Mécanismes de traces efficaces pour programmes multithreadés

Vincent DANJEAN — Pierre-André WACRENIER

Projet INRIA Runtime
LaBRI / INRIA-Futurs, Université Bordeaux 1
351, cours de la Libération
33405 Talence Cedex - France
{Vincent.Danjean,Pierre-Andre.Wacrenier}@labri.fr

RÉSUMÉ. *Aujourd'hui, il est très difficile d'observer et de comprendre finement les performances des applications reposant sur des supports d'exécution multithreadés, en particulier lorsque la plateforme de threads utilisée est complexe (ordonnancement multi-niveaux). Nous proposons un environnement permettant d'observer précisément le comportement des applications multithreadées donnant ainsi accès à des informations telles que le nombre de cycles consommés par une fonction donnée ou l'ordonnancement exact des threads utilisés.*

ABSTRACT. *Nowadays, observing and understanding the performance of a multithreaded application is untrivial, especially within a complex thread environment (multilevel scheduling). Thanks to our environment, the run of a multithreaded application can be precisely analyzed. In particular, our environment allows the programmer to measure the number of CPU cycles used by a given function or the precise scheduling of the threads of the application.*

MOTS-CLÉS : *Profilage, application multithreadée, ordonnancement à deux niveaux, SMP*

KEYWORDS: *Profiling, multithreaded application, two-level scheduling, SMP*

1. Introduction

Déterminer les goulots d'étranglement, vérifier le bon comportement d'une application dans des circonstances très précises ou encore comprendre la portée réelle d'une optimisation nécessitent une analyse extrêmement fine de l'exécution d'une application. Pour ce faire on utilise des outils de *profilage* ; ces outils peuvent être séparés en deux catégories suivant le point de vue (noyau ou applicatif) qu'ils adoptent pour observer l'exécution de l'application.

Le point de vue *noyau* est obtenu essentiellement par une instrumentation légère du système d'exploitation. Cette instrumentation est souvent intégrée au code source du noyau par le programmeur (LINUX TRACE TOOLKIT [YAG 00], FAST KERNEL TRACE [RUS 01]), mais elle peut l'être aussi dynamiquement (KERNINST [TAM 99]). Ces outils permettent une analyse très fine des applications : dans [RUS 01], une analyse des performances de la pile TCP est menée, afin de comprendre le manque d'efficacité relatif d'un circuit de calcul des sommes de contrôle.

Le point de vue *applicatif* est obtenu par l'instrumentation du programme exécutable, souvent au moyen d'outils intervenant à la génération de code. De nos jours, toutes les plates-formes de développement sont pourvues de ce type d'outils ; parmi ceux-ci, on peut citer les utilitaires GNU `gcov` et `gprof` [GRA 82] ou encore l'outil d'INTEL `VTune` qui est capable de tirer partie des compteurs de performance des processeurs INTEL et de suivre le cheminement des threads ordonnancés par le système d'exploitation.

Cependant, on distingue plusieurs familles de bibliothèques de threads suivant que l'ordonnancement est réalisé par le système d'exploitation (on parle alors de *thread noyau* comme ceux de la bibliothèque standard LINUXTHREAD [LER 96]) ou directement par l'application (*thread utilisateur* comme ceux de la bibliothèque GNU-PTH [ENG 03]). Les threads utilisateurs permettent l'obtention de meilleures performances et une plus grande flexibilité car ils sollicitent peu le système d'exploitation et n'ont que peu de dépendance à son égard, en contrepartie ils ne permettent pas d'exploiter pleinement les machines multiprocesseurs (SMP). Afin de concilier le meilleur des deux mondes, une troisième famille de bibliothèques de threads a vu le jour : les bibliothèques de threads à deux niveaux. Dans ce cadre, un ensemble de threads noyaux servent à ordonnancer les threads de niveau utilisateur de l'application[1]. Ces threads noyaux peuvent être gérés par l'application comme dans la bibliothèque NGPT [ABT 02], ou bien être gérés en partie ou totalement par le système d'exploitation (cas des systèmes SOLARIS et de ceux basés sur le mécanisme des *scheduler activations* [AND 91, DAN 00]).

Ainsi, l'ordonnancement à deux niveaux des threads d'une application sur une machine multiprocesseur dépend de nombreux facteurs et il est en pratique imprévisible ;

1. La bibliothèque de threads MARCEL [DAN 02] de l'environnement de programmation multithreadé distribué PM^2 [NAM 95] permet de choisir entre un ordonnancement à deux niveaux et un ordonnancement au seul niveau utilisateur.

il est tout aussi difficile de déterminer, pour une application multithreadée donnée, si la politique d'ordonnancement du noyau (ou celle de la bibliothèque de threads utilisée) a conduit à une exploitation optimale ou non des ressources disponibles. De nos jours, les développeurs des supports d'exécution basés sur le multithreading (OPENMP [Ope99], PADICOTM [DEN 02], MPICH/MADELEINE [AUM 03]) ont un besoin crucial des informations d'ordonnancement afin d'évaluer et de comprendre les performances de ces environnements.

Le but de notre travail est la réalisation d'un environnement permettant d'analyser finement le comportement précis d'un programme multithreadé, même lorsque les mécanismes d'ordonnancement sont complexes. Cet environnement permettra, par exemple, de déterminer le temps CPU individuellement consommé par chaque thread de l'application et sur quel(s) processeur(s) il s'est exécuté. Pour parvenir à nos fins, nous devons concilier plusieurs impératifs :

Être très peu intrusif. Non seulement le surcoût doit être faible afin de modifier le moins possible les performances du programme, mais aussi pour ne pas influer sur l'ordonnancement *normal* des threads de l'application (par exemple en augmentant fortement le temps d'exécution des sections critiques, en créant de nouveaux points de changement de contexte pour l'ordonnanceur, etc.)

Savoir gérer une grande quantité d'informations. La connaissance du comportement précis d'une application multithreadée nécessite la récolte de beaucoup d'informations (décisions d'ordonnancement, entrée et sortie des threads pour les fonctions qu'ils exécutent, etc.) Il convient d'être attentif à la masse de données générées, de l'ordre du Mo/s, afin que l'application n'en souffre pas (consommation mémoire, etc.)

Maîtriser les ordonnancements complexes. Les techniques développées devront permettre de mettre en évidence les ordonnancements réels des threads de l'application sur les plates-formes complexes (SMP avec bibliothèque de threads à deux niveaux). À cette fin, il apparaît clairement que l'on aura besoin d'informations de l'ordonnanceur utilisateur (en particulier les changements de contexte des threads utilisateurs) mais aussi d'informations du système d'exploitation (quand et sur quel processeur s'exécutent les threads noyaux).

Après un rappel des techniques de recueil de performances, nous présentons notre propre technique qui repose sur l'utilisation d'une trace noyau et d'une trace utilisateur pour observer l'exécution des applications multithreadées ; nous présentons ensuite notre environnement de profilage et détaillons, en quelques points, les techniques que nous avons mises en œuvre pour garantir la qualité des mesures obtenues.

2. Techniques pour le recueil des performances

2.1. *Un exemple de base :* `gprof`

L'outil `gprof` [GRA 82] fournit pour chaque fonction (ou ligne) du programme son nombre exact d'appels et une estimation du temps pris par cette exécution. Ce profilage se fait en trois étapes :

1) instrumention du programme cible à la compilation : un appel à une fonction de comptage est inséré au début de chaque fonction afin d'incrémenter le compteur attaché à l'arc `(fonction appelée, adresse de retour)` ;

2) recueil des informations lors de l'exécution : outre les informations nécessaires au graphe d'appel, la valeur du compteur ordinal est enregistrée par échantillonnage dans un « histogramme » (tous les centièmes de seconde par exemple, ou mieux, en variant de façon aléatoire la fréquence d'échantillonnage) ;

3) synthèse *post-mortem* des informations : le graphe d'appel est reconstitué grâce aux compteurs attachés aux arcs et la durée d'exécution des fonctions est estimée grâce à l'histogramme du compteur ordinal.

Ainsi `gprof` exploite les deux techniques de base du profilage des performances : l'enregistrement d'une trace d'événements et l'échantillonnage du compteur ordinal. Contrairement au traçage d'événements, la technique d'échantillonnage ne nécessite pas l'instrumentation du code et elle est moins intrusive ; cependant, les durées d'exécution ainsi estimées reflètent rarement la réalité. Par exemple, l'estimation de la durée d'exécution d'une fonction est calculée par `gprof` une fois pour toutes, indépendamment de la valeur des paramètres effectifs ou du chemin d'appel.

2.2. *Éléments matériels pour l'évaluation des performances*

La seule mesure du temps ne peut mettre en évidence des contre-performances d'origine matérielle (tels des défauts de cache répétés), c'est pourquoi tous les processeurs modernes intègrent des *compteurs de performances*.

À titre d'exemple, le PENTIUM IV dispose d'un compteur de cycles sur 64 bits et de 18 registres compteurs de performances où 45 types d'événements peuvent y être dénombrés [Int03]. Ces compteurs de performances sont spécialisés (4 concernent plutôt le bus et la mémoire, 4 concernent l'horloge et les sauts, 4 concernent les flottants et 6 concernent le pipeline). Chaque registre dispose d'un compteur (40 bits) et de bits de contrôle qui servent (1) à définir l'ensemble des événements à compter et (2) à fixer le nombre d'occurrences à atteindre pour déclencher une interruption. Il est ainsi possible de baser l'échantillonnage du compteur ordinal, non plus seulement sur le temps, mais aussi sur le décompte d'événements déterminés (tous les 1000 défauts de cache de niveau L2, par exemple) ; cette technique est mise en œuvre par le profileur `VTune`.

Si de nombreuses bibliothèques spécialisées facilitent l'accès aux compteurs de performances, les bibliothèques PAPI [BRO 00] et PCL [BER 98] se dégagent en ce sens qu'elles proposent des API portables exploitant les compteurs de performances des processeurs actuels. PAPI propose en outre des fonctionnalités plus complexes comme le multiplexage temporel des compteurs (mise à disposition de compteurs virtuels au détriment de la précision des mesures) et la création d'histogrammes basés sur des compteurs de performances.

Notons enfin que l'enregistrement d'un registre de performance coûte avec PAPI de l'ordre de 1300 cycles sur PENTIUM [MUC 02] et devrait être bientôt ramené à 230 cycles (le coût d'accès à un registre de performance étant de l'ordre de 100 cycles).

2.3. *Instrumentation des applications réparties*

À la base, l'analyse des applications réparties réclame l'instrumentation des applications et de leur support d'exécution afin de relever non seulement les valeurs des compteurs de performances mais aussi de conserver la trace de certains événements (entrée/sortie de procédure, changement de contexte de threads, réception/envoi de message, etc.)

Dans sa thèse [SHE 01], Shende définit des techniques pour relier des observations de bas niveau aux requêtes exprimées à un niveau plus abstrait et propose une stratégie d'instrumentation multi-niveaux. Shende montre ainsi que l'analyse d'une application complexe peut nécessiter des interventions aux niveaux suivants : code source, préprocesseur, compilateur, bibliothèque, support d'exécution et machine virtuelle. Shende a validé son approche en s'appuyant sur la suite logicielle Tau [SHE 98] qui propose une API et un ensemble d'outils portables pour l'analyse des programmes parallèles écrits en C, C++, Fortran ou Java. Notons que Tau est relativement peu intrusif puisque, d'après [MAL 04], le coût d'un enregistrement est de l'ordre de 1400 cycles (500*ns* sur un PENTIUM IV cadencé à 2,8 GHz). Cependant, en environnement multithreadé, Shende [SHE 01] utilise des verrous de haut niveau, ce que nous désirons absolument éviter.

Notons que l'analyse des programmes MPI a retenu une attention soutenue, sans doute parce que, outre sa popularité dans le milieu des hautes performances, MPI comporte une API de profilage standardisée (PMPI). Ainsi, depuis quelques années déjà, on dispose d'environnements de profilage MPI performants ; le lecteur trouvera dans [MOO 01] un panorama des outils de profilage MPI tels que Tau et Vampir de la société PALLAS / INTEL. Parmi ces environnements, on peut distinguer le projet KOJAK [MOH 03] qui propose une analyse automatique des performances des programmes MPI et OPENMP. Ainsi KOJAK intègre :

– des bibliothèques d'accès aux compteurs de performances (PAPI et PCL) ;

– un logiciel d'instrumentation de programmes et de support d'exécution (Tau) ;

– une bibliothèque de stockage et de manipulation de traces (EPILOG) ;

– un outil d'analyse automatique des performances piloté par un langage de spécification des propriétés de performance (EXPERT, [WOL 03]) ;

– un outil de visualisation, celui de l'environnement de profilage Vampir, par exemple.

La neutralité de l'observation et l'interprétation des performances obtenues constituent donc deux problèmes qui prennent une dimension supérieure dans le cadre des applications parallèles ou réparties. Par exemple, dans [MAL 04], Malony et Shende proposent et étudient une technique de compensation à la volée des erreurs d'estimation introduites par la prise de traces et d'échantillons. Ils montrent que leur technique est adaptée aux programmes séquentiels et améliore la précision des mesures des programmes parallèles. Toutefois, ils montrent que cette technique ne permet pas d'estimer l'effet intrusif de la prise d'une mesure sur un processus autre que celui qui la réalise. Enfin, l'observation fine d'une exécution d'un programme parallèle implique l'utilisation de mécanismes de synchronisation (pour la simple sauvegarde des mesures, par exemple) qui peuvent modifier radicalement le comportement parallèle du programme et, au bout du compte, dénaturer l'expérience.

3. Des traces noyaux à un environnement de profilage multithread

3.1. *Les Fast Kernel Traces*

FKT (FAST KERNEL TRACE) [RUS 01] est une méthode permettant d'obtenir une trace du flot d'exécution du système d'exploitation LINUX sur machines multiprocesseurs. Cette trace peut être aussi précise que voulue par le programmeur noyau. La méthode consiste à placer des macro-commandes spéciales directement dans le code source du noyau. Ainsi, contrairement à des outils d'instrumentation dynamique comme KernInst, l'instrumentation du noyau ne peut se faire à la volée : la modification d'un point d'enregistrement nécessite la modification du code source et donc la recompilation du noyau et son redémarrage. Néanmoins, l'enregistrement de la trace dans un tampon peut être démarré ou interrompu à tout instant depuis l'espace utilisateur. Le mécanisme de sauvegarde de ce tampon est très optimisé[2] [THI 03].

Détaillons les arguments de la macro-commande présentée dans la Figure 1 :

– Le premier argument, KEYMASK, allié à la variable globale du noyau fkt_active, permet de valider ou non l'enregistrement de l'événement lors de l'exécution. La variable globale fkt_active peut être positionnée depuis l'espace utilisateur grâce à un nouvel appel système. Notons qu'il suffit d'utiliser directement la fonction fkt_header pour éviter un test inutile.

2. Ce tampon est en effet confondu avec la projection en mémoire du fichier associé (son *buffer-cache*) et les pages de ce dernier, gérées comme un buffer tournant, sont automatiquement sauvegardées et évincées par le gestionnaire des pages (la *TLB*), évitant ainsi toute recopie et même tout gaspillage mémoire.

```
#define FKT_PROBE2(KEYMASK,CODE,P1,P2)                \
  do {                                                \
      if( KEYMASK & fkt_active )                      \
         fkt_header( ((unsigned int)(CODE)),          \
            (unsigned int)(P1), (unsigned int)(P2) );\
        } while(0)
```

Figure 1. *Définition simplifiée d’une macro-commande* FKT

– Le deuxième argument, `CODE`, doit être caractéristique de l’événement à enregistrer : il servira à repérer les différentes occurrences de l’événement dans la trace collectée. Quelques événements sont prédéfinis, comme celui marquant les changements de contexte, les autres codes étant à la disposition du programmeur.

– Le programmeur a ensuite, à sa disposition, jusqu’à 5 arguments entiers supplémentaires (ici 2).

La fonction `fkt_header` enregistre l’événement en utilisant la meilleure estampille disponible puisque FKT fait appel au registre compteur de cycles (le *time-stamp counter register* du PENTIUM) qui fournit sur 64 bits le nombre de cycles écoulés depuis la mise sous tension du processeur ; cette horloge est suffisamment dimensionnée puisque sa période est supérieure à 136 années (2^{32} s) pour un processeur cadencé à 4 GHz (2^{32} Hz).

Ainsi, FKT est un système de trace noyau peu intrusif (une mesure consomme entre 80 et 230 cycles) et flexible puisque l’utilisateur a la capacité de placer et de prendre ses mesures où et quand il le désire. Il devra être étendu pour atteindre notre objectif.

Nous reviendrons plus en détail sur FKT dans la section traitant de l’implémentation, cependant le lecteur pourra se référer à [RUS 02] pour la description et l’analyse complètes de ce système.

3.2. *Les contraintes issues de l’ordonnancement à deux niveaux*

Afin de tracer précisément le comportement d’un programme multithreadé, nous devons être capables de déterminer, à tout instant, quel thread s’exécute sur quel processeur physique. Cette simple requête mène aux constats suivants :

Les informations du noyau seules ne suffisent pas. Le noyau, n’ayant pas connaissance des threads utilisateurs des applications utilisant une bibliothèque de threads à deux niveaux, est dans l’incapacité de donner leur ordonnancement au sein des threads noyaux.

Les informations de l’application seules ne suffisent pas. Supposons qu’une exécution d’un programme utilise quelques threads utilisateurs s’exécutant au sein de trois threads noyaux sur une machine biprocesseur. S’il est possible de déterminer *sans information du noyau* quel thread noyau exécute quel thread utilisateur[3], on ne peut cependant pas déterminer la date des changements de contexte du thread de niveau noyau. Plus problématique encore, sans information du noyau, on ne peut pas connaître les dates de préemption des trois threads noyaux par le système d'exploitation. Autrement dit, la préemption d'un thread utilisateur due à la préemption du thread noyau sous-jacent ne laisse pas de trace dans l'espace utilisateur ; dans ces conditions, il est généralement impossible d'obtenir des mesures exactes sur un thread donné, comme par exemple, son temps d'occupation des CPU.

La communication espace noyau / espace utilisateur est trop coûteuse. Une première solution serait de créer de nouveaux appels systèmes pour, par exemple, connaître le processeur exécutant l'enregistrement de l'événement ou encore pour que l'application informe le noyau de ses changements de contexte utilisateur. Mais cette solution est trop intrusive : outre le temps d'exécution prohibitif d'un appel système (cf. les micro benchmarks de la section 5.3), cette solution aurait également pour effet de fournir à l'ordonnanceur l'occasion de prendre des décisions d'ordonnancement bien plus fréquemment que lors d'une exécution non instrumentée. Une seconde solution serait l'utilisation par le noyau d'*upcalls*, c'est-à-dire que le noyau force l'application à appeler une fonction, comme le fait le mécanisme des signaux, pour transmettre ses événements à l'application. Toutefois, cette seconde solution est coûteuse car elle nécessite la sauvegarde de l'ensemble des registres.

3.3. *Notre proposition*

La solution retenue est de laisser FKT générer des événements relatifs au noyau et de développer un système similaire, appelé FAST USER TRACE (FUT), pour la génération des traces de l'application. Les deux points clés de cette solution sont :

– les traces noyau et applicatives sont nécessaires à l'obtention d'une analyse complète de l'exécution de l'application ;

– les traces noyau et applicatives partagent la même référence de temps afin de pouvoir reconstruire l'ordonnancement complet des threads de l'application sur la machine.

Une fois les deux traces fusionnées en une unique *supertrace*, on dispose pour chaque événement, en plus de son code, de ses paramètres, de son estampille et de l'identité du processeur, de celle du thread noyau et celle du thread utilisateur à l'origine de cet événement. En conciliant ainsi les points de vue noyau et utilisateur, cet environnement de profilage permet non seulement de reconstituer le déroulement d'une application multithreadée complexe mais aussi de replacer ce déroulement dans son contexte d'exécution, les événements majeurs du noyau pouvant être aussi tracés et analysés.

---

3. Il suffit de connaître le premier thread utilisateur sur chaque thread noyau puis de suivre les changements de contexte utilisateur.

Sur le plan pratique, FUT est très similaire à FKT à la fois dans son principe de fonctionnement et dans son utilisation. Seuls quelques champs de la trace enregistrée ont été adaptés pour une utilisation en espace utilisateur ; une description du format des traces est détaillée en Section 5.1.

4. Description de notre environnement

Après avoir présenté notre plate-forme d'implémentation, nous détaillons l'interface d'instrumentation des applications puis nous nous intéresserons au traitement des traces obtenues.

4.1. *Plate-forme d'implémentation*

Nous avons implémenté notre environnement de profilage sur l'architecture INTEL $x86$ à la fois dans le noyau LINUX et dans MARCEL, la bibliothèque de threads purement utilisateurs ou à deux niveaux de l'environnement PM^2.

Pour avoir le support de FKT, on instrumente le noyau en un ensemble de points en lui appliquant un *patch*. Ainsi, des événements sont mémorisés lors des changements de contexte, lors des entrées et sorties d'interruption matérielle (IRQ, etc.) ou logicielle (appels systèmes, etc.).

L'instrumentation de la bibliothèque de threads MARCEL permet de mémoriser les événements nécessaires à la reconstruction de l'ordonnancement global (changement de contexte des threads utilisateurs, création et terminaison de threads noyaux). Cette instrumentation permet aussi l'analyse de toutes les fonctions de la bibliothèque MARCEL. De cette façon, on peut, par exemple, tracer précisément les performances de la bibliothèque et/ou voir les causes de préemption des threads (temps écoulé, attente de verrou, etc.) L'instrumentation de la bibliothèque MARCEL n'a pas demandé beaucoup de travail et pourrait être très facilement reproduite dans toute autre bibliothèque de threads.

4.2. *Outils d'instrumentation*

Nous avons repris l'interface de FKT. Ainsi, l'enregistrement d'événements repose principalement sur l'utilisation des macros `FUT_PROBE`x`()` et quelques types d'événements sont prédéfinis, comme ceux marquant le début ou la fin des threads.

Notons aussi que nous avons développé un outil de pré-traitement. Celui-ci attribue automatiquement un code d'entrée et un code sortie à chaque fonction. Pour instrumenter un programme, il suffit alors d'insérer les macros `PROF_IN()` et `PROF_OUT()` au début et à la fin des fonctions à profiler. L'insertion de ces macros peut être faite manuellement ou automatiquement : le compilateur `gcc` dispose de fonctionnalités d'instrumentation des fonctions compilées et ce, au moyen de deux appels de fonctions supplémentaires.

4.3. *Analyse des traces*

Une fois les traces utilisateur et noyau générées, celles-ci sont fusionnées dans une *supertrace* au moyen de l'outil `supertrace`. Dans cette supertrace, les événements sont interclassés temporellement et chaque événement est relié au processeur, au thread noyau et au thread utilisateur qui l'ont enregistré. Toutefois, certains événements, comme ceux enregistrés dans les routines d'interruptions, ne peuvent être associés à un thread utilisateur.

L'analyse de cette supertrace peut être menée par un utilitaire en ligne de commande, appelé `sigmund`. Il permet d'appliquer à la supertrace un ou plusieurs filtres. On peut, par exemple, extraire les événements concernant un processeur, un thread noyau ou un thread utilisateur ; on peut aussi sélectionner les événements produits entre deux dates ou durant l'exécution d'une fonction spécifiée, etc.

Des informations telles que la mesure du temps actif, du temps d'exécution réel des threads ou des fonctions sélectionnées peuvent être extraites des événements ainsi sélectionnés. Il est alors possible de profiler précisément les fonctions d'un processus multithreadé, même avec une charge importante sur le système.

4.4. *Exemples*

```
int fonction_instrumentee(int foo, int bar) {
  int res;

  PROF_IN();
  if (foo) {
    FUT_PROBE2(code_perso, foo, bar);
    ...
    PROF_OUT();
    return res;
  }
  ...
  PROF_OUT();
  return 0;
}
```

Figure 2. *Exemple d'instrumentation de code*

La Figure 2 montre comment instrumenter une fonction dans le code de l'application à l'aide des macros `PROF_IN()`/`PROF_OUT()` pour que la fonction génère un événement à chaque entrée et à chaque sortie.

```
$> sigmund --trace-file supertrace.log --thread 15 \
   --event CONTEXT_SWITCH --list-events
type  date_tick  pid cpu thr   code name                 param(s)
[...]
USER  97615576  7137   1   7  23014 USER_CONTEXT_SWITCH  15
USER  97757052  7137   1  15  23014 USER_CONTEXT_SWITCH  8
USER  98006248  7136   0   6  23014 USER_CONTEXT_SWITCH  15
KERN  98139183  7136   0  15  23014 KERN_CONTEXT_SWITCH  6152
KERN  98638163  2352   2   ?  23014 KERN_CONTEXT_SWITCH  7136
USER  99060185  7136   2  15  23014 USER_CONTEXT_SWITCH  7
[...]
$> sigmund --trace-file supertrace.log --thread 15 --active-time
Temps actif total = 130193845 cycles
```

Figure 3. *Interrogation de la supertrace avec sigmund*

La macro `FUT_PROBE2(code_perso, foo, bar)` permet de générer un événement dans la trace dont le code sera `code_perso`. Cet événement aura deux arguments qui lui seront associés (`foo` et `bar`).

La Figure 3 montre deux exemples d'informations fournies par notre environnement. Le programme instrumenté est un programme multithreadé exécuté sur une machine biprocesseur XEON SMT (soit 4 processeurs logiques). La bibliothèque de threads utilisée est une bibliothèque à 2 niveaux utilisant 4 threads noyaux pour propulser ses threads utilisateurs. On essaie d'obtenir des informations concernant l'ordonnancement d'un thread particulier (le numéro 15 sur cet exemple).

La première requête sert à connaître les événements de la trace relatifs aux changements de contexte concernant le thread sélectionné. On peut voir ici quelles sont les informations enregistrées dans la supertrace pour chaque événement, à savoir :

type	le type d'événement (noyau ou utilisateur) ;
date_tick	sa date ;
`pid`	le thread noyau s'exécutant à cet instant ;
`cpu`	le processeur qui a enregistré la trace ;
`thr`	le thread utilisateur s'exécutant à cet instant (s'il est profilé) ;
`code`	le code de l'événement ;
`name`	le nom de l'événement ;
`param(s)`	les paramètres éventuels associés à l'événement.

Sur cet exemple, on voit que le thread utilisateur 15 a été ordonnancé par le thread noyau 7137 sur le processeur 1. Puis, il a été ordonnancé par un autre thread noyau (le 7136) sur le processeur 0. Ce thread noyau a été préempté par le noyau (qui a donné la main à une autre application) pour être réordonnancé sur le processeur 2 un peu plus tard. Le thread utilisateur a ensuite passé la main.

Grâce à ce type d'analyse, on peut voir quelle a été l'affinité d'un thread donné avec les processeurs physiques. Sur cet exemple, on voit que le thread 15 n'a pas eu d'affinité marquée avec un thread noyau particulier et qu'un thread noyau lui-même a été déplacé par le système.

La seconde requête permet de connaître le temps CPU total consommé par un thread utilisateur qui a été ordonnancé par différents threads noyaux sur différents processeurs.

4.5. *Vers une présentation graphique de la trace*

Une fois la supertrace calculée, nous disposons de l'ordonnancement effectif des threads de l'application sur la machine ainsi que des événements enregistrés qui leur sont associés. Le programme `sigmund` permet alors d'interroger cette supertrace et d'en extraire les informations désirées. Cependant, une analyse globale du comportement du programme est humainement difficile à mener en mode texte ; une représentation graphique peut s'avérer mieux adaptée.

Plutôt que de développer notre propre outil de présentation graphique, nous avons préféré utiliser les travaux effectués dans ce domaine. Notre choix s'est porté vers PAJÉ [KER 00] qui est un outil de visualisation interactif présentant le flot d'exécution d'applications parallèles où un nombre, potentiellement grand, de threads s'exécutent concurremment[4]. PAJÉ est facile à interfacer car il a été conçu comme un outil générique : la structure des flots d'exécution à représenter n'est pas définie dans le logiciel mais elle est décrite dans le fichier de données lui-même. Ainsi l'écriture d'un filtre a suffi pour la conversion de la supertrace au format accepté par le logiciel PAJÉ.

Sur la Figure 4, on peut observer un extrait du déroulement d'un programme multithreadé. Sur cet exemple, la machine a quatre processeurs physiques (il s'agit d'un biprocesseur XEON SMT). L'application observée (nommée `traces`) utilise deux threads noyaux pour ordonnancer ses quatre threads utilisateurs : trois threads sont destinés aux calculs d'un produit de matrice (nommés `calcul1`, `calcul2` et `calcul3`) et le quatrième assure des communications. Les informations sont représentées dans trois zones matérialisant les points de vue thread utilisateur, thread noyau et processeur[5]. Détaillons ces différentes vues :

Informations sur chaque thread noyau :

1) identité du processeur physique utilisé (lorsque ce thread est ordonnancé par le système d'exploitation) ;

2) identité du thread utilisateur qu'il ordonnance (si le thread noyau exécute l'application observée) ;

4. PAJÉ a été initialement conçu pour suivre l'état des threads de niveau utilisateur de l'environnement de programmation distribué ATHAPASCAN [BRI 97].
5. Si tous les threads noyaux et tous les processeurs du système sont représentés, seuls les threads utilisateurs de l'application espionnée sont observables.

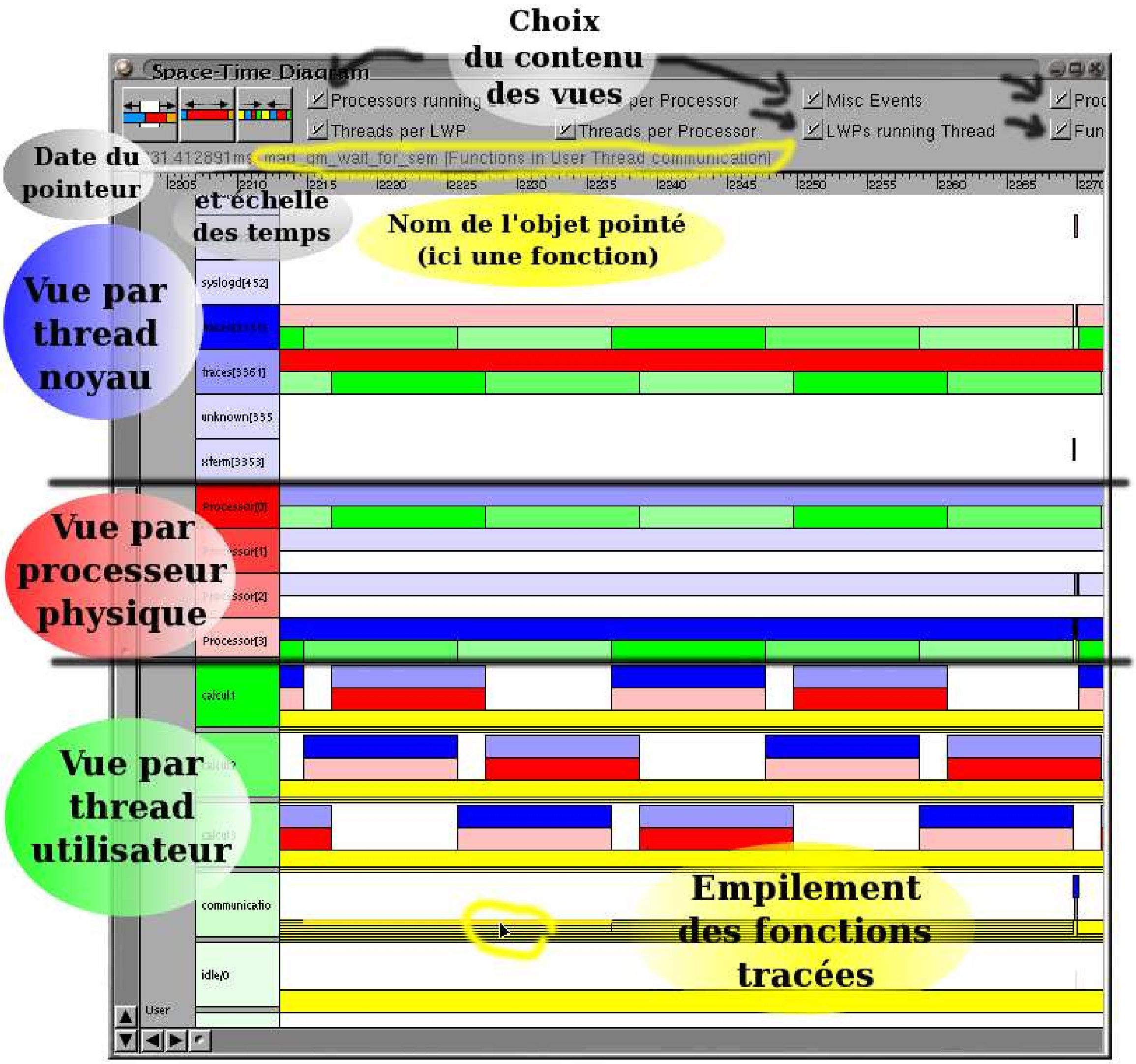

Figure 4. *Visualisation d'une trace avec* PAJÉ

Informations sur chaque processeur :

1) identité du thread noyau exécuté ;

2) identité du thread utilisateur exécuté (s'il s'agit de l'exécution de l'application observée) ;

Informations sur chaque thread utilisateur :

1) identité du thread noyau propulseur (lorsqu'il est sélectionné par l'ordonnanceur de l'application) ;

2) identité du processeur physique lorsqu'il est réellement exécuté (il doit alors être aussi ordonnancé par un thread noyau) ;

3) emboîtement des appels de fonction et des événements tracés dans l'application.

Il est facile d'observer sur la représentation graphique que le système d'exploitation laisse bien la main aux deux threads noyaux de l'application ; ceux-ci s'exécutent de façon continue chacun sur un processeur de la machine (les processeurs 0 et 3). En revanche, pour les threads utilisateurs, on observe que le thread de communication est rarement pris en charge par un thread noyau : il est en fait bloqué sur un sémaphore (information révélée en positionnant la souris sur la dernière fonction appelée par ce thread). Le graphique montre tout aussi clairement que les threads utilisateurs de calcul passent d'un thread noyau à l'autre à chaque quantum. Cela induit un va-et-vient d'un processeur à l'autre, ce qui peut être préjudiciable à l'application quant à l'utilisation du cache processeur par les threads utilisateurs.

Sans ces traces à deux niveaux, il aurait plus été difficile de se rendre compte de la mauvaise utilisation du cache par les threads utilisateurs. On « voit » ainsi que l'ordonnanceur des threads de niveau utilisateur ne tient pas compte des affinités thread utilisateur/thread noyau. Si l'on peut facilement améliorer l'exécution de ce programme[6], le problème de fond, lui, est non trivial : il s'agit d'améliorer le dialogue entre la machine et l'application.

5. Éléments d'implémentation

Afin de garantir la qualité des mesures récoltées, nous avons été confronté à quelques problèmes techniques ; nous évoquons ici les points cruciaux : la mesure du temps, la synchronisation (en particulier sur SMP) et la sauvegarde des événements.

5.1. *L'estampillage des événements et le format de traces*

Rappelons que FKT utilise le registre compteur de cycles qui fournit sur 64 bits le nombre de cycles écoulés depuis la mise sous tension du processeur. Il est très avantageux d'utiliser ce registre pour dater les événements, en effet :

6. Il suffit de faire coopérer non pas trois, mais deux ou quatre threads au calcul du résultat.

– les mesures obtenues via ce registre sont très précises et ce, pour le coût d'*une seule instruction assembleur* à la fois en mode utilisateur comme en mode noyau (contrairement à un appel à une fonction comme `gettimeofday()`);

– cette horloge est parfaitement synchronisée entre les modes utilisateur et noyau, et même entre tous les processeurs d'une machine SMP.

En mode noyau, seuls les 32 bits de poids faibles sont sauvegardés : il y a en effet suffisamment d'événements noyaux (ne serait-ce que les interruptions de l'horloge) pour que l'on soit assuré d'avoir moins de 2^{32} cycles (de l'ordre de la seconde) entre chacun des événements noyaux et de pouvoir reconstituer à coup sûr l'ordonnancement des événements noyau. Cet argument ne tient plus en mode utilisateur, car on ne peut faire aucune hypothèse générale sur la prise de mesure d'un thread utilisateur. Il est donc nécessaire d'enregistrer les 64 bits du registre.

offset	taille	description
0	4	32-bits bas du cycle d'horloge
4	2	pid du thread noyau
6	2	#CPU (en SMP)
8	1	taille de l'entrée (en octet)
9	3	code
12	4	paramètre 1 (si présent)
16	4	paramètre 2 (si présent)
20	4	paramètre 3 (si présent)
24	4	paramètre 4 (si présent)
28	4	paramètre 5 (si présent)

Tableau 1. *Format d'une trace d'un événement noyau*

offset	taille	description
0	8	64-bits du cycle d'horloge
8	1	taille de l'entrée (en octet)
9	3	code
12	4	TID du thread (en SMP)
16	4	paramètre 2 (si présent)
20	4	paramètre 3 (si présent)
24	4	paramètre 4 (si présent)
28	4	paramètre 5 (si présent)

Tableau 2. *Format d'une trace d'un événement utilisateur*

Les tableaux 1 et 2 décrivent le format des événements ; on y précise les champs requis (partie supérieure) où on retrouve l'estampillage et les champs optionnels (partie inférieure). On peut observer que les enregistrements des plus petits événements (ceux sans arguments) n'occupent que 12 ou 16 octets dans la trace, autorisant leur mémorisation en grand nombre.

```
int cmpxchgl(int* ref, int oldvalue, int newvalue) {
        if (*ref != oldvalue)
                return 0;
        *ref=newvalue;
        return 1;
}
```

L'instruction processeur `cmpxchgl` exécute ce code de manière atomique

Figure 5. *Pseudo code de l'instruction `cmpxchgl`*

5.2. *Les mécanismes d'exclusion mutuelle*

Sur une machine SMP, nous devons tenir compte des accès concurrents aux tampons en écriture et adopter une politique efficace pour régler ces conflits. L'utilisation de mécanismes d'exclusion mutuelle de haut niveau (mutex, etc.) est proscrite car bien trop coûteuse. En collaboration avec R. Russell, nous avons résolu ce problème en utilisant l'instruction machine atomique `cmpxchgl` (décrite par la Figure 5). Après avoir obtenu l'adresse du premier emplacement libre dans le tampon, l'adresse du prochain emplacement libre est calculée en fonction de la taille de l'événement à écrire ; cette nouvelle valeur remplace de manière atomique l'ancienne adresse. La procédure est itérée en cas d'échec. Par conséquent, les événements enregistrés dans la trace peuvent être temporellement désordonnés ; charge est laissée à l'utilitaire `supertrace` de réordonner temporellement les événements lors de la fusion des traces noyau et utilisateur.

5.3. *Surcoût induit par l'enregistrement des événements*

L'enregistrement des traces doit être le plus rapide possible afin de réduire au mieux les perturbations engendrées par les mesures. Pour ce faire, les événements estampillés sont écrits dans des tampons (un par trace) alloués lors de l'initialisation.

Fonction/Macro		Nombre de cycles
Macro	`PROF_IN`	260
Appel système	`getpid()`	1900
Fonction standard d'E/S tamponnée	`printf("test")`	672

Tableau 3. *Micro benchmarks*

Le Tableau 3 permet de comparer les différents coûts (appel système, appel de bibliothèque d'entrée/sortie, enregistrement d'un événement). Il apparaît clairement qu'un appel système est beaucoup plus intrusif qu'un appel de fonction. On observe aussi que l'enregistrement d'un événement dans le tampon dédié est deux à trois fois plus efficace que l'écriture d'une chaîne constante par la fonction `printf`.

	temps d'exécution	nombre d'événements enregistrés	(taille)	débit (Mo/s)
Amorce trace	-	0	(788 ko)	-
Programme `Sumtime`				
non profilé	220 ms	-		-
profilé	269 ms (+22%)	119 577	(3,530 Mo)	10,2
profilé (complet)	426 ms (+93%)	837 325	(14 422 Mo)	32
Programme `ProdMat`				
non profilé	8.1 s	-		-
profilé	8.1 s (+0%)	2 618	(0,874 Mo)	0,010
profilé complet	8.2 s (+1%)	695 068	(12,589 Mo)	1,456

Tableau 4. *Performance pour des applications types*

Nous avons ensuite cherché à évaluer le surcoût induit par la prise des traces ainsi que la taille des traces générées. Ces deux quantités sont très fluctuantes et dépendent fortement du niveau d'instrumentation du code. On peut en effet se contenter de tracer uniquement l'ordonnancement des threads sur le système ou bien, à l'extrême, on peut aussi tracer l'ensemble des fonctions de l'application cible ainsi que celles des bibliothèques utilisées, ce qui permet alors d'obtenir une vue très précise du déroulement du programme.

Le Tableau 4 montre le surcoût dû au profilage lors de l'exécution de deux applications types. On compare l'exécution normale (application et bibliothèques compilées sans code d'instrumentation), l'exécution profilée où l'on enregistre que les changements de contexte et l'exécution profilée où on enregistre tous les appels de fonction et tous les appels système. Dans les deux derniers cas, il s'agit du même binaire ; seule une option au lancement permet de choisir le masque actif pour l'enregistrement des événements. Ces mesures ont été obtenues sur une machine sous LINUX 2.6.4 équipé de processeurs XEON SMT cadencé à 2,8 GHz (quatre processeurs exploitées par LINUX). La bibliothèque de threads utilisée est la bibliothèque de thread à deux niveaux MARCEL qui, dans les versions profilées des programmes est entièrement instrumentée (toutes les fonctions de la bibliothèque sont instrumentées, pas uniquement les changements de contextes).

Une trace débute toujours par une *amorce* contenant des informations sur l'exécutable et le noyau (liste des noms des fonctions, liste des processus actifs, etc.) ce qui permet de relire cette trace sans avoir besoin du noyau et/ou de l'application pour retrouver le nom des événements enregistrés. La taille de cette amorce est donnée dans la première ligne du tableau. On a ensuite, sur chaque ligne, le temps d'exécution du programme, le nombre d'événements enregistrés et la taille de la trace générée, et enfin le débit en écriture dans le fichier de trace au cours de l'exécution de l'application.

La première application (`Sumtime`) est un test intensif de la bibliothèque MARCEL : elle crée, synchronise et détruit $2n - 1$ threads en sommant les entiers de 1 à n et ce, de façon dichotomique. Pour calculer une somme des entiers de $[a, b]$, un thread va en créer deux autres qui traiteront chacun une moitié de l'intervalle, il attend ensuite la terminaison de ses fils et renvoie à son tour la somme des deux entiers qu'il a obtenu. Ici, beaucoup d'événements sont enregistrés, y compris lorsqu'on se contente d'enregistrer uniquement les changements de contextes car les threads se synchronisent (donc se bloquent et passent la main) en permanence. Lorsque l'on n'enregistre que les changements de contexte, on observe une pénalité de 22%. On a alors le déroulement exact de l'ordonnancement des threads de l'application. Si l'on désire connaître en plus dans quelle fonction ils se trouvent à tout instant, la pénalité passe à 93%. Ces chiffres élevés s'expliquent par le fait que l'application n'a pas de code de calcul propre : toutes ses fonctions sont très courtes. L'application génère par conséquent un nombre d'événements très important. Il s'agit là d'un pire cas pour notre environnement.

Le second programme (`ProdMat`), que l'on a déjà étudié à l'aide de PAJÉ, est un calcul de multiplication de matrices : le domaine de calcul est divisé en un nombre prédéterminé de sous domaines, chacun pris en charge par un thread. Le déroulement interne des boucles de calcul ne génère pas d'événement particulier, on désire seulement recueillir l'ordonnancement des threads et les temps de calcul de chacune de leurs tâches. On observe alors que le surcoût dû à l'enregistrement d'événements devient négligeable devant les autres coûts de l'application : les perturbations dues aux démons sur le système engendrent des variations du temps d'exécution plus importantes que celles dues à l'instrumentation. Ce type d'observation est plus représentatif de ce que l'on souhaite observer lorsque l'on veut analyser une application multithreadée. Et rien n'empêche d'avoir des portions de code instrumentées de façon plus complète que d'autres, ou encore de modifier le masque des événements enregistrés une fois que l'on a déjà ciblé en partie les périodes les plus intéressantes ou les plus critiques.

6. Conclusion

Les ordonnanceurs multi-niveaux permettent de tirer pleinement partie des architectures de type SMP, deux threads d'un même processus pouvant s'exécuter réellement en parallèle, tout en conservant une grande efficacité sur le plan de la gestion des threads (création, terminaison, etc.) Ce progrès complexifie notablement le recueil et l'analyse des performances des applications multithreadées, analyse cruciale dans un contexte hautes performances. Les logiciels que nous avons développés constituent une réponse à ce besoin : les outils proposés permettent d'instrumenter rapidement et précisément l'application cible afin d'en analyser finement l'exécution.

Pour bénéficier de cet environnement de profilage, il est nécessaire d'utiliser un système d'exploitation et une bibliothèque de threads adaptés. À cette fin, nous fournissons un patch pour le noyau LINUX sur `Pentium` et un second pour la bibliothèque de threads MARCEL [7].

Notons que les modifications nécessaires pour supporter notre environnement de profilage sont très localisées, les bibliothèques de threads comme NGPT [ABT 02], NPTL [DRE 03] ou LINUXTHREAD [LER 96] peuvent donc être adaptées sans difficultés majeures.

Le portage de notre environnement sur d'autres architectures nécessite la présence d'une instruction assembleur semblable à `cmpxchgl` (instruction présente sur tous les processeurs récents) et une référence de temps précise accessible à faible coût. L'architecture de l'ITANIUM répond par exemple parfaitement à ces contraintes et cette adaptation est en cours de réalisation. Il serait également intéressant de généraliser les mesures en utilisant les compteurs de performance et ce, pourquoi pas, au travers d'une API telle que PAPI.

Nous envisageons également d'améliorer le rendu des informations présentées par le logiciel de visualisation PAJÉ, ainsi que d'offrir des filtres de conversion de notre *supertrace* vers d'autres formats afin de bénéficier d'outils de traitement existant.

Nous avons réalisé ce travail dans le cadre de l'ACI GRID RMI consistant à concevoir une plateforme distribuée à objets multi-applications (PADICOTM [DEN 02]) capable d'exploiter de multiples technologies réseaux de manière transparente. Nos travaux vont permettre de comprendre et d'analyser les performances de cet environnement complexe qui permet de faire collaborer des codes MPI, Corba, Java, etc. au sein d'un même exécutable.

Remerciements

Nous remercions Émmanuelle Becker et Clément Ménier qui ont participé au développement de la première version de `FUT` ainsi que Samuel Thibault pour les critiques et les améliorations qu'il a bien voulu apporter à FUT/FKT.

7. Bibliographie

[ABT 02] ABT B., DESAI S., HOWELL D. P., GONZALEZ I. P., MCCRACKEN D., « Next Generation POSIX Threading Project », `http://www-124.ibm.com/developerworks/oss/pthreads/`, 2002.

[AND 91] ANDERSON T., BERSHAD B., LAZOWSKA E., LEVY H., « Scheduler Activations : Efficient kernel support for the user-level managment of parallelism », *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP 91)*, octobre 1991, p. 95–105.

7. Développée dans le cadre de la plateforme de développement multithreadée distribuée PM^2.

[AUM 03] AUMAGE O., MERCIER G., « MPICH/MADIII : a Cluster of Clusters Enabled MPI Implementation », *proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2003)*, Tokyo, Japan, mai 2003, p. 26–33.

[BER 98] BERRENDORF R., ZIEGLER H., « PCL - The Performance Counter Library : A Common Interface to Access Hardware Performance Counters on Microprocessors », rapport, 1998, julie, `http://www.fz-juelich.de/zam/docs/autoren98/berrendorf3.html`.

[BRI 97] BRIAT J., GINZBURG I., PASIN M., PLATEAU B., « Athapascan runtime : efficiency for irregular problems », LENGAUER C. et al., Eds., *EURO-PAR'97 Parallel Processing*, vol. 1300 de *Lect. Notes in Comp. Science*, Springer, août 1997, p. 591-600.

[BRO 00] BROWNE S., DONGARRA J., GARNER N., HO G., MUCCI P., « A Portable Programming Interface for Performance Evaluation on Modern Processors », *The International Journal of High Performance Computing Applications*, vol. 14, n° 3, 2000, p. 189–204.

[DAN 00] DANJEAN V., NAMYST R., RUSSELL R., « Integrating Kernel Activations in a Multithreaded Runtime System on Linux », *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, Lect. Notes in Comp. Science, Cancun, Mexico, mai 2000, Springer-Verlag.

[DAN 02] DANJEAN V., « Réactivité aux événements d'entrées/sorties dans les environnements multithreads », *Actes des Rencontres francophones du parallélisme (RenPar 14)*, Hammamet, Tunisie, avril 2002.

[DEN 02] DENIS A., PÉREZ C., PRIOL T., « PadicoTM : An Open Integration Framework for Communication Middleware and Runtimes », *IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, mai 2002, IEEE Computer Society, p. 144–151.

[DRE 03] DREPPER U., MOLNAR I., « The Native POSIX Thread Library for Linux », `http://people.redhat.com/drepper/nptl-design.pdf`, janvier 2003.

[ENG 03] ENGELSCHALL R. S., « GNU-Pth », `http://www.gnu.org/software/pth/`, 2003.

[GRA 82] GRAHAM S., KESSLER P., MCKUSICK M., « gprof : A call graph execution profiler », *SIGPLAN Notices*, 1982.

[Int03] Intel, « IA-32 Intel Architecture Software Developer's Manual Volume 3 : System Programming Guide », 2003.

[KER 00] DE KERGOMMEAUX J. C., DE OLIVEIRA STEIN B., « Pajé : an extensible environment for visualizing multi-threaded programs executions », *Proceedings of the 6th International EuroPar Conference*, EuroPar2000, 2000.

[LER 96] LEROY X., « The LinuxThreads library », `http://pauillac.inria.fr/~xleroy/linuxthreads`, 1996.

[MAL 04] MALONY A. D., SHENDE S. S., « Overhead Compensation in Performance Profiling », *Proc. Europar 2004 Conference*, LNCS, 2004, à paraître.

[MOH 03] MOHR B., WOLF F., « KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications », *Proc. of the European Conference on Parallel Computing (EuroPar)*, LNCS 2790, Springer, août 2003, p. 1301-1304.

[MOO 01] MOORE S., CRONK D., LONDON K., DONGARRA J., « Review of Performance Analysis Tools for MPI Parallel Programs », VERLAG S., Ed., *8th European PVM/MPI Users' Group Meeting*, n° 2131 Lect. Notes in Comp. Science, 2001, p. 241–248.

[MUC 02] MUCCI P., « Dynaprof and PAPI : A Tool for Dynamic Runtime Instrumentation and Performance Analysis », ScicomP 6, `http://icl.cs.utk.edu/papi`, août 2002.

[NAM 95] NAMYST R., MÉHAUT J.-F., « PM2 : Parallel Multithreaded Machine. A computing environment for distributed architectures », *Parallel Computing (ParCo '95)*, Elsevier Science Publishers, septembre 1995, p. 279–285.

[Ope99] « OpenMP Fortran Application Program Interface », novembre 1999, `http://www.openmp.org`.

[RUS 01] RUSSELL R. D., CHAVAN M., « Fast Kernel Tracing : a Performance Evaluation Tool for Linux », *Proc. 19th IASTED International Conference on Applied Informatics (AI 2001)*, IASTED, février 2001.

[RUS 02] RUSSELL R. D., « FKT : Fast Kernel Tracing », rapport n° 00-02, 2002, University of New Hampshire.

[SHE 98] SHENDE S., MALONY A., CUNY J., LINDLAN K., BECKMAN P., KARMESIN S., « Portable Profiling and Tracing for Parallel Scientific Applications using C++ », *Proceedings of SPDT'98 : ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, août 1998.

[SHE 01] SHENDE S., « The Role of Instrumentation and Mapping in Performance Measurement », PhD thesis, University of Oregon, août 2001.

[TAM 99] TAMCHES A., MILLER B. P., « Using Dynamic Kernel Instrumentation for Kernel and Application Tuning », *The International Journal of High Performance Computing Applications*, vol. 13, n° 3, 1999, p. 263–276.

[THI 03] THIBAULT S., « Developping a software tool for precise kernel measurements », Master's thesis, University of New Hampshire, 2003.

[WOL 03] WOLF F., MOHR B., « Automatic performance analysis of hybrid MPI/OpenMP applications », *J. Syst. Archit.*, vol. 49, n° 10-11, 2003, p. 421–439, Elsevier North-Holland, Inc.

[YAG 00] YAGHMOUR K., DAGENAIS M. R., « Measuring and Characterizing System Behavior Using Kernel-Level Event Logging », *Proceeding of the 2000 USENIX Annual Technical Conference*, juin 2000.

***Vincent Danjean** a obtenu son doctorat à l'École Normale Supérieur de Lyon en 2004 et est actuellement post-doctorant à la Direction des Applications Militaires du CEA. Ses recherches se focalisent autour l'utilisation des processus légers dans le domaine du calcul hautes performances. Les travaux de Vincent sont intégrés au sein de la bibliothèque portable de processus légers à deux niveaux* MARCEL.

***Pierre-André Wacrenier** a obtenu son doctorat à l'Université de Sciences et Technologies de Lille en 1993 et est actuellement Maître de Conférences à l'Université de Bordeaux 1. Initialement théoriques (théorie des traces, algorithmique répartie), ses activités de recherche concernent maintenant les environnements de supports exécutifs hautes performances pour les architectures parallèles.*