Gestion d'un cache

Loris Marchal – CR08 ordonnancement

30 novembre 2011

1 Introduction

On s'intéresse au problème de la gestion d'une mémoire temporaire, couramment appelée cache. Plutôt qu'un cache de page web (qui sont de tailles différentes), on considère des pages de mêmes tailles ; la capacité du cache est donc donnée par le nombre maximal de pages qu'il peut stocker, noté k.

Un utilisateur (ou programme) requiert une séquence de pages. Si la page demandée est dans le cache, il n'y a rien à faire. Sinon, il faut d'abord la charger dans le cache (en évinçant éventuellement une autre page si celui-ci est plein) afin de la fournir à l'utilisateur. Le but est de minimiser le nombre de chargements. En général, la séquences des requêtes n'est pas connue à l'avance, on est donc dans un contexte *online*.

Ce problème est courant dans les systèmes d'exploitation, où la mémoire est paginée (organisée en pages). Les caches ne contiennent qu'un nombre limité de pages, et il faut souvent choisir quelle page éliminer du cache pour charger une nouvelle page. En pratique, la politique utilisée est très souvent LRU pour *Least Recently Used*, qui se souvient pour chaque page du cache de sa dernière occurrence, et évince la page dont la dernière occurrence est la plus ancienne.

2 Ratio de compétitivité

On considère le cas *offline*, c'est-à-dire qu'on connaît à l'avance toute la séquence des requêtes. Dans ce cas, la politique optimale consiste, lorsqu'on doit évincer une page du cache, à choisir celle dont la prochaine occurrence est la plus tardive (résultat proposé par Belady [1], mais preuve plus récente [3]).

On rappelle la définition du facteur de compétitivité d'un algorithme *online*.

Définition 1. *On note $C_A(I)$ le coût de l'algorithme A sur l'entrée I. On dit qu'un algorithme A est c-compétitif s'il existe une constante a telle que pour toute entrée I et tout algorithme B,*

$$C_A(I) \leq c \times C_B(I) + a.$$

En suivant [5], on peut montrer qu'il existe une borne sur le ration de compétitivité d'un algorithme *online*.

Théorème 1. *Tout algorithme* online *déterministe est au plus k-compétitif.*

On va prouver une version plus général de ce résultat. On considère un algorithme A *online* quelconque pour le problème du cache, et l'algorithme optimal ci-dessus, noté OPT. On suppose que A dispose d'un cache de taille k_{A} et OPT d'un cache de taille k_{OPT}, avec $k_{\mathrm{OPT}} \leq k_{\mathrm{A}}$.

Théorème 2. *Pour tout algorithme* online *A, il existe des séquences s telles que*

$$C_A(s) \geq \frac{k_A}{k_A - k_{\mathrm{OPT}} + 1} C_{\mathrm{OPT}}(s).$$

Démonstration. Initialement, les caches sont pleins : l'ensemble des pages initialement dans le cache de A est noté S_A^{init} alors que l'ensemble des pages dans le cache de OPT est noté $S_{\mathrm{OPT}}^{\mathrm{init}}$.

On considère la séquence de pages décrit par ces deux étapes :

1. Les $k_{\mathrm{A}} - k_{\mathrm{OPT}} + 1$ premières pages de la séquence ne sont ni dans S_A^{init} ni dans $S_{\mathrm{OPT}}^{\mathrm{init}}$. On note S_1 l'ensemble de ces pages.
2. Les $k_{\mathrm{OPT}} - 1$ pages suivantes sont définies de la façon suivante : à chaque instant, on choisit une page de $S_{\mathrm{OPT}}^{\mathrm{init}} \cup S_1$ qui n'est pas dans le cache de A.

Pour la deuxième étape, on peut toujours trouver une page de $S_{\mathrm{OPT}}^{\mathrm{init}} \cup S_1$ qui n'est pas dans le cache de A : en effet, on travaille sur $k_{\mathrm{A}} + 1$ pages alors que le cache de A est de taille k_{A} : il y a toujours au moins une page évincée, c'est celle qu'on choisit comme prochaine page accédée.

A fait un chargement pour chaque requête, donc au total $(k_{\mathrm{A}} - k_{\mathrm{OPT}} + 1) + (k_{\mathrm{OPT}} - 1) = k_{\mathrm{A}}$. Au contraire, l'optimal garde à la fin de la première étape uniquement les $k_{\mathrm{OPT}} - 1$ pages de la deuxième étape (plus la dernière page de la première étape). Donc il ne fait un chargement que pour les pages de la première étape, au total $k_{\mathrm{A}} - k_{\mathrm{OPT}} + 1$ chargements.

Donc tout algorithme *online* avec un cache de taille k_{A} est au plus $\frac{k_{\mathrm{A}}}{k_{\mathrm{A}} - k_{\mathrm{OPT}} + 1}$-compétitif par rapport à l'optimal avec un cache de taille k_{OPT}. □

Il se trouve que la politique LRU utilisée en pratique atteint cette borne.

Théorème 3. *Pour tout séquence s,*

$$C_{\mathrm{LRU}}(s) \leq \frac{k_{\mathrm{LRU}}}{k_{\mathrm{LRU}} - k_{\mathrm{OPT}} + 1} C_{\mathrm{OPT}}(s) + k_{\mathrm{OPT}}$$

Démonstration. On considère une sous-séquence t de s requêtes pendant laquelle LRU fait f chargements de pages, avec $f \leq k_{\mathrm{LRU}}$. On appelle p la requête qui précède immédiatement t. LRU et MIN ont tous deux p dans leur cache en commençant t. On considère plusieurs cas :

1. Si LRU fait 2 chargements d'une même page pendant t, c'est que t contient au moins $k_{LRU}+1$ requêtes vers des pages distinctes. Donc MIN fait au moins $k_{LRU}+1-k_{\mathrm{OPT}} \geq f + 1 - k_{\mathrm{OPT}}$ chargements sur t.
2. Si LRU fait un chargement de p pendant t c'est que t contient au moins $k_{LRU} + 1$ requêtes vers des pages distinctes. Idem, MIN fait au moins $k_{LRU} + 1 - k_{\mathrm{OPT}} \geq f + 1 - k_{\mathrm{OPT}}$ chargements sur t.

3. Sinon, LRU fait f chargements de pages distinctes et différentes de p. Comment au début de t, MIN a p dans sa mémoire, il fait au moins $f-(k_{\text{OPT}}-1) = f+1-k_{\text{OPT}}$ chargements pendant t.

On partitionne s en sous-séquences $s_0, s_1, \ldots, s_k$ telles que LRU fait exactement k_{LRU} chargements sur s_i pour $i > 0$, et au plus k_{LRU} chargements sur s_0. Pour $i > 0$, on a

$$\frac{C_{\text{LRU}}(s_i)}{C_{\text{OPT}}(s_i)} \leq \frac{k_{\text{LRU}}}{k_{\text{LRU}} - k_{\text{OPT}} + 1}$$

Pendant s_0, MIN fait au moins $f_0 - k_{\text{OPT}}$ chargements, ce qui conclut la preuve. □

3 Algorithme *online* randomisé

On vient de montrer que tout algorithme *online* avait un ratio de compétitivité d'au moins k pour un cache de taille k. Cependant, ce ratio ne s'applique qu'aux algorithmes déterministes, dont les choix sont prévisibles. On peut concevoir un algorithme non-déterministe (randomisé) qui a un meilleur ratio. On s'intéresse à un problème très proche et plus général, le problème des k serveurs. Soit G un graphe à n sommets, dont les arêtes sont munies de longueurs qui vérifient l'inégalité triangulaire. On dispose de k serveurs qui occupent les sommets du graphe. Étant donnée une séquence de requêtes de serveurs, on doit décider de comment déplacer les serveurs de telle sorte que le sommet de la requête courante soit couvert par un serveur. Le but est de minimiser la distance totale parcourue par les serveurs. Lorsque le graphe est homogène (toutes les distances valent 1), on se ramène au problème initial.

On présente l'algorithme de [2]. Initialement, les sommets couverts par des serveurs sont $1, 2, 3 \ldots, k$. L'algorithme maintient un ensemble de sommets marqués, qui sont initialement les sommets couverts. À chaque requêtes, les marques sont mises à jour, puis les serveurs sont déplacés comme suit :

Marquage Le sommet de la requête est marquée. Si $k+1$ sommets sont marqués, toutes les marques sauf la dernière sont effacées.

Déplacement Si le sommet de la requête n'est pas couvert par un serveur, on choisit aléatoirement un serveur qui est sur un sommet non marqué, et on le déplace sur le sommet de la requête.

Cet algorithme correspond à une forme randomisée de LRU : plutôt que de maintenir une file de serveurs, on les divise en deux files, les marqués (requêtes récentes) et les autres (requêtes moins récentes). Quand on a besoin d'un serveur, on pioche dans la file des serveurs non marqués. Quand cette file est vide, on la remplace par l'autre file, en la mélangeant aléatoirement.

Théorème 4. *L'algorithme précédent, noté M est $2H_k$-compétitif sur un graphe homogène, où H_k est le $k^{\text{ème}}$ nombre harmonique :*

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}.$$

NB : H_k est proche de $\ln(k)$ ($\ln(k+1) \leq H_k \leq \ln(k)+1$), ce qui donne un bien meilleur ratio que pour les algorithmes déterministes.

Démonstration. Soit $\sigma = \sigma(1), \sigma(2), \ldots$ un séquence de requêtes. L'algorithme découpe implicitement cette séquence (sauf quelques requêtes au début) en phases. Une phase commence quand une requête nécessite un chargement et réinitialise l'ensemble des sommets marqués : au début d'une phase, k sommets sont marqués et couverts d'un serveur, et la première requête correspond à un sommet non marqué.

On dit qu'un sommet qu'il est *nouveau* s'il n'était pas présent dans la phase précédente et qu'il n'a pas encore été vu dans la phase courante. Un sommet est *vieux* s'il était présent dans la phase précédente mais n'a pas encore été visité dans la phase courante.

On va montrer que pour tout algorithme A, l'algorithme M ne fait pas plus de $2H_k$ fois le nombre de déplacement que A. On va comparer le nombre de déplacements des algorithmes sur chaque phase. Leur coût dépend de l, le nombre de sommets nouveaux dans la phase.

Sans perte de généralité, on considère que A est paresseux, c'est-à-dire qu'il déplace un serveur uniquement pour couvrir un sommet requis.

On va montrer que le coût de l'algorithme M sur une phase est en moyenne $l/2$. Soit d le nombre de serveurs de A qui ne correspondent pas un serveur de M au début de la phase, et d' ce nombre à la fin de la phase. Soit C_A le coût de A sur cette phase. On a $C_A \geq l - d$ car parmi les l requêtes vers des nouveaux sommets, au plus d sommets sont déjà couverts, vu que M n'en couvre aucun.

De plus, on considère S, l'ensemble des sommets marqués à la fin de la phase, donc couverts par les serveurs de M à cet instant. Il y a d' serveurs de A qui ne couvrent pas des sommets de S. Comme pendant cette phase, seuls les sommets de S ont été requis et que A est paresseux, au moins d' serveurs de A n'ont pas été utilisés pendant cette phase. Les $k - d'$ autres ont dû couvrir les k sommets de S, donc $C_A \geq d'$.

En combinant les deux inégalités :

$$C_A \geq \max(l - d, d') \geq \frac{1}{2}(l - d + d')$$

En sommant sur toutes les phases, les termes d et d' se téléscopent, et on obtient $C_A \geq \frac{1}{2}l$ en moyenne sur chaque phase.

Il nous reste à borner le coût de M durant une phase. Il y a l requêtes vers des nouveaux sommets qui coûtent chacun 1. Il reste $k - l$ requêtes vers des vieux sommets et le coût de chaque sommet est la probabilité qu'il n'y ait pas de serveur sur ce sommet quand il apparaît dans une requête. Cette probabilité dépend du nombre de vieux sommets s et du nombre de sommets nouveaux déjà visités c. L'espérance du coût d'une requête vers un vieux sommet est c/s car....

Le pire cas pour M correspond à l requêtes vers des nouveaux sommets, suivies de $k-l$ requêtes vers des anciens sommets. Le coût de ces dernières requêtes est alors donné par

$$\frac{l}{k} + \frac{l}{k-1} + \frac{l}{k-2} + \cdots \frac{l}{l+1} = l(H_k - H_l)$$

Le coût total de la phase pour M est donc borné par $l(H_k - H_l + 1) \leq lH_k$. □

Cette borne peut être légèrement améliorée : on peut montrer que le ratio de compétitivité de tout algorithme *online* (même randomisé) peut être borné par H_k. Il existe d'ailleurs un algorithme randomisé plus complexe qui atteint cette borne [4].

4 Conclusion

Plusieurs points méritent d'être soulevés :

- Utiliser des algorithmes randomisés permet d'obtenir de meilleurs ratio de compétitivité (l'analyse est un peu différente puisqu'on compare le coût moyen de l'algorithme randomisé, ce qui lui est plus favorable que le pire cas dans le cas déterministe).
- Il y a eu beaucoup d'étude sur ce domaine, à la fois pratiques et théoriques (ces deux domaines sont souvent disjoints).
- L'analyse de compétitivité est un outil intéressant pour juger un algorithmes *online*, mais elle n'est pas toujours suffisante pour donner une idée de l'intérêt pratique d'un algorithme (étude dans le pire cas). D'autres outils ont été proposés pour rapprocher étude théorique et applications de ces algorithmes.

Références

[1] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2) :78–101, 1966.

[2] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel Dominic Sleator, and Neal E. Young. Competitive paging algorithms. *J. Algorithms*, 12(4) :685–699, 1991.

[3] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2) :78–117, 1970.

[4] Lyle A. McGeoch and Daniel Dominic Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6) :816–825, 1991.

[5] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2) :202–208, 1985.