Création d'affinité mémoire pour l'efficacité dans ICARE

Anne-Marie Kermarrec & Christine Morin

IRISA
Campus universitaire de Beaulieu
35042 Rennes Cedex FRANCE
Phone: (33) 99 84 71 00 - Fax (33) 99 84 71 71
email: {akermarr,cmorin}@irisa.fr

1 Introduction

Nous avons mis en œuvre une mémoire virtuelle partagée (MVP) recouvrable appelée ICARE au dessus d'une plate-forme d'expérimentation de systèmes distribués (ASTROLAB [8]). La mise en œuvre d'ICARE repose sur une approche pessimiste de retour arrière [7] qui permet d'éviter la reprise d'une application depuis son état initial en cas de défaillance d'un des éléments du système. En effet, périodiquement, un état cohérent du système est sauvegardé et cet état est restauré en cas de défaillance. ICARE tolère de multiples fautes transitoires et une faute permanente simple. Afin d'améliorer l'efficacité d'ICARE, nous proposons des algorithmes qui permettent de diminuer le nombre de défauts de page à la fois en fonctionnement normal et à la reprise après une défaillance, en contrôlant la réplication des données à l'établissement d'un point de récupération. Les algorithmes proposés créent de l'affinité mémoire, concept issu de celui plus connu d'affinité de cache, entre le contenu de la mémoire d'un nœud et les processus qui s'y exécutent. Notre solution tire parti de la réplication nécessaire à l'implémentation du mécanisme de tolérance aux fautes mis en œuvre dans ICARE pour améliorer sa performance. La conception des solutions repose sur l'observation des résultats obtenus lors d'évaluations préliminaires où plusieurs cas particuliers ont été relevés, pour lesquels les temps d'exécution observés avec ICARE étaient inférieurs à ceux observés avec une MVP standard. La première partie de cet article présente, outre le cadre et les hypothèses de l'étude, les premiers résultats observés. La seconde partie décrit les algorithmes proposés. Le dernier paragraphe conclut.

2 ICARE : une MVP recouvrable

2.1 Cadre de l'étude

Le système que nous considérons est composé d'une interconnexion de nœuds de type *silence sur défaillance* contenant chacun un processeur et un module mémoire. Cette caractéristique assure que les nœuds sains ne sont pas affectés par le nœud défaillant. Nous supposons disposer d'un réseau d'interconnexion fiable et d'un mécanisme efficace de détection d'erreur. L'unité de défaillance dans un tel système est le nœud. Nous avons mis en œuvre un mécanisme de retour arrière dans lequel un point de récupération est sauvegardé régulièrement sur un support stable assurant les propriétés de permanence et de mise à jour atomique des données de récupération [6]. La permanence des données de récupération est assurée par leur sauvegarde sur deux modules mémoire indépendants vis-à-vis des défaillances. La mise en œuvre d'un protocole de validation à deux phases [7] assure la mise à jour atomique d'un point de récupération. Pour assurer la cohérence de l'état du système, nous considérons un point de récupération coordonné global [2]. Ceci assure que l'état du système sauvegardé est cohérent, évitant ainsi tout risque d'effet domino [9].

ICARE a été mis en œuvre en étendant le protocole de cohérence à invalidation sur écriture d'une MVP standard [5]. Dans la version standard de la MVP, une page peut se trouver dans l'un des états suivants : *invalide* (INV), *partagé* (P), lorsque plusieurs exemplaires de la page existent et *exclusif* (EXC) lorsque la page est accédée en écriture et que par conséquent un seul exemplaire de la page existe. Dans ce dernier cas, la page est unique dans le système. Le *propriétaire* d'une page possède l'exemplaire le plus à jour de la page et est chargé de gérer les informations de cohérence concernant cette page, notamment son état et la liste des nœuds en possédant un exemplaire (*liste_des_répliques*).

2.2 Réplication des données dans ICARE

ICARE intègre la gestion des données de récupération et des données courantes[1] au sein du protocole de cohérence. Sa mise en œuvre repose sur l'extension du protocole standard [5]. On introduit deux nouveaux états pour distinguer les pages de récupération des pages courantes et les identifier : *partagé-PR*[2](P-PR) et *invalide-PR* (I-PR). L'état P-PR est relatif aux pages de récupération n'ayant pas été modifiées depuis le dernier point de récupération. Ces pages restent accessibles en lecture en fonctionnement normal et un exemplaire I-PR est créé à la première modification. Une page I-PR est utilisée seulement en cas de défaillance. L'établissement d'un point de récupération consiste à sauvegarder pour chaque page, deux exemplaires de récupération. Cette opération relève de la responsabilité du propriétaire, les deux exemplaires étant respectivement dans l'état P-PR1 (le nœud possédant une page P-PR1 en est le propriétaire) et P-PR2. À l'établissement d'un point de récupération, chaque nœud parcourt sa mémoire. On distingue deux types de traitement selon le nombre d'exemplaires d'une page dans le système.

- **exemplaire unique** : la page locale est transformée en page P-PR1 et une seconde page est créée sur le nœud voisin[3] dans l'état P-PR2.

- **exemplaires multiples** : l'exemplaire local de la page est transformé en P-PR1 et l'un des autres exemplaires existants est *exploité* pour être transformé en P-PR2. Cette solution évite la création systématique de pages et limite la taille des messages échangés.

Lorsqu'une défaillance est détectée, le dernier point de récupération sauvegardé formé de l'ensemble des pages P-PR1, P-PR2 et I-PR est restauré, les autres pages I, P ET EXC devant être invalidées.

1. Celles utilisées pour l'exécution d'une application.
2. PR : point de récupération
3. Nous considérons que les nœuds sont numérotés.

2.3 Mise en œuvre et Résultats

ICARE a été mis en œuvre sur la plate-forme d'expérimentation ASTROLAB composée de PCs actuellement connectés par Ethernet et très prochainement par ATM. Le système d'exploitation d'Astrolab repose sur l'utilisation du micronoyau *Chorus*. Nous avons effectué des mesures préliminaires sur une multiplication de matrices avec une fréquence d'établissement d'un point de récupération toutes les 2 secondes. Les mesures observées ont permis de dégager certains comportements qui sont à l'origine des solutions proposées pour améliorer l'efficacité. Nous avons mesuré, outre le surcoût en temps d'exécution, le nombre de défauts de pages (en lecture et en écriture), le nombre de pages créées et le nombre de pages exploitées. Nous avons également mesuré le nombre de lecture effectuées sur des pages P-PR2, c'est-à-dire des pages créées pour les besoins de la tolérance aux fautes. Ce nombre correspond à la quantité de défauts de pages anticipés. Enfin, nous avons mesuré le nombre d'écriture sur des pages P-PR1 c'est-à-dire des défauts de page introduits par la mise en œuvre de la tolérance aux fautes qui correspondent à des pages EXC, accessibles en écriture, transformées en P-PR1, accessibles seulement en lecture, à l'établissement d'un point de récupération.

Multiplication de matrices 256*256									
Nombre de processeurs		Surcoût	Défauts en écriture	Défauts en lecture	Pages créées	Pages exploitées	Lecture sur pages P-PR2	Écriture sur pages P-PR1
2	1	-20%	320	0	310	0	0	6
	2		64	192	220	0	180	0
4	1	+16%	288	0	290	0	0	2
	2		160	32	102	91	160	1
	3		160	32	33	29	32	3
	4		160	32	28	0	0	4

Multiplication de matrices 512*512								
Nombre de processeurs		Surcoût	Défauts en écriture	Défauts en lecture	Pages créées	Pages exploitées	Lecture sur pages P-PR2	Écriture sur pages P-PR1
2	1	-10%	1280	0	1310	0	0	30
	2		256	768	1051	0	768	32
4	1	-14%	1152	0	1179	0	0	27
	2		128	640	706	79	640	17
	3		128	640	497	79	431	17
	4		128	640	293	0	350	21

Les résultats montrent que dans plusieurs cas, le temps d'exécution avec ICARE est inférieur, (jusqu'à -20% dans l'exécution à 2 processeurs de la multiplication 256*256) à celui observé avec la MVP standard. Ces résultats sont dus à l'utilisation pour le calcul de pages P-PR2 créées pour les besoins de la tolérance aux fautes. Ce phénomène évite jusqu'à 90% des défauts de page et compense le surcoût engendré par la résolution des augmentations d'accès. Afin d'améliorer l'efficacité d'ICARE nous proposons des algorithmes d'établissement de points de récupération, reposant sur la création *d'affinité mémoire*, pour anticiper des défauts de pages en fonctionnement normal et à la reprise après une défaillance. Ces algorithmes reposent sur le contrôle de l'endroit de réplication des données à l'établissement d'un point de récupération et assurent, en outre, que la charge reste bien équilibrée dans le système après une défaillance.

3 Création d'affinité mémoire pour l'efficacité

3.1 Affinité mémoire

Définition 1 (Affinité mémoire) *Il existe de l'affinité entre un module mémoire M d'un nœud N et un processus p si M contient au moins une page utilisée durant l'exécution de p sur N. Le degré d'affinité dépend du nombre de pages contenues dans M et utilisées par p.*

Le concept d'affinité mémoire est issu de celui d'affinité de cache [10] utilisé dans l'ordonnancement des processus dans les multiprocesseurs à mémoire partagée. Cette stratégie consiste, au moment du réordonnancement des processus bloqués, à mesurer l'affinité entre les différents caches et le processus et à le réordonnancer de préférence sur un processeur dont le cache contient déjà un sous-ensemble de son ensemble de travail. Dans ICARE, l'affinité des processus avec les mémoires n'est pas mesurée mais créée et ce, sans surcoût car la réplication est inhérente à la mise en œuvre même du mécanisme de tolérance aux fautes implémenté.

3.2 Amélioration de l'efficacité en fonctionnement normal

Définition 2 (Principe de localité temporelle) *La localité temporelle repose sur la réutilisation de pages par un même nœud au cours de l'exécution d'une application et stipule que des pages accédées récemment ont une forte probabilité d'être à nouveau accédées dans un futur proche par le même nœud [3].*

Le premier algorithme proposé vise à améliorer l'efficacité d'ICARE en anticipant dynamiquement les défauts de pages au moment de l'établissement d'un point de récupération. Notre proposition repose sur la localité temporelle généralement observée dans les accès mémoire des applications parallèles. Afin de bénéficier de cette caractéristique, nous proposons une stratégie de réplication qui permet de créer de l'affinité mémoire entre un processus et un module mémoire au moment de l'établissement d'un point de récupération. Cette stratégie consiste à enregistrer l'historique d'une page afin de créer l'exemplaire P-PR2 d'une page unique dans le système, sur un nœud l'ayant déjà utilisée par le passé (et ayant donc une forte probabilité de l'utiliser à nouveau dans le futur), plutôt que de la créer systématiquement sur le nœud voisin. Afin de conserver l'historique d'une page, c'est-à-dire l'identité des nœuds ayant référencé la page récemment, un champ est ajouté dans le descripteur de page (*liste_histo*). Lorsque qu'un nœud i accède en écriture à une page p dont le propriétaire est le nœud j, i devient le nouveau propriétaire et le champ *liste_histo* de la page p sur le nœud i est construite de la manière suivante :

$$liste\_histo_i(p) = liste\_des\_répliques_j(p) + j.$$

L'algorithme d'établissement d'un point de récupération est décrit à la figure 1.

3.3 Amélioration de l'efficacité à la reprise

3.3.1 Algorithme de contrôle des données privées

Dans la suite de cet article, nous distinguons les pages privées d'un processus relatives à son état et qui sont locales à un nœud, de ses pages partagées. L'algorithme de contrôle de la

```
début {création du nouveau point de reprise}
{pour chaque page p dont le nœud est propriétaire
{
 p.état = partagé-PR1;
  exemplaire de reprise local}
 cas p.état dans {

    partagé : {un autre exemplaire de la page existe déjà}
         nœud = p.liste_des_répliques.premier
         { le premier élément de la liste est choisi}
         exploitation(nœud, p);

    exlusif ou partagé : {la page existe en exemplaire unique}
         Si p.liste_histo ≠ {∅} {la liste d'historique n'est pas vide}
             alors
                 nœud = p.liste_histo.premier
                 {le nœud choisi est le premier élément de la liste}
             sinon
                 nœud= (N + 1)mod (nb_nœud);
                 {le nœud par défaut est le nœud voisin}
         fsi
         réplication(nœud,p);
         { création d'un exemplaire P-PR2 sur nœud }
    partagé-PR1 : skip;
    invalide-PR :
         p.état = I;
 } fcas
} fpour
}fin
```

FIG. 1 – *Point de récupération des données partagées avec contrôle de la réplication pour l'efficacité en fonctionnement normal*

réplication des données privées, figure 2, permet de grouper sur un nœud les pages relatives à l'état d'un même processus. En cas de faute permanente, le nœud dont la mémoire contient l'intégralité des données de récupération privées d'un processus est chargé de sa reprise. Il n'est cependant guère envisageable d'utiliser la stratégie initiale de réplication pour les pages privées (uniques par définition) car une telle solution consisterait à dédier chaque nœud comme *nœud de secours* de son voisin et doublerait la charge de ce dernier en cas de faute permanente du premier nœud. L'algorithme proposé permet de distribuer la responsabilité des processus d'un nœud sur l'ensemble des autres nœuds en vue de maintenir la charge bien équilibrée dans le système à la reprise après une faute permanente. Une structure de données, la *table de localisation*, est ajoutée sur chaque nœud et contient pour chaque processus qui s'y exécute, l'identité du nœud responsable de sa reprise en cas de défaillance du premier nœud.

3.3.2 Algorithme de contrôle des données partagées

Le dernier algorithme, figure 3, permet de créer de l'affinité mémoire entre mémoires et processus afin de réduire le nombre de défauts de pages à la reprise et notamment d'éliminer la période transitoire de rechargement des mémoires à la reprise après une faute permanente. Cet algorithme contrôle la réplication des données partagées à l'établissement d'un point de récupération afin de créer les pages P-PR2 référencées par un processus sur le nœud qui aura la responsabilité de la reprise de ce processus en cas de défaillance. Ainsi, les pages privées d'un processus et les pages partagées qu'il utilise pour son exécution sont sauvegardées sur un même nœud. La *table de localisation* définie au paragraphe précédent est consultée pour connaître l'emplacement des données de récupération d'un processus. Les pages partagées relatives à un processus sont répliquées sur le même nœud que celui contenant les pages privées correspondantes.

```
début {création du nouveau point de reprise sur le nœud N}
{   nœud = voisin (N)
 {le premier nœud sollicité est le nœud voisin}

Pour chaque file d'ordonnancement
{{file des processus prêts, file des processus suspendus, file
des processus bloqués}
 Pour chaque processus Pi
{    Pour chaque page p privée du processus Pi
{
      sauvegarde_locale(p);
      {création de la première copie de reprise localement}

      sauvegarde(p , nœud);
      {création de la seconde copie de reprise}
}fpour
      table_loc.Pi = nœud;
      {mise à jour de la table de localisation}
      nœud = (nœud +1)mod(nb_nœud);
      si nœud = N alors
             nœud = voisin(N)
             {la seconde copie ne doit pas être locale}
      fsi
} fpour
} fpour
} fin
```

Figure 2: Point de récupération des données privées avec contrôle de la réplication

```
début {création du nouveau point de reprise}
{pour chaque page p dont le nœud est propriétaire
{
 sauvegarde_locale(p)
 cas (etat) dans {

    page_multiple: {un autre exemplaire de la page existe
déja}
        exploitation( p);

   page_unique:
        proc = p.desc.process;
        {consultation du champ identificateur de processus
dans le descripteur de pages
         nœud = table_loc.proc;
        {nœud est le nœud sur lequel se trouvent les données
         de reprise privées du processus proc}
        réplication(nœud,p);
        {la seconde copie de reprise de la page p est
sauvegardée sur nœud}
 }fcas
} fpour
}fin
```

Figure 3: Point de récupération des données partagées avec contrôle de la réplication

4 Travaux apparentés et conclusion

Le principe de base des algorithmes proposés dans cet article est de tirer parti de la réplication de données inhérente au mécanisme de retour arrière mis en œuvre dans ICARE pour améliorer l'efficacité en fonctionnement normal et limiter le nombre de défauts de page liés à la reprise après une faute permanente dans le système. Ceci est possible car les données de récupération sont stockées dans les mémoires vives du système. Cette solution qui évite de développer du matériel spécifique ([14, 4]) dédié au stockage des données de récupération assurant leurs propriétés de stabilité, est également adoptée dans [1, 13]. En outre, dans ICARE, les données de récupération peuvent être utilisées comme des données courantes tant qu'elles n'ont pas été modifiées depuis le dernier point de récupération. Les algorithmes proposés dans cet article consistent à contrôler la réplication des données de récupération, nécessaire lorsque les pages existent en un seul exemplaire, de manière à créer de l'affinité mémoire entre les mémoires des nœuds et les processus qui s'y exécutent. La plupart des autres approches associant fonctionnement normal et tolérance aux fautes reposent sur l'exploitation des mécanismes standard pour limiter le coût de la tolérance aux fautes [11, 12]. Dans [11], la réplication des données, mécanisme standard permettant de réduire le coût des accès distants, est utilisée pour limiter le surcoût lié à la reprise. Au contraire, dans ICARE, nous exploitons les mécanismes de tolérance aux fautes afin d'améliorer la performance en fonctionnement normal. Les différentes stratégies de contrôle de la réplication, présentées séparément dans cet article pour des raisons de clarté, peuvent facilement être intégrées au sein du même algorithme d'établissement de point de récupération. Il est toutefois nécessaire de privilégier le contrôle de la réplication pour l'efficacité en fonctionnement normal par rapport à l'efficacité à la reprise. Ces algorithmes sont en cours d'implémentation sur la plate-forme ASTROLAB.

Références

[1] L. Brown and J. Wu. Dynamic snooping in a fault-tolerant distributed shared memory. In *Proc. of the 14th International conference on Distributed Computing Systems*, 1994.

[2] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3 (1):63–75, February 1985.

[3] J. Henessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 1990.

[4] B. Janssens and W.K. Fuchs. Reducing interprocessor dependence in recoverable distributed shared memory. In *Proc. of the 13th Symposium on Reliable Distributed Systems*, October 1994.

[5] A-M Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proc .of the 25th International Symposium on Fault Tolerant Computer Systems*, June 1995. Pasadena, USA.

[6] B. Lampson. *Distributed Systems and Architecture and Implementation*, volume 105 of *Lectures Notes in Computer Science*, chapter Atomic Transactions, pages 246–265. Springer Verlag, 1981.

[7] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*, volume 3 of Dependable Computing and Fault-Tolerant Systems. Springer Verlag, second revised edition, 1990.

[8] G. Muller and M. Banâtre. Proposition d'une architecture pour l'expérimentation de systèmes distribués extensibles. Technical report, INRIA, 1995.

[9] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software engineering*, 1, June 1975.

[10] M. Squillante and E. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. Technical report, IBM Research Division, 1991.

[11] P. Triantafillou. Recovering in large distributed systems with replicated data. In *Proc. of Second International Conference on Parallel and Distributed Information Systems*, January 1993.

[12] T.A. Varvarigou and J.A. Trotter. Distributed reconfiguration of multiprocessor systems. In *Proc. of the 13th Symposium on Reliable Distributed Systems*, October 1994.

[13] T.J. Wilkinson. *Implementing Fault Tolerance in a 64-bit Distributed Operating System*. PhD thesis, City University, London, July 1993.

[14] K.L. Wu and W.K. Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on computers*, 39(4), April 1990.