Chapitre 7 : Analyse

7.1 Un programme comporte 6 fonctions, main, A, B, C, D et E. L'outil gprof calcule le nombre d'appels de chaque fonction, avec leur provenance, et prend à intervalle de temps régulier un échantillon du compteur de programme. Pour un programme, il fournit les informations suivantes sur chaque fonction (nombre d'appels avec leur appelant; nombre d'échantillons du compteur de programme) : main (1 appel, pas d'appelant; 0 échantillon), A (5 appels, main:3, B:2; 4 échantillons), B (5 appels, main:5; 2 échantillons), C (6 appels, A:6; 6 échantillons), D (8 appels, A:5, B:3, 16 échantillons), E (8 appels, B:8; 12 échantillons). Pour chaque fonction, donnez le pourcentage du temps passé dans la fonction elle-même (self) et le pourcentage du temps passé dans la fonction et celles appelées récursivement (self+childs). Sur plusieurs exécutions avec les mêmes entrées (le programme n'utilise pas de variable aléatoire), est-ce que les nombres d'appels et d'échantillons peuvent varier? Quelle est la fiabilité des temps que vous avez calculés (self et self+childs)?

Le pourcentage de temps passé dans chaque fonction est donné par la proportion des échantillons. Le nombre total d'échantillons est de 4 + 2 + 6 + 16 + 12 = 40. Si les échantillons sont pris aux 1ms, ceci veut dire que le programme s'est exécuté pendant 40ms au total. Ceci donne pour chaque fonction : main 0%, A 4/40 = 10%, B 2/40 = 5%, C 6/40 = 15%, D 16/40 = 40%, E 12/40 = 30%. On peut ensuite imputer le temps self+childs de chaque fonction à ses appelants, en commençant par les fonctions qui n'ont pas appelé d'autres fonctions (feuilles de l'arbre d'appel) et pour lesquelles self = self+childs. Ainsi, pour C, D et E, self+childs = self et donne C 15%, D 40%, E 30%. Le temps de C 15% est imputé à A (6/6 appels en provenance de A), celui de D est imputé 5/8 x 40% = 25% à A et 3/8 x 40% = 15% à B, et celui de E 30% est imputé à B. Cela donne pour A childs = 15% (de C) + 25% (de D) = 40% et self+childs = 40% + 10% = 50%. Le temps de A est imputé 3/5 = 30% à main et 2/5 = 20% à B. Pour B on a childs = 20% (de A) + 15% (de D) + 30% (de E) = 65% et self+childs = 65% + 5% = 70%. Le temps de B est imputé entièrement à main, son seul appelant. Cela donne main childs = 30% (de A) + 70% (de B) = 100% et self+childs = 100% + 0% = 100%. On constate sans surprise que main et les fonctions appelées représentent 100% du temps d'exécution. Le nombre d'appels, et leur provenance, est compté explicitement et ne devrait pas changer d'une fois à l'autre pour un programme déterministe avec les mêmes entrées. Le nombre d'échantillons dans chaque fonction peut facilement varier en fonction du comportement en partie aléatoire du système d'exploitation, de l'interaction avec d'autres tâches, des interruptions et des fautes de cache. Si le nombre d'échantillons est grand, (ce qui n'était pas le cas ici pour simplifier les calculs), le résultat devrait tout de même être assez fiable. Il peut toutefois y avoir des cas

pathologiques où les échantillons seront biaisés, par exemple si le programme utilise une minuterie corrélée avec celle d'échantillonnage pour synchroniser des attentes ou l'éxécution de fonctions spécifiques. L'imputation aux appelants pour calculer le self+childs est basée sur l'hypothèse que chaque appel prend le même temps en moyenne, quel que soit l'appelant, ce qui est loin d'être nécessairement le cas et est très peu fiable a priori.

7.2 Un programme comme Oprofile ou Perf est utilisé pour prendre des échantillons du compteur de programme à intervalle de temps régulier. Que peut-on calculer à partir de ces échantillons? Si les échantillons sont plutôt récoltés basés sur des interruptions générées par les compteurs de performance (e.g., à tous les 100000 décomptes), par exemple celui des fautes de cache de données L1, que peut-on calculer? Finalement, on décide de plutôt utiliser un outil comme CPU Profile qui échantillonne à intervalle régulier le contenu de la pile d'appel (la fonction courante et celles dans le chemin d'appel). Lorsqu'un échantillon se répète, ce qui arrive souvent, un facteur multiplicatif est ajouté plutôt que de sauver le même chemin d'appel deux fois dans le fichier de sortie. Voici les échantillons récoltés, calculez le temps passé dans chaque fonction, self et self+childs : (main, A) 3 fois, (main,A,C) 6 fois, (main,A,D) 6 fois, (main,B,A) 1 fois, (main,B) 2 fois, (main,B,D) 10 fois, (main,B,E) 12 fois.

Avec un outil comme Oprofile ou Perf, on peut calculer le temps self. Si la bse d'échantillonnage est plutôt un compteur de performance, on va plutôt compter la veleur self pour cette métrique, ici le nombr de fautes de cache de données L1. Avec un outil comme CPU Profile, pour chaque échantillon, il faut ajouter le facteur multiplicatif au temps self de la fonction au sommet de la pile et le facteur multiplicatif au temps self+childs de chaque fonction dans le chemin d appel. On peut diviser par le nombre d'échantillons, ici 3+6+6+1+2+10+12 = 40, afin de faire le calcul en pourcentage. Ainsi, pour main, il n'est jamais en sommet de pile (self = 0/40 = 0%). Par contre, main est présent dans tous les échantillons (self+childs = 40/40 = 100%). Pour A, en sommet de pile, nous avons self = (3 + 1)/40 = 10%. Autrement, A est présent dans les échantillons, self+childs = (3+6+6+1)/40 = 40%. Pour B, nous avons self = 2/40 = 5% et self+childs = (1+2+10+12)/40 = 62.5%. Pour C, nous avons self = 6/40 = 15% et self+childs = 6/40 = 15%. Pour D nous avons self = (6+10)/40 = 40% et self+childs = (6+10)/40 = 40%. Finalement, pour E nous avons self = 12/40 = 30% et self+childs = 12/40 = 30%. Il est intéressant de comparer ces chiffres avec ceux obtenus à l'exercice précédent. Ces deux analyses pourraient correspondre à la même exécution et néanmoins donner des chiffres différents pour self+childs. Les chiffres pour self sont identiques. Les nombres d'appels seraient les mêmes, s'il s'agit de la même exécution, mais cette information n'est pas disponible ici. Le temps self+childs obtenu ici est aussi fiable que celui pour self, puisqu'il est extrait directement des échantillons. Ainsi, on constate ici que les appels de A à partir de B passent beaucoup moins de temps dans A, et font peu ou pas appel à C ou D, comparés aux appel de A directement à partir du main. Ceci explique

que le temps self+childs pour B est finalement moins élevé que celui estimé à l'exercice précédent (62.5% au lieu de 70%).

7.3 La version usuelle de malloc/new a été remplacée par une version qui, avant d'appeler la fonction usuelle, prend un échantillon de la pile d'appel (chemin d'appel) et annote le nombre d'octets alloués afin de créer un outil semblable à Heap Profile. L'information récoltée est donc une liste de chemins d'appel avec pour chacun le nombre d'octets alloués. Peut-on faire exactement le même traitement que celui réalisé par CPU Profile? Dans ce cas, qu'est-ce que le total pour self et self+childs donnera? Si, afin de réduire le surcoût de la collecte d'information, on ne collecte un échantillon de pile d'appel que pour 1 invocation de malloc/new sur 10, que donnera l'information résultante? On aimerait en plus détecter les objets non libérés ou libérés 2 fois, comment pourrait-on effectuer cette vérification efficacement si on intercepte en plus les appels aux fonctions free/delete?

Si on effectue le même traitement que pour CPU Profile mais avec les octets alloués au lieu du nombre de fois que le chemin d'appel a été trouvé en exécution, cela donnera pour chaque fonction elle-même le nombre d'octets alloués (self) ainsi que cette même métrique pour la fonction et celles appelées récursivement (self+childs). Si on ne conserve l'information que pour 1 appel sur 10 à malloc/new, on peut estimer qu'en moyenne il y aura 10 fois plus d'octets alloués que ce qui est mesuré. On peut donc multiplier les chiffres obtenus par 10, tout en sachant que ces valeurs ne sont pas exactes mais une estimation statistique des octets alloués. Pour détecter les problèmes de libération d'objets, à chaque fois qu'un objet est alloué, on peut noter que l'objet à cette adresse est alloué. Ceci est noté dans une variable associée, dans l'entête de l'objet, un bit dans un vecteur de bits où chaque bit représente l'adresse d'un mot, ou encore une entrée pour l'adresse de l'objet alloué dans une table de hachage. Lors d'une libération de l'espace, on enlève cette note. Si un objet libéré n'a pas de note correspondante, il s'agit d'une libération incorrecte (double libération ou mauvaise adresse). A la fin du programme, s'il reste encore des objets alloués selon l'information notée, alors cela veut dire que tout l'espace n'a pas été libéré.

7.4 Un outil comme Helgrind note tous les appels de synchronisation (e.g., prise et relâchement de verrou, barrière, opération atomique…) ainsi que les accès en mémoire de variables partagées (variables globales ou dans le monceau). Expliquez comment cet outil pourra distinguer les cas problématiques des cas corrects dans les situations suivantes : deux thread incrémentent une même variable i) sans protection, ii) avec une opération atomique, iii) avec un verrou; iv) plusieurs threads calculent chacun une partie différente d'un vecteur, attendent à un rendez-vous global et ensuite chaque thread accède l'ensemble du vecteur en lecture; v) les ajouts à une queue sont protégés par un verrou alors que les recherches ne sont pas protégées par des verrous en utilisant la technique RCU.

En i), si un premier thread écrit la variable, elle devient possédée par ce thread. Lorsque le second thread vient pour l'écrire, on regarde les verrous qu'il détient versus ceux associés à cette variable en mémoire lors du dernier accès. Dans ce cas-ci, puisqu'aucun verrou n'est pris, il n'y a pas d'intersection et un accès non protégé est détecté. En ii), l'opération atomique peut être assimilée à une prise de verrou (verrou spécifique à cette variable), l'accès et le relâchement du verrou, ce qui est équivalent au cas iii). Lors de chaque accès en écriture, on vérifie les verrous pris par le thread et on peut valider qu'à chaque fois au moins un verrou fait partie de l'intersection entre les verrous détenus par le thread et l'intersection de ceux détenus les fois précédentes. En iv), au début, chaque thread accède des variables différentes et il n'y a pas de partage à vérifier. Ensuite, le rendez-vous fait qu'il s'agit d'un segment de temps disjoint pour chaque thread et les accès en écriture précédents ne sont plus pris en compte. Par la suite, les multiples accès en lecture font que les variables seront considérées comme partagées en lecture et cela est tout à fait acceptable. En v), l'outil ne peut facilement savoir qu'un accès en lecture non protégé est en fait correct car RCU mise sur le fait que les accès en mémoire pour un pointeur aligné sont atomiques, sans qu'il soit requis d'appeler explicitement une opération atomique.

7.5 Quel serait un bon outil pour chacune des tâches suivantes : i) trouver une fuite de mémoire dans un programme, ii) détecter de la corruption de mémoire dans un programme, iii) optimiser le temps CPU moyen des fonctions d'un processus, iv) trouver un problème de fautes de cache, v) vérifier si un programme souffre de blocages dans le pipeline d'exécution (e.g., cycles perdus en raison d'une mauvaise prédiction des sauts conditionnels), vi) détecter un problème de course dans un logiciel multi-thread, vii) détecter un problème intermittent de latence indue dans une application répartie, viii) valider qu'une section de code est correcte quel que soit l'ordonnancement possible des instructions.

Pour i), des outils comme Memcheck ou Address Sanitizer instrumentent les appels à malloc/new/free/delete et font toutes les vérifications nécessaires pour détecter facilement ces problèmes. En ii), ces mêmes outils peuvent détecter plusieurs cas de corruption de mémoire en vérifiant si seulement des cases allouées sont accédées et seulement des variables initialisées sont lues. De plus, un espace est laissé entre chaque allocation afin d'aider à la détection des débordements de tableau. Pour iii) optimiser le temps moyen des fonctions d'un processus, les outils comme gprof, Oprofile, Perf ou CPU Profile peuvent faire le travail efficacement. Pour les problèmes de cache iv), un outil basé sur les compteurs de performance, comme Oprofile ou Perf, est très efficace. Les mêmes outils peuvent aussi détecter les blocages dans le pipeline v) en sélectionnant la source adéquate pour les compteurs de performance. Les problèmes de course vi) sont ciblés par les outils helgrind et Thread Sanitizer. Pour détecter un problème intermittent vii), par exemple dans une application répartie, les outils de traçage sont souvent l'ultime recours. Avec un traceur

efficace et de bons outils pour analyser les traces, par exemple LTTng et Trace Compass, il peut être assez facile de trouver et comprendre le problème. Afin de vérifier formellement un modèle viii) en essayant toutes les combinaisons possibles, Spin/Promela peut être un excellent outil en autant que la section à valider soit courte.