RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Mécanismes de traces efficaces pour programmes multithreadés

Vincent Danjean

LaBRI, Université Bordeaux 1,
351, cours de la Libération
33405 Talence Cedex - France
Vincent.Danjean@labri.fr

Résumé
Actuellement, il est très difficile d'observer et de comprendre finement les performances des environnements et des applications multithreadés, en particulier lorsque la plate-forme de threads utilisée est complexe (ordonnancement multi-niveaux). Nous proposons un environnement permettant d'observer finement 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 utilisé. Les résultats obtenus à l'aide notre plate-forme permettent de mesurer, d'interpréter et d'améliorer les performances des supports d'exécutions et des applications multithreadées.

Mots-clés : Profilage, Application multithreadée, Ordonnancement à deux niveaux, 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 fine de l'exécution de l'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, instrumentation souvent directement intégrée au code source du noyau par le programmeur. Ainsi les outils Linux Trace Toolkit (LTT)[17] et Fast Kernel Trace (FKT)[16, 15] permettent de connaître le comportement précis du noyau Linux. Ces outils permettent une analyse très fine des applications : dans [16], une analyse des performances de la pile TCP est menée afin d'analyser et d'expliquer le manque d'efficacité relatif d'un circuit de calcul des sommes de contrôle.

Le point de vue applicatif, commun à tous les programmeurs, est obtenu par l'instrumentation du programme exécutable et ce souvent au travers d'outils intervenant à la génération de code. Parmi les nombreux outils de profilage, on peut citer les utilitaires GNU `gcov` et `gprof`[12].

On peut regretter que ces outils grand public ne soient destinés qu'aux applications monothreadées, aucun support particulier n'étant prévu pour gérer les threads. Le logiciel Pajé[8] a été développé à cette fin, il permet de suivre l'état des threads de l'environnement de programmation distribué Athapascan[5] ; cependant les données recueillies ne sont pas complètes : seul l'état des threads utilisateurs est observable, indépendamment de l'occupation concrète des processeurs physiques. Ce type d'observation est inadéquat aux architectures de type SMP.

En effet, rappelons qu'on distingue plusieurs familles de bibliothèques de threads suivant que l'ordonnancement est réalisé par le système d'exploitation (on parle alors de *threads noyaux* comme ceux de la bibliothèque standard LinuxThread [13]) ou directement par l'application (*thread utilisateur* comme ceux de la bibliothèque GNU-Pth[11]). Les threads utilisateurs permettent l'obtention de meilleures performances et/ou une plus grande flexibilité, cependant, le système d'exploitation n'étant plus sollicité, les threads utilisateurs ne permettent pas d'exploiter pleinement les machines multiprocesseurs (SMP). Afin de remédier à ce défaut, 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 (ou LWP) 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[2], ou bien être gérés en partie ou totalement par le système d'exploitation (threads du système d'exploitation Solaris ou utilisation de mécanisme dérivant des *scheduler activations*[3, 7]).
Ainsi, l'ordonnancement à deux niveaux des threads d'une application sur une machine multiprocesseur dépend de nombreux facteurs, elle est en pratique imprévisible, et il est difficile de recueillir des informations concernant l'ordonnancement des threads d'une exécution donnée sans support particulier. Plus difficile encore est de mesurer, 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[1], PadicoTM[9], MPICH/Madeleine[4]) 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. En effet, non seulement le surcoût doit être faible afin de pouvoir utiliser les informations produites pour analyser les performances des diverses parties du programme analysé, mais en plus il ne faut 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 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 plate-formes complexes (SMP avec bibliothèque de threads à deux niveaux). Pour cela, 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).

Dans ce papier, après avoir justifié l'utilisation d'une trace noyau et d'une trace utilisateur pour observer l'exécution des applications multithreadées, nous présentons notre environnement de profilage dédié aux applications multithreadées et détaillons, en quelques points, les techniques que nous avons mises en oeuvre pour garantir la qualité des mesures obtenues.

2. Des traces noyaux à un environnement de profilage multithread

2.1. Les Fast Kernel Traces

Pour récupérer les informations liées au système d'exploitation, nous avons utilisé la bibliothèque FKT. Cet outil permet de générer des traces afin de récolter des informations sur le déroulement du flot d'exécution des différents threads du noyau. Chaque événement est daté avec une grande précision grâce au registre *compteur de cycle* des Pentium. Le principe de fonctionnement consiste à enregistrer dans un tampon les événements générés par l'écriture dans le code source de quelques macros aux points clés. L'instrumentation du noyau peut très facilement être étendue ou réduite en ajoutant ou enlevant ces macros dans le code. On a bien ici un système de trace peu intrusif, mais il ne concerne en l'état que des événements noyau. Il devra être complété ou étendu pour atteindre notre objectif.
Le lecteur pourra se référer à [15] pour une description et une analyse complète de ce système.

[1] La bibliothèque de threads Marcel [6] de l'environnement de programmation multithreadé distribué PM^2 [14] permet un ordonnancement purement utilisateur ou à deux niveaux au choix, l'utilisation de threads utilisateurs pour l'application permettant de faire migrer les threads d'une machine à une autre.

2.2. 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. Avec une bibliothèque de threads utilisateurs à deux niveaux, le noyau ne connaît pas l'existence des threads utilisateurs de l'application. Il est donc dans l'incapacité de donner leur ordonnancement au sein des threads noyaux fournis à l'application.

Les informations de l'application seules ne suffisent pas. En effet, imaginons un programme avec quelques threads utilisateurs s'exécutant au sein de 3 threads noyaux sur une machine bi-processeurs. Sans information du noyau, on peut certes déterminer quel thread noyau exécute quel thread utilisateur (il suffit de connaître le premier thread utilisateur sur chaque thread noyau puis de suivre les changements de contexte utilisateur), cependant on ne peut pas déterminer le processeur exécutant ces threads utilisateurs. Et plus important encore, on ne connaît pas les dates où le système d'exploitation préempte au moins un des trois threads noyaux pour faire tourner les autres. Ainsi, si un thread utilisateur est préempté parce que le thread noyau sous-jacent l'est, aucune information dans l'espace utilisateur ne nous fournit cette information. Il serait alors impossible de connaître par exemple le temps d'occupation exacte des CPU pour un thread utilisateur donné.

Demander les informations manquantes lors des événements est trop coûteux. On pourrait imaginer que, lors de l'enregistrement d'un événement, l'application demande au noyau sur quel processeur le thread se trouve, ou bien que l'application informe le noyau de ses changements de contexte utilisateur (tout cela avec des appels systèmes dédiés). Cependant, cette solution est beaucoup trop intrusive : outre le temps d'exécution prohibitif d'un appel système (cf. les micro benchmarks de la section 4.3), cela aurait également pour effet de fournir à l'ordonnanceur l'occasion de prendre des décisions d'ordonnancement bien plus souvent que lors d'une exécution normale. Par ailleurs, l'utilisation par le noyau d'upcalls (grâce à des signaux par exemple) pour transmettre ses événements à l'application serait également beaucoup trop coûteuse.

2.3. Notre proposition

La solution retenue est de laisser FKT s'occuper 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.

Sur le plan pratique, FUT est vraiment semblable à FKT à la fois dans son principe de fonctionnement et dans son utilisation, on pourra donc se référer à [15] pour une description précise de 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 4.1.

3. 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.

3.1. 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 (choix à la compilation) de l'environnement PM^2. En appliquant le patch noyau pour avoir le support de FKT, on obtient un ensemble de points déjà instrumentés. Ainsi, on génère des événements à chaque changement de contexte bien évidemment, mais aussi à chaque entrée et sortie d'interruption matérielle (IRQ, etc.) mais aussi logicielle (appels systèmes, etc.). Il est aisé de rajouter de nouveaux points d'instrumentation grâce aux macros fournies si l'on désire profiler quelques points plus particuliers du noyau.

La bibliothèque de threads Marcel a aussi été instrumentée. On obtient donc des événements pour chaque changement de contexte utilisateur et chaque création/destruction de thread noyau (ces événements sont nécessaires pour reconstruire l'ordonnancement global), mais aussi pour toutes les fonctions de la bibliothèque. Ainsi, 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.

3.2. Outils d'instrumentation

L'enregistrement d'événements dans les traces (noyau | utilisateur) repose principalement sur l'utilisation des macros `(FKT|FUT)_PROBEx()` qui possèdent un argument `code` typant l'événement et jusqu'à `x`(≤ 5) arguments supplémentaires (de type `int`). Quelques types d'événements sont prédéfinis, comme ceux marquant les changements de contexte ou bien le début ou la fin des threads, les autres codes étant à la disposition du programmeur.

Des fonctions sont aussi disponibles pour démarrer/arrêter le profilage (initialisation des tampons / écriture des tampons sur fichiers) et démarrer/arrêter la collecte d'événements afin de minimiser la taille des tampons, cependant les changements de contexte ont besoin d'être enregistrés tout au long du profilage.

Notons aussi que nous avons développé un outil de pré-traitement pour l'instrumentation des programmes applicatifs, celui-ci réalise l'attribution automatique d'un code d'entrée et de sortie de chaque fonction. L'utilisateur n'a qu'à utiliser les macros `PROF_IN()` et `PROF_OUT()` au début et à la fin des fonctions qu'il veut profiler comme l'avons fait pour profiler les fonctions de Marcel.

3.3. Analyse des traces

Une fois les traces utilisateur et noyau générées, elles sont fusionnées au moyen d'un outil appelé `supertrace`; celui-ci génère une *supertrace* dans laquelle chaque événement est rattaché à son thread utilisateur, à son thread noyau et au processeur qui l'a exécuté. Dans cette supertrace, les événements sont classés par date. Bien évidemment, certains champs de ces enregistrements peuvent rester vides si l'information n'est pas pertinente. Par exemple, aucun thread utilisateur ne peut être associé à un événement noyau dans une routine d'interruption.

L'analyse de cette supertrace peut être menée par un utilitaire en ligne de commande, appelé `sigmund`. Cet utilitaire 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, sélectionner les événements produits entre deux dates ou durant l'exécution d'une fonction spécifiée, etc.

Des informations telle 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.

3.4. Exemples

La Figure 1 montre comment instrumenter une fonction dans le code de l'application à l'aide des macros `PROF_IN()/PROF_OUT()` pour qu'elle génère un événement à chaque entrée et sortie. Une solution exploitant l'option `-finstrument-functions` de `gcc` est en cours d'évaluation. Avec cette option, il n'y a plus besoin de modifier le code, `gcc` insère automatiquement un appel de fonction au début et à la fin de chaque fonction compilée. L'inconvénient est que cela génère un appel de fonction supplémentaire par rapport à une macro.

La macro `PROF_PROBE2(code, arg1, arg2)` permet de générer un événement dans la trace dont le code sera `code`. Cet événement aura deux arguments qui lui seront associés (`arg1` et `arg2`).

La Figure 2 montre deux exemples d'informations fournies par notre environnement. Le programme instrumenté est un programme multithreadé tournant sur une machine SMP à 4 processeurs. La bibliothèque de thread utilisée est une bibliothèque à 2 niveaux utilisant 4 threads noyaux (LWP) 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 interrogation sert à connaître les événements de la trace relatifs aux changements de

```
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;
}
```

FIG. 1 – Exemple d'instrumentation de code

```
$> 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
```

FIG. 2 – Interrogation de la supertrace avec sigmund

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 (si le LWP appartient à l'application profilée) ;

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 15 a été ordonnancé par le LWP 7137 sur le processeur 1 pendant un certain temps. Puis, il a été ordonnancé par un autre LWP (le 7136) sur le processeur 0. Ce LWP a été préempté par le noyau (qui a donné la main à une autre application) et réordonnancé sur le processeur 2 un peu plus tard. Le thread utilisateur a ensuite passé la main.

Grâce à ce genre d'analyse, on peut voir quel a été l'affinité d'un thread particulier avec les processeurs physiques (c'est important pour les caches, en particulier pour les machines NUMA). Sur cet exemple, on voit que le thread 15 n'a pas eu d'affinité marquée avec un LWP particulier et qu'un LWP lui-même a été changé de processeur par le système.

La seconde interrogation permet de connaître une information calculée à partir des informations des traces. En l'occurrence, `sigmund` nous permet de connaître le temps CPU total consommé par un thread

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)

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)

Format d'une trace d'un événement utilisateur

TAB. 1 – Format des enregistrements des traces en mémoire

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

FIG. 3 – Pseudo code de l'instruction `cmpxchgl`

utilisateur particulier (bien que ce thread ait été ordonnancé par plusieurs LWP et sur différents processeurs).

4. Éléments d'implémentation

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

4.1. L'estampillage des événements et le format de traces

Le registre compteur de cycles (*time-stamp counter register*, présent dans les processeurs à partir du Pentium) 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 :

- 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 utilisateurs et noyau, et même entre tous les processeurs d'une machine.

Alors qu'en mode utilisateur il est nécessaire d'enregistrer les 64 bits de ce registre, on n'enregistre pour la trace noyau que les 32 bits de poids faibles : il y a suffisamment d'événements noyaux (interruptions d'horloge régulières) pour que l'on soit assuré d'avoir moins de 2^{32} cycles entre chacun des événements noyaux et de pouvoir reconstituer à coup sûr l'ordonnancement des événements.

Le Tableau 1 décrit 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.

4.2. 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. Nous avons résolu ce problème en utilisant l'instruction machine atomique `cmpxchgl` (décrite par la Figure 3). Après avoir obtenu l'adresse du

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

TAB. 2 – Micro benchmarks

	Temps d'exécution (ms)	
Programme	Sumtime	Work
Instrumenté et profilé	104	1045
Instrumenté mais profilage désactivé	32	1044
Non instrumenté (programme et bibliothèque)	20	1044

TAB. 3 – Performance pour des applications types

premier emplacement libre dans le tampon, l'adresse du prochain emplacement libre est calculé 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 laissé à l'utilitaire `supertrace` de réordonner temporellement les événements lors de la fusion des traces noyau et utilisateur.

4.3. 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, une solution beaucoup plus rapide que l'écriture dans un fichier. Le Tableau 2 permet de comparer les différents coûts (appel système, appel de bibliothèque d'E/S, enregistrement d'un événement). Il apparaît clairement qu'il est inenvisageable d'utiliser des appels systèmes lors de la génération de traces utilisateur si l'on veut que la prise de traces soit peu intrusive. On peut voir aussi que l'enregistrement d'un événement dans le tampon dédié est deux à trois fois plus efficace qu'utiliser les fonctions de la bibliothèque standard pour écrire dans un fichier (en outre, dans notre benchmark, `printf` se contente d'écrire une chaîne constante de manière tamponnée).
Le Tableau 3 montre le surcoût dû à nos traces lors de l'exécution réelle de programmes. On compare les exécutions instrumentées et profilées, les exécutions des programmes avec le code d'instrumentation mais sans activer le profilage (même programme que le précédent mais avec une option au lancement) et enfin une version du programme sans code d'instrumentation.
Le premier programme (`Sumtime`) est un test intensif de la bibliothèque Marcel. Il se contente de créer, synchroniser et détruire des threads. Son temps d'exécution dépend directement de l'efficacité des primitives de la bibliothèque Marcel. Lorsqu'on l'instrumente, le surcoût semble important car on enregistre beaucoup d'événements (plus de 500 000 événements dans ce cas). `Sumtime` représente le pire cas pour notre environnement. En revanche, le second programme (`Work`) utilise des threads, mais exécute du calcul à l'intérieur de ceux-ci. Les appels à la bibliothèque de threads sont moins fréquents, mais cette application est plus représentative des applications réelles. On voit alors que le surcoût dû à l'enregistrement d'événements devient négligeable devant les autres coûts de l'application.

5. 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. 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 [2].

Notons que les modifications nécessaires pour supporter notre environnement de profilage sont très localisées, les bibliothèques de threads comme NGPT[2], NPTL[10] ou LinuxThread [13] pourront donc être adaptées.

Le portage de notre environnement sur d'autres architectures nécessite la présence d'une instruction assembleur semblable à `cmpxchgl` (présent sur tous les processeurs actuels de nos jours) et une référence de temps précise accessible à faible coup. L'architecture de l'Itanium (IA-64) répond parfaitement à ces attentes par exemple, et nous comptons réaliser bientôt ce portage.

Nous avons réalisé ce travail dans le cadre de l'ACI Grid RMI consistant à concevoir une plate-forme distribuée à objets multi-applications (PadicoTM[9]) 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 au sein d'un même exécutable du code MPI, Corba, Java, etc.

À plus long terme, nous envisageons de convertir notre supertrace en un format standard afin de bénéficier d'outils de traitement existant.

Remerciements

Nous remercions Émmanuelle Becker et Clément Ménier qui ont développé la première version de `FUT`, `supertrace` et `sigmund` au cours d'un projet.

Bibliographie

1. *OpenMP Fortran Application Program Interface*, November 1999. `http://www.openmp.org`.
2. Bill Abt, Saurabh Desai, David P. Howell, Inaky Perez Gonzalez, and Dave McCracken. Next Generation POSIX Threading Project. http ://www-124.ibm.com/developerworks/oss/pthreads/, 2002.
3. T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations : Efficient kernel support for the user-level managment of parallelism. In *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP 91)*, pages 95–105, October 1991.
4. Olivier Aumage and Guillaume Mercier. MPICH/MADIII : a Cluster of Clusters Enabled MPI Implementation. In *proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2003)*, pages 26–33, Tokyo, Japan, May 2003.
5. J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime : efficiency for irregular problems. In C. Lengauer et al., editor, *EURO-PAR'97 Parallel Processing*, volume 1300 of *LNCS*, pages 591–600. Springer, August 1997.
6. Vincent Danjean. Réactivité aux événements d'entrées/sorties dans les environnements multithreads. In *Actes des Rencontres francophones du parallélisme (RenPar 14)*, Hammamet (Tunisie), April 2002.
7. Vincent Danjean, Raymond Namyst, and Robert Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, Lect. Notes in Comp. Science, Cancun, Mexico, May 2000. Springer-Verlag.
8. J. Chassin de Kergommeaux and B. de Oliveira Stein. Pajé : an extensible environment for visualizing multi-threaded programs executions. In *Proceedings of the 6th International EuroPar Conference*. EuroPar2000, 2000.
9. Alexandre Denis, Christian Pérez, and Thierry Priol. PadicoTM : An Open Integration Framework for Communication Middleware and Runtimes. In *IEEE International Symposium on Cluster Computing and the Grid (CC-Grid2002)*, pages 144–151, Berlin, Germany, May 2002. IEEE Computer Society.
10. Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. http ://people.redhat.com/drepper/nptl-design.pdf, January 2003.
11. Ralf S. Engelschall. GNU-Pth. http ://www.gnu.org/software/pth/, 2003.
12. S. Graham, P. Kessler, and M. McKusick. gprof : A call graph execution profiler. In *SIGPLAN Notices*, 1982.
13. Xavier Leroy. The LinuxThreads library. http ://pauillac.inria.fr/˜xleroy/linuxthreads.
14. Raymond Namyst and Jean-François Méhaut. PM2 : Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.
15. Robert D. Russell. FKT : Fast Kernel Tracing. Technical Report 00-02, University of New Hampshire, 2002.
16. Robert D. Russell and Mrinalini Chavan. Fast Kernel Tracing : a Performance Evaluation Tool for Linux. In *Proc. 19th IASTED International Conference on Applied Informatics (AI 2001)*. IASTED, February 2001.

[2] développée dans le cadre de la plate-forme de développement multithreadée distribuée PM^2

17. Karim Yaghmour and Michel R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proceeding of the 2000 USENIX Annual Technical Conference*, June 2000.