures.

j'ai trouvé que le scrolling vertical fonctionne mal pour
des très grandes tailles, zoom=4:1. Il 'sature' à 6xxx alors
que le tunnel est haut de 16384... fâcheux. mais 1:1 est OK.

Je trouve que le code a pas mal ralenti. le speedup est moins
fort pour les grosses tailles. voyons ce qui peut nous freiner.

Première raison : il serait temps de se préoccuper de
l'associativité des caches ! je crois me souvenir que la
L1 du Pentium a des sets de 4KO, ce qui cause quelques paniques
pour les lignes très larges. Cela a été 'intégré' dans les mesures
précédentes et n'a pas fait de vagues, donc voyons autre chose.

Deuxième raison possible: les listes de modification foutent
la merde dans la cache. Les listes ne sont pas ordonnées et
modifient l'ordre et l'endroit où les données sont cachées,
quelles qu'elles soient. essayons avec moins de segments.

résultat sur grosse taille (1024*16384): gain entre 2%
(strip=1) et 5% (strip=6).
1024*16384, 13 segments: Strip=6   min=18824483  strip=1:41468567
1024*16384, 4 segments:  strip=6   min=17930777  strip=1:40559429

Il est donc intéressant de penser à un "garbage collector intelligent"
qui permettrait de gagner quelques % en réorganisant les données
et en les "compressant" encore plus. On frise le compilateur LISP ici.

Autre essai en essayant de "sortir" du bourbier des lignes de cache:
1088*16384, 4 segments, strip=6, min=19386521
                                 ratio:  1.08118688/    1088/1024=1.0625
1024*16384, 4 segments, strip=6, min=17930777
                                 ratio:  1.063542265/   1024/960=1.0666/
 960*16384,             strip=6, min=16859487

 Pas de grosse différence, sauf que 1088/1024 < 1024/960
alors que 19386521/17930777 > 17930777/16859487. Pas de beaucoup
mais la tendance est intéressante à signaler.
Cela rejoint la remarque que la largeur du tunnel est
plus pénalisante que la hauteur. le multiprocessing par
division de la largeur du tunnel (bandes verticales)
est donc une voie sûre d'amélioration des performances.


Mesures brutes: 13 segments, zoom=4:1
                                          min/taille
 taille   strip    min    strip=1 speedup (cycles par cel.)
 256*256   25     92168    147963 1.60536  1.406372
 384*384   17    182768    310230 1.69739  1.239474
 512*512   12    301497    532305 1.76553  1.150119
 768*768    9    646880   1231260 1.90338  1.096733
1024*1024   6   1227273   2736858 2.23004  1.170418
```

```
1536*1536   4   2986700   5912358 1.97956  1.265928
2048*2048   3   6036906  10464761 1.73346  1.439310
3072*3072   2  16731691  23226581 1.38817  1.772953
4096*4096   2  35138297  41666952 1.18579  2.094405

le speedup diminue sensiblement, probablement à cause du buffer
de trame qui pompe la moitié de la L1.
Mais il y a VRAIMENT un problème de cache pour les grosses trames.
pourquoi le speedup a-t-il baissé ????
Aucun calcul n'est effectué.
Mais pour les petites tailles, on tourne à 1,1 cycles
par cellule, ce qui reste raisonnable pour une procédure de
déplacement qui n'est pas optimisée. En effectuant
le mouvement hexagonalement au lieu d'horizontalement seulement
comme au début, la vitesse est réduite de seulement la moitié,
tout en permettant d'avoir des parois arbitraires.
Mais je crois surtout que la diminution de la vitesse par 2 est
dûe au buffer qui prend la moitié de la L1.

A 1,096733 cycles par cellule, on tourne à 183Mc/s en crête.
le calcul diminuera cette vitesse d'environ 2 ou 3 (pour la crête),
ce qui reste dans les premières estimations.

Curieusement, la "crête" s'est déplacée de 512*512 (taille de la L2)
vers 768*768 (2*L2). explication ???

2,1 cycles par cellule font tourner à 100Mc/s. encore correct.
mais on est toujours "memory bound" pour une raison qui m'échappe.
Il faudrait un autre "type" de bécane mais je n'ai pas assez d'argent...

La courbe de performance lors des calibrations s'est aussi notablement
modifiée, remontant plus haut que la vitesse pour strip=1.
Je me pose de sérieuses questions, mais je n'ai aucune idée.
Je sais seulement que chaque "amélioration" de l'algorithme
va diminuer cette performance au fur et à mesure des ajouts,
comme cela se produit depuis le début des mesures.
Tout ce que je sais c'est qu'il s'agit d'un problème
d'architecture mémoire.

Il serait aussi temps de se préoccuper de l'ordre des champs dans
les cellules du tunnel.

le "discrétiseur" n'a montré aucun défaut, du moins pour "make_cell".
la procédure centrale (génération des traits) devra cependant être
refaite. plus tard, un jour. mais j'aurais bien besoin de générer
un cercle lisse :-)



mardi: fait des mesures sur le PII.
le comportement est un peu différent des simulations précédentes,
mais la L2 dans le même boîtier permet de garder des performances
raisonnables, toujours de l'ordre de 1 à 2 cycles par cellule.

PII 266 monoproc, SP27.asm, 13 segments
taille      strip       min       strip=1
 256*256     25       92168        147963
 384*384     17      182768        310230
 512*512     12      301497        532305
 768*768      9      646880       1231260
1024*1024     6     1227273       2736858
1536*1536     4     2986700       5912358
2048*2048     3     6036906      10464761  strip=8:max=11210693
3072*3072     2    16731691      23226581  strip=6:max=25761657
4096*4096     2    35138297      41666952

L'algo ne se comporte plus très bien pour les grandes tailles,
largeur et hauteur*largeur.

Essais avec grandes tailles et largeur différentes:
2048*8192     3    23471642      40212128
1024*16384    6    18824483      41468567
la largeur pose un gros problème.
```

```
essai avec 4 segments seulement:
1024*16384    6    17930777      40559429
                   5% de gagnés  2,2% de gagnés
La complexité des structures pose un petit problème.
il faudrait fabriquer le "garbage collector" !!!


essais avec largeurs légérement différentes:
1088*16384    6    19386521
760*16384     7    16859487
(pas très convaincant)



mercredi: je passe à sp28, j'essaie d'adapter le source de Zaleski.
Il semble utiliser la même technique d'equations que celle à laquelle
j'ai conclu, c'est à dire : XOR à la fin, constitution du masque de
XOR au fur et à mesure. c'est une technique assez intelligente qui
n'est pas évidente pour un compilo VHDL ou les tables de Karnaugh,
d'où la supériorité de l'homme sur la machine :-)
Un compilo n'est pas assez intelligent pour détecter la réversibilité
de certaines configurations, et le bénéfice immense du XOR.

Le fait d'inclure un programme GPLé dans le mien rend le mien aussi
GPL de fait. maintenant au moins, j'ai l'étiquette GPL rajoutée dans
l'en-tête. C'est officiel, non seulement je suis ouvert mais en plus
j'en ai l'étiquette.

Je crois que je viens de comprendre le sens de "collision_left" et
"collision_right": c'est pour savoir la chiralité de la collision.
<confirmation Zaleski>

Par contre, il n'utilise pas la même table de collision :
la collision triangulaire débouche sur une autre configuration
triangulaire. tout le travail est donc à refaire.
Au moins, je sais que ma piste n'est pas miteuse.

Equation en 0-x-x: ça a l'avantage d'être assez compact.
problème: faire un generateur de nombres aléatoires à 3 sorties
dont une seule est active à la fois. Je n'arrive pas à réduire la
collision à un cas où 2 sorties sont possibles.

Relecture des notes de jusssieu, page 3':
Cs (vitesse du son) = sqrt (3/7) = 0,654653... unités par pas de temps.
Re* = 2,220
Fmax=0,285 (densité optimale)
 or, 0,285 est très proche de 2/7 (au millième), ce qui veut dire que
 2 particules par site sont nécesaires.
Je cherchais une fraction simple pour initialiser le fluide mais
c'est encore plus évident que je ne l'avais pensé :-)))
le truc est donc de mettre deux fois plus de particules que de sites,
et ça tournera à plein régime à 2,2 Re par cellule.

g=invariance galliléenne,
7(1-2F)/12(1-F),
F=2/7-> g=7(1-(4/7))/12(1-(2/7)) = 7(3/7)/12(5/7) = 3/(79/7)=21/79=0.2658...
bon, là je suis paumé car je ne sais pas dans quelle dimension s'exprime
cette valeur...
Lecture complète effectuée, il apparait que c'est le coefficient de transport
des turbulences... il faut s'approcher de g=1 pour simuler un fluide réel,
mais cela force à ralentir fortement l'efficacité du modèle !
une solution existe avec un modèle à 3 particules au repos et atteint
g=1,07 (7% de fiabilité, wow), et même 1 pour d=0,2. mais le Re chute
alors terriblement.


Samedi matin
Je pense qu'on peut fabriquer un modèle FHP3 avec une meilleure invariance
en reprenant le tableau et en "favorisant" la création des particules
immobiles par rapport à leur "destruction". En clair cela se
traduit par le fait que les transitions qui "créent" des particules
immobiles soient plus nombreuses que celles qui les détruisent.
Le problème est alors de "choisir" (il n'y a jamais de choix)
```

quelles collisions vont les détruire, ou comment (nombre aléatoire ?).

Je crois aussi que la table des collisions (6 lignes et 3 colonnes)
donne aussi la solution de comment faire les lois pour le modèle à 3
particules immobiles.

Mais pour l'instant j'en suis à rechercher les invariants dans le
modèle à 1 particule immobile.

les collisions 0−x−x sont mieux connues. Elles modifient toutes 4
particules, selon 2 cas de figures:
−2 paires opposées dans le cas d'une collision binaire−binaire
−3 particules consécutives et une particule immobile dans le cas
  d'une collision binaire−ternaire.
On retrouve ce deuxième cas de figure dans les collisions 1−x−x
donc je ne peux pas commencer la programmation.

Il m'est aujourd'hui évident que ce deuxième cas "xor ternaire"
est typique d'une collision avec échange d'une particule immobile.
les xor quaternaires sont des "sauts dans la même case".
Reste à comprendre pourquoi cette configuration−là et pas une autre.

En tous cas, une chose est maintenant sûre : il y a toujours 4 particules
qui changent de direction. ce qui veut dire que trois ne sont pas
déviées, quel que soit le cas de figure. C'est important car on peut
diminuer à coup sûr le nombre d'opérations SI on considère les particules
qui ne sont pas déviées, au lieu des particules déviées.
Cela veut dire qu'il faudra inverser le masque de XOR avant de l'appliquer
sur les données. Cela rajoute quelques instructions en assembleur à la
fin du "calcul" mais j'espère réduire d'un quart sa lourdeur.
J'espère ne pas m'être trompé car l'enjeu est immense.

J'ai appris cette nuit qu'on pouvait faire des écoulements alternatifs
de Von Karman avec une paroi oblique. Cela m'évite la fabrication
d'un cercle mais me force quand même à faire une ligne oblique.
De toutes façons, la variance galliléenne est trop forte pour
faire un écoulement correct de ce type.

J'ai décidé de sacrifier une fraction de chiralité pour simplifier
les collisions sortantes des 0−3−x. Il faudra aussi une version paire
et impaire. Il faut réexaminer toutes les équations sur le papier.
Toutes les équations programmées devront être testées en vrai sur
l'ordinateur... pour vérifier qu'elles fonctionnent bien comme prévu.
J'ai l'impression que le papier à listing n'est plus assez grand,
qu'il me faudra des rouleaux de papier peint ou de nappes en papier
pour contenir la quantité de choses à écrire...

Durant le développement de ce logiciel, j'ai développé aussi une méthode
de débogage itérative. La première chose à faire est d'isoler la
partie "fautive", sans laquelle aucune erreur se produit. De nombreux
essais peuvent être nécessaires mais on est sûr de trouver.
La deuxième partie est de trouver le "POURQUOI" après le "QUOI":
une analyse du flot d'instructions et des dépendences de données
(quelle est la valeur de chaque registre à chaque moment) permet
dans la plupart des cas de trouver la partie fautive. Parfois,
en "remontant" les instructions, on arrive à trouver la vraie raison,
qui fait qu'un code éprouvé ne se comporte pas comme il devrait.
Une grande partie des causes de ces bugs sont de vraies erreurs
d'inattention (la moitié).

samedi soir:

Maryvonne Lebrun:
>Quelles sont les applications que tu as en vue? (AERONAUTIQUE?).
>Sais tu que dans mon domaine on manque de logiciels pour calculer les
>souffles d'explosion "a cote" de l'axe. (ca forme theoriquement un coeur).
>Cela conduit a penaliser les industriels dans bien des cas.
>Si jamais les "explosions" t'interessent je pourrais de donner des
>coordonnees de gens qui font ca.
C'est vrai qu'on peut simuler des explosions.
j'en avais simulé plein dans mes premiers programmes.

```
C'était même assez amusant :-)


Pour la collision :
à la fin:
[A]^= ~xorA
[B]^= ~xorB
[C]^= ~xorC
[D]^= ~xorD
[E]^= ~xorE
[F]^= ~xorF
[G]^= ~xorG

en asm MMX: (remplacer ? par la direction ou un numéro de registre)
pcmpeqb mm?,mm? ; met mm? tout à 1
pxor mm?, mm?? ou [xor?] ; mm? contient le masque inversé
pxor mm?, [?]
movq [?],mm?

Autre solution : partir d'un masque XOR entièrement à 1 et
"enlever" des bit à chaque fois. mais ça ne fait que "repousser" le
problème puisqu'il faut mettre les masques à 0.
Mais vu qu'il faut aussi le mettre à 0 au début.....

Remarque : pour mesurer les collisions, le OR de 3 masques XOR
opposés (disons A, C et E) est suffisant (au lieu de 7 OR).


COL=xorA | xorC | xorE entrelacé avec la modification des masques :


movq mm0,[xorA] ; charge A
pcmpeqb mm4,mm4 ; mm4=11111....
movq mm1,[xorC] ; charge C
movq mm3,mm0    ; crée le compteur de collisions
movq mm2,[xorE] ; charge E
pxor mm0,mm4    ; inverse le masque A
por mm3,mm1     ; change le compteur de collisions
pxor mm0,[A]    ; modifie A
pxor mm1,mm4    ; inverse le masque C
por mm3,mm2     ; change le compteur de collisions
pxor mm1,[C]    ; modifie C
pxor mm2,mm4    ; inverse le masque E
movq [A],mm0    ; écriture de A, mm0 est maintenant libre
movq mm0,mm4    ; mm0=1111 pour Xor futur avec la mémoire
movq [col],mm3  ; sauve le compteur de collisions
movq mm3,mm4    ; mm4=1111 pour Xor futur avec la mémoire
pxor mm2,[E]    ; modifie E
;ici: mm3=mm4=mm0=1111
; sauver mm1 (C), mm2 (E)
; puis traiter B,D,F et G
movq [C],mm1
movq mm1,mm4
pxor mm4,[xorB]
; ici il n'y a plus rien à entrelacer :-(
movq [E],mm2
pxor mm0,[xorD]
pxor mm4,[B]
pxor mm3,[xorF]
pxor mm0,[D]
movq [B],mm4
pxor mm1,[xorG]
pxor mm3,[F]

pxor mm1,[G]
movq [D],mm0

movq [F],mm3

movq [G],mm1

32 instructions, 5 registres utilisés.
La difficulté de ce code est de ne pas mettre plus d'un accès
à la mémoire par paire d'instructions afin que le PMMX soit content.
```

```
Mais à la fin, il n'y a plus d'autre choix, il n'y a plus rien qui
n'accède pas à la mémoire pour entrelacer.

Les règles de codage importantes ici sont (autant que possible):
-1 un seul accès à la mémoire par paire d'instructions
  (à la fin, il n'y a pas d'autre chose à faire, donc on abandonne)
-2 repousser les dépendances de registres au moins à 2 instruction,
    plus pour les écritures de registres.

Il faut voir le code qui viendra avant pour trouver des valeurs existant
déjà dans les registres et enlever quelques accès à la mémoire.


On peut commencer par les collisions quaternaires 0-2-x-/R et 0-4-x-R
qui sont très simples à coder: tout se passe entre paires.
la particule immobile ne change pas, donc il faut l'inverser ;-)
Il reste donc une paire à modifier (celle qui ne change pas).
Elle est choisie au hasard entre deux configurations.

Le générateur de nombres aléatoires est finalement assez rustique :
un simple registre 64 bits qui est changé à chaque calcul par une simple
addition. on peut initialiser le registre par l'horloge si on veut.
Le fait d'additionner une cellule (n'importe laquelle, A par exemple)
au compteur lui fait très vite donner des valeurs incohérentes,
pas besoin d'utliser de congruence !


Un examen papier de 0-x-x conclut pour les collisions quaternaires:


NG=!G ; inverse la particule immobile

; 1) paires opposées identiques:
PA=A^D
PB=B^E
PC=C^F
SYM=!(PA + PB + PC) ; 1 s'il y a 3 paires

Si SYM <> 0 continuer:

; 2: il faut deux paires à 1:

PA=(A^NG)SYM
PB=(B^NG)SYM
PC=(C^NG)SYM

SYM= !(PA^PB^PC)
    (PA|PB)

;SYM=1 si une collision quaternaire est valide,
;on détecte le cas où deux (seulement) paires sont à 1
;(cas 0,1,3 ignorés car non quaternaires).


; 3: sortie: l'algorithme est une sélection séquentielle.
XORG=SYM ; ne pas modifier la particule immobile...
R=RANDOM
XORA=XORD= R A SYM
R^=A
XORB=XORE= R B SYM
R^=B
XORC=XORF= R C SYM

26 opérations booléennes pour 6 collisions (sans compter tous les
mouvements...) ça fait 4 opérations par collision :-/

Je crois que ça suffira pour l'instant. Demain je m'attaque à la réduction
des équations ternaires. Avant ça, un test en réel de l'equation
quaternaire serait utile... (SP29.asm)



Lundi matin: les collisions quaternaires fonctionnent !
il a fallu aménager la règle initiale car elle n'était pas
```

```
exacte mais ça y est.
Le code reste en chantier (je dirais même: le bordel)
pour permettre le développement de l'algorithme dans
des conditions acceptables, on gagnera des cycles plus tard
(au moins la moitié !).

Le code:

1) paires opposées identiques:
------------------------------
PA=A^D
PB=B^E
PC=C^F

;charge les données
 movq mm0,[edi+A]
 movq mm1,[temp_B]
$base1__ equ $-4
 movq mm2,[temp_C2]
$base1__ equ $-4
 movq mm3,[edi+D]
 movq mm4,[edi+E]
 movq mm5,[edi+F]
 pcmpeqb mm6,mm6 ; mm6=-1
 pxor mm3,mm0 ; PA=mm3
 pxor mm4,mm1 ; PB=mm4
 pxor mm5,mm2 ; PC=mm5

SYM=!(PA + PB + PC) = 1 s'il y a 3 paires identiques
 por mm3,mm4
 por mm3,mm5
 pxor mm3,mm6 ; NOT mm3

NG=!G : inverse la particule immobile
 pxor mm6,[edi+G]  ; mm6=1 xor G



2: il faut deux paires à 1:  (cas 0,1 et 3 inintéressants)
---------------------------

PA=(A^NG)SYM
PB=(B^NG)SYM
PC=(C^NG)SYM
 pxor mm0,mm6
 pxor mm1,mm6 ; ^NG
 pxor mm2,mm6
 pand mm0,mm3
 pand mm1,mm3 ; SYM
 pand mm2,mm3

SYM= !(PA^PB^PC) (PA|PB)
 movq mm4,mm0
 movq mm5,mm0
 pxor mm4,mm1 ; mm4=PA^PB
 por mm5,mm1  ; mm5=PB|PA
 pxor mm4,mm2 ; mm4=PA^PB^PC
 pandn mm4,mm5 ; SYM=mm4

Si SYM <> 0 continuer...

SYM=1 si une collision quaternaire est valide,
on détecte le cas où deux (seulement) paires sont à 1
(cas 0,1,3 ignorés car non quaternaires).

ici:
mm0=PA
mm1=PB
mm2=PC
mm3=
mm4=SYM
mm5=
mm6=NG (inutilse)
mm7=
```

```
3: sortie: l'algorithme est une sélection/élimination séquentielle.
----------

XORG=SYM ; ne pas modifier la particule immobile... (????)
R=RANDOM
SYM=!SYM
XORA=XORD= (R PA) | SYM
R^=PA
XORB=XORE= (R PB) | SYM
R^=PB
XORC=XORF= (R PC) | SYM

 pcmpeqb mm7,mm7 ; mm7 = -1
 pxor mm4,mm7
 movq [xorG],mm7
$base1__ equ $-4


 movq mm5,mm0     ;r1=mm5
 movq mm3,[RANDOM];mm3=rand
$base1__ equ $-4

 pand mm5,mm3     ;and r1,rand
 movq mm6,mm1     ;r2=B

 pxor mm3,mm0     ;xor rand,A
 por mm5,mm4      ;or r1,sym

 movq mm7,mm2     ;r3=C

 movq [xorA],mm5  ;mov [xorA],r1
$base1__ equ $-4
 pand mm6,mm3     ;and r2,rand

 movq [xorD],mm5  ;mov [xorD],r1
$base1__ equ $-4
 por mm6,mm4      ;or r2,sym

 pxor mm3,mm1     ;xor rand,B
 movq [xorB],mm6  ;mov [xorB],r2
$base1__ equ $-4

 pand mm7,mm3     ;and r3,rand
 movq [xorE],mm6
$base1__ equ $-4

 por mm7,mm4      ;or r3,sym
 movq [xorC],mm7
$base1__ equ $-4
 movq [xorF],mm7
$base1__ equ $-4


J'ai remarqué que des particules "traversent" les parois, je ne comprends pas
exactement pourquoi mais elles le font. Il faut placer le déplacement
des particules autre part !

SP30 tente de résoudre ce problème.
La solution la plus simple et la plus radicale est d'interdire
les collisions avec réarrangement (le "calcul") avec un masque
(la "somme" des masques de modification).
Avec un léger aménagement, le programme fonctionne très bien.
Il faudra seulement penser à faire le ménage après, et il
y aura du travail !

Ce qui me chagrine c'est que je ne peux pas directement tester
les collisions 0-4-x, car la création des configurations
de particules est un peu difficile en assembleur...
Alors je mets plein de particules G et je regarde ce qui se passe !
L'expérience consiste à saturer le tunnel de particules G et
mettre beaucoup de particules mobiles, puisque les collisions
0-4-x ne s'effectuent qu'à haute densité. L'affichage des
particules G est désactivé. En quelques temps de circulation,
```

```
l'écran est complétement noyé dans le brouillard, sans effet
de bord visible. J'estime cela suffisant pour continuer.

SP31: On passe au reste !
J'ai mis en place un "banc d'essais" des collisions.
il faut encore les tester une à une et réassembler à chaque
fois mais au moins ça marche, sans empirisme. Les collisions
quaternaires (0-2-x <->0-2-x et 0-4-x <-> 0-4-x) sont validées.

SP32: ready



Mardi: le jour où le Bonto a été réinventé.

Les collisions quaternaires étant en place, il faut se soucier
des collisions ternaires 0-2-x-R <-> 0-3-x <-> 0-4-x-/R.
les états allant vers 0-3-x sont simples, il y a deux solutions
et le générateur de nombres aléatoires est adapté.

Le problème est que les collisions partant vers 0-2-x et 0-4-x
ont chacune 3 possibilités !!!!! Je me suis refusé à baisser
les bras devant un problème aussi simple, car l'option de
"briser" la symétrie du modèle est désagréable à penser.

La première solution pour faire 3 sorties est "statique";
elle utilise les lignes paires et les lignes impaires,
dans lesquelles on a:
lignes paires:    0-3-1 -> 0-x-1     0-3-2 -> 0-x-2 OU 0-x-3
lignes imppaires: 0-3-1 -> 0-x-2     0-3-2 -> 0-x-1 OU 0-x-3

<arbre binaire>

La programmation est simple mais le résultat est assez
mauvais, car il ne correspond pas aux cas étudiés dans les équations.

J'étais presque parti pour le mettre en place lorsque j'ai repensé
à la solution que j'avais déjà mise en place pour l'élimination/sélection
progressive pour la sortie des collisions quaternaires.

Celà reviendrait en premier lieu à sélectionner/éliminer une
première partie des sorties par un masque aléatoire:
on obtiendrait A=1 B=0 C=1 ou A=0 B=1 C=0 selon le masque (0 ou 1).
Ensuite, avec une autre valeur aléatoire (un deuxième masque)
on sélectionnerait une seule sortie s'il en reste plusieurs:
Si B=0 alors A=masque et C=/masque.

Le problème est que statistiquement, B a une chance sur 2 d'apparaître
alors que A et C ont une chance sur quatre d'être tirés.

<arbre binaire>

Une autre solution est d'inverser le rôle de A et B (ou B et C)
selon la parité de la ligne ou un troisième nombre aléatoire.

<2 arbres binaires>

Dans ce cas, une sortie aura 1 chance sur 4 d'être sélectionnée
alors que les deux autres sorties auront 3 chances sur 8. mauvais.

<HT width=50>

Il faut se rendre à l'évidence, on ne peut pas faire un 3 avec une
combinaison de 2. 2<>3, 2^2=4<>3, 2^3=8<>9, donc tout tirage ne peut
donner qu'une combinaison de 2. Il faudrait introduire un 3 qui viendrait
de quelque part (afin de rendre le tout multiple de 3) mais notre équation
ne permet pas de trouver de combinaison de 3 sorties dont une SEULE est à 1,
car si cette règle d'exclusion n'est pas respectée on retombe encore
dans le cas polinômial !

Une voie possible est de travailler en base 3, mais ce domaine est riqué
puisqu'on risque de retomber dans le cas polinômial si on ne fait
pas attention. En plus, les équations booléennes permettant de
travailler dans cette base où deux bits codent 3 possibilités
```

sont plus compliquées que nécessaire. Comme 3 n'est pas un grand nombre,
on peut cependant travailler avec trois variables dont une seule est à
1. Ce sera notre générateur aléatoire.

Le problème, maintenant que le nombre est défini, c'est d'avoir une
suite aléatoire de nombres. Et c'est là que la situation se dénoue :
c'est une suite qu'il nous faut.

Les 3 nombres dont un seul est à 1 peuvent être "décalés", subir une
rotation, selon la valeur du nombre aléatoire, dont la valeur est 0 ou 1.
Si le nombre est 1, alors le nombre A passe dans B, B devient C et C devient
A, cycliquement. Un peu comme dans le jeu de "bonto" (merci maman !)

<dessin de rotation de bits>

sauf que l'ordre de commutation n'est pas important.
Ce qui compte c'est que la valeur du "vecteur" A,B,C soit inconnue
pour tout pas de temps N>3 (qui est une valeur de "démarrage" très
raisonnable). Evidemment, il reste la possibilité de prévoir avec
une chance sur deux la valeur du "nombre" en connaissant la valeur
précédente, une chance sur deux de connaitre la valeur en connaissant
la valeur N−2, mais pour des temps très grands, il est impossible
de prévoir cette valeur si le générateur de nombres bianires est imprévisible.

Cela vient du fait que la probabilité de chaque valeur ne dépend plus
d'un arbre binaire de profondeur limité (correspondant à une fraction),
mais la probabilité dépend de la manière dont est fait l'échantillonnage.
Si l'échantillonnage est fait après un "long" temps N, alors la fraction
se rapprochera, "tendra" vers une approximation de 1/3.
On a résolu le problème de statistique en introduisant le temps, qui
a priori n'est pas connu (statistique), donc lui aussi imprévisible.
D'ailleurs, cette "aventure" pose une question:

L'infini est−il divisible par trois ?

Quelle que soit la réponse, on a obtenu un algorithme simple et adaptable
pour d'autres cas. Ses propriétés sont satisfaisantes a priori
pour être utilisé dans le modèle FHP.

pseudocode:

```
int A,B,C,RANDOM,A2,B2,C2

A2=A
B2=B
C2=C

A=(B2 RANDOM) | (A2 !RANDOM)
B=(C2 RANDOM) | (B2 !RANDOM)
C=(A2 RANDOM) | (C2 !RANDOM)
```

c'est simple, non ? c'est comme si on faisait une rotation bit à bit,
avec un nombre de rotation de 0 ou 1.
Cette rotation peut s'effectuer assez rarement, pour chaque trame par exemple.

en asm MMX:

1) charger RANDOM

```
movq mm0,[RANDOM]
```

2) le recopier dans 3 registres pour faire un PANDN:

```
movq mm1,mm0
movq mm2,mm0
movq mm3,mm0
```

3) charger les valeurs initiales:

```
movq mm4,[RAND_A]
movq mm5,[RAND_B]
movq mm6,[RAND_C]
```

```
4) les masquer avec PANDN pour avoir (?2 !RANDOM)

pandn mm1,mm4
pandn mm2,mm5
pandn mm3,mm6

5) masquer les originaux avec RANDOM (? RANDOM)

pand mm4,mm0
pand mm5,mm0
pand mm6,mm0

6) résultat: OR et entrelaçage

por mm4,mm2
por mm5,mm3
por mm6,mm1

7) écriture du résultat

movq [RAND_A],mm4
movq [RAND_B],mm5
movq [RAND_C],mm6


Les dépendances de registres sont fortes. il faut réécrire et
réorganiser les instructions. L'allocation des registres semble
toutefois correcte.


Remarque: les performances du programme diminuent un peu, le STRIP aussi,
à cause du plus grand nombre de variables temporaires utilisées, qui prennent
de la mémoire. Il faudra trouver une solution à ce problème.


Réécriture du code de rotation: il y a un problème : les accès à la
mémoire. sur PII on ne peut faire d'écriture qu'une fois par "paquet",
toutes les 3 isntructions. sur PMMX on ne peut accéder à la mémoire
que toutes les 2 instuctions. Or il y a 3 accès à effectuer successivement
par "groupe". J'espère que le code où ce morceau sera mis aura des
instructions à entrelacer.

D'abord, rechercher le "chemin critique" : c'est la plus longue
chaine de dépendances. Ensuite, les autres instructions seront
entrelacées dans les parties restantes, où les dépendances de registres
laissent de la place.


L'analyse papier n'a pas trouvé de problème particulier avec
le code ci-dessus, on va donc le garder pour l'instant, avec quelques
entrelaçages:

;*****************************

; RANDOM NUMBER GENERATOR UPDATE:
;1) load RANDOM
 movq mm0,[RANDOM]
$base1__ equ $-4

;2) copy before PANDN:
 movq mm1,mm0
;3) load the initial values:
 movq mm4,[RAND_A]
$base1__ equ $-4
 movq mm2,mm0
;4) mask out with PANDN (x1 !RANDOM)
 pandn mm1,mm4
 movq mm5,[RAND_B]
$base1__ equ $-4
;5) (x2 RANDOM)
 pand mm4,mm0
 movq mm3,mm0
 movq mm6,[RAND_C]
$base1__ equ $-4
```

```
 pandn mm2,mm5
 pand mm5,mm0
;6) result
 por mm4,mm2
 pandn mm3,mm6
 pand mm6,mm0
;7) write the result
 movq [RAND_A],mm4
$base1__ equ $-4

 por mm5,mm3
 por mm6,mm1

 movq [RAND_B],mm5
$base1__ equ $-4
 movq [RAND_C],mm6
$base1__ equ $-4

;test:
 movq [dword 0],mm4
randa equ $-4
 movq [dword 0],mm5
randb equ $-4
 movq [dword 0],mm6
randc equ $-4
```

La dernière partie affiche les résultats à l'écran.
après un certain temps (pour que le premier générateur de nombres
sans queue ni tête se mette en marche) le résultat semble satisfaisant.
Il faut compter une trentaine de pas (mais pas les pas de temps, car
un appel à la procédure de calcul calcule plusieurs pas de temps).

Question: dans quelle mesure l'algorithme "bonto" influence
la chiralité du modèle ?

Je ne pense pas qu'il y ait une influence quelconque puisque si
on peut peut-être connaitre la valeur du nombre aléatoire en
fonction de sa valeur précédente, il n'y a aucun rapport entre
ce générateur et les valeurs qu'il modifie. En effet, le nombre
aléatoire n'est pas dépendent des 8 configurations de particules
initiales qu'il modifie...

Mercredi matin, SP33:

Je commence à voir comment les collisions restantes de 0-x-x peuvent
être programmées. La tâche est un peu plus compliquée car il y a plus
de paramètres à prendre en compte.

D'abord il faut reconnaître les 0-2-x-R et 0-4-x-/R: ils ont en fait
déjà reconnus presque totalement par une opération effectuée précédemment
pour les collisions quaternaires:

```
NG=!G ; inverse la particule immobile
; 1) paires opposées identiques:
PA=A^D
PB=B^E
PC=C^F
SYM=!(PA + PB + PC) ; 1 s'il y a 3 paires
ici on rajoute:
SYM3=A^B B^C PA PB PC pour détecter 0-3-x

; 2: il faut deux paires à 1:
PA=(A^NG)SYM
PB=(B^NG)SYM
PC=(C^NG)SYM
SYM= !(PA^PB^PC)
    (PA|PB)
```

```
Lundi 12 juillet 1999:

Au prix d'une astuce, je crois que les collisions 1-x-x peuvent être
simplifiées à 3 rotations de 3 cas.

L'astuce pas encore trouvée consiste à savoir si au moins 3
particules sur les 6 possibles sont présentes.

En fait on "plie" les combinaisons 1-4-x et 1-5-x vers 1-2-x et 1-1-x.
On inverse le sens de

000000 >=3 interessant
000001
000010
000011
000100
000101
000110
000111 1
001000
001001
001010
001011 1   1
001100
001101 1   1
001110 1
001111 1
010000
010001
010010
010011 1   1
010100
010101 1   Y
010110 1   1
010111 1
011000
011001 1   1
011010 1   1
011011 1
011100 1
011101 1   1
011110 1
011111 1   1
100000
100001
100010
100011 1
100100
100101 1   1
100110 1   1
100111 1
101000
101001 1   1
101010 1   Y
101011 1   1
101100 1   1
101101 1
101110 1   1
101111 1   1
110000
110001 1
110010 1   1
110011 1
110100 1   1
110101 1   1
110110 1
110111 1   1
111000 1
111001 1
111010 1   1
111011 1   1
111100 1
```

```
111101 1   1
111110 1   1
111111 1
```

le problème est de trouver l'equation donnant 1 au moins pour les
cas présentés. Je crois qu'un comptage partiel et judicieux devrait
faire l'affaire mais je ne sais pas exactement comment.

Je sais aussi qu'il faut reprendre les equations 0-2-x et 0-4-x
afin d'intégrer dedans les collisions allant vers 0-3-x.

Mardi 18 juillet 5H20 du matin:

je viens de "résoudre" l'équation permettant
de trouver la "majorité" de 6 nombres. seuls les cas où il y a 4, 5 ou 6
bits à 1 mettent le résultat à 1.

L'algo de base est un "population count" à base de "half adder" (AND et XOR).
la somme de 6 nombres de 1 bits est effectuée et seul le MSB est gardé.
l'arbre de dépendence a montré que deux half adders peuvent être parallélisés
dans la limite des registres. on est juste en dessous de la limite des
8 registres mais ça tient bien. il y a cependant "l'amorçage" avec ses
6 chargements de mémoire qui peuvent poser problème mais ce n'est pas
une surprise.

l'algo a été optimisé de manière "absolue", sans trop tenir compte des
particularités de chaque processeur sauf pour la mémoire. je ne pense pas
qu'il y ait d'algo meilleur car l'arbre de dépendences a été analysé avec
soin et toute dépendence a été repoussé aussi loin que possible.
je ne pense même pas qu'un compilo fasse (beaucoup) mieux.
Il y a en tout 32 instructions :

```
movq mm2,[A]
movq mm1,[B]
movq mm6,mm2
movq mm5,[C]
movq mm7,mm1
movq mm3,[D]
pxor mm2,mm5
pxor mm1,mm3
movq mm0,[E]
pand mm5,mm6
movq mm4,[F]
pand mm3,mm7
;Half Add stage 2:
movq mm7,mm2
movq mm6,mm0
pxor mm2,mm1
pxor mm0,mm4
pand mm1,mm7
pand mm4,mm6
;HA 3:
movq mm6,mm1
pand mm2,mm0
pxor mm1,mm5
pand mm5,mm6
;HA 4:
movq mm6,mm1
movq mm7,mm4
pand mm1,mm3
pand mm4,mm2
pxor mm2,mm6
pxor mm3,mm7
;OR:
por mm1,mm4
pand mm2,mm3
por mm1,mm5
por mm1,mm2
```

MM1 contient : Sigma(bits)>3 (4,4,6)

```
mercredi 3H du matin

J'ai "unifié" les collisions ternaires et quaternaires de 0−x−x
(sans les collisions sortantes des triangulaires). Ce n'est pas
évident mais vu que j'y suis arrivé... il suffit d'y aller un pied
devant l'autre. mais vu qu'il faut aussi souvent reculer, ça
ressemble plus à une danse.

Je suis reparti des 0−2−x−/G et 0−4−x−G en logique positive.
j'ai "mélangé" les parties communes de 0−2−x−G et 0−4−x−/G,
qui ont des caractéristiques similaires, entre autres le XOR
des entrées par la particule G pour profiter de la "dualité".

finalement, à la fin du calcul il faut utiliser des entrées
"inversées" car les deux type de code G et /G les inversent.
donc il faut revoir l'équation en considérant les entrées inversées
dès le début. Comme pour la première version, on inverse
simplement G et le tour est joué.

Autre difficulté : le faible nombre de registres. On est obligé
d'utiliser quelques variables en mémoire pour faciliter le travail
et garder autant que possible les variables importantes en registre.
PA, PB et PC sont les meilleurs candidats car leur utilisation
est indépendente et successive.

Finalement, les deux algorithmes de modification de la fin sont:
− pour 02x/G et 04xG, l'algorithme d'élimination/sélection progressive
    décrit plus tôt. Problème : il faut aussi inverser le résultat.
− pour 02xG et 04x/G, le principe est plus simple : une seule paire
    est 'active' à la fois, il suffit de sélectionner un des éléments
    de la paire au hasard et de l'envoyer sur le XORx correspondant.


Formule de base:

G^=1
A^=G
B^=G
C^=G
D^=G
E^=G
F^=G
PA=AD
PB=BE
PC=CF
X1=A^D+B^E+C^F (paires identiques)
X2=PA^PB^PC    (parité : cas g ou /g)
Q=/(B.C).X2.X1 (collisions ternaires: 1 paire à 1)
T=/X2.(B+C).X1 (2 paires à 1)
XORG=T
T1=RAND T
T2=/RAND T
PA:
P1=/(PA RAND) Q
RAND^=PA
XORA=T1 PA
XORD=T2 PA
PB:
P2=/(PB RAND) Q
RAND^=PB
XORB=T1 PB
XORE=T2 PB
PC:
P1=/(PC RAND) Q
XORC=T1 PC
XORF=T2 PC


Il faut encore fabriquer la fomule pour 0−3−x−x puis l'intégrer
dans la formule ci−dessus, avant de construire l'arbre de dépendence
qu'il faudra ensuite "plier" pour faire tenir tout ça dans 8 registres...
```

```
0-3-x-x:

DELTA=(A^D B^E C^F) (A^B B^C)
XORG|=DELTA
XORA|=DELTA RAND1-3  A
XORD|=DELTA RAND1-3 /A
XORB|=DELTA RAND2-3 /A
XORE|=DELTA RAND2-3  A
XORC|=DELTA RAND3-3  A
XORF|=DELTA RAND3-3 /A

Mais comme les entrées (A,B,C,... même G) sont inversées,
je ne suis pas sûr de la "polarité" de A. J'utilise donc le A
d'origine.

Cette formule s'insère assez facilement dans la précédente
mais la fait grossir assez conséquemment. Elle était déjà compliqué
mais elle l'est encore plus maintenant. Si une erreur apparait
expérimentalement, elle sera difficile à détecter.


G^=1
A^=G
B^=G
C^=G
D^=G
E^=G
F^=G
PA=AD
PB=BE
PC=CF
X1=A^D+B^E+C^F (paires identiques)
X2=PA^PB^PC    (parité : cas g ou /g)
DELTA=(A^D B^E C^F) (A^B B^C)
Q=/(B.C).X2.X1 (collisions ternaires: 1 paire à 1)
T=/X2.(B+C).X1 (2 paires à 1)
XORG=T | DELTA
T1=RAND T
T2=/RAND T
PA:
P1=/(PA RAND) Q
RAND^=PA
XORA=(T1 PA) | (DELTA RAND1-3  A)
XORD=(T2 PA) | (DELTA RAND1-3 /A)
PB:
P2=/(PB RAND) Q
RAND^=PB
XORB=(T1 PB) | (DELTA RAND2-3 /A)
XORE=(T2 PB) | (DELTA RAND2-3  A)
PC:
P3=/(PC RAND) Q
XORC=(T1 PC) | (DELTA RAND3-3  A)
XORF=(T2 PC) | (DELTA RAND3-3 /A)


En reconsultant mes notes, je me suis rappelé d'un détail:
la formule de 0-3-x-G est mauvaise, car il faut XORer A avec G.

G^=1
A^=G
B^=G
C^=G
D^=G
E^=G
F^=G
PA=AD
PB=BE
PC=CF
X1=/(A^D+B^E+C^F) (paires identiques)
X2=PA^PB^PC    (parité : cas g ou /g)
DELTA=(A^D B^E C^F) (A^B B^C)
Q=/(B.C).X2.X1 (collisions ternaires: 1 paire à 1)
T=/X2.(B+C).X1 (2 paires à 1)
```

```
XORG=T | DELTA
T1=RAND T
T2=/RAND T
DELTA1=DELTA  (A^G)
DELTA2=DELTA /(A^G)
PA:
P1=/(PA RAND) Q
RAND^=PA
XORA=(T1 PA) | (DELTA1 RAND1-3)
XORD=(T2 PA) | (DELTA2 RAND1-3)
PB:
P2=/(PB RAND) Q
RAND^=PB
XORB=(T1 PB) | (DELTA2 RAND2-3)
XORE=(T2 PB) | (DELTA1 RAND2-3)
PC:
P3=/(PC RAND) Q
XORC=(T1 PC) | (DELTA1 RAND3-3)
XORF=(T2 PC) | (DELTA2 RAND3-3)

68 opérations logiques environ, ce qui fait presque une centaine
d'instructions.

erreur /X1 corrigée (oubli d'inversion) ainsi que plusieurs fautes
de frappe...


17H45: l'arbre de dépendences prend deux pages...
il est 3 fois plus complexe que le popcount à 6 entrées.



jeudi 2H du matin:
j'ai réussi à écrire environ 60 instructions de 0-x-x-x
en partant de la fin ("applatissement d'arbre" à la main).
je souffre cruellement du manque de registres et du format
à 2 opérandes : pour 33 instructions logiques, 29 instructions
sont nécessaires pour "bouger" les données, dont 9 pour copier
un registre dans un autre. 27 accès à la mémoire sont effectués,
avec la quasi certitude de "toucher" la cache L1.

Il me reste environ 26 opérations logiques à programmer,
soit encore une cinquantaine d'instructions.



5H10 du matin : 108 instructions !
J'ai réussi ce pari. Le code n'est pas 100% parfait, probablement
98% mais le reste c'est du débugging, du chipottage et un peu de
renommage de registres pour enlever une ou deux instructions.

 pcmpeqb mm0,mm0          ; 108
 pxor mm0,[G]
;$base1__ equ $-4
 movq mm7,[A]
;$base1__ equ $-4
 movq mm3,[D]
;$base1__ equ $-4
 pxor mm7,mm0
 pxor mm3,mm0
 movq mm4,mm7
 pxor mm7,mm3
 pand mm3,mm4             ; 100
 movq mm2,[E]
;$base1__ equ $-4
 movq mm6,mm7
 movq mm1,[B]
;$base1__ equ $-4
 pxor mm2,mm0
 pxor mm1,mm0
 movq [PA],mm3
$base1__ equ $-4
 movq mm3,mm2
 pxor mm4,mm1
```

```
 pand mm2,mm1
 pxor mm3,mm1             ; 90
 movq mm5,[C]
;$base1__ equ $-4
 por mm6,mm3
 pand mm7,mm3
 movq mm3,[F]
;$base1__ equ $-4
 pxor mm5,mm0
 pxor mm3,mm0
 movq [PB],mm2
$base1__ equ $-4
 movq mm2,mm3
 pxor mm5,mm3
 pand mm2,mm3             ; 80
 por mm6,mm5
 pand mm7,mm5
 movq [PC],mm2
$base1__ equ $-4
 movq mm5,mm3
 movq mm3,mm0   ; some renaming to do since here....
 pand mm4,mm7
 pxor mm2,[PB]
$base1__ equ $-4
 movq mm0,mm1
 pxor mm3,mm6
 movq mm6,mm5             ; 70
 pxor mm2,[PA]
$base1__ equ $-4
 pand mm1,mm5
 pxor mm6,mm0
 pand mm3,mm2
 por mm5,mm0
 pandn mm1,mm3
 pand mm4,mm6
 movq mm7,[RAND]
$base1__ equ $-4
 pandn mm2,mm5            ; 60
 movq [Q],mm1
$base1__ equ $-4
 movq mm5,mm7
 movq mm6,mm2
 movq mm1,[G]
;$base1__ equ $-4
 pandn mm7,mm2
 por mm6,mm4
 pxor mm1,[A]
;$base1__ equ $-4
 movq mm3,mm4
 pand mm2,mm5
 movq [T2],mm7
$base1__ equ $-4
 pand mm4,mm1
 movq [XORG],mm6          ; 50
$base1__ equ $-4
 pandn mm1,mm3
 movq mm3,[PA]
$base1__ equ $-4
 movq mm7,mm5
 movq mm0,[RANDA]
$base1__ equ $-4
 pand mm7,mm3
 movq mm6,mm3
 pandn mm7,[Q]
$base1__ equ $-4
 pxor mm5,mm3
 movq [P1],mm7            ; 40
$base1__ equ $-4
 movq mm7,mm0
 pand mm0,mm4
 pand mm3,mm2
 pand mm7,mm1
 pand mm6,[T2]
$base1__ equ $-4
```

```
 por mm3,mm0
 movq mm0,[PB]
$base1__ equ $-4
 por mm6,mm7
 movq mm7,mm5
 movq [XORA],mm3          ; 30
$base1__ equ $-4
 pand mm7,mm0
 movq [XORD],mm6
$base1__ equ $-4
 pandn mm7,[Q]
$base1__ equ $-4
 movq mm3,[RANDB]
$base1__ equ $-4
 movq mm6,mm2
 movq [P2],mm7
 movq mm7,mm3
 pxor mm5,mm0
 pand mm6,mm0
 pand mm3,mm1             ; 20
 pand mm7,mm4
 pand mm0,[T2]
$base1__ equ $-4
 por mm6,mm3
 movq mm3,[RANDC]
$base1__ equ $-4
 por mm7,mm0
 movq mm0,[PC]
$base1__ equ $-4
 pand mm1,mm3
 movq [XORB],mm6
$base1__ equ $-4
 pand mm3,mm4
 movq [XORE],mm7          ; 10
$base1__ equ $-4
 pand mm5,mm0
 pand mm2,mm0
 pand mm0,[T2]
$base1__ equ $-4
 por mm2,mm3
 pandn mm5,[Q]
$base1__ equ $-4
 por mm0,mm1
 movq [XORC],mm2
$base1__ equ $-4
 movq [P3],mm5
$base1__ equ $-4
 movq [XORF],mm0          ; 1
$base1__ equ $-4


 6H30:
J'ai repéré quelques fautes donc je doute que la formule fonctionne
du premier coup...
Sur PMMX, cette fonction devrait durer environ 60 ou 65 cycles,
soit 1 cycle par cellule...
38 accès à la mémoire, soit un tous les deux cycles au moins.
Pour l'essayer, il faudra mettre au point un cadre spécial pour visualiser
les résultats P1, P2, P3, XORx etc PUIS modifier la partie de déplacement.

A+


18H45: SP34 en mise au point, mire binaire en place.
j'ai modifié légérement l'utilisation de quelques variables :
j'ai "réuni" P1 et PA, P2 et PB, P3 et PC. d'autres variables pourraient
encore bénéficier de cette réduction de variable.
Q=XORF, T2=XORC
l'avantage de ce type de code linéaire c'est que s'il bugge, il a peu
de chance de rebooter la machine :-) ce qui accélère le développement.


23H20: j'ai déjà repéré une erreur assez simple (propagation d'une
mauvaise valeur). Une deuxième, moins facile à réparer, propage une
```

```
autre mauvaise valeur, F au lieu de C. l'arbre doit être "touché"
pour résoudre le problème.


jeudi, 0h30: le problème a été résolu. DELTA et Q sont validés

1H10: T résolu.

3H10: eh bien non, T est inversé avec Q. j'ai repris les équations
du départ et je n'avais pas pris en compte l'inversion des entrées.
heureusement il n'y a pas beaucoup de choses à changer, mais quand
même... cela correspond à une observation précédente mais que j'avais
crue erronée. En plus, T et Q se confondent lorsque G est à 1...
Ce dernier bug est plus gênant car l'origine n'est pas connue,
probablement une mauvaise programmation.

formule corrigée:

G^=1
A^=G
B^=G
C^=G
D^=G
E^=G
F^=G
PA=AD
PB=BE
PC=CF
X1=/(A^D+B^E+C^F) (paires identiques)
X2=PA^PB^PC    (parité : cas g ou /g)
DELTA=(A^D B^E C^F) (A^B B^C)
T=/(B.C).X2.X1 (collisions ternaires: 1 paire à 1)
Q=/X2.(B+C).X1 (2 paires à 1)
XORG=T | DELTA
T1=RAND T
T2=/RAND T
DELTA1=DELTA  (A^G)
DELTA2=DELTA /(A^G)
PA:
P1=/(PA RAND) Q
RAND^=PA
XORA=(T1 PA) | (DELTA1 RAND1-3)
XORD=(T2 PA) | (DELTA2 RAND1-3)
PB:
P2=/(PB RAND) Q
RAND^=PB
XORB=(T1 PB) | (DELTA2 RAND2-3)
XORE=(T2 PB) | (DELTA1 RAND2-3)
PC:
P3=/(PC RAND) Q
XORC=(T1 PC) | (DELTA1 RAND3-3)
XORF=(T2 PC) | (DELTA2 RAND3-3)

maintenant, je me demande si je n'avais pas fait
l'erreur en recopiant la formule l'autre jour...


4H15: comme un con, j'avais simplement oublié de recopier
le code dans la deuxième moitié de la section... Gott sei dank,
ç'aurait pu être pire...



4H40: SP35.asm, im00000014.bmp
la formule semble fonctionner, du moins pour la détection des
conditions. on passe à la suite pour vérifier le fonctionnement
plus en détail mais d'abord il faut modifier à fond la procédure
de déplacement.

6H15: heureusement que j'ai le générateur de tables car j'en suis
à 262 labels ! j'ai été forcé d'utiliser $base2__ car $base__ et
$base1__ ont atteint la centaine.
en contrepartie, j'ai une belle faute de protection...
à première vue, elle n'est pas dûe à une erreur de frappe d'un label.
```

```
elle se produit dans la partie de calcul, je vais tout simplement
isoler le problème.

7H: SP36.asm
la GPF semble produite dans le code de déplacement.

-la GPF est produite dans la partie des lignes paires
(enfin, la deuxième copie). la première partie fait
des zolis dessins :-)

l'examen du code a révélé un label mal placé
(après une instruction qui n'accédait pas à la mémoire),
ainsi que quelques oublis de labels et un mauvais registre
modifié, ce qui explique les zolies zébrures...

Maintenant, les particules semblent ne pas se soucier des
parois. je crois que c'est dû à la mauvaise utilisation
de WALLMASK. comme il reste un registre disponible, on va
s'en occuper...


7H30: les murs ne sont pas complétement résolus.
des particules semblent même disparaitre.
du boulot en perspective...

8H30: j'ai trouvé une petite erreur de renommage de registre.
il a fallu comparer terme à terme deux versions du programme
(avant et après renommage) pour trouver l'erreur, principalement
d'inattention, dans la section de déplacement.


SP37 FONCTIONNE !
je procède aux essais des combinaisons de collisions.

9H: j'ai peur d'avoir à me taper les 128 combinaisons, mais il
y a un problème dans le test de tenue de route : il y a des
particules créées, ce qui sature le tunnel... :-/

19H15: la "nuit" a porté conseil et je me suis souvenu
que je n'avais pas testé les collisions DELTA->Q.
elles sont effectivement en cause, car elles sortent
dans le "mauvais sens". il faut inverser DELTA1 et DELTA2,
ou tout simplement le A XOR G... mais ça ne dispense pas
d'une analyse exhaustive des collisions.

19h40: SP37 est "stable", recopié vers sp38. quelque effets
hydrodynamiques apparaissent mais il manque tout 1-x-x-x !


23H30: j'ai fait fonctionner SP37 plusieurs heures et tout va bien.
j'ai ensuite fait SP39: il montre les collisions, en passant par le
buffer temporaire 2D. l'effet visuel est assez étonnant, proche
d'aurore boréales :-)



13 aout 99, 4H30 :
je me suis penché depuis longtemps sur 1-x-x.
Cela passe par un additionneur (popcount) 6->3.
le résultat peut être utilisé a peu de frais pour l'affichage
des densités. Le vieux problème de QUELLE méthode d'"explosion"
(passer de 64 bits à 64 octets) resurgit puisque:

1ère méthode :

movq mm1,mm0 (mm0=debut)
psrlq mm0,1
psrlb mm1,7[varie]  (octet = 0 ou 1)
padd mm1,[ptr]
movq [ptr],mm1

prend 5 instructions, soit 40 pour les 64 octets, ou 20 cycles
dans le cas idéal.
Problème : il faut "désembrouiller" les octets à l'affichage car
```

```
on les récupère dans cet ordre :

paquet 0:  0  8 16 24 32 40 48 56
paquet 1:  1  9 17 25 33 41 49 57
paquet 2:  2 10 18 26 34 42 50 58
paquet 3:  3 11 19 27 35 43 51 59
paquet 4:  4 12 20 28 36 44 52 60
paquet 5:  5 13 21 29 37 45 53 61
paquet 6:  6 14 22 30 38 46 54 62
paquet 7:  7 15 23 31 39 47 55 63

La méthode d'explosion a l'avantage d'être très rapide mais
nécessite un algorithme au moins aussi rapide pour réorganiser
les octets lors de l'affichage.

La deuxième méthode est plus expéditive :

movq mm1,mm0
psrlq mm0,8
punpcklbw mm1,mm1
punpckldw mm1,mm1
punpckldq mm1,mm1
;ici on a l'octet recopié 8 fois.
; il faut donc trier ici les bits :
pand mm1,[=0x8040201008040201]
pcmpeqb mm1,0 [reg ou mem]

ici, on doit faire un masque 0x0101010101010101
mais on peut s'en passer si on fait uns soustraction
car le résultat de pcmpeqb est 0 ou 255,
ce qui épargne une instruction.

Le problème est que la réorganisation ne se fait qu'une
fois lors de l'affichage pour l'algorithme 1, ce qui
est "amorti" par un grand nobre de passes de strip mining.
En plus, cela utilise 2x moins d'instructions et de registres
que la version 2. Quel est le code qui peut
changer ces foutus octets le plus VITE possible ????
une solution inacceptable est de réorganiser les
bits au niveau du tableau : l'algorithme de déplacement
deviendrait infernal.


Un algorithme de "base" c'est à dire idiot, manipulerait
les octets un à un, ce qui nécessiterait probablement
plus de 100 instructions, ce qui est absolument inacceptable.
Je crois que comme pour le cas de l'affichage du curseur,
les instructions punpkXXX peuvent nous aider beaucoup.



les instructions punpckhbw et punpcklbw permettent d'obtenir :
paquet 0:  0  1 16 17 32 33 48 49 \
paquet 1:  8  9 24 25 40 41 56 57 /
paquet 2:  2  3 18 19 34 35 50 51 \
paquet 3: 10 11 26 27 42 43 58 59 /
paquet 4:  4  5 20 21 36 37 52 53 \
paquet 5: 12 13 28 29 44 45 60 61 /
paquet 6:  6  7 22 23 38 39 54 55 \
paquet 7: 14 15 30 31 46 47 62 63 /

l'opération de base est :
mm0=paquet(0+n*2)
mm1=paquet(1+n*2)
movq mm2, mm0 (copie temporaire)
p'0=punpcklbw mm0,mm1
p'1=punpckhbw mm2,mm1

avec 8 isntructions d'opération, et probablement plus de "mouvement"
car il n'y a pas assez de registres.

ensuite avec punpckhwd et punpcklwd on peut avoir:
paquet 0:  0  1  2  3 32 33 34 35 \
paquet 1:  8  9 10 11 40 41 42 43  \
```

```
paquet 2: 16 17 18 19 48 49 50 51  /
paquet 3: 24 25 26 27 56 57 58 59 /
paquet 4:  4  5  6  7 36 37 38 39 \
paquet 5: 12 13 14 15 44 45 46 47  \
paquet 6: 20 21 22 23 52 53 54 55  /
paquet 7: 28 29 30 31 60 61 62 63 /
```

et finalement punpckhdq et punpckldq permet d'obtenir le tableau
désiré en 24 opérations, mais Dieu sait combien de copies de registres...
on a pu passer de O(n) à O(log2(n)) au prix d'une belle prise de tête.

Le choix est donc fait, sur le principe que la complexité doit être
repoussée loin de la boucle principale. Il faut maintenant analyser
l'arbre de dépendences pour déterminer le flot d'instructions.
Une contrainte supplémentaire d'ajoute : les paquets doivent
être affichés dans l'ordre afin que le processeur puisse constituer
des "paquets" pour les "bursts" 256 bits du bus processeur.

L'arbre de dépendences ressemble beaucoup à une FFT−radix8 : l'aspect
log2 de l'arbre et la dépendence de chaque résultat à chaque entrée
fait que le faible nombre de registres ainsi que les instructions à 2
opérandes rendent le travail désespérant, mais enfin on a vu pire...
On doit utiliser des variables en mémoire, probablement 4 au moins.

Pour commencer, on fait en radix−4 puisque le code est utilisé 2 fois,
et ça tient dans les registres. les résultats du premier seront mis
en mémoire et réutilisés à la fin. Cela correspond à une
sorte de balayage en "zig−zag" des gros arbres.

premier résultat pour radix−4 :

```
movq mm0, [0]     **
;                 ||
movq mm4, [8]     ||* *
movq mm2,mm0      ||| |*
movq mm1, [16]     ||*||*
punpcklbw mm0,mm4  ||||||   * *
movq mm3,mm1         ||||  *| |
movq mm5, [32]        |||**|| |
punpckhbw mm2,mm4     |||||||*| *
punpcklbw mm1,mm5       |||||||*|
punpcklbw mm3,mm5         |||||||*
movq mm4,mm0                ||||||*
movq mm5,mm2                 ||||||*
punpcklwd mm0,mm1             ||||||
punpcklwd mm2,mm3               ||||
punpckhwd mm4,mm1                 ||
punpckhwd mm5,mm3                 ||
```

```
movq [t0],mm0
movq [t1],mm2
movq [t2],mm4
movq [t3],mm5
```

```
radix 2:
movq mm6, [40]
movq mm7,
```

radix−4 x 2 : on utilise les 2 registres restants pour assurer l'interim

```
radix 2 commence par:
movq mm6, [40]
movq mm7, [48]
```

on l'entrelace avec le code radix−4 en essayant de
ralonger la "distance" moyenne entre les utilisations
de registres tout en essayant de ne pas saturer l'unité de mémoire
(1 accès par cycle seulement). la distance minimale est

de 3 instructions. dès qu'un registre se libère, on continue le calcul:

```
movq mm0, [0]     **
;cycle mort       ||
movq mm4, [8]     ||* *
movq mm2,mm0      #|| |*
movq mm1, [16]     ||*||*
punpcklbw mm0,mm4  ##|||| *   *        ---- long
movq mm6, [32]    *  |||| |   |        ---- très long
movq mm3,mm1      |  #||| | * |
movq mm5, [24]    |   |||*|*| |
punpckhbw mm2,mm4 |   ##|||||*|   *    --- assez long...
movq mm7, [40]    |*    |||||||   |    ---- très long aussi
punpcklbw mm1,mm5 ||    ##|||||* *|
movq mm4,mm0      ||      #|||||*||    ---- déplacé
punpcklbw mm3,mm5 ||       ##||||||**
movq mm5,mm2      ||         #|||||||*
punpcklwd mm0,mm1 || *        ##||||||
punpckhwd mm4,mm1 || |*         ##||||
movq mm1,[48]     ||*||           ||||
punpcklwd mm2,mm3 |||||*          ##|| ---- déplacé
punpckhwd mm5,mm3 ||||||*           ##
movq mm3,mm6      #|||||||*
movq [t2],mm4     ||||#|||
punpcklbw mm6,mm7 ##|| |||*    *
movq mm4,[56]      ||| ||||* * |
punpckhbw mm3,mm7  #|| ||#||*| | *
movq [t1],mm2       || #| |||| | |
movq mm2,mm1        #|  | ||||*| |
movq [t3],mm5       ||  # |||||| |
movq mm7,mm6      * ||    #||||| |
punpcklbw mm1,mm4 | #|     #||||*|
movq mm5,mm3      |* |      #|||||
punpckhbw mm2,mm4 || |       ##|||*
movq mm4,mm0      || #         ||||  *
punpcklwd mm6,mm1 || |         ##||* |
punpckhwd mm7,mm1 #| |  *    *  #||| |
movq mm1,[t1]      |*|  |    |   ||| |
punpcklwd mm3,mm2  |||* | *  |   ##| |
punpckhwd mm5,mm2  #||| | |  |    #| |  *
punpckldq mm0,mm6   |#| | |  |     #*|  |
movq mm2,mm1        # | | |* |     |||  |
punpckhdq mm4,mm6   | | | || |     #|#* |
movq mm6,[t2]       | |*| || |      | | |
punpckldq mm1,mm3   # #||*|| |      | | |
movq [v0],mm0          ||||| |      # | |
movq mm0,mm6          #||||*|        | |
movq [v32],mm4        |||||||       # |
punpckldq mm6,mm7     ##|||||*         |
movq mm4,[t3]           ||||||*        |
punpckhdq mm2,mm3       |##||||  *     |
movq [v8],mm1           #  ||||  |     |
punpckhdq mm0,mm7          ##||  |*    |
movq mm3,mm4                 |#* ||    |
punpckldq mm4,mm5            |#|*||    #
movq [v16],mm6               # ||||    |
;                              ||||    |
movq [v24],mm4                |#||    |
punpckhdq mm3,mm5             # ||*   #
movq [v40],mm2                  #||
;                                ||
movq [v48],mm0                   #|
;                                 |
movq [v56],mm3                   #
```

Calcul : ma carte video absorbe 28MO/s et j'envoie 64 octets en 20 ns.
tiendra−t−elle le coup ?

en attendant de la faire exploser, j'essaie avec NASM/DEBUG.

−une erreur de recopie : punpcklbw au lieu de punpckhbw
−plus sérieux : l'échange de valeurs à la fin : l'accès à

```
la mémoire vidéo se fait encore plus dans le désordre ! :-(
essayons de voir ce qui peut être réarrangé:

movq mm0, [m0]    ; **
;cycle mort       ; ||
movq mm4, [m8]    ; ||* *
movq mm2,mm0      ; #|| |*
movq mm1, [m16]   ;  ||*||*
punpcklbw mm0,mm4 ;  ##|||| *   *        ---- long
movq mm6, [m32]   ; *  |||| |   |        ---- très long
movq mm3,mm1      ; |  #||| | * |
movq mm5, [m24]   ; |   |||*|*| |
punpckhbw mm2,mm4 ; |   ##|||||*|   *    --- assez long...
movq mm7, [m40]   ; |*    |||||||   |    ---- très long aussi
punpcklbw mm1,mm5 ; ||    ##|||||* *|
movq mm4,mm0      ; ||      #|||||*||    ---- déplacé
punpckhbw mm3,mm5 ; ||       ##||||||**
movq mm5,mm2      ; ||         #|||||||*
punpcklwd mm0,mm1 ; || *        ##||||||
punpckhwd mm4,mm1 ; || |*         ##||||
movq mm1,[m48]    ; ||*||           ||||
punpcklwd mm2,mm3 ; |||||*          ##|| ---- déplacé
punpckhwd mm5,mm3 ; ||||||*           ##
movq mm3,mm6      ; #||||||*
movq [t2],mm4     ; ||||#|||
punpcklbw mm6,mm7 ; ##|| |||*    *
movq mm4,[m56]    ;  ||| ||||* * |
punpckhbw mm3,mm7 ;  #|| ||#||*| | *
movq [t1],mm2     ;   || #| |||| | |
movq mm2,mm1      ;   #|  | ||||*| |
movq [t3],mm5     ;   ||  # |||||| |
movq mm7,mm6      ; * ||    #||||| |
punpcklbw mm1,mm4 ; | #|     #||||*|
movq mm5,mm3      ; |* |      #|||||
punpckhbw mm2,mm4 ; || |       ##|||*
movq mm4,mm0      ; || #         ||||  *
punpcklwd mm6,mm1 ; || |         ##||* |
punpckhwd mm7,mm1 ; #| |  *    *  #||| |
movq mm1,[t1]     ;  |*|  |     |   ||| |
punpcklwd mm3,mm2 ;  |||* | *  |   ##| |
punpckhwd mm5,mm2 ;  #||| | |  |    #| |  *
punpckldq mm0,mm6 ;   |#| | |  |     #*|  |
movq mm2,mm1      ;   # | | |* |     |||  |
punpckhdq mm4,mm6 ;   | | | || |     #|#* |
movq mm6,[t2]     ;   | |*| || |      | | |
punpckldq mm1,mm3 ;   # #||*|| |      | | |
movq [v0],mm0     ;      |||||  |      # | |
movq mm0,mm6      ;      #||||*|        | |
punpckldq mm6,mm7 ;      ##|||||*       | |
movq [v8],mm4     ;        ||||||       # |
punpckhdq mm2,mm3 ;        |##|||   *     |
movq mm4,[t3]     ;        |  |||*  |     |
punpckhdq mm0,mm7 ;        |  ##||  |*    |
movq [v16],mm6    ;        |    #|  ||    |
movq mm3,mm4      ;        |     #* ||    |
movq [v24],mm0    ;        |     || |#    |
punpckldq mm4,mm5 ;        |     #|*|     #
movq [v32],mm1    ;        #      |||     |
punpckhdq mm3,mm5 ;               #|| *   #
movq [v40],mm2    ;                |# |
;                                  |  |
movq [v48],mm4    ;                #  |
;                                     |
movq [v56],mm3    ;                   #

OK: bswap.asm est validé. et en plus, il accède à l'écran dans l'ordre :-)))
58 instructions, temps estimé à 31 cycles P200, soit 16 ns pour 64 octets.
ça va saigner.

22h: j'ai chronométré la boucle sous MSDOS et je trouve environ 130 cycles
processeur. je ne comprends pas car toutes les données et le code
sont en cache et j'ai enlevé les IRQ. Le plus probable problème est
un mauvais respect des règles de pairage, probablement que les instruction
punpckXXXX ont un quota par cycle...
```

```
23H30: la consultation des docs d'Intel a dévoilé une partie du mystère:
l'accès à la mémoire ne se fait qu'avec le pipe U, et il n'y a qu'un seul
shifter par cycle. l'accès à la mémoire pourrait expliquer le reste :
"small load after big store" et vice versa.


movq mm0, [m0]    ; **
;cycle mort       ; ||
movq mm4, [m8]    ; ||* *
movq mm2,mm0      ; #|| |*
movq mm1, [m16]   ;  ||*||*
punpcklbw mm0,mm4 ;  ##|||| *   *        ---- long
movq mm6, [m32]   ; *  |||| |   |        ---- très long
movq mm3,mm1      ; |  #||| | * |
movq mm5, [m24]   ; |   |||*|*| |
punpckhbw mm2,mm4 ; |   ##|||||*|   *    --- assez long...
movq mm7, [m40]   ; |*    |||||||   |    ---- très long aussi
punpcklbw mm1,mm5 ; ||    ##|||||* *|
movq mm4,mm0      ; ||      #|||||*||    ---- déplacé
punpckhbw mm3,mm5 ; ||       ##||||||**
movq mm5,mm2      ; ||         #|||||||*
punpcklwd mm0,mm1 ; || *       ##||||||
punpckhwd mm4,mm1 ; || |*        ##||||
;*
movq mm1,[m48]    ; ||*||           ||||
punpcklwd mm2,mm3 ; |||||*          ##|| ---- déplacé
punpckhwd mm5,mm3 ; ||||||*           ##
movq mm3,mm6      ; #||||||*
movq [t2],mm4     ; ||||#|||
punpcklbw mm6,mm7 ; ##|| |||*    *
movq mm4,[m56]    ;  ||| ||||* * |
punpckhbw mm3,mm7 ;  #|| ||#||*| | *
movq [t1],mm2     ;   || #| |||| | |
movq mm2,mm1      ;   #|  | ||||*| |
movq [t3],mm5     ;   ||  # |||||| |
movq mm7,mm6      ; * ||    #|||||  |
punpcklbw mm1,mm4 ; | #|     #||||*|
movq mm5,mm3      ; |* |      #|||||
punpckhbw mm2,mm4 ; || |       ##|||*
movq mm4,mm0      ; || #         ||||  *
punpcklwd mm6,mm1 ; || |         ##||* |
;*
punpckhwd mm7,mm1 ; #| |  *    *  #||| |
;*
movq mm1,[t1]     ;  |*|  |    |   ||| |
punpcklwd mm3,mm2 ;  |||* | *  |   ##| |
punpckhwd mm5,mm2 ;  #||| | |  |    #| |  *
;*
punpckldq mm0,mm6 ;   |#| | |  |     #*|  |
movq mm2,mm1      ;   # | | |* |     |||  |
punpckhdq mm4,mm6 ;   | | | || |     #|#* |
;*
movq mm6,[t2]     ;   | |*| || |      | | |
punpckldq mm1,mm3 ;   # #||*|| |      | | |
movq [v0],mm0     ;      ||||| |      # | |
movq mm0,mm6      ;     #||||*|        | |
punpckldq mm6,mm7 ;     ##|||||*       | |
;*
movq [v8],mm4     ;        ||||||       # |
punpckhdq mm2,mm3 ;        |##|||   *     |
movq mm4,[t3]     ;        |  |||*  |     |
punpckhdq mm0,mm7 ;        |  ##||  |*    |
movq [v16],mm6    ;        |    #|  ||    |
movq mm3,mm4      ;        |     #* ||    |
movq [v24],mm0    ;        |     || |#    |
punpckldq mm4,mm5 ;        |     #|*|     #
movq [v32],mm1    ;        #      |||     |
punpckhdq mm3,mm5 ;               #|| *   #
movq [v40],mm2    ;                |# |
;                                  |  |
movq [v48],mm4    ;                #  |
;                                     |
```

```
movq [v56],mm3    ;                   #
```

En tenant compte des règles de pairage, il y a 6 problèmes détectés,
six cycles de perdus. Cela n'explique pas la différence entre les 34 cycles
théoriques et les 130 mesurés. on m'en vole 100 ! 3/4 de perte.
Je préfère arrêter ici car de toutes manières la mémoire vidéo ne peut pas
absorber un tel débit. c'est équivalent à envoyer 64 octets à 15MHz, soit
960MO/s alors que le bus PCI est limité en théorie à 133MO/s.
Bonne nouvelle cependant : ça laisse du "temps" pour rajouter des traitements
statistiques intermédiaires :-/ Le PII en profitera moins cependant.

Minuit :
Pour afficher, nous avons donc la procédure de "retournement" des
octets. Pour voir quelque chose, il faut aussi la palette adaptée.
Le projet prévoit un algorithme min/max d'optimisation de la palette
au fur et à mesure du calcul. Pour l'instant, on ne peut pas trop se
le permettre car le reste est plus important (collisions 1-x-x-x)
alors on met une palette fixe. On ne va pas pouvoir afficher les
murs par contre pour l'instant. seul le zoom 1:1 sera implémenté bien
sûr.

Pour la palette, elle commence par le noir et prend des niveaux de gris
de plus en plus clairs. il suffit de la programmer.

```
 xor al,al
 mov dx,3C8h
 out dx,al
 inc dl
loop_palette_:
 out dx,al
 out dx,al
 out dx,al
 add al,12
 jnc loop_palette_ ; boucle 21 fois
[rajouté à SP40]
```

Ensuite, pour voir qqc, il faut qqc à voir. On va commencer par
les collisions : ça prend un seul bit. Je me suis cassé le bec
sur le champ de vélocité.

La procédure d'accumulation utilise une méthode légèrement différente
pour deux raisons : la première était mauvaise, et elle utilisait deux
décalages. je préfère utiliser un registre supplémentaire et éviter
quelques dégâts en utilisant un masque. voilà ce que ça donne
quand on essaie d'être plus malin qu'on ne l'est.

```
pxor mm1,mm1
pcmpeqb mm2,mm2

movq mm0,[COLL] ; là où la collision est sockée
$
psubb mm1,mm2   ; mm1=0x0101010101010101
                ; mm2 libre
movq mm2,mm0
movq mm3,mm0

pand mm0,mm1
psrlq mm2,1

paddb mm0,[m0]
movq mm4,mm2

psrlq mm3,2
pand mm2,mm1

paddb mm2,[m8]
psrlq mm4,2

movq [m0],mm0
movq mm5,mm3

movq [m8],mm2
```

```
pand mm3,mm1

movq mm6,mm4
pand mm4,mm1

paddb mm3,[m16]
psrlq mm5,2

paddb mm4,[m24]
movq mm7,mm5

psrlq mm6,2
pand mm5,mm1

movq [m16],mm3
movq mm0,mm6

movq [m24],mm4
psrlq mm7,2

paddb mm5,[m32]
pand mm6,mm1

pand mm7,mm1
psrlq mm0,2

movq [m32],mm5
pand mm0,mm1

paddb mm6,[m40]

paddb mm7,[m48]

paddb mm0,[m56]

movq [m40],mm6

movq [m48],mm7

movq [m56],mm0



La conclusion de cette journée est que non seulement FHP est memory bound
mais MMX l'est encore plus ! 24 cycles pour faire 8 AND...
Je ne sais pas ce que ça aurait donné si j'avais triplé l'arbre
au lieu de me contenter de le doubler. ça n'aurait pas changé que
Intel c'est merdique.

3H10: comme on pouvait s'en douter, il y a un petit bug.
je ne crois pas qu'il s'agisse d'assembleur, on verra ça.
Mais la bonne nouvelle c'est que le code tourne en 26 cycles,
soit seulement 2 de plus que prévu. 186 cycles "à sec" mais
ça boucle à la vitesse escomptée.

3H30: bon, 'spèce d'abruti, le "bug" était que je n'avais pas mis
la procédure de désembrouillage à la fin : "explode1.asm" fonctionne
très bien. "Il n'y a plus qu'à" mettre tout ça dans SP40.ASM.

4H: l'ajout de la fonction d'accumulation ralentit le code de 15% en
zone de performance maximale, ce qui est raisonnable si l'on se souvient
qu'il accède à la grosse zone tampon.

4H20: je viens de me rendre compte qu'une variable temporaire est lue
mais pas écrite. bizarre !
j'avais simplement remplacé le mode d'adressage, enlevant le "t".
je fatigue...
Je remplace t1,t2 et t3 par PA, PB et PC qui existent dèjà et ne
sont pas utilisés. c'est incroyable ce que cette formule génère comme
variables temporaires...

Je renomme SP41.
Premier essai : il faut vider le plan temporaire et élucider le
mystère des bandes verticales.
```

1− une petite modif de la procédure d'initialisation "étend" la
vidange au buffer contigu. mais ce n'est pas suffisant.
2− il faut modifier display_tunnel car il affiche des données
erronnées maintenant que le buffer temporaire est utilisé.
je me contente de le mettre entre parenthèses pour l'instant.
c'est juste une histoire d'interface, elle devra être refondue
plus tard de toute façon.
3− les bandes verticales... là, c'est le bouquet.
La fonction d'accumulation suivie de l'affichage fonctionne
bien mais pas au bon endroit. Cela vient d'une utilisation différente
du buffer pour le déplacement et l'accumulation : le pointeur va
dans un sens utile au déplacement des particules mais donne un résultat
faux pour l'accumulation.
Au premier balayage de fenêtre, la première ligne de buffer
sert pour accumuler ET déplacer la première ligne du tunnel
mais au deuxième balayage, la première ligne du buffer sert
pour la deuxème ligne du tunnel. l'accumulation se fait entre
des lignes différentes ce qui provoque des effets inattendus.
Je pense que le mieux serait de revoir le déplacement, car
l'accumulation manipule une plus grosse masse de données (les 4/5è).

7H50:
Je me demande aussi s'il est intéressant de rendre le buffer d'accumulation
statique afin d'améliorer l'affichage. Cela occupe deux fois plus
de mémoire mais ne change pas vraiment le temps d'exécution car
la bande passante est la même je crois. La gestion des pointeurs
sera plus simple aussi, de même que l'affichage.

C'est parti pour NA00.asm.

D'abord il faut changer le format des cellules:
mettre les accumulateurs au milieu, à côté de WALL. Cela fait
des cellules de 128 octets.

Il faut aussi changer le format et le calcul d'une tonne de formules
et de SMC. ça va être marrant mais au moins l'affichage ne sera pas
soumis aux caprices du calcul.

Je ne pense pas qu'il soit nécessaire d'augmenter la taille de la
fenêtre de strip mining car la pratique a montré que les 32 lignes
étaient rarement utilisées.

L'algorithme de balayage et de déplacement n'est pas modifié.

Dimanche 1h30:
hier matin j'ai compilé l'aditionneur 6/7 bits qui va s'occuper
à la fois de 1−x−x et de l'affichage des densités. C'est un aditionneur
2*3 bits suivi d'un 2 bits sans entrée de retenue pui d'un incrémenteur
pour le 7ème bit, afin d'avoir le résultat de l'adition sur 6 bits
avant. C'était assez facile à faire, avec l'habitude.

```
movq mm0,[A]
;*
movq mm2,[B]
movq mm1,mm0
movq mm3,[D]
pand mm0,mm2 ; HA1
movq mm5,[E]
movq mm4,mm3
movq mm6,[C]
pxor mm2,mm1 ; HA1
movq mm7,[F]
pand mm3,mm5 ; HA3
pxor mm5,mm4 ; HA3
movq mm1,mm2
movq mm4,mm7
pand mm2,mm6 ; HA2
pand mm7,mm5 ; HA4
pxor mm6,mm1 ; HA2
pxor mm5,mm4 ; HA4
```

```
por mm0,mm2
por mm3,mm7
movq mm4,mm6
movq mm1,mm0
pand mm6,mm5 ; HA5
pand mm0,mm3 ; HA6
pxor mm5,mm4 ; HA5
movq mm7,[G]
pxor mm3,mm1 ; HA6
movq mm4,mm5
movq mm1,mm3
pand mm5,mm7 ; HA8
pand mm3,mm6 ; HA7
pxor mm6,mm1 ; HA8
por mm0,mm3
pxor mm7,mm4 ; HA8
movq mm4,mm5
pand mm6,mm5 ; HA9
pxor mm5,mm4 ; HA9
movq [ST],mm0
por mm0,mm6
movq [S0],mm7
;*
movq [S1],mm5
;*
movq [S2],mm0

39 instructions, 20 opérations logiques en oubliant un étage.
42 inst. 22 OP en corrigé :−| C'est raisonnable si on considère
que cela effectue 2 opérations à la fois.

Maintenant, il faut savoir comment effectuer les collisions 1−x−x−x
en essayant de réduire les ubiquités avec 0−x−x−x !


6H45: j'ai levé un lièvre. je divise XORx en EPSILON1 et EPSILON2.
cela permet de traiter 3 directions seulement à la fois, ce qui réduit
la pression sur les registres. XORx est alors EPSILON1x AND EPSILON2x.
le AND sera réduit au fur et à mesure du calcul ce qui permet de mettre
les EPSILON dans les emplacements XORx.

Il faudra donc refaire, revoir et repenser les collisions quaternaires.
je crois que les anciennes equations tiennent toujours, mais on peut s'aider
des nouvelles.



Jeudi 19 août 1999, 1h30:
j'ai trouvé une formule satisfaisante:

t1 = F xor B
t2 = /t1 and (A xor B)
EA1 = t2 and (B xor G)
PA1 = EA1 and A
EA2 = NOT ((A EA1) OR RAND1/2t1 or PA1 or (RAND2/ t2 B)))

t1 = A xor C
t2 = /t1 and (B xor C)
EB1 = t2 and (C xor G)
PB1 = EB1 and B
EB2 = NOT ((B EB1) OR RAND1/2t1 or PB1 or (RAND2/ t2 C)))

t1 = B xor D
t2 = /t1 and (C xor D)
EC1 = t2 and (D xor G)
PC1 = EC1 and C
EC2 = NOT ((C EC1) OR RAND1/2t1 or PC1 or (RAND2/ t2 D)))

t1 = F xor B
t2 = /t1 and (D xor E)
ED1 = t2 and (E xor G)
PA2 = ED1 and D
ED2 = NOT ((A ED1) OR RAND1/2t1 or PA2 or (RAND2/ t2 E)))
PA = PA1 PA2
```

```
XORA = EA2 ED1
XORD = EA1 ED2
XORG = XORA or XORD

t1 = F xor B
t2 = /t1 and (E xor F)
EE1 = t2 and (F xor G)
PB2 = EE1 and E
EE2 = NOT ((A EE1) OR RAND1/2t1 or PB2 or (RAND2/ t2 F)))
PB = PB1 PB2
XORB = EB2 EE1
XORE = EB1 EE2
XORG = XORG or XORB or XORE

t1 = F xor B
t2 = /t1 and (F xor A)
EF1 = t2 and (A xor G)
PC2 = EA1 and F
EF2 = NOT ((A EF1) OR RAND1/2t1 or PC2 or (RAND2/ t2 E)))
PC = PC1 PC2
XORC = EC2 EF1
XORF = EC1 EF2
XORG = XORG or XORC or XORF



estimation: 150-200 instructions. PA-PC n'est pas complétement
calculé mais le gros du calcul est fait.

Le calcul est organisé autour d'une utilisation des registres
"à tour de rôle" pour contenir les données des 3 entrées.
J'en ai profité pour influencer l'equation afin de n'utiliser
qu'une fois une des variables d'entrées afin de disposer d'un
maximum de registres...
RAND1/2, RAND2/3, G(xor st) sont conservés en mémoire.
les EPSILONx sont conservés dans XORx.
Je crois même que j'en ai oublié XORG dans l'histoire...
<corrigé>

Pour PA-PC on verra après.



5h45: j'ai fini le graphe des collisions ternaires.
il tient sur six pages. je n'ai pas fait XORG.
à cause de sa taille,
et ses 200 instructions (à vue de nez) il faudra faire une vérification
sérieuse à toutes les étapes du codage, ainsi que des tests sur le terrain
pour éviter tous les problèmes.
Le code se "pipeline" assez bien, les six étapes se recouvrent entre voisines
ce qui rend sa mise en boucle sous-efficace car l'allocation des registres
est délicate. j'ai quand même trouvé des registres "fixes" ce qui
facilite le codage.
maintenant, j'ai les doigts pleins de taches de feutre de couleur...
mais mes doigts ne sont pas des graphes !



23H je viens de me rappeler qu'il faut aussi voir comment on peut
"entrelacer" le code d'adition et celui de collision...

0H30: une petite modification dans le code de l'adition est suffisante:
ST reste dans MM0 et MM6 est ORé avec mm0 avant d'écrire S2.

movq mm0,[A]  ; U
;*
movq mm2,[B]  ; U
movq mm1,mm0
movq mm3,[D]  ; U
pand mm0,mm2  ; HA1
movq mm5,[E]  ; U
movq mm4,mm3
movq mm6,[C]  ; U
```

```
pxor mm2,mm1  ; HA1
movq mm7,[F]  ; U
pand mm3,mm5  ; HA3
pxor mm5,mm4  ; HA3
movq mm1,mm2
movq mm4,mm7  ; U
pand mm2,mm6  ; HA2
pand mm7,mm5  ; HA4
pxor mm6,mm1  ; HA2
pxor mm5,mm4  ; HA4
por mm0,mm2   ;    mm2 free
movq mm2,[G]  ; mm2=G original
por mm3,mm7
movq mm4,mm6
movq mm1,mm0
pand mm6,mm5  ; HA5
pand mm0,mm3  ; HA6
pxor mm5,mm4  ; HA5
movq mm7,mm2
pxor mm3,mm1  ; HA6
movq mm4,mm5
movq mm1,mm3  ; U
pand mm5,mm2  ; HA8
pand mm3,mm6  ; HA7
pxor mm6,mm1  ; HA8
por mm0,mm3   ;     mm3 free  MM0=ST
pxor mm7,mm4  ; HA8
movq mm3,[B]  ; mm3 = B original
movq mm4,mm6 ; erreur de recopie
pand mm6,mm5  ; HA9
pxor mm5,mm4  ; HA9 mm4 free
movq mm4,[A]  ; mm4 = A original
por mm6,mm0   ; last computation for S2
movq [S0],mm7 ; mm7 free
pxor mm2,mm0  ; mm2=G!
movq [S1],mm5 ; mm5 free
movq mm5,mm3  ; copy B
movq [S2],mm6 ; mm6 free
movq mm7,mm3  ; copy B
movq mm6,[F]
pxor mm3,mm0  ; B!
movq [G2],mm2 ; save G!
pxor mm5,mm4  ; BxA
movq mm1,[RAND2/3]
pxor mm7,mm6  ; BxF
pxor mm4,mm0  ; A!
pxor mm2,mm3  ; GxB
pand mm1,mm3  ; RAND2/3 B
movq mm6,mm7  ; copy BxF
pandn mm7,mm5 ; BxA (BxF)
movq mm5,mm4  ; copy A!
pand mm1,mm7  ; U
pand mm2,mm7  ; EPSILON_A1
movq mm7,mm4
por mm6,mm1   ;
pandn mm5,mm2 ; U
pand mm7,mm2  ; PA
movq [XORD],mm2 ; save EA1
por mm6,mm7
movq mm1,[C]  ; ;;; ; ; ; ;
pcmpeqb mm2,mm2
pand mm6,[RAND1/2]
;
movq [PA],mm7 ; mm7 free
por mm6,mm5   ; mm5 free
;
pxor mm6,mm2  ; EA2 mm2 free
;
;
;
;
movq [XORA],mm6 ; save EA2 mm6 free
```

Ce joli petit code a été chronométré à 51 cycles pour 41 cycles

théoriques. je me demande d'où vient ce blocage.

Essai de l'aditionneur: S1 ne marche pas.
erreur de recopie : copie d'un mauvais registre, ce qui execute
l'equivalent de xor m5,m5 ce qui fait que m5 est nul quand il est
sauvé dans la mémoire.

ST et S0−2 sont validés.

PA est faux, par définition de formule. il capte F/AB/G et /FA/BG
alors qu'il devrait capter /FA/B/G et F/ABG.
il y a donc une bifurcation dans la formule que j'avais mal définie.

lundi 30 août, 4h50 du matin :
après le fiasco de l'autre fois, j'ai fais plus attention en construisant
ma formule. je suis parti du cas général afin de simplifier le "gros"
de la troupe (les 5 autres parties) et profiter de l'expérience
de l'applatissement précédent où j'arrivais à répéter certaines parties.

j'ai d'abord conçu la formule, que j'ai explosée en graphe de dépendences.
j'ai brisé une longue branche, ce qui m'oblige à utiliser une variable en
mémoire. j'ai éjecté ST des registres car il ne sert pas beaucoup
une fois que le calcul est lancé.

en faisant le graphe j'ai fait attention à ne pas prendre de décisions
à la va−vite, j'ai utilisé des noeuds à plusieurs entrées : la
commutativité du OR par exemple permet de générer plusieurs graphes.
Pour choisir le meilleur, j'ai analysé le "chemin critique" en numérotant
les noeuds en partant par le dernier. j'ai ainsi pu réduire le chemin
critique global et garder des branches assez courtes. le chemin critique
est d'environ 8 opérations.

Ensuite il faut trouver quelles branches deviennent des "faisceaux",
des branches correspondant à un registre à la fois. la technique
est d'abord de voir ce que les instructions obligent à faire, comme
NAND qui 'force' un 'chemin' lors des bifurcations : le registre
qui contient le résultat est celui dont l'entrée est inversée.

Pour le reste de la constitution des faisceaux, il faut profiter de
la commutativité de certains opérateurs et ne pas hésiter à
inverser des opérandes si nécessaire. et la dernière méthode utilisée est:
constituer des faisceaux les plus courts possibles : en partant
du bas, on remonte et lorsqu'une bifurcation est commutative, on
choisit la branche qui donne le chemin le plus court, par exemple
trouver le chemin le plus court vers un chargement de donnée
à partir d'un registre ou de la mémoire.

Ensuite, l'algorithme est très simple : on prend la dernière branche
(celle de la fin en bas), on remonte le faisceau que l'on alloue à un
registre. ensuite, lorsque le faisceau se termine, on choisit un autre
faisceau qui se termine juste avant le début du faisceau qu'on
vient de remonter, on alloue ce faisceau au registre courant, on remonte
le faisceau, etc.
arrivé en haut du graphe, on revient en bas et on recommence avec
un autre registre pour le faisceau qui finit le plus bas : on le
remonte, on choisit un faisceau qui se termine juste avant ce point etc.

arrivé là, j'ai eu la chance de voir que 6 registres étaient nécessaires.
le graphe était donc assez "étroit" mais il faut une variable en mémoire.
21−23 opérations logiques environ par "tour", 12 opérations de mouvement,
environ 14 accès à la mémoire. 300 opérations, ou 150 cycles pour les 6
tours.

voyons le code:
j'ai attribué 5 registres "fixes" qui délimitent chaque tour,
et 3 registres "tournant" mm0, mm1 et mm2 qui contiennent les "entrées"
de l'eq. Comme avant, l'entrée qui n'est plus utilisée pour le tour
suivant sert de faisceau pour le tour courant. c'est cette valeur qui
nécessite une variable en mémoire. les registres tournants sont
nommés RA, RB et RC dans le code.

```
          RC      RB     RA
tour A    F mm0   A mm1  B mm2
tour B    A mm1   B mm2  C mm0
tour C    B mm2   C mm0  D mm1
tour D    C mm0   d mm1  e mm2
tour E    d mm1   e mm2  f mm0
tour F    e mm2   f mm0  a mm1


ensuite je n'ai qu'à "lire" le graphe, en partant du haut et en
suivant chaque "ligne", chaque "étape" du chemin critique, ici en
partant de la droite de la ligne. cette amélioration à ma technique de
"compilation" permet d'avoir des distances assez égales entre deux
opérations interdépendantes :-) (si le graphe s'y prette)

ligne1:
movq mm5,RB
movq mm4,RA
movq [RC(scratch)],RC

ligne2:
movq mm6,RC
movq mm3,[G!]
pxor mm5,RC
pxor mm4,RC

ligne3:
movq mm7,RB
pand mm6,RA
pandn mm3,RB
pandn mm4,mm5
movq mm5,RC
pxor mm5,[G!]

ligne4:
pandn mm7,mm6
pand mm3,mm4
pand RC,mm4
movq mm6,RB
movq mm4,RB
pxor mm5,RA

ligne5:
por mm7,mm3
(pand mm3,[PA])
pand mm6,RC
(pand RC,[EPSILONA1])
pandn mm4,mm5
movq mm5,[RC(scratch)]

ligne6:
por mm6,mm4
pand mm7,[RAND1/3]
por mm5,RB

ligne7:
[
por mm6,mm7
[
por mm5,RA
pcmpeqb mm7,mm7

ligne8:
pand mm6,[rand1/2]
pxor mm7,mm5

ligne 9:
por mm7,mm6

ligne10:
(pand mm7,[EPSILANA2])
```

```
movq [EPSILONA2],mm7

je n'ai pas pris en compte le scheduling des accès à la mémoire
ainsi que les problèmes PII/PMMX aux règles différentes,
je reste en PMMX et je crois que ça suffira.
de plus j'ai lâchement oublié la fabrication d'un des termes
tournants à l'entrée de l'équation...


movq RA,[B]
;
pxor RA,[ST]
;
movq mm5,RB
movq mm4,RA
movq [RC(scratch)],RC
;;
movq mm6,RC
movq mm3,[G2]
pxor mm5,RC
pxor mm4,RC
;;
movq mm7,RB
pand mm6,RA
pandn mm3,RB
pandn mm4,mm5
movq mm5,RC
pxor mm5,[G2]
;;
pandn mm7,mm6
pand mm3,mm4
pand RC,mm4
movq mm6,RB
movq mm4,RB
pxor mm5,RA
;;
por mm7,mm3
(pand mm3,[PA])
pand mm6,RC
(pand RC,[EPSILONA1])
pandn mm4,mm5
movq mm5,[RC(scratch)]
;;
por mm6,mm4
pand mm7,[RAND1/3]
por mm5,RB
;;
movq [PA],mm3
por mm6,mm7
movq [EPSILONA1],RC
por mm5,RA
pcmpeqb mm7,mm7
;;
pand mm6,[rand1/2]
pxor mm7,mm5
;;
por mm7,mm6
;;
(pand mm7,[EPSILANA2])


movq [EPSILONA2],mm7

************************************

tentative de "fusion" :
(avec contre-vérification au passage)

movq RA,[B]
;x
pxor RA,[ST]
;x
movq mm5,RB
movq mm4,RA
```

```
movq [RC(scratch)],RC
movq mm6,RC
movq mm3,[G2]
pxor mm5,RC
pxor mm4,RC
_____________________
movq mm7,RB
pand mm6,RA
pandn mm3,RB
pandn mm4,mm5
movq mm5,RC
pxor RC,[G2]
pandn mm7,mm6
pand mm3,mm4
pand RC,mm4
movq mm6,RB
movq mm4,RB
pxor mm5,RA
por mm7,mm3
(pand mm3,[PA])
pand mm6,RC
(pand RC,[EPSILONA1])
pandn mm4,mm5
movq mm5,[RC(scratch)]
por mm6,mm4
pand mm7,[RAND1/3]
por mm5,RB
movq [PA],mm3
por mm6,mm7
movq [EPSILONA1],RC
por mm5,RA
***movq RA,[B]
pcmpeqb mm7,mm7
***pxor RA,[ST]
pxor mm7,mm5
pand mm6,[rand1/2]
***movq mm5,RB
por mm7,mm6
***movq mm4,RA
(pand mm7,[EPSILONA2])
***movq mm6,RC
***movq mm3,[G2]
***pxor mm5,RC
***movq [RC(scratch)],RC
***pxor mm4,RC
movq [EPSILONA2],mm7
_____________________

****************************
tentative de terminaison:


_____________________
movq mm7,RB
pand mm6,RA
pandn mm3,RB
pandn mm4,mm5
movq mm5,RC
pxor RC,[G2]
pandn mm7,mm6
pand mm3,mm4
pand RC,mm4
movq mm6,RB
movq mm4,RB
pxor mm5,RA
por mm7,mm3
pand mm3,[PA]
pand mm6,RC
pand RC,[EPSILONA1]
pandn mm4,mm5
movq mm5,[RC(scratch)]
por mm6,mm4
pand mm7,[RAND1/3]
por mm5,RB
```

```
movq [PA],mm3
por mm6,mm7
movq [EPSILONA1],RC
por mm5,RA
;
pcmpeqb mm7,mm7
;
pxor mm7,mm5
pand mm6,[rand1/2]
;
por mm7,mm6
;
pand mm7,[EPSILONA2]
;
;
;
;
;
movq [EPSILONA2],mm7
_______________________

*********************************


amorçage avec l'adition:

movq mm7,[A]  ; U
;
movq mm3,[B]  ; U
movq mm4,mm7
movq mm2,[C]  ; U
pand mm7,mm3  ; HA1
movq mm5,[D]  ; U
movq mm1,mm2
movq mm0,[E]  ; U
pxor mm3,mm4  ; HA1
movq mm6,[F]  ; U
pand mm2,mm5  ; HA3
pxor mm5,mm1  ; HA3
movq mm4,mm3
movq mm1,mm6  ; U
pand mm3,mm0  ; HA2
pand mm6,mm5  ; HA4
pxor mm0,mm4  ; HA2
pxor mm5,mm1  ; HA4
por mm7,mm3   ;    mm2 free
movq mm3,[G]  ; mm3=G original (copie pour le XOR ensuite)
por mm2,mm6
movq mm1,mm0
movq mm4,mm7
pand mm0,mm5  ; HA5
pand mm7,mm2  ; HA6
pxor mm5,mm1  ; HA5
movq mm6,mm3   ; copie de G pour le calcul
pxor mm2,mm4  ; HA6
movq mm1,mm5
movq mm4,mm2  ; U
pand mm5,mm3  ; HA8
pand mm2,mm0  ; HA7
pxor mm0,mm4  ; HA8
por mm7,mm2   ;     mm2 free  MM7=ST
pxor mm6,mm1  ; HA8 (G xor MM1->mm6, mm3=G)
 movq mm2,[B]  ; mm2 = B original
movq mm1,mm0 ;!!!
pand mm0,mm5  ; HA9
pxor mm5,mm1  ; HA9 mm1 free
 movq mm1,[A]  ; mm1 = A original
por mm0,mm7   ; last computation for S2
movq [S0],mm6 ; mm6 free
 pxor mm3,mm7  ; mm3=G!    ****
movq [S1],mm5 ; mm5 free
 pxor mm2,mm7  ; B!
movq [S2],mm0 ; mm0 free
 pxor mm1,mm7  ; A!
```

```
 movq mm0,[F]
movq mm5,mm1
 movq [G2],mm3 ; save G!
 pxor mm0,mm7 ; F!
movq [ST],mm7
movq mm4,mm2 ; copie B!
movq mm6,mm0
pxor mm5,mm0
pxor mm4,mm0
;stall, V
movq [RCscratch],mm0
_______________________

P:mm3:G2
M:mm7:ST
O:mm4
R:mm5
K:mm6
          RC  L    RB  N    RA  Q
tour A    F mm0    A mm1    B mm2

J'ai renommé les registres de la partie d'addition
afin d'intégrer celle-ci dans l'amorce du premier tour.
le code semble correct mais il faut le tester pour en être sûr.
en plus, l'additionneur n'est pas fantastique, j'aurais
peut-être dû le refaire avec la nouvelle méthode.



midi 20:
S0, S1 et S2 sont OK mais ST est dans les choux.
j'ai peur que le renommage des registres l'ait
perturbé. mais ce n'est pas grave : ce n'est qu'une
trentaine d'instructions...

23H : ST était bêtement surécrit par une vieille
partie du code dans lequel j'ai greffé l'extrait.
faute d'inattention.


23H45: PA et EPSILONA1 sont validés.
il faut encore valider EPSILONA2 avec les 4 combinaisons d'entrées
pour les variables aléatoires.

Minuit: houra ! la formule fonctionne.
2 ennuis tout proches cependant :
-1) RANDOM 1/2 à munir d'une version inversée.
 dans mon empressement, je n'ai pas inclus de version PANDN
 dans le graphe.
-2) les XORx ne seront peut-être pas suffisants, 3 autres
 variables sont nécessaires car l'ordre de chargement a changé:
 il n'y a plus de recouvrement avec les registres.

Tout de même le grand moment est arrivé : la mise bout à bout
de toutes les sections. dès que ça fonctionne, il faut chronométrer
tout ça !


3H30: je me suis trompé pour les XORx, ils se comportent comme prévu
auparavant. reste tout de même le problème de RAND1/2 qu'il faut inverser.
Finalement, quand c'est bien conçu, le tout est assez "indolore"...


4H: bonne nouvelle : PA, PB et PC fonctionnent.
mauvaise nouvelle: ils ne font que la détection, il faudra
rajouter encore un peu de code (déjà connu: sélection progressive)
pour faire la phase de "décision" en utilisant RANDOM.

Pour ce qui est du code, il y a en tout 300 lignes et 750 octets
de binaire en mode 16 bits, ce qui donnera environ 1KO en
mode 32 bits. Pour ce qui est de savoir si ça va marcher,....


J'ai récupéré SP39 car je ne peux rien faire avec NA00 qui est inachevé.
```

```
je m'en sers seulement pour tester les collisions.

C'est bizarre mais en regardant ces vieux sources, je me rends compte que
j'ai oublié de calculer XORG...



3 septembre, 21H:
j'ai résolu le problème de XORG dans la partie de translation.
je teste SP51 "à vide", sans la partie calcul.
la partie de translation a besoin d'un grand coup d'optimisation !!!


adaptation du coeur du calcul:



  ;
  ;  POPCOUNT:
  ;

  movq mm7,[edi+A]  ; U
  ;
  movq mm3,[temp_B]  ; U
$base3__ equ $-4
  movq mm4,mm7
  movq mm2,[temp_C2]  ; U
$base3__ equ $-4
  pand mm7,mm3  ; HA1
  movq mm5,[edi+D]  ; U
  movq mm1,mm2
  movq mm0,[edi+E]  ; U
  pxor mm3,mm4  ; HA1
  movq mm6,[edi+F]  ; U
  pand mm2,mm5  ; HA3
  pxor mm5,mm1  ; HA3
  movq mm4,mm3
  movq mm1,mm6  ; U
  pand mm3,mm0  ; HA2
  pand mm6,mm5  ; HA4
  pxor mm0,mm4  ; HA2
  pxor mm5,mm1  ; HA4
  por mm7,mm3   ;    mm2 free
  movq mm3,[edi+G]  ; mm3=G original (copie pour le XOR ensuite)
  por mm2,mm6
  movq mm1,mm0
  movq mm4,mm7
  pand mm0,mm5  ; HA5
  pand mm7,mm2  ; HA6
  pxor mm5,mm1  ; HA5
  movq mm6,mm3   ; copie de G pour le calcul
  pxor mm2,mm4  ; HA6
  movq mm1,mm5
  movq mm4,mm2  ; U
  pand mm5,mm3  ; HA8
  pand mm2,mm0  ; HA7
  pxor mm0,mm4  ; HA8
  por mm7,mm2   ;     mm2 free  MM7=ST
  pxor mm6,mm1  ; HA8 (G xor MM1->mm6, mm3=G)
   movq mm2,[temp_B]  ; mm2 = B original
$base3__ equ $-4
  movq mm1,mm0 ;!!!
  pand mm0,mm5  ; HA9
  pxor mm5,mm1  ; HA9 mm1 free
   movq mm1,[edi+A]  ; mm1 = A original
  por mm0,mm7   ; last computation for S2
  movq [S0],mm6 ; mm6 free
$base3__ equ $-4
   pxor mm3,mm7  ; mm3=G!    ****
  movq [S1],mm5 ; mm5 free
$base3__ equ $-4
   pxor mm2,mm7  ; B!
  movq [S2],mm0 ; mm0 free
$base3__ equ $-4
```

```
   pxor mm1,mm7  ; A!
   movq mm0,[edi+F]
  movq mm5,mm1
   movq [G2],mm3 ; save G!
$base3__ equ $-4
   pxor mm0,mm7 ; F!
  movq [ST],mm7
$base3__ equ $-4
  movq mm4,mm2 ; copie B!
  movq mm6,mm0
  pxor mm5,mm0
  pxor mm4,mm0
  ;stall, V
  movq [RCscratch],mm0
$base3__ equ $-4

  ;---------------
  ; A: RC:F:mm0 RB:A:mm1 RA:B:mm2
  ;---------------
  movq mm7,mm1
  pand mm6,mm2
  pandn mm3,mm1
  pandn mm4,mm5
  movq mm5,mm0
  pxor mm0,[G2]
$base3__ equ $-4
  pandn mm7,mm6
  pand mm3,mm4
  pand mm0,mm4
  movq mm6,mm1
  movq mm4,mm1
  pxor mm5,mm2
  por mm7,mm3
  pand mm6,mm0
  pandn mm4,mm5
  movq mm5,[RCscratch]
$base3__ equ $-4
  por mm6,mm4
  pand mm7,[RAND03]
$base3__ equ $-4
  por mm5,mm1
  movq [PA],mm3
$base3__ equ $-4
  por mm6,mm7
  movq [XORD],mm0
$base3__ equ $-4
  por mm5,mm2
     movq mm0,[temp_C2]  ;***
$base3__ equ $-4
  pcmpeqb mm7,mm7
     pxor mm0,[ST] ;***
$base3__ equ $-4
  pxor mm7,mm5
  pand mm6,[RAND02]
$base3__ equ $-4
     movq mm5,mm2  ;***
  por mm7,mm6
     movq mm4,mm0  ;***
     movq mm6,mm1  ;***
     movq mm3,[G2] ;***
$base3__ equ $-4
     pxor mm5,mm1  ;***
     movq [RCscratch],mm1 ;***
$base3__ equ $-4
     pxor mm4,mm1  ;***
  movq [XORA],mm7
$base3__ equ $-4

  ;---------------
  ; B RC:A:mm1 RB:B:mm2 RA:C:mm0
  ;---------------
  movq mm7,mm2
  pand mm6,mm0
  pandn mm3,mm2
```

```
  pandn mm4,mm5
  movq mm5,mm1
  pxor mm1,[G2]
$base3__ equ $-4
  pandn mm7,mm6
  pand mm3,mm4
  pand mm1,mm4
  movq mm6,mm2
  movq mm4,mm2
  pxor mm5,mm0
  por mm7,mm3
  pand mm6,mm1
  pandn mm4,mm5
  movq mm5,[RCscratch]
$base3__ equ $-4
  por mm6,mm4
  pand mm7,[RAND03]
$base3__ equ $-4
  por mm5,mm2
  movq [PB],mm3
$base3__ equ $-4
  por mm6,mm7
  movq [XORE],mm1
$base3__ equ $-4
  por mm5,mm0
     movq mm1,[edi+D]
  pcmpeqb mm7,mm7
     pxor mm1,[ST]
$base3__ equ $-4
  pxor mm7,mm5
  pand mm6,[RAND12]
$base3__ equ $-4
     movq mm5,mm0
  por mm7,mm6
     movq mm4,mm1
     movq mm6,mm2
     movq mm3,[G2]
$base3__ equ $-4
     pxor mm5,mm2
$base3__ equ $-4
     movq [RCscratch],mm2
     pxor mm4,mm2
  movq [XORB],mm7
$base3__ equ $-4

  ;---------------
  ; C
  ;---------------
  movq mm7,mm0
  pand mm6,mm1
  pandn mm3,mm0
  pandn mm4,mm5
  movq mm5,mm2
  pxor mm2,[G2]
$base3__ equ $-4
  pandn mm7,mm6
  pand mm3,mm4
  pand mm2,mm4
  movq mm6,mm0
  movq mm4,mm0
  pxor mm5,mm1
  por mm7,mm3
  pand mm6,mm2
  pandn mm4,mm5
  movq mm5,[RCscratch]
$base3__ equ $-4
  por mm6,mm4
  pand mm7,[RAND13]
$base3__ equ $-4
  por mm5,mm0
  movq [PC],mm3
$base3__ equ $-4
  por mm6,mm7
  movq [XORF],mm2
```

```
$base3__ equ $-4
  por mm5,mm1
     movq mm2,[edi+E]
  pcmpeqb mm7,mm7
     pxor mm2,[ST]
$base3__ equ $-4
  pxor mm7,mm5
  pand mm6,[RAND02]
$base3__ equ $-4
     movq mm5,mm1
  por mm7,mm6
     movq mm4,mm2
     movq mm6,mm0
     movq mm3,[G2]
$base3__ equ $-4
     pxor mm5,mm0
     movq [RCscratch],mm0
$base3__ equ $-4
     pxor mm4,mm0
  movq [XORE],mm7
$base3__ equ $-4

  ;---------------
  ; D
  ;---------------
  movq mm7,mm1
  pand mm6,mm2
  pandn mm3,mm1
  pandn mm4,mm5
  movq mm5,mm0
  pxor mm0,[G2]
$base3__ equ $-4
  pandn mm7,mm6
  pand mm3,mm4
  pand mm0,mm4
  movq mm6,mm1
  movq mm4,mm1
  pxor mm5,mm2
  por mm7,mm3
  pand mm3,[PA]
$base3__ equ $-4
  pand mm6,mm0
  pand mm0,[XORA]
$base3__ equ $-4
  pandn mm4,mm5
  movq mm5,[RCscratch]
$base3__ equ $-4
  por mm6,mm4
  pand mm7,[RAND13]
$base3__ equ $-4
  por mm5,mm1
  movq [PA],mm3
$base3__ equ $-4
  por mm6,mm7
  movq [XORA],mm0
$base3__ equ $-4
  por mm5,mm2
     movq mm0,[edi+F]
  pcmpeqb mm7,mm7
     pxor mm0,[ST]
$base3__ equ $-4
  pxor mm7,mm5
  pand mm6,[RAND12]
$base3__ equ $-4
     movq mm5,mm2
  por mm7,mm6
     movq mm4,mm0
  pand mm7,[XORD]
$base3__ equ $-4
     movq mm6,mm1
     movq mm3,[G2]
$base3__ equ $-4
     pxor mm5,mm1
     movq [RCscratch],mm1
```

```
$base3__ equ $-4
     pxor mm4,mm1
  movq [XORD],mm7
$base3__ equ $-4

  ;---------------
  ; E
  ;---------------
  movq mm7,mm2
  pand mm6,mm0
  pandn mm3,mm2
  pandn mm4,mm5
  movq mm5,mm1
  pxor mm1,[G2]
$base3__ equ $-4
  pandn mm7,mm6
  pand mm3,mm4
  pand mm1,mm4
  movq mm6,mm2
  movq mm4,mm2
  pxor mm5,mm0
  por mm7,mm3
  pand mm3,[PB]
$base3__ equ $-4
  pand mm6,mm1
  pand mm1,[XORB]
$base3__ equ $-4
  pandn mm4,mm5
  movq mm5,[RCscratch]
$base3__ equ $-4
  por mm6,mm4
  pand mm7,[RAND23]
$base3__ equ $-4
  por mm5,mm2
  movq [PB],mm3
$base3__ equ $-4
  por mm6,mm7
  movq [XORB],mm1
$base3__ equ $-4
  por mm5,mm0
     movq mm1,[edi+A]
$base3__ equ $-4
  pcmpeqb mm7,mm7
     pxor mm1,[ST]
$base3__ equ $-4
  pxor mm7,mm5
  pand mm6,[RAND02]
$base3__ equ $-4
     movq mm5,mm0
  por mm7,mm6
     movq mm4,mm1
  pand mm7,[XORE]
$base3__ equ $-4
     movq mm6,mm2
     movq mm3,[G2]
$base3__ equ $-4
     pxor mm5,mm2
     movq [RCscratch],mm2
$base3__ equ $-4
     pxor mm4,mm2
  movq [XORE],mm7
$base3__ equ $-4

  ;---------------
  ; F
  ;---------------

  movq mm7,mm0
  pand mm6,mm1
  pandn mm3,mm0
  pandn mm4,mm5
  movq mm5,mm2
  pxor mm2,[G2]
$base3__ equ $-4
```

```
  pandn mm7,mm6
  pand mm3,mm4
  pand mm2,mm4
  movq mm6,mm0
  movq mm4,mm0
  pxor mm5,mm1
  por mm7,mm3
  pand mm3,[PC]
$base3__ equ $-4
  pand mm6,mm2
  pand mm2,[XORC]
$base3__ equ $-4
  pandn mm4,mm5
  movq mm5,[RCscratch]
$base3__ equ $-4
  por mm6,mm4
  pand mm7,[RAND23]
$base3__ equ $-4
  por mm5,mm0
  movq [PC],mm3
$base3__ equ $-4
  por mm6,mm7
  movq [XORC],mm2
$base3__ equ $-4
  por mm5,mm1
  ;
  pcmpeqb mm7,mm7
  ;
  pxor mm7,mm5
  pand mm6,[RAND12]
$base3__ equ $-4
  ;
  por mm7,mm6
  ;
  pand mm7,[XORF]
$base3__ equ $-4
  ;
  ;
  ;
  ;
  ;
  movq [XORF],mm7
$base3__ equ $-4



10H 30:
entre deux appels de ma soeur obnubilée par ses taches ménagères, je
débugge SP52. il a souffert d'abord d'un trop grand nombre de labels,
puis de nombreuses erreurs de translation/copie/inattention.



11H30: il y a vraiment un problème avec l'équation.

minuit 30: l'equation fonctionne correctement, sans PA-PC.
renommé vers SP54.

6H45: l'algo a été aplati pour PA-PC:

PA = PA OR RANDOM
R  = PA OR RANDOM2
PB = PB OR R
R2 = R AND PB
PC = PC OR R2

remarque 1 : c'est beaucoup plus léger que la première version
(cf plus haut).
remarque 2: c'est tout aussi lourd car seulement 3 instructions
sur 12 n'accèdent pas à la mémoire. dur dur pour seulement 5 opérations
logiques! mais ici, pas de problème d'allocation des registres...

 movq mmA,[PB]
$base4__ equ $-4
```

```
 movq mmB,[PA]
$base4__ equ $-4
 movq mmC,[RANDOM]
$base4__ equ $-4
 movq mmD,mmA
 por mmC,MMB
 por mmB,[RANDOM2]
$base4__ equ $-4
 pand mmD,mmB
 por mmB,mmA
 por mmD,[PC]
$base4__ equ $-4
 movq [PA],mmC
$base4__ equ $-4
 movq [PB],mmB
$base4__ equ $-4
 movq [PC],mmD
$base4__ equ $-4


7H50 du matin,
j'ai oublié de masquer la sortie... il faut donc
faire un OR de tous les Pn puis faire un AND de toutes les sorties...
j'en profite pour entrelacer ceci avec l'épilogue des collisions...

9H:
la formule n'est pas stable. observation.

certaines collisions frontales ont apparemment le mauvais XOR,
PB et PC semblent inversés.

en plus,maintenant, une particule G est ajoutée...




jeudi 23 septembre 99:
le zu est revenue. a part ça, la formule est "en dérangement".
les derniers jours ont vu une certaine agitation de ce côté-là aussi.
Avec FHP, plus on cherche, plus on trouve, et plus il faut chercher.

A B C G | Pn E1 E2
------------------
0 0 0 0 |       1
0 0 0 1 |       1
0 0 1 0 |       1
0 0 1 1 |       A
0 1 0 0 | 1    1/3
0 1 0 1 |    1 1/2
0 1 1 0 |       1
0 1 1 1 |       1
1 0 0 0 |       1
1 0 0 1 |      /A
1 0 1 0 |    1  1
1 0 1 1 |      1/3
1 1 0 0 |       1
1 1 0 1 |       1
1 1 1 0 |       1
1 1 1 1 |       1

E2 est vraiment différent des premiers essais.
il faut noter que les collisions impossibles peuvent
avoir une sortie à 1. rappel : "collision impossible"
veut dire : nombre de particules >=4 ou G=!G.
Ainsi, quasiment toutes les sorties sont à 1, la
complexité de l'équation est réduite.
Le problème est sur les variables aléatoires,
comme lors des derniers essais !
la partie 1/3 ne pose pas de problème.
la partie A et /A non plus, mais 1/2
doit avoir la valeur A puis /A lors
des six étapes du calcul.
```

```
On va donc ruser. Il faut que dans le calcul, RANDOM
soit chargé de la mémoire, inversé dans un autre registre,
et ainsi lors de la phase de "renommage", on peut
faire un "movq" qui copie RANDOM ou son inverse.
ça fout la pression sur les registres mais je crois
qu'on peut s'en tirer. Dans un sens, ça réduit le nombre
d'accès à la mémoire, ce qui n'est pas plus mal.



8H30 : bon, on ne va pas "ruser". grâce à une "petite"
simplification logique (qu'aucun compilo n'aurait pu
détecter) on peut utiliser la bonne vielle méthode de
renomamge manuel. Cela veut aussi dire qu'il faut se
trimbaler RANDOM et son inverse tout le temps mais ce
n'est pas trop grave.
La nouvelle "formule" donne :

A/B= G  B (A xor C) (A xor RANDOM1/2)    <- c'est la nouvelle astuce
Pn=(A xor B)  (A xor C)  (A xor G)
E1=(A xor B)  (A xor C)  (A xor G)
1/3=Pn Random1/3
Pn=Pn B
1/2=E1 RANDOM1/2 <- c'est ce random qu'on renomme

Là où le problème commence, c'est quand on veut constituer E2.
comme indiqué dans le tableau du haut, presque toutes les
sorties sont à 1. donc, comme E2 est une somme de produits,
on peut inverser la sortie du OR. MAIS (il y en a toujours un)
comment faire pour la probabilité de 1/3 ?

En toute logique, l'inverse est la probabilité de 2/3.
Vérification faite, on peut initialiser le générateur
de nombres aléatoires avec n'importe quelle configuration
puisque les bits sont juste "décalés" à chaque cycle.
On peut donc mettre 2 bits au lieu d'un pour chaque position,
ce qui donne bien la probabilité de 2 sur 3. Si on inverse
le résultat, on se retrouve avec la probabilité de 1/3.

Pour l'occasion, SP55 est renommé en SP56.

E2= NOT (A/B or 1/3 or 1/2)

Ce NOT est "cher". On peut en éviter une moitié,
lorsque E1 and E2 est effectué à la fin : il faut faire
"pandn E2,[E1]".



dimanche 3 octobre 99 :

AXG = A xor G
AXC = A xor C  -> C disparait
AXB = A xor B
copy B
A/B = /B AXC
ABC = AXB /AXC
load RANDOM2
cp ABC (?)
E1 = ABC AXG
Pn = ABC /AXG
A/B = A/B G
AXR = A xor RANDOM2
// E1 = E1 [E2]
A/B = A/B AXR
1/3 = Pn [RAND1/3]
sauve E1
Pn = Pn B
1/2 = E1 (/)RANDOM2
E2= A/B or 1/3
E2 = E2 or 1/2
X1 = 1               //  X1 = [E1(x-3)]
```

```
E2 = E2 xor X1       // E2 = E2 X1
sauve [E2]

reste à faire le travail sur graphe.


lundi soir :
avant de se lancer dans le travail de graphe,
il faut vérifier si la méthode est correcte.
Il faut donc transformer bêtement la formule
ci-dessus en assembleur et tester son fonctionnement.

!movq _R2,[RANDOM] ; load RANDOM2
!movq _G,[G] ; load G

movq _RA,[A]
movq _AXB,_RB     ; AXB = A xor B
pxor _AXC,_RA     ; AXC = A xor C  -> C disparait et AXC le remplace
pxor _AXB,_RA

movq _A/B,_RB     ; A/B = /B AXC
pandn _A/B,_AXC
pandn _AXC,_AXB   ; ABC = AXB /AXC  AXB:fini, AXC->ABC

movq _AXG,_G     ; AXG = A xor G (déplacé)
pxor _AXG,_RA

movq _Pn,_AXC     ; cp ABC (?)
pand _AXC,_AXG    ; E1 = ABC  AXG         E1=_AXC
pandn _AXG,_Pn    ; Pn = ABC /AXG         Pn=_AXG  (_Pn fini)
pand _A/B,_G      ; A/B = A/B G

movq _AXR,_RA     ; AXR = A xor RANDOM2
pxor _AXR,_R2
//pand _AXC,[E2]  ; // E1 = E1 [E2]

pand _A/B,_AXR    ; A/B = A/B AXR
movq _1/3,[RAND13]; 1/3 = Pn [RAND1/3]
pand _1/3,_AXG
movq [E1],_AXC    ; sauve E1
pand _AXG,_RB     ; Pn = Pn B

por _A/B,_1/3     ; E2= A/B or 1/3
movq _1/3,_R2
pand(n) _1/3,_AXC  ; 1/2 = E1 (/)RANDOM2
por _A/B,_1/3      ; E2 = E2 or 1/2

pcmpeqb _X1,_X1   ; X1 = 1
// movq _X1,[E1]  ;  //  X1 = [E1(x-3)]
pxor _A/B,_X1     ; E2 = E2 xor X1
// pand _A/B,_X1  ;  // E2 = E2 X1
movq [E2], _A/B   ; sauve [E2]

Assignation des registres :
mm0: _RA
mm1: _RB
mm2: _RC/_AXC
mm3: _AXB/_AXG/_X1
mm4: _A/B
mm5: _Pn/_1/3/_AXR
mm6: _R2
mm7: _G


renommage des registres :

!movq mm6,[RANDOM] ; load RANDOM2
!movq mm7,[G] ; load G

movq _RA,[A]
movq mm3,_RB      ; AXB = A xor B
pxor _AXC,_RA     ; AXC = A xor C  -> C disparait et AXC le remplace
pxor mm3,_RA
```

```
movq mm4,_RB      ; A/B = /B AXC
pandn mm4,_AXC
pandn _AXC,mm3    ; ABC = AXB /AXC  AXB:fini, AXC->ABC

movq mm3,mm7      ; AXG = A xor G (déplacé)
pxor mm3,_RA

movq mm5,_AXC     ; cp ABC (?)
pand _AXC,mm3     ; E1 = ABC  AXG         E1=_AXC
pandn mm3,mm5     ; Pn = ABC /AXG         Pn=_AXG  (_Pn fini)
pand mm4,mm7      ; A/B = A/B G

movq mm5,_RA      ; AXR = A xor RANDOM2
pxor mm5,mm6
//pand _AXC,[E2]  ; // E1 = E1 [E2]

pand mm4,mm5      ; A/B = A/B AXR
movq mm5,[RAND13] ; 1/3 = Pn [RAND1/3]
pand mm5,mm3
movq [E1],_AXC    ; sauve E1
pand mm3,_RB      ; Pn = Pn B

por mm4,mm5       ; E2= A/B or 1/3
   /// pand mm3,[Pn] ; (oublié)
movq mm5,mm6
pand(n) mm5,_AXC  ; 1/2 = E1 (/)RANDOM2
por mm4,mm5       ; E2 = E2 or 1/2

movq [Pn],mm3     ; sauve Pn
pcmpeqb mm3,mm3   ; X1 = 1
// movq  mm3,[E1]  ;  //  X1 = [E1(x-3)]
pxor mm4,mm3      ; E2 = E2 xor X1
// pand mm4,mm3   ;  // E2 = E2 X1
movq [E2], mm4    ; sauve [E2]


Première observation : le code semble plus court.
il n'est pas encore optimisé mais ça semble prometteur.


mercredi :

SP57 n'implémente pas encore les Pn. il faut que
je retrouve exactement la formule et que je voie ce qu'elle fait.


19H: grand houra, SP57 fonctionne pour les cas étudiés.
sauf un. 5è colonne, 3è ligne.
je me demande si ce n'est pas un problème d'initialisation des particules.
ensuite je chercherai une faute quelconque de recopie.
recopie dans SP58.asm.

il y a aussi un problème de chiralité. les collisions triangulaires
avec particule au repos ont des probabilités inégales.

21H30: le problème de la collision qui n'avait pas lieu a été
résolu. il faut encore observer le cas des probabilités fausses.
je me demande si ce n'est pas lié à la manière dont le générateur
de nombres aléatoires est constitué (y aurait-il du Zaleskisme
derrière tout ça ?)
solution (?) : essayer de voir ce qui se passe si le générateur
ne bouge pas.
solution 2 : initialiser une "mer" de particules aléatoires.

22H30: des observations plus poussées laissent entendre qu'il n'y a
pas lieu de s'inquiéter. il faut juste faire attention à ce que la
"bouillie de bits" soit uniforme, ce qui est vrai si le "fluide" est
"lâché", livré à lui-même. Sinon, les caractéristiques géométriques
du fluide se retrouvent dans la chiralité.

J'ai aussi une pensée pour le travail qui reste à réaliser. même si
presque tout est au point, il y a tout de même des milliers de lignes
de code à optimiser à tous les niveaux.
```

```
Maintenant, il faut tester le reste des collisions.
le source fait actuellement 172KO et le binaire fait 12KO.
il faut ajouter les Pn (collisions frontales) pour que toutes les
collisions à deux éléments soient testées.


1H30 : SP59 est prêt pour la compilation.
j'ai réactivé les Pn dans le code de mouvement et j'ai refait
le ménage dans la sélection progressive. deux colonnes sont ajoutées
pour les tests de collisions.

remarque 1 : les "vecteurs de test" sont un "peu boggés".

remarque 2 : problème apparemment au niveau de la sélection progressive:
 la sortie des collisions est toujours la même.

remarque 3 : il y a toujours le problème de la troisième ligne à régler.

remarque 4 : interférence avec XORG. mais ce qui est dingue c'est que
ce n'est pas pour toutes les collisions (sortie=AD pour certaines lignes).

remarque 5 : collisions frontales à 5 éléments : on n'en parle même pas...

En faisant tourner le programme, la population des particules a légèrement
augmenté mais pas "explosivement". Cela, plus l'expérience précéndente,
prouve qu'il n'est pas suffisant de faire tourner le programme
sans anomalies avant de le déclarer valide. Souvenez-vous de Zaleski...


15H30: j'ai trouvé le bug du XORG.
le por mm5,[PC] était placé avant le movq mm7,mm5
dans la deuxième instance du code de mouvement.
XORG avait donc en plus les bits de PC...

Il faut aussi débogger la sélection progressive.
SP60.asm
Il semble aussi que le cas où il y a plus de 4 particules est mal géré.

voyons d'abord la sélection progressive.

il y a probablement un problème au niveau des dépendences de données.


jeudi, 0h5: un petit tour du côté de chez Debug.exe a montré que
l'algo de sélection est boggé.
la question reste la même: qu'est-ce qui ne va pas ?????

session debug:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
debug sel.com<comm >co

type co
-g


Le programme s'est terminé normalement
-d160

2E73:0160  FF 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:0170  00 FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:0180  00 00 FF 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:0190  0F 0F 0F 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:01A0  FF 0F 0F 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:01B0  FF FF F0 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:01C0  00 F0 FF 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:01D0  DC 8C C0 40 8E C0 83 EF-10 26 01 1D 48 8E C0 EB   ...@.....H...
-q
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

le problème est avec PA, la première colone. Pour le reste, PB et PC sont
parfaits.

J'ai enfin compris le problème : c'est à cause du OR pour PA2.
```

```
Résultat, il faut refaire les équations. heureusement elles
ne sont pas aussi lourdes que la formule centrale.


Nouvelle formule:

PA2 = PA or  Rand
PB2 = PB or /Rand
PC2 = PC or (PA Rand) or (PB /rand)

Cette fois-ci ça devrait aller.

PA2 = PA or  Rand
R1  = PA   Rand
PB2 = PB or /Rand
R2  = PB  /Rand
PC2 = PC or R1 or R2

...

load PA
/R  = -1
load PB
OR3 = PA
load PC
/R  = /R xor Rand
OR3 = OR3 or PB
R1  = PA
OR3 = OR3 or PC
R2  = PB
PA  = PA or Rand
PB  = PB or /R
R1  = R1 Rand
PA  = PA OR3
R2  = R2 /R
PC  = PC or R1
PB  = PB OR3
PC  = PC or R2
save PA
PC  = PC OR3
save PB
save PC

.... ess_sel1.asm :

; movq PA,[PA]
;;$base4__ equ $-4
; pcmpeqb /R,/R
; movq PB,[PB]
;;$base4__ equ $-4
; movq OR3,PA
; movq PC,[PC]
;;$base4__ equ $-4
; pxor /R,Rand
; por OR3,PB
; movq R1,PA
; por OR3,PC
; movq R2,PB
; por PA,Rand
; por PB,/R
; pand R1,Rand
; pand PA,OR3
; pand R2,/R
; por PC,R1
; pand PB,OR3
; por PC,R2
; movq [PA],PA
;;$base4__ equ $-4
; pand PC,OR3
; movq [PB],PB
;;$base4__ equ $-4
; movq [PC],PC
;;$base4__ equ $-4
```

```
; assignation des registres :
; PA=mm0
; PB=mm1
; PC=mm2
; /R=mm3
; OR3=mm4
; R1=mm5
; Rand=mm6
; R2=mm7

 movq mm0,[PA]
;$base4__ equ $-4
 pcmpeqb mm3,mm3
 movq mm1,[PB]
;$base4__ equ $-4
 movq mm4,mm0
 movq mm2,[PC]
;$base4__ equ $-4
 pxor mm3,mm6
 por mm4,mm1
 movq mm5,mm0
 por mm4,mm2
 movq mm7,mm1
 por mm0,mm6
 por mm1,mm3
 pand mm5,mm6
 pand mm0,mm4
 pand mm7,mm3
 por mm2,mm5
 pand mm1,mm4
 por mm2,mm7
 movq [PA2],mm0
;$base4__ equ $-4
 pand mm2,mm4
 movq [PB2],mm1
;$base4__ equ $-4
 movq [PC2],mm2
;$base4__ equ $-4

donne le résultat suivant :

-g


Le programme s'est terminé normalement
-d160

2E73:0160  FF 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:0170  00 FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:0180  00 00 FF 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:0190  0F 0F 0F 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:01A0  FF 0F 0F 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:01B0  F0 FF F0 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:01C0  0F F0 FF 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2E73:01D0  C7 46 FA 00 00 E9 93 00-8A 46 FA D0 E0 8B 56 FC   .F.......F....V.
-q


Ce qui est le résultat désiré !
on revient de loin...


2H30 : collision frontale avec une G : sur la troisième ligne, la
collision donne toujours le même résultat. comme ce phénomène a
déjà eu lieu, je me doute que c'est un mauvais vecteur de test :
la même configuration, une ligne plus bas, donne le bon résultat.


2H45 : problème résolu pour la collision frontale + G.
la colonne suivante a un problème assez similaire.

le problème avec ce truc c'est qu'on est jamais parfaitement sûr
que la collision a bien lieu de la manière désirée, à l'endroit
```

voulu. il suffit qu'une particule soit décentrée et la collision
n'a pas lieu correctement.

Collisions testées :

```
 A+G ->   F+B
 F+B ->   A+G
 A+D ->   B+E ou C+F
 A+C+G -> A+B+D ou B+C+F
 A+B+D -> A+C+G
 B+C+F -> A+C+G
 A+D+G -> A+C+E ou B+D+F
 A+C+E -> A+D+G ou B+E+G ou C+F+G
 A+C+E+G -> A+D+B+E ou B+E+C+F ou C+F+A+D
```

SP61.asm : essai de simplification des vecteurs de tests
on utilise une petite fonction qui interprète un octet.

4H30 : l'utilisation d'une petite fonction pour faire le "sale travail"
allège le programme de 20KO et diminue les "chances" d'erreurs.

Mais ABDEG est boggé jusqu'à la moelle...

SP62: transition réussie du code. mais le déboggage doit continuer.
les 51 contre-exemples sont en place, pas de surprise de ce côté-là.

5H15:
rectification : il n'y a que 45 vecteurs de contre-vérification.
les "manquants" sont le cas 0 et les cas A-F qui fonctionnent sans
problème, heureusement. cela porte le nombre de collisions "impossibles"
à 52, soit 76 collisions possibles, ce qui correspond exactement
au modèle FHP saturé.

Par contre il y a un vrai problème lorsque popcount>3, sur la cinquième
ligne. j'avais déjà remarqué cela pour le complémentaire de la collision
frontale simple, mais là je me demande ce qui se passe.

12H45:
la nuit porte conseil. d'abord, compléter les vecteurs de tests.

13H20 : il y a effectivement un bos grug.
le problème c'est que les symptômes sont mélangés.
jusqu'à maintenant, j'ai pu discerner les différents cas
mais je ne suis plus sûr.

Comme le problème apparaît pour popcount>3, il faut probablement
voir du côté de l'additionneur. il a déjà été testé avec succès,
donc je suis très étonné.

Autre symptôme, une collision ABCD donne un ABDG. Il faut peut-être
voir si le XOR[ST] est bien effectué.

14H: j'ai trouvé le pot aux roses:
j'ai oublié de XORer l'amorce. F et A n'étaient pas xorés,
même si G l'était. Xing fingers.

14H15 : le contrôle visuel est : OK !
renommé vers SP64.asm.

samedi, 2h40:
après ce succès (aber alles ist relativ) il faut songer
à l'optimisation, même si le plus gros est déjà fait.
tout ? non.
après l'optimisation au niveau des instructions (microscopique),
l'optimisation au niveau des blocs (macroscopique, et ses gros graphes)

```
il faut songer maintenant à l'optimisation générale.
Cela veut dire à l'heure actuelle : mettre tous les blocs ensemble
et regarder ce qu'ils font. Je suis sûr que des optimisations
importantes restent à faire. Une fois que cela sera fait, on recodera
les blocs comme d'habitude, en faisant beaucoup plus attention, cette
fois-ci. enfin, on regardera les compteurs de performance pour voir
où sont perdus les cycles.

Voyons voir :

1) parois
     il n'y a rien à y faire. c'est entièrement "data dependent"
et déjà assez bien codé. On peut cependant tester si WALLMASK est
complétement à 1, pour éviter de passer l'étape de calcul.
Cela nécessite aussi de faire une version "allégée" du code
de déplacement. il faut tester cela apreès le pré-mouvement.

2) pré-mouvement, variables temporaires :
     Ces quelques instructions peuvent être facilement fusionnées avec
le popcount. cela joue seulement avec les variables temporaires de
B et C mais réduit légèrement le nobre de registres libres.

D'ailleurs je viens de penser à l'instant à un moyen d'éviter
la duplication du code pour les lignes paires et impaires,
grâce à l'instruction de décalage avec un registre au lieu d'un immédiat.

problème : cela rajoute des instructions et oblige à accéder (inutilement)
à 2 emplacements mémoire. la version actuelle, bien que plus
lourde, tient dans la cache d'instructions et ne pose donc
pas de problème. ça fonctionne déja bien. Il faudrait utiliser
cette astuce si le code ne tenait plus en L1.

vérification faite auprès de Andy Glew, il n'y a pas de problème
jusqu'à 16KO.

3) popcount:
load A-G, store ST, S0-S2.
il faut refaire la coloration des registres, et fusionner
avec le prologue de la détection. il faut aussi intégrer le prémouvement.

4) détection * 6
load ST,RandA-C, A-F (progressivement)
store XORA-F, Pn
Il faut fusionner la sélection (Pn) avec la détection.
il faut aussi faire qqc pour le OR des XOR, mais il n'y a
pas assez de registres...

5) sélection : dans 4)
load/store P, Random
penser à sauver le OR des Pn pour afficher les collisions...

problème : où effectuer le masquage des Pn ?
il faut effectuer un OR de tous les Pn, effectuer la "sélection"
puis masquer le tout par le OR des données initiales.
si 5 est dans 4, il y aura beaucoup d'accès à la mémoire car
il n'y a pas assez de registres. il faudra sauver le OR, et
plein d'autres choses. finalement il vaut mieux faire ça séparément,
on n'a vraiment pas assez de registres. si le or des XOR était
plus sobre, on pourrait peut-être effectuer la sélection dans
la partie de déplacement.

6) déplacement
load XORs, A-G, store A-G (modifiés) + temporaires
il y a le problème des OR qui prennent beaucoup de registres.
attention : un seul shift par cycle !
c'est à recoder entièrement.

7) sommation/densité
load S0-S2 densité, store densité.
Selon les ordres de l'utilisateur, il faut pouvoir
changer de module (visualiser les collisions ou la densité ?)


5H:
```

```
maintenant que j'y pense, le code de sommation que j'ai créé
n'effectue qu'une sommation d'un bit. il est souhaitable de pouvoir
effectuer la sommation sur 3 bits pour afficher les particules,
et pas seulement les collisions. approx 110 instructions.

Comme on commence à applatir un graphe à partir du bas, c'est par
cette fonction, la dernière, qu'on va commencer le travail.

registre 0-2: valeurs originales
registre 3 : masque
registre 4 : valeur accumulée
registre 5 : valeur temporaire (masquée+shift(!))

il y a donc une petite marge mais pas assez pour effectuer le
dédoublement de l'arbre comme auparavant.

quoique ?

registre 6 : valeur accumulée [2]
registre 7 : valeur temporaire (masquée+shift(!)) [2]

je ne suis pas sûr. il faudra jouer très serré à cause du décalage
de la valeur initiale, qui nécessite de l'adresse.

of course le prologue de cette fonction sera entrelacé avec
l'épilogue de la partie de déplacement.



lundi, 4h du mat. :

je repense au code qui en fait est complétement
asphyxié par ... les masques. il en faut 3 en fait.
il ne reste donc plus que deux registres pour faire notre
travail.

on a t1,t2,v1,v2,v3,m1,m2,m3

t1=v1
// t2=v2
v1>>=1
// v2>>=1
t1m1
// t2m2
t1+=t2
t1+=[mem] (optionnel ?)
t2=v3
v3>>=1
t2m3
t2+=t1
sauve t2


noter que le v?>>=1 peut être déplacé à volonté.
mais la troisième partie est difficile car il n'y a
pas grand-chose à entrelacer. à part les décalages ?


le graphe donne:

movq t1,v1
psrlq v1,1

; sauve t2 ici: (cycle précédent)
movq [mem],t2
movq t2,v2

pand t1,m1
pand t2,m2

psrlq v2,1
por t2,t1
```

```
movq t1,v3
psrlq v3,1

(padd t2,[mem])/stall
pand t1,m3

padd t2,t1
/stall

on voit qu'il y a au minimum 1 stall, sinon plus.
il faut 7 cycles par tour, après l'initialisation
du "pipeline".


cette initialisation prend deux ou trois tours:
il faut faire la création des masques et
aussi shifter quelques valeurs:

0: créer le masque 0101010101010101

cycle 1: masque 1 existe, m2 et m3 sont libres.
  cela permet de bien s'étaler dans les registres.

2: m2=m1<<1

cycle 2: m1 et m2 existent, m3 permet encore de s'étaler.

3: m3=m2<<1

la suite a été traitée plus haut.
il faut donc faire du boulot autour des cycles 1 et 2 :

movq v1,[S0]
movq v2,[S1]
movq v3,[S2]

pxor m1,m1
pcmpeqb m2,m2
psubb m1,m2   ; m1=0x0101010101010101, m2 libre

movq t1,v1
movq t2,v2
movq t3,v3

pand t1,m1
pand t2,m1
pand t3,m1

psrlq v1,1
psllq t2,1
psllq t3,2

por t2,t1
paddb t3,[mem]
paddb t2,t3    ; noter l'"arbre" qui brise les dépendances parasites.

movq m2,m1
psllq m2,1
movq [mem],t2

***

movq t1,v1
movq t2,v2
movq t3,v3

pand t1,m1
pand t2,m2
pand t3,m2

psrlq v1,1
psrlq v2,1
psllq t1,1
```

```
por t2,t1
paddb t3,[mem]
paddb t2,t3

movq m3,m2
psllq m3,1
movq [mem],t2



PostIt : penser à faire un interpréteur de commandes.
ça facilitera énormément l'automatisation des expériences !

penser aussi à sauver les résultats des calibrations dans des fichiers
au format gnuplot !!!


résultats du graphe:

movq t1,[S0]
;$
pcmpeqb m1,m1 ; m1=-1

movq t2,[S1]
;$
movq v1,t1

movq t3,[S2]
;$
pxor m3,m3    ; m3=0

movq v2,t2
psubb m3,m1   ; m3=0x0101010101010101

movq v3,t3
pand t3,m1

pand t2,m1
psllq t3,2

pand t1,m1
psllq t2,1

paddb t3,[m0]
por t2,t1

movq m1,m3
paddb t2,t3

movq t3,v3
psllq m3,1

movq t2,v2
psrlq v1,1

movq [m0],t2
pand t3,m3

movq t1,v1
psllq t3,1

pand t2,m3
psrlq v1,v1

paddb t1,[m1]
por t2,t1

psrlq v2,1
paddb t2,t1

movq m2,m3   ; t3 se transforme en m2 ici
psllq m3,1   ; 34 instructions sans stall ! 17 cycles (optimistement)
```

```
Maintenant je me demande s'il n'est pas mieux de mettre m3 dans
une variable globale. ça fait un accès mémoire en plus mais plus
d'opportunités pour le parallélisme/ILP. ah. j'adore ce mot.

<graphe>
en effet, avec un accès à la mémoire:

movq t1,v1
psrlq v1,1

movq t2,v2
psrlq v2,1

movq [M],t3
movq t3,v3

pand t1,m1
pand t2,m2

pand t3,[m3]
;$
por t2,t1

paddb t3,[M]
psrlq v3,1

paddb t3,t2
//stall

13 instructions et 1 stall.
pas grand-chose de changé, sauf que c'est moins tordu...
et lorsque le "paddb t3,[M]" est enlevé, il n'y a pas de stall,
ce qui réduit la boucle à 6 cycles.

Pour enlever le stall et gagner un cycle sur 15,
il faut traiter deux occurences de la boucle ensemble,
il suffit pour cela de déplacer les accès à la mémoire.

;1
movq t1,v1
psrlq v1,1

movq t2,v2
psrlq v2,1

movq [M],t3
movq t3,v3

pand t1,m1
pand t2,m2

pand t3,[m3]
;$
por t2,t1

paddb t3,[M]
psrlq v3,1

paddb t3,t2
movq t1,v1

psrlq v1,1
movq t2,v2

movq [M],t3
psrlq v2,1

movq t3,v3
pand t1,m1

pand t3,[m3]
;$
pand t2,m2
```

```
paddb t3,[M]
por t2,t1

psrlq v3,1
paddb t3,t2


et voilà ! 13 cycles au lieu de 14 !

mais on a beau faire, on ne peut pas diminuer le nombre d'opérations.
le scheduling des instructions doit faire le reste.
il a fallu cependant revoir l'organisation des données pour arriver
à réduire au maximum le nombre de cycles (PMMX seulement, mais
ça marche toujours sur P6). ce processus de développement est itératif,
il est fortement dépendent des algorithmes à écrire et il n'y
a donc pas de recette miracle, mais une bonne dose d'aprofondissement.
pour cela, il faut tout mettre à plat et retourner tout dans tous
les sens pour trouver une faille.

En tous cas il faut un peu refaire l'amorce,
ensuite, vérifier que ça marche !


movq t1,[S0]
;$
pcmpeqb m1,m1 ; m1=-1

movq t2,[S1]
;$
movq v1,t1

movq t3,[S2]
;$
pxor m2,m2    ; m2=0

movq v2,t2
psubb m2,m1   ; m2=0x0101010101010101

movq v3,t3
pand t3,m2

pand t2,m2
psllq t3,2

pand t1,m2
psllq t2,1

paddb t3,[_M0]
por t2,t1

movq m1,m2
paddb t2,t3

movq t3,v3
psllq m2,1

movq t1,v1
psrlq v1,1

movq [_M0],t2 ; damnit ! problème de dépendence !!! lock t2. renommage avec t1
pand t3,m2

pand t1,m1 ; oublié.

movq t2,v2
psllq t3,1

pand t2,m2
psrlq v1,1

paddb t2,[_M1]
por t3,t1
```

```
psrlq v2,1
paddb t3,t2



PRLLSUM0.ASM : chronométré à 59 cycles.
le comptage manuel augurait 57 cycles mais je ne me plains pas.
et comme le code semble fonctionner, je peux enfin respirer :-)


PRLLSUM1.ASM: sans l'addition (initialisation du buffer)

   BOUDIOU de "'{(=}=+ø0 !!!!!! 122 cycles !

j'ai lâché ma hargne sur comp.arch.


disons que l'algo fonctionne.
il faut maintenant choisir entre 4 codes:
-code de début de bloc ou non
-code de densité ou de collision.

le code de début de bloc ne lit pas la donnée du buffer
d'affichage, ce qui évite d'avoir à le remettre à 0.

le code de densité est 3x plus lourd que le code de collision.

je pense qu'on peut séparer les codes destiné aux collisions
et aux densités. il faut donc recopier la boucle centrale
et modifier l'interface pour inclure cette option.

pour savoir si on est à la première ligne d'un bloc, on va
utiliser un flag dans EBX. le bit 16 sera à 1 dans ce cas.
à la fin de la ligne, il sera remis à zero.
à l'intérieur de la boucle de ligne, avant d'accumuler,
on testera le flag et on dirigera vers la procédure adaptée.



ROUND 2: le déplacement.
ici aussi, prévoir une version "light" au cas où WALLMASK=-1.
c'est un peu la jungle, comme on peut le remarquer.
jusqu'à maintenant, le code a été pensé pour être lisible et
pour fonctionner, pas pour aller vite. c'est aussi un endroit
très sensible au niveau des accès à la mémoire.

D'abord, le OR des XORs. Il faut penser à garder des points
de sortie pour les collisions et XORG.

niveau 1: les variable d'entrée

movq _ma,XORF
movq _mb,XORA
movq _mc,XORB
movq _ma,XORC
movq _mb,XORD
movq _mc,XORE
movq _ma,XORF
movq _mb,XORA

niveau 2: les OR de premier niveau

movq _ma,[XORF]
movq _mb,[XORA]
por _ma,_mb
movq _mc,[XORB]
por _mb,_mc
movq _ma,[XORC]
por _mc,_ma
movq _mb,[XORD]
por _ma,_mb
movq _mc,[XORE]
por _mb,_mc
movq _ma,[XORF]
```

```
por _mc,_ma
movq _mb,[XORA]
por _ma,_mb

niveau 3: les OR de deuxième niveau

movq _ma,[XORF]
movq _mb,[XORA]
por _ma,_mb
movq _mc,[XORB]
por _ma,_mc
por _mb,_mc
movq _ma,[XORC]
por _mb,_ma
por _mc,_ma
movq _mb,[XORD]
por _mc,_mb
por _ma,_mb
movq _mc,[XORE]
por _ma,_mc
por _mb,_mc
movq _ma,[XORF]
por _mb,_ma
por _mc,_ma
movq _mb,[XORA]
por _mc,_mb

niveau 4: XORG + petite simplification de la fin

movq _ma,[XORF]
movq _mb,[XORA]
por _ma,_mb
movq _mc,[XORB]
por _ma,_mc
por _mb,_mc
movq _ma,[XORC]
por _mb,_ma
por _mc,_ma
movq _mb,[XORD]
por _mc,_mb
por _ma,_mb
movq [XORG],_mc
movq _mc,[XORE]
por _ma,_mc
por _mb,_mc
movq _ma,[XORF]
por _mb,_ma
por _mc,_ma
por _mc,[XORA]
por _mc,[XORG]


niveau 5: /WALLMASK et Pn

movq _ma,[XORF]
movq _mb,[XORA]
por _ma,_mb
movq _mc,[XORB]
por _ma,_mc
por _mb,_mc
por _ma,[PA]
pand _ma,_wm

xor _ma,[A]

movq _ma,[XORC]
por _mb,_ma
por _mc,_ma
por _mb,[PB]
pand _mb,_wm

xor _mb,[B]

movq _mb,[XORD]
```

```
por _mc,_mb
por _ma,_mb
movq [XORG],_mc
por _mc,[PC]
pand _mc,_wm

xor _mc,[C]

movq _mc,[XORE]
por _ma,_mc
por _mb,_mc
por _ma,[PA]
pand _ma,_wm

xor _ma,[D]

movq _ma,[XORF]
por _mb,_ma
por _mc,_ma
por _mb,[PB]
pand _mb,_wm

xor _mb,[E]

por _mc,[XORA]
movq _ma,_mc
por _mc,[PC]
pand _mc,_wm

xor _mc,[F]

por _ma,[XORG]
movq _mb,[PnOr]
pand _ma,_wm
por _mb,_ma

xor _ma,[G]

pand _mb,_wm
movq [coll],_mb ; pas utilisé : passe directement à la passe suivante.


à ce niveau-là, le scheduling est quasi inexistant.
l'allocation des registres se fera à la fin, dans le
graphe qui incluera le déplacement.


partie paire:

; A: pair et impair
     movq mm1,[edi+A]
     pxor mm1,mm7    ; modification
     movq mm0,mm1 ; A
     psrlq mm1,1  ; A
     psllq mm0,63 ; A'
     movq [edi+A],mm1; A
     por mm0,[edi+A-CEL] ; A'
     movq [edi+A-CEL],mm0; A'

     paddd mm1,[RANDOM]
$base1__ equ $-4
     movq [RANDOM],mm1
$base1__ equ $-4



; B: (last part)
; pair
     movq mm0,[temp_B]
$base__ equ $-4
     pxor mm0,mm6
     movq [esi+72],mm0 ; result-> temp B

;impair:
```

```
; B: (last part)
     movq mm1,[temp_B]
$base1__ equ $-4
     pxor mm1,mm6
     movq mm0,mm1
     psrlq mm1,1
     psllq mm0,63
     movq [esi+72],mm1 ; result-> temp B
     por mm0,[esi+72-80]
     movq [esi+72-80],mm0



;

;C: (last part)
;pair:
     movq mm2,[temp_C2]
$base__ equ $-4
     movd mm1,[temp_C]
$base__ equ $-4
     pxor mm2,mm5
     movq mm0,mm2
     psllq mm2,1
     psrlq mm0,63
     por mm2,mm1
     movd [temp_C],mm0
$base__ equ $-4
     movq [esi+64],mm2 ; result-> temp C

;impair:
;C: (last part)
     movq mm0,[temp_C2]
$base1__ equ $-4
     pxor mm0,mm5
     movq [esi+64],mm0 ;bug ; result-> temp C


; D:  pair et impair
     movq mm2,[edi+D]
     movd mm1,[temp_D]
$base__ equ $-4
     pxor mm2,mm7
     movq mm0,mm2
     psllq mm2,1
     psrlq mm0,63
     por mm2,mm1
     movd [temp_D],mm0
$base__ equ $-4
     movq [edi+D],mm2

; E: pair
     movq mm0,[edi+E]
     movd mm2,[temp_E]
$base__ equ $-4
     pxor mm0,mm6
     movq mm1,mm0
     psrlq mm0,63
     psllq mm1,1
     por mm1,mm2
     movd [temp_E],mm0
$base__ equ $-4
     movq [edi+1000000],mm1
_LINE_E equ $-4

;impair:
; E:
     movq mm0,[edi+E]
     pxor mm0,mm6
     movq [edi+10000000],mm0
_LINE_E2 equ $-4


; F:
```

```
; pair:
     movq mm0,[edi+F]
     pxor mm0,mm5
     movq [edi+10000000],mm0
_LINE_F equ $-4
; F:
; impair
     movq mm0,[edi+F]
     pxor mm0,mm5
     movq mm1,mm0
     psrlq mm0,1
     psllq mm1,63
     movq [edi+10000000],mm0
_LINE_F2 equ $-4
     por mm1,[edi+1000000]
_LINE_F_CELL equ $-4
     movq [edi+1000000],mm1
_LINE_F_CELL2 equ $-4



; G:  pair et impair
    pxor mm7,[edi+G]
    movq [edi+G],mm7


il faut donc faire de nombreux graphes légèrement différents:
-collision paire
-collision impaire
-densité paire
-densité impaire
suivis par un morceau avec accumulation ou non (saut conditionnel).
j'abandonne provisoirement l'idée d'un court-circuit sur Wallmask==-1.
il est plus simple d'avoir un flag=1 sur le LSB du pointeur de liste
des murs.



dimanche, 2h du matin:
j'ai retrouvé ça dans les archives:

Received: from or.mime.univ-paris8.fr (or.mime.univ-paris8.fr
     [193.54.153.27]) by front3.grolier.fr (8.9.0/MGC-980407-Frontal-No_Relay)
     with ESMTP id QAA12691 for <whygee@club-internet.fr>;
     Thu, 10 Dec 1998 16:27:53 +0100 (MET)
Received: from dichlore.mime.univ-paris8.fr (dichlore.mime.univ-paris8.fr
     [193.54.153.26])
     by or.mime.univ-paris8.fr (8.9.0/8.8.8) with ESMTP id QAA04638
     for <whygee@mime.univ-paris8.fr>; Thu, 10 Dec 1998 16:18:39 +0100 (CET)
Received: from mime.univ-paris8.fr (localhost [127.0.0.1])
     by dichlore.mime.univ-paris8.fr (8.8.8/8.8.8) with ESMTP id QAA01534;
     Thu, 10 Dec 1998 16:24:42 +0100 (CET)
     (envelope-from whygee@mime.univ-paris8.fr)
Sender: whygee@dichlore.mime.univ-paris8.fr
Message-ID: <366FE7B9.A5DFE1CF@mime.univ-paris8.fr>
Date: Thu, 10 Dec 1998 16:24:41 +0100
From: "Whygee (Yann Guidon)" <whygee@mime.univ-paris8.fr>
Reply-To: whygee@mime.univ-paris8.fr
Organization: The Whygee Corporation
X-Mailer: Mozilla 4.05 [en] (X11; I; FreeBSD 3.0-980105-SNAP i386)
MIME-Version: 1.0
Newsgroups: comp.arch
To: Andy Glew <glew@cs.wisc.edu>
Subject: Re: Is it possible to optimize Intel PII instruction flows ?
References: <366DEDBD.3267EEED@mime.univ-paris8.fr>
 <366EC0FC.64B36FE6@cs.wisc.edu> <366F1FF9.426955FE@mime.univ-paris8.fr>
 <366F2CCD.4948C354@cs.wisc.edu>
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
X-UIDL: 710a138d99b1c9910744e848d285403a
Status: RO
X-Mozilla-Status: 8015

Andy Glew wrote:
```

```
> > It appears that the usual lookup-table oriented method is
> > suboptimal and not suitable for the PII (we have one at the university)
> > since it needs a lot of masks and ORs, and there are a LOT of
> > partial register stalls that did not appear in the previous
> > processor generations.
> You can almost 100% definitely eliminate all of these partial stalls,
> by, as you say, masks and ORs.
yes but this requires additional instructions (shifts+OR ) that were not
required before. the 7 cycles per table update is much too high.

> By the way, the i486 and P5 had partial stalls, just not so bad.
yes, but 7 cycles can't remain unnoticed.
I have (the numbers speak) to use the boolean approach.

> > so the (renamed) register is updated at the next cycle ?
> Yes.

> > That's why i abandon the PMMX for the PII (i have no PMMX anyway).
> Sorry. We wanted to give you 2, or even 4, loads per cycle, but
> back in  1991 we ran out of hardware.
that's not fair :-(

I presume, the PII successors will include more units ?
i believe the Intel's ooo achitecture is not yet mature.
furthermore, ie for tight loops, there should be a uops cache
(as does another chip, i don't remember which).

> > Anyway, how much can i count on the out of order execution ?
> ???
(sorry)
how much can the ooo scheme help me ignore nasty details like in the Pentium ?
the 40 renamed registers seem to be enough to delay some instructions during
about 5 cycles. so, bothering only of the instruction translation
seems enough to ensure a good performance (to a certan point).
the code i have to write is about 400 instructions long, has no jump and uses
only about ten kinds of instructions, so a reduced number of parameters are
to be taken in account (i hope). at least, 95% of the memory accesses hit
the L1 cache :-)

>   If your code is flat-out capable of using 64 bits of data from memory
> every clock cycle, out-of-order won't help.
even i L1 ?

> Where it will help is if
> you have a burst in one place, and a burst in another.  Or, even more
> likely, if you have cache misses that can be overlapped.
Since i optimized the algo for Pentium-like processors, i don't
see your point...

> In the presence of data cache misses a Pentium II beats an Alpha,
> at least until the 21264 comes along.
If i had an Alpha...
Unfortunately, there is no DOS-like OS for ALPHA
that allows me to use the yummy possibilities of that chip.
We run LINUX on the Alphas we have and this is a multithreading OS,
and unfortunately task switches break the strip mining's performance gains.

In a word, i prefer to write a little compiler for the PII than
an OS for the ALPHA :-/

> > > Similarly, stores require two uops, just like every other P6 instruction
> > > that contains a store: store-address and store-data.
> > Maybe "i should read the docs" but i'm not sure that i understand it right.
> > Are the two uops executed simultaneously ?
> No, the second depends on the first. But they can overlap operations from
> other instructions in the window.
so the overall store throughput is one per cycle...

> > So, in the scope of instruction scheduling the question now is: what
> > matters most ?
> > feeding all the execution units or feeding all the uops translation units ?
> > I think that a compound solution is better but all the latencies and the
> > other stages make the compiler very complex.
>
```

```
> My recommendation: run a piece of sample code, and profile it with VTUNE,
> looking for places where 0, 1, 2, or 3 instructions retire per cycle.
> In the vicinity of places with low performance, look for the following
> bottlenecks:
>
>     a) more than one taken branch per cycle
nope

>     b) more than 2 ALU ops per cycle
decoding cycle ? translation cycle ?

>     c) more than 1 load per cycle
(idem as above)

>     d) the 411 template
handled by the code generator

>     e) more than 16 bytes of instruction per cycle
mmm....
as previously said, instructions are 3 or 7 bytes long.
gasp :-(
So this is another constraint (i have not seen it in the opt. manual)
for the code generator :-(

> I usually do this by totalling up the instructions in a loop,
> and seeing how many cycles it should take to execute according to each
> of the rules above.  If one rule is unduly lower than the others
> - e.g. if the loop has 6 ALU instructions => 3 ALU cycles, but also
> 6 loads => loads are the bottleneck at 6 cycles - I try to optimize that
> bottleneck.
Since the code will be automatically generated, i probably won't use this
technique. i'll have to measure the code anyway, and tune the optimisation
techniques this way.
wow...
it was about fluid simulation at the beginning :-/

WHYGEE
~~~~~~~~~~~~~~~~~~~~
"I'm a poor, a lonesome coder..."
SHARCPAGE: http://www.mime.up8.edu/~whygee/sharcpage.html



et aujourd'hui, j'ai eu une autre réponse d'Andy concernant
le problème du code deux fois plus lent. Je ne m'attendais pas à
une réponse aussi rapide alors j'avais commencé à travailler sur le reste.

From aglew@students.wisc.edu Sat Oct 16 23: 00:57 1999
Received: from or.mime.univ-paris8.fr (or.mime.univ-paris8.fr [193.54.153.27])
by front5m.grolier.fr (8.9.3/No_Relay+No_Spam_MGC990224) with ESMTP id XAA10508
     for <whygee@club-internet.fr>; Sat, 16 Oct 1999 23:00:56 +0200 (MET DST)
Received: from mail1.doit.wisc.edu (mail1.doit.wisc.edu [144.92.9.40])
     by or.mime.univ-paris8.fr (8.9.3/8.9.3) with ESMTP id WAA16040
     for <whygee@mime.univ-paris8.fr>; Sat, 16 Oct 1999 22:54:11 +0200 (CEST)
Received: from [204.71.144.26] by mail1.doit.wisc.edu
     id PAA62522 (8.9.1/50); Sat, 16 Oct 1999 15:07:48 -0500
Message-ID: <00b401bf1812$c8a41e20$1a9047cc@epoxy>
From: "Andy Glew" <aglew@students.wisc.edu>
To: <whygee@mime.univ-paris8.fr>
X-Mailer: Microsoft Outlook Express 5.00.2314.1300
Subject: Re: PMMX memory access: now i'm furious.
Date: Sat, 16 Oct 1999 15:12:33 -0500
X-Priority: 3
X-MSMail-Priority: Normal
X-MimeOLE: Produced By Microsoft MimeOLE V5.00.2314.1300
Status:
X-Mmail: \Recent
X-M-Uid: 1627.940107657
X-Mozilla-Status: 8013
X-Mozilla-Status2: 00000000
X-UIDL: 1627.940107657

Is this a P55C chip?
I.e. a Pentium with MMX? I think they were sold up to 200MHz.
```

```
*Not* a Pentium II with MMX, right? Not a P6 core?

Anyway, if a P55C, the answer is obvious: P55C has a no-write-allocate
cache. It doesn't put things in the cache unless they have been read.
Thus, when you remove the loads, the writes to the same cache address
that follow are guaranteed cache misses.

Prefetching for writes by scheduling reads ahead of time is/was
a well-known technique for P5 family. It's on all the "how to code well"
web pages.  P6s, however, have a write-allocate cache, so this
prefetch technique is much less necessary - it sometimes helps,
since it may start a memory request earlier, but sometimes it hurts.
That's one of the problems of reading generic documentation,
like "Optimizing for Intel's 32 bit processors" - they are shy
about optimizations that are not uniform across all chips.


C'est clair : dans le contexte actuel il est préférable
de garder la forme générale de la procédure, celle qui lit les
données pour accumuler. Il faut vider le plan temporaire
après l'affichage.
je suis à la fois en train de payer et de profiter de l'optimisation
agressive du code. dans un sens j'ai la certitude que mon code fonctionne
comme prévu, en le testant bloc par bloc, et dans l'autre je passe
beaucoup de temps sur des détails. ces détails sont moins insignifiants
cependant lorsque le code est 2 ou 3 fois plus lent que prévu...
et c'est là que réside un grand sujet que le mémoire doit traiter.
y apporter des réponses est encore mieuxxxxx.



dimanche soir, 2h40:
comment faire, donc ?
la nouvelle situation est qu'on dispose d'un autre plan, immense,
où est stockée l'information à afficher. Cela permet donc de faire des
scrollings et de sauver la totalité de l'image sans avoir à tout
recalculer. Cela veut aussi dire qu'il ne faut vider une ligne que
juste avant de commencer le calcul, lorsqu'on déplace la fenêtre,
lorsqu'un nouvelle ligne est commencée.

Le problème est un problème de mémoire cache. comme Andy le dit,
il faut charger la donnée en mémoire avant d'y écrire.
2 solution viennent à l'esprit : avoir une petite procédure qui
vide la ligne tout en effectuant un préfetch, ou modifier
la procédure d'accumulation pour que le chargement ne soit pas
utilisé. mais cette deuxième solution est hors de question car
1- la forme de l'accès ne permet pas cela
2- il faudrait rajouter 8 instructions qui seraient utilisées
dans tous les cas, ce qui ralentirait globalement le code.

Il faudrait, durant toute la partie du calcul, avoir du code
qui précharge les données et qui les efface conditionnellement,
avec du code ia32 à prédication. mais même cela:
-1 le PMMX ne dispose pas de cela dans les tables "officielles",
-2 ça rajoute du code inutile dans la plupart des cas,
  même si ça sert de bouche-trou
dans tous les cas, je ne sais pas comment faire.


Qu'est-ce qui est prêt ?

A) l'accumulation de densité:
(PRLLSUM0.ASM) (PARLLSUM.ASM est un peu boggé,
l'allocation des registres a apparemment dévoilé un bug)

org 100h

;t1=mm0
;t2=mm1
;t3=mm2
;m1=mm3
;m2=mm4
;v1=mm5
```

```
;v2=mm6
;v3=mm7

cli

xor eax,eax
xor edx,edx
mov ecx,16

wrmsr
mov cx,ax

align 16
loop_entry:

;;0
movq mm0,[S0]
;$
pcmpeqb mm3,mm3 ; m1=-1

movq mm1,[S1]
;$
movq mm5,mm0

movq mm2,[S2]
;$
pxor mm4,mm4    ; m2=0

movq mm6,mm1
psubb mm4,mm3   ; m2=0x0101010101010101

movq mm7,mm2
pand mm2,mm4

pand mm1,mm4
psllq mm2,2

pand mm0,mm4
psllq mm1,1

paddb mm2,[_M0]
por mm1,mm0

movq mm3,mm4
paddb mm1,mm2

movq mm2,mm7
psllq mm4,1           ; 10 cy

movq mm0,mm5
psrlq mm5,1

movq [_M0],mm1 ; damnit ! problème de dépendence !!! lock t2. renommage avec t1
pand mm2,mm4

pand mm0,mm3 ; oublié.

movq mm1,mm6
psllq mm2,1

pand mm1,mm4
psrlq mm5,1

paddb mm1,[_M1]
por mm2,mm0

psrlq mm6,1
paddb mm2,mm1


;;1
movq mm0,mm5
psrlq mm5,1
```

```
movq mm1,mm6
psrlq mm6,1

movq [_M1],mm2
movq mm2,mm7

pand mm0,mm3
pand mm1,mm4   ; 20 cy

pand mm2,[m3]
;$
por mm1,mm0

paddb mm2,[_M2]
psrlq mm7,1

paddb mm2,mm1
movq mm0,mm5

psrlq mm5,1
movq mm1,mm6

movq [_M2],mm2
psrlq mm6,1

movq mm2,mm7
pand mm0,mm3

pand mm2,[m3]
;$
pand mm1,mm4

paddb mm2,[_M3]
por mm1,mm0

psrlq mm7,1
paddb mm2,mm1

;;2
movq mm0,mm5
psrlq mm5,1      ; 30 cy

movq mm1,mm6
psrlq mm6,1

movq [_M3],mm2
movq mm2,mm7

pand mm0,mm3
pand mm1,mm4

pand mm2,[m3]
;$
por mm1,mm0

paddb mm2,[_M4]
psrlq mm7,1

paddb mm2,mm1
movq mm0,mm5

psrlq mm5,1
movq mm1,mm6

movq [_M4],mm2
psrlq mm6,1

movq mm2,mm7
pand mm0,mm3

pand mm2,[m3]
;$
pand mm1,mm4     ; 40 cy
```

```
paddb mm2,[_M5]
por mm1,mm0

psrlq mm7,1
paddb mm2,mm1

;;3
movq mm0,mm5
psrlq mm5,1

movq mm1,mm6
psrlq mm6,1

movq [_M5],mm2
movq mm2,mm7

pand mm0,mm3
pand mm1,mm4

pand mm2,[m3]
;$
por mm1,mm0

paddb mm2,[_M6]
psrlq mm7,1

paddb mm2,mm1
movq mm0,mm5

psrlq mm5,1
movq mm1,mm6       ; 50 cy

movq [_M6],mm2
psrlq mm6,1

movq mm2,mm7
pand mm0,mm3

pand mm2,[m3]
;$
pand mm1,mm4

paddb mm2,[_M7]
por mm1,mm0

psrlq mm7,1
paddb mm2,mm1     ; 55 cy

; stall!

dec cx
movq [_M7],mm2
jnz near loop_entry

rdtsc
sti

mov [time],eax
mov [time+4],edx

db 0CCh

ret


align 16

time dd 0,0,0,0
S0 db 0,-1,0,-1,0,-1,0,-1, 0,0,0,0,0,0,0,0
S1 db 0,0,-1,-1,0,0,-1,-1, 0,0,0,0,0,0,0,0
S2 db 0,0,0,0,-1,-1,-1,-1, 0,0,0,0,0,0,0,0
m3 db 4,4,4,4,4,4,4,4,     0,0,0,0,0,0,0,0
 db 0;...
```

```
align 16
_M0 dd 0,0,0,0
_M1 dd 0,0,0,0
_M2 dd 0,0,0,0
_M3 dd 0,0,0,0
_M4 dd 0,0,0,0
_M5 dd 0,0,0,0
_M6 dd 0,0,0,0
_M7 dd 0,0,0,0
db 0
align 16
```

B) l'accumulation des collisions:
ligne 3280

Il faut aussi penser à inclure une procédure de détection de la souris et de la mémoire. ensuite faire l'interprétation des lignes de commande et des fichiers de commande, la sauvegarde des mesures et des images... optionware...

il faut cependant trouver une solution pour notre petit problème de buffer à mettre à zero. le problème étant le préfetch, non ?

Lundi 3 am:
j'ai lu plein de docs d'Intel.
de 1: pour charger une valeur dans un compteur de performance avec WRMSR, il suffit d'utiliser EAX, EDX est ignoré.
de 2: l'explication pour les write miss est là, mais assez floue. Intel préfère se concentrer sur le P6.
de 3: j'ai fait pas mal de mesures, et il se trouve que je me suis rendu compte que WBINVD vide TOUTES les caches, pas seulement la cache de données. alors comment je vais faire pour vérifier l'effet du code de préfetch ????
de 4: j'ai imprimé les docs qui traitent des compteurs de performance.

je pense donc traiter par les essais le problème du préfetch. j'en reviens donc au premier cas où le code d'accumulation est sélectionné. Cette solution est envisageable si l'on considère que le problème à l'origine du blocage est résolu. yo.

premiers essais de préfetch: on passe de 122 à 76 cycles. encourageant. maintenant il faut trouver la meilleure configuration et gagner encore une quinzaine de cycles.

4am:
les compteurs de performance ont l'air de bien fonctionner. c'est vraiment très utile !!! malheureusement, les vérifier tous prend du temps.

J'ai réussi à descendre à 66 cycles, mais en m'apercevant que cette "#'{−|`^ç@àø+= de CPU traite les données par paquet de 128 bits, pas 256... il faut effectuer 4 opérations de préfetch au lieu de 2 comme je le pensais.

(! voir ligne 6788)

Je suis descendu à 58 cycles en enlevant les instructions de préfetch du corps de la boucle, elles effectuent des opérations assez lourdes. je me demande quelle opération accède à la mémoire sans bloquer le pipeline ? en tous cas, ça montre qu'on peut prefetcher à l'extérieur de la boucle.

Une autre mesure montre qu'il faut encore gagner 5 cycles par rescheduling, c'est à dire repairage, et on ne pourra pas faire mieux.

5am: je suis à 55 cycles !

```
lundi soir, 10h:
le PII semble assez insensible aux conditions de la memoire.
il stagne toujours à qqs pourcents autour de 59 cycles pour la boucle.
je ne sais pas a priori comment le booster. je verrai plus tard, je
pense.

11h: j'ai essayé d'assembler sp65.asm.
resize a planté. pas de compilo pascal sous la main : j'ai récupéré
mon DD 1G2 et l'ai placé dans le rack. j'ai isolé le problème qui,
si je m'en souviens, m'est déjà arrivé. il a suffit d'enlever
uses crt,dos pour en finir : l'adresse de l'erreur pointait
vers cette partie-là dans le ficher .MAP :-)

v.exe fonctionne mais la souris ne répond pas. pourtant elle fonctionne
bien avec le reste des applications et le programme ne signale pas d'erreur.

11h20: il suffisait de brancher la souris sur COM1 et pas COM2...
d'où l'intérêt de se faire son propre détecteur de souris.
le code n'a même pas réussi à trouver le pot aux roses.


11H50: le BSWAP (BYTEREV?) est chronométré à 38 cycles à vide sur PII.
je suis rassuré...

2h15: actuellement, j'ai 4 morceaux de code prêts.
A) bswap -> affichage
B) accumulation densité    \
C) initialisation densité  / 55 cycles
D) accumulation collisions = 26 cycles
BCsont environ 2 fois plus lents alors qu'ils effectuent
3 fois plus d'operations. cela vient du fait qu'il y a autant
d'accès à la mémoire dans tous les cas : load/store de 8 mots de 64 bits.
naturellement, il reste à faire l'initialisation des collisions.



mardi soir, 1h15 du matin:
hier soir avant de me coucher, j'avais mis au point une technique de
"software pipelining". elle a la contrainte de nécessiter autant de
registres que la boucle a d'opérations. je l'ai adaptée un peu
(problème de durée de boucle) pour rescheduler l'"initialisation
collisions" et le résultat est pas mal : ça permet pour des
boucles au corps simple de se passer de l'analyse des graphes.

Comme d'habitude, on commence le scheduling par la fin.
Je m'aperçois que l'initialisation à 0x0101010101010101
du masque est commune à toutes les parties de code (collision/densité
et accumulation/initialisation).

mmH=masque
mmG=valeur de la collision.

pcmpeqb mmF,mmF

pxor mmH,mmH
movq mmA,mmG

psubb mmH,mmF
movq mmB,mmG

psrlq mmA,1
pand mmG,mmH

movq mmC,mmB
psrlq mmB,2
movq [_M0],mmG
pand mmA,mmH

movq mmD,mmC
psrlq mmC,3
movq [_M1],mmA
```

```
pand mmB,mmH

movq mmE,mmD
psrlq mmD,4
movq [_M2],mmB
pand mmC,mmH

movq mmF,mmE
psrlq mmE,5
movq [_M3],mmC
pand mmD,mmH

movq mmG,mmF
psrlq mmF,6
movq [_M4],mmD
pand mmE,mmH

psrlq mmG,7
pand mmF,mmH

movq [_M5],mmE
pand mmG,mmH      ;16

movq [_M6],mmF

movq [_M7],mmG

18 cycles théoriques.
si on y rajoute le cycle de bouclage, on trouve les 19 cycles
mesurés. encore une fois, l'importance du préfetch est cruciale.
il faut trouver un endroit dans les instructions précédentes pour
caser les 4 instructions de préfetch.


2h10 : j'ai 5 morceaux prêts.
il faut maintenant passer à l'étape de déplacement, ce qui commence
par une analyse du code déjà écrit : il faut en tirer le graphe
et analyser le besoin en registres.
Ensuite on entrelace les graphes entre eux, pairs et impairs,
on rajoute le code de OR des XORs et on applatit le tout.

5h15: pour chaque cas, il suffit au maximum de 3 registres,
il reste donc un registre pour respirer. cela permet d'entrelacer
plus intimement les blocs entre eux.


vendredi, 19h:
j'ai analysé le graphe de dépendence pour les déplacements pairs.
85 instructions, 40 accès à la mémoire, 30 étages de graphe.
je n'ai même pas compté dedans les instructions de préfetch.

20h: consultation des docs: l'instruction de prefetch
sera un "test al,[mem]" car elle est pairable U et V
mais ce n'est pas pairable avec une instruction MMX qui
accède à la mémoire.

3h:
47 cycles en comptant les préfetch, soit une demi-douzaine
de cycles morts, principalement à cause du grand nombre d'accès à la
mémoire (presque la moitié).



déplacement pair:
en entrée,t4=PA, t3=PB

buggé.

;1
movq mA,[XORF]
pcmpeqb mM,mM
;2
movq mB,[XORA]
;                   STALL
```

```
;3
movq mC,[XORB]
por mA,mB
;4
pxor mM,[WallMask]
por mA,mC
;5
movq t1,[A]
por mB,mC
;6
movq [PA],t4        ;t4 contenait PA
por mA,t4
;7
movq [PB],t3        ;t3 contenait PB
pand mA,mM
;8
test al,[_M0]       ; prefetch
pxor t1,mA
;9
movq mA,[XORC]
por mB,t3
;10
movq t2,t1
por mB,mA
;11
por mC,mA
movq t3,t1
;12
paddd t2,[RANDOM]
pand mB,mM
;13
por mA,t4
psrlq t1,1
;14
movq t4,[temp_B]
psllq t3,63
;15
movq [RANDOM],t2
pxor t4,mB
;16
movq mB,[XORD]
;                   STALL
;16'
por t3,[A-CEL]
;                   STALL
;17
movq [A],t1
por mC,mB
;18
movq [A-CEL],t3
por mA,mC
;19
movq [XORD],mC
;                   STALL
;20
movq [esi+80],t4
;                   STALL
;21
por mC,[PC]
;                   STALL
;22
movq t1,[temp_C2]
pand mC,mM
;23
por mB,[PB]
pxor t1,mC
;24
movq mC,[XORE]
movq t3,t1
;25
movd t4,[temp_C_32]
por mA,mC
;26
movq t2,[edi+D]
```

```
psrlq t1,63
;27
pand mA,mM
test al,[_M2]       ; prefetch
;27'
por mB,mC
psllq t3,1
;28
movd [temp_C_32],t1
pxor t2,mA
;29
movq mA,[XORF]
por t3,t4
;30
test al,[_M4]       ; prefetch
movq t1,t2
;31
movq [esi+64],t3 ;(C)
por mB,mA
;32
movq t3,[edi+E]
por mC,mA
;33
movq t4,[temp_D]
psrlq t1,63
;34
test al,[_M6]       ; prefetch
psllq t2,1
;35
movq [temp_D],t1
pand mB,mM
;36
movq mA,mC
por t2,t4
;37
por mA,[XORD]
pxor t3,mB
;38
por mC,[PC]
movq t1,t3
;39
movq mB,[edi+F]
pand mC,mM
;40
pand mM,mA
test al,[_M0]       ; prefetch
;41
por mA,[PnOr]  ; optionnel
pxor mB,mC
;42
movq [edi+D],t2
psllq t1,1
;43
pxor mM,[edi+G]
psrlq t3,63
;44
movq [edi+F-line],mB
;                   STALL
;45
por t1,[edi-LINE+E] ; ???
;                   STALL
;46
movq [temp_E],t3  ; 32b ???
;                   STALL
;47
movq [G],mM
;                   STALL
;48
movq [edi-LINE+E],t1
;                   STALL


sortie :mA=collisions
```

```
50 cycles, 11 stalls, quelques instructions oubliées.
la manipulation manuelle de la (presque) centaine (90)
d'instructions a eu quelques petites failles
mais le bon sens n'est pas complétement endormi.

bonne nouvelle, il n'y a pas de grosse modification à faire
pour passer au mode sans collisions.



4H30 : MON DIEU QUEL ABRUTI !!!
je viens de comprendre pourquoi les lignes de cache semblaient
faire 128 bits au lieu de 256...
c'est parce que j'alignais les données sur 16 octets pour que
ça soit lisible sous DEBUG...

5h20: le code s'exécute en 53 cycles, exécute 88 instructions
et en paire 37. j'espère me rattrapper sur la version impaire
en mettant à profit les erreurs que j'ai faites pour ce code.
je ne peux plus rien faire pour lui, maintenant, il faudra aussi
penser à vérifier son parfait fonctionnement. enfin, il faudra
l'adapter au reste du code car l'accès aux variables sera changé
d'ici peu...
la dernière version est dans PRLLSUM2.ASM.



samedi 19h20:
déplacement impair: 45 cycles, 9 stalls.
j'ai été plus soigneux pour le codage mais je crois que la technique
utilisée est mauvaise, il faut revenir à la technique de la remontée
de graphes. l'allocation des registres est assez contraignante.
néanmoins, vu la saturation des L/S, je ne crois pas que beaucoup
de cycles pourraient être épargnés, je m'arrête donc là.
de plus, avec 50 cycles en moyenne pour le déplacement, on est
en-dessous d'un cycle par cellule.

entrée: t4=PA, t3=PB

;1
movq mmA,[XORF]
pcmpeqb wm,wm
;2
movq mmB,[XORA]
;                   STALL
;3
movq mmC,[XORB]
por mmA,mmB
;4
pxor wm,[WallMask]
por mmA,mmC
;5
movq t1,[A]
por mmB,mmC
;6
movq [PA],t4
por mmA,t4
;7
movq [PB],t3
pand mmA,wm
;8
por mmB,t3
pxor t1,mmA
;9
mov mmA,[XORC]
movq t2,t1
;10
por mmB,mmA
movq t3,t2
;11
paddd t2,[RANDOM]
por mmC,mmA
```

```
;12
por mmA,t4
test al,[_M0]       ; prefetch
;13
movq t4,[temp_B]
pand mmB,wm
;14
movq [RANDOM],t2
psrlq t1,1
;15
;                   STALL
;                   STALL  préfetch ?
;15'
movq [A],t1
pxor t4,mmB
;16
movq mmB,[XORD]
movq t2,t4
;17
movq t1,[temp_C2]
por mmC,mmB
;18
psllq t2,63
por mmA,mmB
;19
movq [XORD],mmC
psrlq t4,1
;20
por mmC,[PC]
;                   STALL
;21
por t2,[ESI-80] ;(B)
pand mmC,wm
;22
movq [esi+64],t4 ; @?
pxor t1,mmC
;23
movq mmC,[XORE]
;                   STALL
;24
movq [esi-80],t2
por mmA,mmC
;25
movq [esi+64],t1 ;(C)
por mmB,mmC
;26
por mmC,[XORA]
pand mmA,wm
;27
por mmB,[PB]
pxor t2,mmA
;28
movq mmA,[XORF]
movq t1,t2
;29
movq t3,[edi+E]
por mmB,mmA
;30
movq t4,[temp_D] ;32b ?
por mmC,mmA
;31
test al,[_M4]       ; prefetch
movq mmA,mmC
;32
por mmC,[PC]
pand mmB,wm
;33
por mmA,[XORD]
psrlq t1,63
;34
pand mmC,wm
psllq t2,1
;35
pxor mmC,[edi+F]
```

```
pand wm,mmA
;36
por mmA,[PnOr]
pxor t3,mmB
;37
pxor wm,[edi+G]
movq mmB,mmC
;38
movd [temp_D],t1
psllq mmC,63
;38'
movq [edi+D],t2
psrlq mmB,1
;39
movq [edi+E],t3
;                   STALL
;40
por mmC,[edi+F-line-cell]
;                   STALL
;41
movq [edi-line+F],mmB
;                   STALL
;42
movq [edi+G],wm
;                   STALL
;43
movq [edi+F-line-cell],mmC
;                   STALL

10 stall, 2 cycles en plus, 45 cycles.
je ne suis même pas sûr que ça marche.

22h: le code tourne en 48 cycles, dont 37 pairés sur 82 instructions.
je n'ai pas vérifié le fonctionnement. il faudra se référer à PRLLSUM2
pour la version la plus récente du code.

Je dispose donc d'une grande partie du code. il faut aussi inclure
la sélection progressive (PnOr) dont une partie est comptée dans
le code de déplacement (sauvegarde de Pa et Pb).



"""
Nouvelle formule:

PA2 = PA or  Rand
PB2 = PB or /Rand
PC2 = PC or (PA Rand) or (PB /rand)

ou:

PA2 = PA or  Rand
R1  = PA   Rand
PB2 = PB or /Rand
R2  = PB  /Rand
PC2 = PC or R1 or R2
"""
Mais on a oublié de masquer la sortie...
donc

Pn=PA or PB or PC
RAND=RAND Pn
/RAND=/RAND Pn
[PnOr]=Pn
PA2 = PA or  Rand
R1  = PA   Rand
PB2 = PB or /Rand
R2  = PB  /Rand
PC2 = PC or R1 or R2

masquer les RAND évite de masquer les sorties, soit 1 opération
d'économisée.

entrée:
```

```
 PB dans mmB
 RAND dans mmC
1 registre inutilisé

;1
movq t1,[PC]
movq t3,mmB
;2
movq t4,[PA]
por mmB,t1
;3
por mmB,t4
;            STALL
;4
movq t2,t1
movq mmA,mmC
;5
movq [PnOr],mmB
pandn mmA,mmB
;6
pand mmB,mmC
movq mmC,t4
;7
por t1,mmB
pand t2,mmB
;8
por t4,mmA
pand mmC,mmA
;9
movq [PC],t1
por t3,t2
; ensuite:
por t3,mmC

18 instructions et 1 stall pour 10 cycles. sympa.
le code précédent en comptait 22, avec la sauvegarde
des variables, on a bien économisé un cycle.

Ce code se "branche" tel quel dans les deux codes précédement
créés.

0h30: création d'un nouveau fichier qui décrit le coeur de la boucle:
c1.asm

la forme étrange de l'arbre précédent est dûe à l'utilisation de PA et PB,
ainsi que leur sauvegarde dans la partie de déplacement. ainsi, PC
doit être sauvé avant la fin de la sélection, et PB se retrouve
dans le chemin critique. Il faut donc finir la rotation de registres
du calcul de détection sur PB, ce qui veut dire qu'on commence par PC, puis
PA,PB,PC,PA,PB. on gagne ainsi un ou deux cycles lors de la transition
entre les blocs.

6h: C1.ASM contient un début de boucle sans :
pré-déplacement pair, popcount, détection, display_line...
j'ai mis en place le "init flag" dans le bit 16 de ebx,
il faudra vérifier son comportement.


dimanche soir, 3h:
j'ai trouvé une stratégie acceptable pour gérer les init/acc:
d'abord, init/acc est sélectionné après la partie de déplacement,
selon la valeur du bit 16 d'EBX. un seul branchement, suivi
d'un appel à la fonction désirée. le retour du branchement
contient le prologue de la boucle, dédoublé pour éviter
un branchement supplémentaire. la manipulation de la pile
perdrait le mécanisme de prédiction des retours (la pile interne),
ce qui ralentirait le retour. l'adresse de branchement, par contre,
est changée par SMC afin de gérer le cas où on affiche les
densités ou les collisions. on évite donc de dédoubler tout le
code une fois de plus.


Vendredi 30, (sam.31) 5h30 am:
chronométrage comparatif des codes FHP:
```

```
1000 itérations sur 1024*640  (655M cellules)
P200MMX+32MO SDRAM

SP65.asm: 15 secondes, strip=6, zoom=4:1   (+/− 43Mc/s)

C:
compile:
gcc −I/usr/X11R6/include −lX11 −L/usr/X11R6/lib −O3 fhp_0.c
run :
nice −−20 time −o perf ./a.out

   58.76 user
    0.44 system
 1:00.87 elapsed
     97% CPU
(0 avgtext+0avgdata 0maxresident) k0 inputs+0outputs
 (153major+178minor)page faults
 0 swaps

−−>11.2 Mc/s

mon code asm est 4 fois plus rapide sur PMMX qu'un code
optimisé en GCC.

cette différence devrait diminuer un peu sur PII à cause (grâce)
à la mémoire cache plus puissante et à 50% d'instructions décodées
chaque cycle en plus.


perf max: 512*512, strip=11, zoom=4:1, 6,3 ms pour 542² cellules,
158Hz*512²=41,6 Mc/s.


dimanche, 3am:
il faut libérer un registre. j'ai choisi pour cela de laisser
RANDOM2 au placard, il sera chargé 2 fois par "tour". l'arbre
ne peut pas se simplifier sans ce cinquième registre. sinon, G,
RA et RB sont toujours en registre. et pour l'instant je n'ai étudié
que la première partie de la détection, qui contient
déjà 28 instructions dont 8 accès à la mémoire. cela sera moins
marrant ensuite.


4am:
j'ai réussi à mettre au point la première moitié de la détection.
estimations: 14 cycles sur PMMX, 11 cycles sur PII.
à 15 cycles en moyenne par tour, cela fait environ 90 cycles sur PMMX
soit 1,5 cycle par cellule :−)


amorce:

movq _RA,[An]     ; U
;V
pxor _RA,[ST]     ; U
movq T3,_RB       ; V
;U
movq T4,RB        ; V

(ne pas oublier d'initialiser _G, _RB et AXC avec les bonnes valeurs,
et xorées de préférence...)
on peut charger ST dans un registre au départ, ce qui est fait de toute
façon car on sort du popcount.

j'avais prévu d'entrelacer les tours un peu plus mais cela n'a pas
été nécessaire. de plus, par chance, le nombre d'instructions est
pair, ce qui évite de désagréables ajustements pour le PMMX.

boucle:

pxor AXC,_RA      ;U1
pxor T3,_RA       ;V2
movq T2,_G        ;U3
```

```
pandn T4,AXC      ;V4
pandn AXC,T3      ;U5
pxor T2,_RA       ;V6
movq T1,AXC       ;U7
pand AXC,T2       ;V8
movq T3,[RAND_n]  ;U9
pandn T2,T1       ;V10
movq T1,[RAND2]   ;U11
pand T4,_G        ;V12
pxor T1,_RA       ;U13
pand T3,T2        ;V14
pand T2,_RB       ;U15
pand T1,T4        ;V16
movq [XORn],AXC   ;U17
por T3,T1         ;V18
pand AXC,[RAND2]  ;U19
pcmpeqb T1,T1     ;V20
movq [Pn],T2      ;U21
por T3,AXC        ;V22
* movq AXC,[An]   ;U23
pxor T1,T3        ;V24
* pxor AXC,[ST]   ;U25
* movq T3,_RB+1   ;V26
movq [XORn+3],T1  ;U27
* movq T4,_RB+1   ;V28

(les * montrent les lignes entrelacées avec le tour suivant/précédent)

reste maintenant à faire la deuxième partie (avec l'"accumulation").


dimanche soir, 2am:
j'ai réussi à faire le graphe pour la deuxième partie de la détection.
30 instructions dont 11 accès à la mémoire, mais je n'ai pas réussi
à éviter un cycle mort (deux bulles dans V). donc 16 cycles en PMMX
et 12 en PII. soit au total (sans l'initialisation) 1.07 cy/site en PII
et 1.4 en PMMX. je n'ai pas pu "boucler" directement avec la partie
précédente, il faudra donc charcuter la code à la jonction.


movq _RA,[A]         ;U1
movq T3,RB           ;V2
movq [XORn+3],T4     ;U3
movq T2,_G           ;V4
pxor _RA,[ST]        ;U5
movq T4,_RB          ;V6
pxor _AXC,_RA        ;U7
pxor T3,_RA          ;V8
pxor T2,_RA          ;U9
pandn T4,_AXC        ;V10
pandn _AXC,T3        ;U11
movq T1,T2           ;V12
movq T3,[RAND2]      ;U13
pandn T2,_AXC        ;V14
pand _AXC,T1         ;U15
pand T4,_G           ;V16
pand _AXC,[XORn+3]   ;U17
pxor T3,_RA          ;V18
movq T1,[RAND_n]     ;U19
pand T3,T4           ;V20
movq [XORn+0],_AXC   ;U21
pand T1,_RA          ;V22
movq T4,[RAND2]      ;U23
pand T2,_RB          ;V24
pand T2,[Pn]         ;U25
por T1,T3            ;V26
pandn T4,_AXC        ;U27
;V28
movq [Pn],T2         ;U29
por T4,T1            ;V30
pandn T4,[XORn+0]    ;U31
;V32

voilà.
```

```
maintenant, faire la jonction avec la sélection progressive des Pn
et le renommage des registres : C3.asm


4am: c'est au tour de la première partie.

je me rends compte que je mise TRES gros car je ne peux pas
à l'heure actuelle vérifier le fonctionnement correct du code.
j'ai maintenant 1700 lignes de code compilable mais probablement
extrêmement instable...

5am: C4.asm manque du popcount et du prédéplacement.
mais surtout il manque la détection de la souris et l'interprétation
des fichiers de scripts...


9h30: SP67 contient le kernel de calcul sans l'affichage moderne.
pas de test encore, mais des adaptations et des vérifications visuelles.
ça va encore être immonde à dévéroler.



Mardi soir, 4h30 am:
j'ai fini de vérifier visuellement le code de déplacement, j'ai eu qqs
problèmes avec Dpair, il manquait certaines instructions.
LABELS2 n'a pas fait de problèmes, il y avait juste une quinzaine
de labels5 en trop. WallMask renomé en WALLMASK.
mais au premier tour, le code a planté. ESC*3.
apparemment une boucle infinie ?

5h : nettoyage dans le code pour ne garder que les mouvements.
A,D,E,F fonctionnent. B et C sont effacés (?).
pointeur fou ? variable incorrectement initialisée ?
au bout de 32 (?) pas de temps, ça plante.

SP68.asm

mercredi 17h30: apparemment un problème de temp_E ?

oui: j'avais oublié un $base5__ sous [temp_E] (45 pair)

deuxième truc troublant : on dirait que RANDOM est écrit
dans le tunnel, à une adresse indépendante de la taille
de ce dernier. je pense à RANDOM car : c'est un mot de
64 bits, ses variations ressemblent à RANDOM en régime
"artificiel" (sans bouillie de bits), il varie avec
les particules A. MAIS: normalement RANDOM est initialisé
avec une certaine valeur...

18h30: j'avais oublié de "déclarer" $base5__ au début du fichier.

manquent:
-pas de collisions
-pas de déplacement en B et C
-E et F ne sont pas déviés par les parois.
(ce qui est normal puisque ça donne B et C qui ne sont pas
encore traités)

l'oubli de la déclaration étant découvert, B et C se mettent à
fonctionner mais B n'a pas le bon angle.

22H: temp_B était décalé deux fois au lieu d'une.

22H10: les collisions fonctionnent, sauf pour un petit détail:
les XORs sont inversés. quasiment rien à modifier...

22H30: problème d'explosion. il faut que j'analyse chaque combinaison.

apparemment le renommage s'est mal déroulé.

11h : apparemment le problème est un peu plus complexe,
il serait lié au fait qu'on commence la boucle sur C
au lieu de A.
```

5h30: après réflexion, le moyen le plus efficace de trouver la bonne
combinaison est d'y aller à coups de %defines dans la partie de calcul.

6h: par chance j'ai trouvé la bonne combinaison au premier essai.
mais les petites erreurs sont maintenant apparentes :
les collisions ABD "ratent" parfois, et les collisions ACE
donnent une mauvaise sortie dans 2 cas sur 3. ici, le nombre
des particules est conservé mais l'impulsion ne l'est pas,
lorsque c'est détecté.

Dimanche 13 novembre, 6h20:
je cherche la cause de mes soucis en isolant le randgen.
Il faudrait aussi faire un fichier pour DEBUG.

lundi 14, 1h du matin:
rien de probant. DEBUG a montré un comportement normal de
la détection. j'en perds mon latin. les tests ont porté sur les
combinaisons qui donnent a priori un mauvais résultat mais
sous DEBUG tout semble aller pour le mieux.

Le seul changement qui a été effectué porte sur l'accès aux données,
les $base ont été enlevés et les entrées de l'équation sont
passées en variables globales. il faut donc encore revérifier
l'écriture du code principal ...

Les faisceaux de preuves s'orientent vers un mauvais
fonctionnement de D et E. Pourtant, si la détection elle-même
s'effectue correctement, alors où se situe le problème ???

D et E m'ont mis sur la voie, BDF m'a presque confirmé, il est déjà
2h du matin. il s'agirait (à confirmer) d'un problème de
renommage des XORx lors du "déphasage" de la détection qui commence
en C et pas en A. En gros, il faudrait tout refaire ! je n'arrive
pas à trouver une cause suffisament localisée et la réécriture du
kernel prendrait encore beaucoup de temps, c'est absolument laborieux
et je n'ai pas d'outils d'aide semi-automatique.

3h: quand bien même je referais le kernel, je ne comprends pas pourquoi
ça ne fonctionne pas. de plus, même si le déplacement n'est pas parfait,
il ne peut pas être beaucoup mieux, de par sa nature memory bound.

Et puis, je ne comprends pas pourquoi XORD et XORE sont inversés avec
XORA et XORB alors que XORC ne l'est pas avec XORF. c'est là que se
noue le problème. j'en suis sûr car je n'arrive pas à l'expliquer,
ce qui complique la chose un peu plus.

4h30
je me suis aperçu que le code de test dans DEBUG ne contenait pas
l'amorce de B et C avant le POPCOUNT. je vérifie donc les résultats
antérieurs ...

1) XORD absent pour BDG-R, ok pour BDG-/R
2) XORE absent pour CEG-R, ok pour CEG-/R
le reste est bon (ACG, DFG, EAG, FBG)

essai avec BCF Co:
ABD,ACF,BDE,ACD,ADE,BEF,BCF,CEF,ADF: ok

BCE, ABE: PB est mis ????
XORx manquant pour BCE et CDF
XORD et XORE sont encore en cause !

<enlever XORG et G2 des variables globales...>

6h: vu la tournure des choses, et puisque toute autre éventualité
a été envisagée, il est envisageable que ce soit une erreur de renommage
de registre. mais il y a 300 lignes à vérifier... une à une...

```
cela expliquerait le PB ?


-trouver dans quelle moitié se trouve la bourde.
on s'aide avec Pn et XORn intermédiaires.
on place un RET en plein milieu.
ensuite, on empêche l'écriture des résultats de la première
partie et on regarde le résultat.


1)

A:          00 FF 00 FF FF 00
B:          00 00 FF 00 FF FF
C:          FF 00 00 FF 00 FF
D:          FF FF 00 00 FF 00
E:          00 FF FF 00 00 FF
F:          FF 00 FF FF 00 00
G:          00 00 00 00 00 00

XORA:       FF FF FF FF 00 F0
XORB:       F0 FF FF FF FF 00
XORC:       00 00 00 00 FF 00
XORD:       00 00 00 00 00 FF
XORE:       FF 00 00 00 00 00
XORF:       FF FF FF FF F0 FF


PnOr:       00 00 00 00 00 00
PA:         00 00 00 00 FF 00
PB:         00 00 00 00 00 FF
PC:         00 00 00 FF 00 00
RANDOM:     0F 0F 0F 0F 0F 0F



RAND_A:     FF FF FF FF FF FF
RAND_B:     FF FF FF FF FF FF
RAND_C:     00 00 00 00 00 00
ST:         00 00 00 00 00 00
S1:         FF FF FF FF FF FF
S2:         FF FF FF FF FF FF
S3:         00 00 00 00 00 00



2)
2E73:04C0  00 00 FF 00 00 00
2E73:04D0  00 00 00 FF 00 00
2E73:04E0  FF 0F FF FF FF FF
2E73:04F0  FF FF 00 FF 00 00
2E73:0500  00 FF FF 00 FF 00
2E73:0510  00 FF 00 00 00 00


PnOr:       FF FF FF 00 00 00
PA:         00 FF 00 00 00 00
PB:         FF FF FF FF FF FF
PC:         FF 0F 0F 00 00 00



samedi 4 décembre , 7h40:
ça fait 2 semaines que je retourne ça dans ma tête sans trouver le repos.
cette fois-ci je ne sais pas si j'aurai le courage de débugger.
j'ai déjà perdu deux semaines dessus.
A la relecture des logs, je vois que seule la détection est fautive,
je peux donc recommencer le codage en tenant compte des erreurs trouvées.
je peux aussi changer l'infrastructure du programme (passage aux noeuds
de 128 octets). c'est peut-être la meilleure chose à faire pour me remettre
dans le bain et avancer quand même.

SP73.asm
```

premier problème, celui des coordonnées avec les coordonnées à l'écran.
il est temps de poser les équations.
mais pour simplifier le code je passe outre le zoom car il n'y a qu'une
seule procédure d'affichage de prête.

vendredi 28 janvier :

de retour de Boston j'installe le nouveau moniteur SONY trinitron :
même si la carte vidéo S3 trio 3D ne l'utilise qu'à 60Hz au lieu des 85Hz
que le moniteur supporte, l'appariston du rouge rend les blancs plus...
blancs !

J'ai rencontré Norman Margolus. sur un P3 à 550MHz de son lago, j'ai
pu monter à 150Mc/s les doigts dans le nez !
je n'ai pas pu monter plus loin pour des raisons idiotes mais je pense
que j'aurais pu atteindre 200Mc/s. le programme est cependant descendu
à 3.6 cycles CPU par octet.

Le plus important est que j'ai pu atteindre l'équivalent d'une CAM8,
même si le P3 et la CAM8 ont de nombreuses années d'écart. J'ai cependant
pu montrer qu'il est possible d'atteindre des performances "similaires"
avec une machine de bureau (même si c'est la toute dernière) :
l'avantage économique est capital et prouve l'intérêt de mon programme.

passons en revue les avantages et inconvénients de nos deux bébêtes :
(PC 550 contre CAM8/25/64MO)

1) la CAM8 est "limitée" à des dimensions qui sont des puissances de deux
alors que le programme n'a qu'une granularité de 64 sites en horizontal :
−> meilleure utilisation de la mémoire lorsque des ratios précis doivent
être utilisés (1 point pour moi)

2) la CAM8 peut effectuer des simulations jusqu'à dimension 32 :
i point pour Norman, mais la CAM8 n'a jamais étée jusque là : on se limite
à dimensions 4 ou 5 dans les cas les plus extrêmes.

3) la CAM8 a des "cellules" codées sur 16 bits (les LUT aussi), qui peuvent
être cascadées (mais pas pipelinées...) pour atteindre 32, 48 bits si
nécessaire. Ainsi, à 25 MHz par carte, on peut faire fonctionner FHP3
à 2 cellules par cycle, ou faire un dérivé à 2 bits par direction :
1 point pour Norman ! il peut faire du FHP3 simple à 2*25*8=400 Mc/s
soit deux fois le record possible sur PC.

4) la CAM8 peut définir n'importe quel voisinage de cellule,
alors que d'un autre côté mon truc permet une fonction similaire,
mais que sur une ligne. Un demi point pour Norman : sa machine est générale.

5) sa machine s'interface avec un système d'exploitation de haut niveau
(Solaris). Je n'en connais pas les détails ni l'occupation de la CPU.
Une interface en mode texte et un affichage graphique sont fonctionnels.
De mon côté j'ai une GUI minimale et un projet de langage script.
Même si Harris Gilliam travaille dessus et que le projet en aie les
capacités, elles ne sont pas développées ou arrivées à maturité.
Mon critère est que l'interactivité n'est pas au centre du projet,
ce sont des outils qui sont rajoutés au système (exception faite du
moniteur VGA qui dispose de sa propre électronique synchrone aux cartes).

Je pense qu'on peut battre la CAM8 actuelle (celle que j'ai vu fonctionner)
avec un bi−P3 à 600MHz environ. Mais de son côté, Norman nous prépare
une super bécane à partir de 128 puces full−custom à 200MHz...
Je ne mets pas en doute sa puissance et sa capacité à fonctionner
mais son approche brute (même si elle est intelligente et reflète son
expérience) me choque un peu, moi qui n'ai que mon ordinateur et une machine
biprocesseur à la fac pour développer son projet. Norman, lui, peut compter
sur la DARPA pour obtenir le million (de francs ou de dollars) que
nécessite son projet ambitieux. On en revient donc toujours
aux problèmes d'argent, et je n'en ai pas.

Les applications les plus récentes de sa machine sont la cryptographie
par AC et les AC quantiques, afin de simuler un ordinateur quantique

```
(on en revient encore à la crypto). Je pense pouvoir mettre en doute
la puissance de la crypto par FHP, après relecture de mon bouquin chéri.


Actuellement je ne me sens plus d'attaque pour refaire le programme de fond
en comble. J'ai acheté le livre de Zaleski et Rothman (j'aurais peut−être dû
rencontrer ce dernier lorsque j'étais à Boston !) donc je ne risque pas de
m'ennuyer, il contient de nombreuses remarques très intéressantes qui
pourraient trouver leur place dans le mémoire. Mais "Laisse les vieux
donner des leçons" (dixit PG).


Quelques additions au programme :
− changer le butterfly d'affichage par réordonnancement des masques
d'accumulation
− mode script "interactif" (ligne de commande) sur appui de "tab"
ou "caps lock"... {−> fontes de textes qu'on peut récupérer dans la table
de fontes VGA...} −> trouver addresse et format des fontes VGA (bible PC ?)
```