Cache Replacement Policies for Multicore Processors

Avinatan Hassidim[*]

September 16, 2009

Abstract

Almost all of the modern computers use multiple cores, and the number of cores is expected to increase as hardware prices go down, and Moore's law fails to hold. Most of the theoretical algorithmic work so far has focused on the setting where multiple cores are performing the same task. Indeed, one is tempted to assume that when the cores are independent then the current design performs well.

This work infirms this assumption by showing that even when the cores run completely independent tasks, there exist dependencies arising from running on the same chip, and using the same cache. These dependencies cause the standard caching algorithms to underperform. To address the new challenge, we revisit some aspects of the classical caching design.

More specifically, we focus on the page replacement policy of the first cache shared between all the cores (usually the L2 cache). We make the simplifying assumption that since the cores are running independent tasks, they are accessing disjoint memory locations (in particular this means that maintaining coherency is not an issue). We show, that even under this simplifying assumption, the multicore case is fundamentally different then the single core case. In particular

1. LRU performs poorly, even with resource augmentation.
2. The offline version of the caching problem is NP complete.

Any attempt to design an efficient cache for a multicore machine in which the cores may access the same memory has to perform well also in this simpler setting. We provide some intuition to what an efficient solution could look like, by

1. Partly characterizing the offline solution, showing that it is determined by the part of the cache which is devoted to each core at every timestep.
2. Presenting a PTAS for the offline problem, for some range of the parameters.

In the recent years, multicore caching was the subject of extensive experimental research. The conclusions of some of these works are that LRU is inefficient in practice. The heuristics which they propose to replace it are based on dividing the cache between cores, and handling each part independently. Our work can be seen as a theoretical explanation to the results of these experiments.

[*]Massachusetts Institute of Technology, Cambridge, 02139, USA. avinatanh@gmail.com

1 Introduction

Almost all of today's computers use multiple cores, and the number of cores is expected to increase as hardware prices go down [4]. Yet, it is not clear what are the important aspects of this change, and how to design the rest of the computer, hardware and software alike, to incorporate it. In the theory community, there have been several recent works on the subject, as well as a dedicated workshop [20]. There are several ways to model this setting, such as the PRAM model [10], and the Bridging model [21] (see [13] for a survey on the need for a good model and the challenges in finding one). However, all these models focus on the limit where many cores are working together on the same data (e.g. sorting a large array [6]). An implicit assumption in almost all the previous theoretical work is that if the cores are running completely independent tasks, or if they are running algorithms which are inherently parallel (such as brute force search in a large space), then there is not much that needs to be done, and the situation is similar to the single core setting. Our goal in this work is to revisit this assumption.

Replacement Policy for a Single Core. In the single core case, the processor presents the cache with a series of requests for memory locations, where each request appears after the last one has been served. The time to serve each request depends on whether the memory location is in the cache (a cache hit occurs) or not (and then a miss occurs and the location is *fetched* from the main memory). If the cache is full, the new memory location enters the cache, and another memory location gets *evicted.* The exact miss time is not important, and we measure algorithms by the number of misses they perform on a given series of requests. In this case, [2] showed that the optimal (offline) algorithm to choose which location to evict is Furthest in the Future (FITF), which evicts the memory location which will be used in the latest possible time.

The more interesting question is the online version of the problem, in which the algorithm sees the next request only after serving the last one. In this case, the common measure for the performance of an algorithm is its *competitive ratio*, or the number of misses it makes divided by the number an optimal offline algorithm would make, on the worst case sequence. For the caching problem, attaining a good ratio is impossible, as all deterministic caching strategies have a competitive ratio which is at least k, the size of the cache. In the same time, we know that in practice very simple strategies give good results. In their seminal paper on caching, Sleator and Tarjan elucidated this disparity, by introducing *resource augmentation*, which compares the online algorithm using a cache of size k to an offline algorithm which uses a smaller cache [19]. Let Least Recently Used (LRU) be the policy which evicts the page for which his last use was the earliest in time. [19] proved that when LRU uses a cache which is a constant factor bigger than the offline algorithm, its competitive ratio is constant.

Not only does LRU have nice theoretical features, it also works well in practice. Although it is a little complicated to exactly keep track of the exact time in which every memory location in the cache was used, heuristics which approximate LRU's behavior are used in most of the standard caches.

Our Results The main contribution of this work is to initiate the theoretical study of the cache problem in multicore machines running (almost) independent tasks. To do this, we show that new ideas are required for the cache replacement policy even when different cores access a disjoint set of memory locations. The main negative result is

Theorem 3.1 (informal) *For any $\alpha > 1$, the competitive ratio of* LRU *is $\Omega(\tau/\alpha)$ when the offline algorithm gets a cache of size k/α, and τ is equal to the miss time divided by the hit time.*

Moreover, we prove that the offline problem is NP complete.

On the positive side, we illustrate that a weak form of FITF does hold:

Theorem 4.1 (informal) *Given that an offline scheduler evicts a memory location which belongs to a specific core, it should evict the memory location which is used furthest in the future.*

Our two theorems essentially say that given the amount of cache which is dedicated to each core, it is easy to decide which location to evict; in contrast the main challenge is to divide the cache between the cores. We use this result to design a Polynomial Time Approximation Scheme (PTAS), when the number of cores and the ratio between miss time and hit time are constant.

The result about the inefficiency of LRU can be seen as a theoretical explanation to practical experiments, which show that caching is problematic in multicore machines. A more interesting connection to practical work comes from the heuristics which are being studied today. These heuristics are phrased in terms of allocating cache to each core, while in its own allocated share the core performs LRU. This fits well with Theorem 4.1, as LRU is a pretty good approximation of FITF (in theory and in practice), and we know that after allocating a part of the cache to each core then FITF is optimal.

Related work There are several recent results on the design of caches for multicore machines, e.g. [12, 14]. These results propose quite radical designs, and show that they have some desirable properties, such as maintaining coherency, but make no attempt to give a competitive analysis of the runtime.

There is a large body of work on sharing a cache, although in very different models. [3] consider the case of sharing a cache among threads, which can be scheduled to perform different parts of the computation. A different approach was taken by [1], which considers a case in which there are n processes sharing a cache, but if one process makes a miss all the processes stop issuing requests until this process gets served. This assumption is not realistic in our case, but simplifies greatly the design of the algorithms.

A more practical approach was taken by [5], who proposed dynamic OS partitioning of the L2 cache between cores. This was continued in the work of [15], who wrote a simulation to evaluate different strategies to partition the caches between two duo-core processors. Other examples for papers which evaluate different sharing policies are [18] (who use Bloom filters) and [8]. The underlying assumption in these works is that the cache should be partitioned between cores, and each core should manage its own cache.

2 Preliminaries

To state the problem of multicore caching, we introduce some notation. Let $\{1, \ldots, n\}$ denote the cores, and let k be an upper bound on the size of the cache. The input to the caching problem is n lists of requests to memory locations, where we let $r^c_1, \ldots r^c_{m_c}$ denote the requests made by core c. We say that two requests are equal if they ask for the same memory location, and also allow empty requests, $r_j^c = \perp$. This type of request means that the core is using its L1 cache, or performing some computation which doesn't require memory access. Let X_c denote the set of memory locations that core c requests. We have $|X_c| \leq m_c$, and $X_c \cap X_d = \emptyset$ for $c \neq d$.

We say that the n lists of requests can be satisfied in time T, if there exist sets $S_1, \ldots, S_T$ (which describe the content of the cache at each time step), and functions $f_1, \ldots, f_n$ (which describe when each core gets served), such that

1. Size constraint: For every time $1 \leq t \leq T$, $|S_t| \leq k$.
2. Requests served in order: $f_c : \{1, \ldots, m_c\} \mapsto \{1, \ldots, T\}$ is strictly monotone.
3. Requests are served: If r_i^c requests item $x^c \in X_c$ then $x^c \in S_{f_c(i)}$.
4. The cache contains memory locations, or is *fetching* memory locations: For every t, we have
$$S_t \subset \bigcup_c (X_c \cup \tilde{X}_c)$$
where $\tilde{X}_c = \{\tilde{x}^c : \ x^c \in X_c\}$ is a set of formal variables which mean that the cache is currently copying the memory location x^c into some cache cell. We say that x_i^c is *evicted* at time t, if $x_i^c \in S_{t-1}$ but $x_i^c \notin S_t$. We say that x_i^c is *fetched* at time t, if $x_i^c, \tilde{x}_i^c \notin S_{t-1}$ and $\tilde{x}_i^c \in S_t$.
5. Fetching takes time τ: If $x^c \in S_t$ and $x^c \notin S_{t-1}$, then for every $t - \tau \leq j < t$, we have $\tilde{x}^c \in S_j$.
6. No prefetching: For every t, if $\tilde{x}^c \in S_t$ then there exists a request r_i^c for the location x^c such that $f_c(i) \leq t$, and $f_c(i+1) > t$

Some of the design choices, e.g. no prefetching, are necessary for the model to have meaning, and are completely standard, while others require explanation. One such convention is to enable the cache to fetch multiple memory locations at the same time, if they are requested by different cores. Any system which does not satisfy this assignment will have a serious bottleneck as the number of cores grow. Practical designs usually allow a bus which is wide and can pass multiple memory locations[1]. Most of our results carry over to the setting where the bus has some width limitations, but this changes the proofs and complicates the presentation.

An important property of the model, is that if one core makes a request which is a miss, this specific core does not issue any more requests until it gets served, but the rest of the cores continue independently. This means that the interleaving of requests that the algorithms sees in the late stages depends on its actions in the earlier stages. Larger values of τ can cause larger deviations in the request sequences. Due to this dependency we focus on the Makespan, and not on the number of misses; most of the results would hold also when counting the number of misses.

[1]The analogy between the model and the real world is tricky here, as real caches operate with cache lines, which span multiple locations. A bus can be more wide than a memory location, and less wide than a cache line. In this case, the design is based on spacial locality of the memory which the processor uses.

A technical convention that we use is that first the contents of a cache cell is evicted, and τ time units later the cell gets the new data from the memory; in the meantime the cell can not be used. This is not important to the results of the paper, and indeed doesn't change the model by much as long as $k \gg n$.

2.1 Real World Parameters

Having presented the model, it is important to see that it captures some aspect of reality. To do so, we present common values of the parameters, as well as the current trends. Modern computers usually have 2 or 4 cores, and this number is expected to grow [4]. For example, the new Nehalem architecture presented by Intel can work with 2,4,6 or 8 cores [22]. Each core has its own private cache, and all the cores share some high level cache[2]. The time to fetch a memory location grows, as it is fetched from more distant caches. For example, in the Intel Pentium M it takes about 3 cycles to fetch from the level 1 cache, 14 cycles to fetch from $L2$, and 240 cycles to fetch from the main memory [7]. When considering the $L2$ cache replacement policy, this means that the ratio between hit time and miss time is $\tau = 240/14 \approx 17$. This ratio is not expected to change much, as technological improvements affect all layers of the cache.

Finally, the parameter k is not the size of the cache, but rather its *associativity*. When a core needs to read a certain memory location, it is usually inefficient to search the entire cache and check if it is there. Thus, the cache is partitioned into sets, such that each memory location can be stored in a single set (usually determined according to the memory address). The size of this set is called the associativity of the cache. If this set has size 1, we the cache implements direct mapping; if the entire cache is such a set, the cache is fully associative; and in any other case the cache is set associative. Most of the caches are set associative (the TLB cache is sometimes fully associative - but it is very small), where the size of the set grows as we get to higher caches. A typical set size for an $L2$ cache is 24 [7] or 32 [22]. Even for this set-size, the heuristic used in practice is often an approximate LRU, as it is hard to keep track of all the uses. Experimental research about multicore caching shows that increasing the associativity improves the runtime for multicore machines. This requires more sophisticated replacement policies, which would be efficient and easy to implement [18].

3 Lower Bounds

3.1 Inefficiency of LRU

Sleator and Tarjan introduced resource augmentation in the design of online algorithms, as a way to explain the poor worst case behavior of online algorithms, as opposed to their success in practical scenarios. In caching, this amounts to comparing an online algorithm which has a cache of size k, with an offline algorithm which has a cache of size h for $h < k$. Indeed, [19] show that FIRST-IN-FIRST-OUT, or FIFO for short, the strategy which evicts the item which first entered the cache[3], has a competitive ratio of $k/(k-h+1)$. This means that for

[2]Usually either each core has its own L1 cache, and they all share an $L2$ cache, or both the $L1, L2$ are private, and there is a shared $L3$ cache, as in the Nehalem. As long as there are private caches and public caches, the hierarchy does not effect our model and results.

[3]The worst case guarantees of LRU can only be worse than those of FIFO. In this case, they are the same.

any k cache misses made by FIFO, any algorithm which has a cache of size h must make at least $(k-h+1)$ misses. In particular, if the online cache is a constant factor better, we get constant competitive ratio. Unfortunately, when there are multiple cores, the scenario changes dramatically, as the following theorem shows

Theorem 3.1. *For any $\alpha > 1$, the competitive ratio of* FIFO *is $O(\tau/\alpha)$ when $h = k/\alpha$. In particular, if we give* FIFO *a constant factor resource augmentation, the ratio is $O(\tau)$. There is a setting with this ratio with just $\lceil\alpha\rceil + 1$ cores.*

Proof. Let $n = \lceil\alpha\rceil + 1$. Consider the following sequence of requests for core c

$$x_1^c, x_2^c, \ldots x_h^c, \ldots, x_1^c, x_2^c, \ldots x_h^c$$

where the request sequence $x_1^c, x_2^c, \ldots x_h^c$ appears $1+\tau$ times.

The offline algorithm can first serve only the first core, then proceed to the second core, etc. Serving the first h requests of each core takes time $\tau \cdot h$ (as they are all misses). The next τh requests also take time τh, as they are all hits. The total runtime of this solution is $2\tau nh = O(\tau h\alpha)$.

The online algorithm tries to serve all cores together. Similarly to the offline, it misses all the requests in the first cycle. However, for each core c, when the request for x_1^c appears in the second time, x_1^c is no longer in the cache - it was evicted to serve request $k/(\rceil\alpha\lceil+1) < h$. A similar analysis shows that every request is a miss. As the online solution fetches n pages at the same time, it has a total runtime of $\tau \cdot (\tau+1)h = O(\tau^2 h)$, as required. □

This theorem is very strong, as a competitive ratio of τ can be achieved by not using the cache at all (in the classical setting when we count misses for the single core case we implicitly assume that τ is unbounded). Note that when the number of cores is $\delta \cdot \alpha$ for $\delta < 1$, the competitive ratio is at most $1/(1-\delta)$, by just giving each core $k/(\delta\cdot\alpha)$ cache space (which is more than the offline algorithm can distribute between all the cores it needs to serve). We note that weaker results hold for other common heuristics, e.g. the randomized MARK [9].

3.2 NP Completeness of the Offline Problem

In this subsection we show that computing the optimal offline schedule is NP complete, using a reduction from 3-Partition. An instance of 3-Partition consists of n numbers $y_1, \ldots, y_n$. Letting $B = \frac{3}{n}\sum_i y_i$, it is NPC to decide if it is possible to partition them into sets $T_1, \ldots T_{n/3}$ such that $|T_j| = 3$ and $\sum_{y_i \in T_j} y_i = B$ for all j. The problem is NP hard in the strong sense (i.e. it is NP complete even when the inputs are given in unary representation [11]), and we can assume without loss of generality that $B/2 \geq y_i \geq B/4$. Given an instance to 3-Partition, we generate the following caching problem:

1. There are n processors.

2. The cache has size $k = n/3$

3. The miss time τ can be any constant

4. The goal is to see if it is possible to schedule all cores in time $T = B + 3\tau - 3$

Core c only requests one page x_1^c, and it requires is y_c times. Formally, the requests of core c are

$$x_1^c, x_1^c, \ldots, x_1^c$$

and there are y_c such requests, that is $r_1^c = r_2^c = \ldots = r_{y_c}^c$.

We sketch a proof for the reduction. It is easy to see that each core must miss at least once. Each miss uses a cache line for time τ, for a total time of τn. If it is possible to finish satisfying the requests until time T, every cell of the cache has to be busy at each time step t (either there is a hit to it or it is used to fetch something from the memory). Also, each cell is used to serve exactly three different cores. The cores who use the i'th cell in the cache constitute the set T_i. In a similar manner, a solution to the 3-Partition admits a solution to the caching instance.

4 Properties of the offline solution

In the case of one core, the optimal solution is Furthest-In-The-Future, or FITF. Upon a request, FITF scans the cache, and attaches each memory location inside the cache the time in which it will be needed again (or infinity if it will not be needed again). It then evicts the location for which this time is maximal [2]. A similar theorem holds in the multicore setting:

Theorem 4.1. *Let S be a solution to the caching problem, and suppose that at time t, S evicts x_i^c. Then there exists another solution $\tilde{S}$ which is identical to S until time t, but at time t, $\tilde{S}$ evicts the page in $S_t \cap X_c$ for which the next usage time is maximal. The cost of $\tilde{S}$ is at most that of S.*

In particular, there is an optimal solution which at each step evicts a page which is used furthest in the future for some core.

The proof of this theorem is somewhat long, and is thus deferred to the full version (a sketch appears in the appendix). This is in stark contrast to the single core case, where the proof of this greedy scheme is very simple. To demonstrate the difference between the single core and the multicore case, we consider a generalization of caching called weighted caching [16]. In this generalization, each memory location x_i is associated with a cost $g(x_i)$, which describes how much time it takes to fetch it from the memory[4]. For a single core, a generalization of FITF is optimal: if an optimal solution evicts a memory location x with cost $g(x)$, the memory location evicted will be the one which is used in the furthest future, out of all the memory location with this cost[5]. This property is no longer true for the multicore case, as the following lemma shows

Lemma 4.2. FITF *is not an optimal strategy, when different pages used by the same core have a different fetching cost.*

[4]This model of caching is especially relevant for web browsing [23, 17]. In a multicore scenario, it can be used if different parts of the high level cache is located at different place on the chip, and thus the fetching time is not uniform.

[5]In fact, sometimes a better model is to assume that each memory location has a different fetching cost, and also takes a different amount of cache space. The appropriate generalization of FITF performs well.

Proof. We present an example with $n = 2$, and $k = 5$. For ease of notation, in the proof of this lemma we denote

$$X_1 = \{X, Y, \alpha, \beta, \gamma\}, \qquad X_2 = \{a, b, c, s\}$$

The cost of fetching $a, b, c, s, \alpha, \beta, \gamma$ is $\tau > 18$, and the cost of fetching X, Y is $T > 20\tau + 18$. To simplify notation, when a core asks for the empty page $\perp$ for r times in a row we denote this by $r_\perp$. The sequences of requests are

$$X, Y, 4\tau_\perp, X, \alpha, \beta, \gamma, \alpha, \beta, \gamma, \alpha, \beta, \gamma, 10\tau_{bot}, \alpha, \beta, \gamma, \alpha, \beta, \gamma, \alpha, \beta, \gamma, Y$$

$$a, b, c, 2T_\perp, s, 6\tau_\perp, a, b, c, a, b, c, a, b, c, 10\tau_{bot}, a, b, c, a, b, c, a, b, c, (T - 3\tau - 2)_{bot}$$

It is possible to satisfy all requests in time $3T + 17\tau + 16$, where both cores finish at the same time. However, this requires that the second core misses at most 4 times (which is also the minimum number of times it has to miss). To see that this makespan is possible, note that the first replacement choice is made at time $2T + 3\tau$, when s is requested by the second core. At that point, the cache contains elements $\{X, Y, a, b, c\}$, and the the optimal algorithm evicts X from the cache. Given this eviction, the rest of the replacement policy becomes very simple. Upon seeing a request for X, the optimal algorithm evicts s at time $2T + 4\tau$. Finally, by the time X is fetched from the main memory, the second core no longer requires the cache (it is in the last stage of requesting the dummy page $T - 3\tau - 2$ times), and therefore a, b, c can be evicted to make room for α, β, γ.

The total time it takes the first core is

$$2T + 4\tau + T + 3\tau + 6 + 10\tau + 9 + 1 = 3T + 17\tau + 16$$

The total time it takes the second core is

$$3\tau + 2T + \tau + 6\tau + 9 + 10\tau + 9 + (T - 3\tau - 2) = 3T + 17\tau + 16$$

Giving a makespan of $3T + 17\tau + 16$ as required.

Consider now solution which makes the FITF decision in time $2T + 3\tau$ upon seeing the request for s. If it evicts a, b or c, the second core will later suffer an extra miss, and satisfying its request will take too long.

If it evicts Y, then it will hit on X. In this case, the requests for the α, β, γ triplets and the requests for the a, b, c triplets will be interleaved, and the makespan will grow.

Note that Theorem 4.1 gives the strong guarantee, that given that we are evicting a memory location which belongs to a specific core, we are evicting the one which is furthest in the future. Thus, to contradict it, we didn't need to consider the case in which the schedule evicts a, b, c, - it's enough to show that evicting X is strictly better than evicting Y. $\square$

Lemma 4.2 shows a complicated scenario, where FITF is not optimal. However, the proof of Theorem 4.1 has to explicitly use the fact that the fetching time of memory locations which belong to the same core is uniform. We note that if each core has a fixed cost for all its possible memory locations, Theorem 4.1 still holds (even if each core has a different fetching cost which is uniform across its pages).

5 The Approximation Algorithm

In this section we present a polynomial time approximation scheme (PTAS), when n, τ are constant. According to theorem 4.1, once the algorithm knows which core should evict a memory location, the decision which location to evict is obvious. Equivalently, each core is facing a cache of varying size, and the algorithm has to decide how much cache to allocate each core at every time step. When considering the algorithms, we will adhere to the latter, and show how much cache each core gets.

We begin by presenting an algorithm when the number of requests each core makes is small. We then extend it to lists of requests of arbitrary length by flushing the cache every once in a while (thus eliminating long term dependencies).

5.1 Short Sequences of Instructions

In this subsection we assume that each core only asks for a small number of locations, and give a PTAS for this case. It is formalized in the next lemma:

Lemma 5.1. *Assume that for some $\alpha > 0$, $k\alpha > m_c$ for all c. If the makespan of the optimal solution is T_k, then for any $\epsilon > 0$, upon input T_k, ϵ, Algorithm 1 finds a solution with makespan $(1+\epsilon)T$. The runtime of the algorithm is exponential in $\tau, \alpha, n, 1/\epsilon$.*

Proof. Let Opt_k denote the optimal solution for a cache of size k, which ends in time T_k. Let $Opt_{k(1-\epsilon/n\tau)}$ be the optimal solution for a cache of size $k(1-\epsilon/n\tau)$, with makespan $T_{k(1-\epsilon/n\tau)}$. To upper bound $T_{k(1-\epsilon/n\tau)}$ in terms of T_k, consider the number of times each cell in the original cache was accessed in Opt_k. Since there are at most nT_k operations, the average cell was accessed at most nT_k/k times. However, if a cell was accessed r times, we can remove it from the cache, by increasing the makespan by at most τr. Removing the cells which were least accessed, gives that

$$T_{k(1-\epsilon/n\tau)} < (1+\epsilon)T_k$$

Let $g_c(t)$ denote the amount of cache that $Opt_{k(1-\epsilon/n\tau)}$ gives core c at time t. It is enough to show that the solution produced by Algorithm 5.1 gives each core c at least $g_c(t)$ cache space at each time step t. As a core can only get one new memory location loaded to its part of the cache every τ time steps, we have that for every time t

$$g_c(t) + 1 \geq g_c(t+\tau)$$

As $\sum_c g_c(t) = k(1-\epsilon/n\tau)$, we also have that g can not decrease too fast, and in particular

$$g_c(t) - n \leq g_c(t+\tau)$$

So g is Lipschitz, with constant n/τ. Let

$$\beta_i^c = \max_{i \cdot r \leq t < (i+1)r} g_c(t)$$

d earlier]

Algorithm 1: SHORT: algorithm for short sequences. Input - lists of requests, T, ϵ.

```
1  $L_1 = L_2 =, \dots, L_n = \emptyset$;
2  $r = \frac{\epsilon k}{n^3}$, set $m = \frac{Tn^3}{\epsilon k} = T/r$;
3  for $c = 0, \dots, n$ do
4      for $(\ell^c_1, \dots \ell^c_m) \in \{0, \dots, k\}^m$ do
5          if core $c$ can finish in time $T$, when $h_c(t) = \ell^c_{t/r}$ then
6              $L_c = L_c \cup \{(\ell^c_1, \dots \ell^c_m)\}$;
7  Let $B$ be the dynamic programming matrix defined as follows:
8  if exists $(\ell^1_1, \dots, \ell^1_m) \in L_1, \dots, (\ell^j_1, \dots, \ell^j_m) \in L_j$ and $\forall 1 \le d \le m$, $\sum_{i=1}^j \ell^i_d < k$ then
9      Set $B[\sum_{i=1}^j \ell^i_1, \dots, \sum_{i=1}^j \ell^i_m, j] = 1$;
10 if $B[i_1, \dots, i_m, n] = 1$ for any tuple $(i_1, \dots, i_m)$ then
11     Return the solution ;
```

where $r = \epsilon k/n^3$ and i is at most to T_k/r . Letting $h_c(t) = \beta^c_{t/r}$, we have $h_c(t) \geq g_c(t)$. Therefore, $\beta^c_i \in L_c$, when we run Algorithm 1, with input T_k, ϵ. Moreover, for every i

$$\begin{aligned}\sum_{c=1}^n \beta^c_i &\leq \sum_{c=1}^n \Big(g_c(i \cdot r) + \frac{rn}{\tau}\Big) = \frac{n^2 r}{\tau} + \sum_{c=1}^n g_c(i \cdot r) \\ &= \frac{\epsilon k}{n\tau} + \sum_{c=1}^n g_c(i \cdot r) \leq \frac{\epsilon k}{n\tau} + k(1 - \frac{\epsilon n}{\tau}) = k\end{aligned}$$

And thus the solution $(\beta 1_1, \dots, \beta^1_m), \dots (\beta n_1, \dots, \beta^n_m)$ survives the consistency check, and some solution is returned.

Analysis of the runtime can be done by following each step. It is deferred to the final version. □

We note that it is not necessary to be precise about the amount of cache given to each core. For example, one can use $O(1/\epsilon)$ different values for β^c_i, instead of k values. Moreover, it is possible to find different approximate solutions. These improvements lead to a better dependency in the parameters, but this is not important for the main algorithm.

5.2 Long Sequences of Requests

After analyzing SHORT, we can use it as a subroutine to handle long sequences of requests, presenting a PTAS for constant n and τ. The approximation is done via dynamic programming. Let B be a matrix, of dimensions $m_1 \times m_2 \dots \times m_n$. We set

$$B[i_1, \dots, i_n] = \min_{p_1, \dots, p_n < 2\tau k/\epsilon} B[i_1 - p_1, \dots, i_n - p_n] + \tau k + COST\left(r^1[i_1 - p_1 : p_1], \dots r^n[i_n - p_n : p_n]\right)$$

where $r^c[i_c - p_c : p_c]$ is a shorthand for $r^c_{i_c - p_c}, \dots, r^c_{i_c}$, and $COST\left(r^1[i_1 - p_1 : p_1], \dots r^n[i_n - p_n : p_n]\right)$ is a $(1+\epsilon/2)$ approximation to the optimal solution for the requests $\left(r^1[i_1 - p_1 : p_1], \dots r^n[i_n - p_n : p_n]\right)$, which is obtained using SHORT. Finally, the output is $B[m_1, \dots m_n]$.

Theorem 5.2. *The dynamic programming procedure returns a $1+\epsilon$ approximation to the makespan of the requests, where the runtime is exponential in $n, \tau, 1/\epsilon$.*

Proof. Let Opt be the optimal solution, with makespan T. Consider a solution $\widetilde{Opt}$, which flushes the cache every $2k\tau/\epsilon$ time steps[6]. We have

$$T_{\widetilde{Opt}} \leq (1+\frac{\epsilon}{2})T_{Opt}$$

where T_{Opt} is the makespan of Opt, and $T_{\widetilde{Opt}}$ is the makespan of $\widetilde{Opt}$.

The solution $\widetilde{Opt}$ is divided into stage, where in each stage, each core performs at most $2k\tau/\epsilon$ instructions. Thus, performing these instructions together will be considered as one of the options when computing the minimum solution in the dynamic programming matrix B. As we loose a multiplicative factor of $(1+\epsilon/2)$ when computing the COST of a short series of requests, the solution given by B satisfies

$$B[m_1, \ldots m_n] \leq (1+\epsilon/2)T_{\widetilde{Opt}} \leq (1+\epsilon)T_{Opt}$$

as required. □

Note that we could improve the runtime guarantee, by uploading n memory cells at the same time each time we flush the cache. Again, this leads to a small improvement in the runtime, but the dependency in n, τ is still exponential, and thus we presented the simpler analysis.

6 Conclusions and Open Questions

Both of the negative results of this work may be theoretically surprising, but they match what is already known to the experimentalists. Theorem 3.1 could have been anticipated by the experiments which show that LRU is not optimal in practice. Theorem 4.1 could have been anticipated by considering the heuristics which perform well - first a division of the cache, and then applying LRU on the part which is given core. There seems to be a difference between LRU and FITF in this context, but since they are very similar from the practical standpoint (or from the theoretical one when introducing resource augmentation), the experimental heuristic and the theorem match.

Two algorithmic questions are obvious: finding approximation algorithms for the offline problem for a wider range of parameters, and finding an online algorithm with a good competitive ratio. Interesting candidates for such an algorithm can come from the heuristics used to partition the cache today (where each core uses LRU on its part).

Even if an algorithm has a nice worst case guarantees, to be actually used it has to be efficient. Moreover, some coherency issues exist even when the cores are running different processes (for example because the operating system is shared). Thus, it is important to perform an analysis of a more realistic system where some conflicts can occur. Finally, it is interesting to identify other areas in the current architecture which are effected by introducing multiple cores, and to model them.

[6]According to the exact definition in the preliminaries $\widetilde{Opt}$ may not be a legal solution, because it violates the technical condition used to prevent prefetching. This point can be overcome by giving a more exact (and less restrictive) definition in the preliminaries, or by a more careful definition of $\widetilde{Opt}$. We keep the notation simple, and ignore this point for the rest of the presentation.

7 Acknowledgments

I would like to thank Alex Andoni, Dror Feitelson, Jeremy Fineman, Piotr Indyk, Edya Ladan Mozes, Charles Leiserson, , Nir Shavit and Bob Tarjan for many stimulating discussions. Of great importance were the technical discussions with Michel Goemans, Aleksander Madry and Jelani Nelson.

References

[1] R.D. Barve, E.F. Grove, and J.S. Vitter. Application-Controlled Paging for a Shared Cache. *SIAM Journal on Computing*, 29(4):1290–1303, 2000.

[2] L.A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM systems journal*, 5(2):78–101, 1966.

[3] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, pages 235–244, 2004.

[4] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. *Technology*, page 1, 2005.

[5] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468. IEEE Computer Society Washington, DC, USA, 2006.

[6] R. Dorrigiv, A. López-Ortiz, and A. Salinger. Optimal speedup on a low-degree multi-core parallel architecture (LoPRAM). In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 185–187. ACM New York, NY, USA, 2008.

[7] U. Drepper. What every programmer should know about memory, 2007.

[8] A. Fedorova, M. Seltzer, and M.D. Smith. Cache-fair thread scheduling for multicore processors. *Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06*, 2006.

[9] 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.

[10] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM New York, NY, USA, 1978.

[11] MR Garey and DS Johnson. " Strong" NP-Completeness Results: Motivation, Examples, and Implications. *Journal of the AssocmUon for Computing Machinery*, 25(3):499–508, 1978.

[12] Z. Guz, I. Keidar, A. Kolodny, and U.C. Weiser. Utilizing shared data in chip multiprocessors with the Nahalal architecture. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 1–10. ACM New York, NY, USA, 2008.

[13] Maurice Herlihy and Victor Luchangco. Distributed computing and the multicore revolution. *SIGACT News*, 39(1):62–72, 2008.

[14] E. Ladan-Mozes and C.E. Leiserson. A consistency architecture for hierarchical shared caches. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 11–22, 2008.

[15] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, pages 367–378, 2008.

[16] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11(2):208–230, 1990.

[17] Evangelos P. Markatos. Main memory caching of web documents. *Computer Networks*, 28(7-11):893–905, 1996.

[18] K. Nikas, M. Horsnell, and J. Garside. An Adaptive Bloom Filter Cache Partitioning Scheme for Multicore Architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on*, pages 25–32, 2008.

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

[20] Vishkin U. Theory and many-cores. *The first workshop on Theory and Multicore computing, adjacent to STOC 2009*, 2009.

[21] L.G. Valiant. A Bridging Model for Multi-core Computing. Springer, SPAA08 and ESA08.

[22] Wikipedia. Intel Nehalem (microarchitecture).

[23] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla, and E.A. Fox. Removal policies in network caches for world-wide web documents. *Computer Communication Review*, 26(4):293–305, 1996.

A Inductive proof for FITF

The following is a sketch of the proof for the optimality of FITF. Assume A throws away y before the first instruction, and B throws away z. Let x_n be the first occurrence of z in the series, and assume wlog that A hits on z (otherwise it should be easy). Also assume wlog that $x_1 = y$. Denote the miss time by T, the hit time is 1, and in any operation x_i which came at time t we

1. Look into the cache to see if x_i is a miss or a hit
2. Compute the state of the cache for the next operation. This is $A(t)$, and we must now have $|A(t)| < k(t)$

If x_i was finished at time t, we denote by $A(t)$ the cache status after we computed the state of the cache. We first increase if required, and then maintain that $|A(t)| \leq k(t)$.

We prove by induction on i that

1. if x_i is finished by B at time $t+1$, then it is finished by A in time $t+T$
2. The cache status after it is finished has the property

$$B(t+1) \cup \{z\} \supset A(t+T)$$

Base case

x_1 (which is y by assumption) ends in time $t = 1$ for B, and T for A. The caches $B(0) = A(0) - \{y\} \cup \{z\}$ (just before the operation $x_1 = y$). Remember that $A(T) \subset A(1) \cup \{y\}$, where $A(1)$ is interpreted again as the status of the cache in the end of the $t = 1$ time. In particular, denoting $\kappa = \min(k(1), |A(0)|)$, we have

$$A(1) = \{\alpha_1, \ldots, \alpha_{\kappa-1}, z\}$$

and also $A(1) \subset A(0) \subset B(0) \cup \{z\}$. We can now therefore throw away elements from $B(1)$, such that we remain in the end with

$$B(1) = \{\alpha_1, \ldots, \alpha_{\kappa-1}, y\}$$

Which satisfies $|B(1)| \leq k(1)$, and gives

$$A(T) \subset A(1) \cup \{y\} \subset B(1) \cup \{z\}$$

as required.

We move to the inductive step. Assume that x_{i-1} was ended by B in time t, and by A in time $t + T - 1$. Also assume that $i < n$, and that $B(t) \cup \{z\} \supset A(t + T - 1)$.

According to the induction hypothesis, A will start taking care of x_i in time $t + T$, and B will start in time $t + 1$. We now need to consider two cases

A.1 x_i is a hit in A or a noop

If x_i is a noop, it takes one unit of time both in A and in B. Therefore, clearly A will also finish in time $t+T$, and B will finish in time $t+1$. We will later consider the status of the cache after the operation.

If x_i is a hit in A, then $x_i \in A(T+t-1) \subset B(t) \cup \{z\}$. But as $i < n$ we know that $x_i \neq z$, and therefore x_i is a hit also on B. Again the action is finished in A in time $t+T$, and in B in time $t+1$, as required.

We now take care of the status of the cache for both operations. If $k(t+1) \geq |B(t)|$, then we can afford to have $B(t+1) = B(t)$, and still $|B(t+1)| < k(t+1)$. As no element entered A in time $t+T$, we have

$$A(t+T) \subset A(t+T-1) \subset B(t) \cup \{z\} = B(t+1) \cup \{z\}$$

where we used the inductive hypothesis.

Finally, we need to consider the case $k(t+1) < |B(t)|$. Denoting $k(t+1) = \kappa$ we have

$$|A(t+T-1)| \leq |A(t+1)| + 1 \leq k(t+1) + 1 = \kappa + 1$$

which is true because in T time steps the size of a cache can grow by at most 1 (because there can be at most one entry). As $z \in A(t)$, we can denote

$$A(t+T-1) = \{\alpha_1, \ldots, \alpha_\kappa, z\}$$

And as $B(t) \cup \{z\} \supset A(t+T-1)$, we can choose to leave in $B(t+1)$ the elements

$$B(t+1) = \{\alpha_1, \ldots, \alpha_\kappa\}$$

This would still give $|B(t+1)| = \kappa = k(t+1)$ so it fits the cache size, and also

$$A(t+T) \subset A(t+T-1) \subset B(t+1) \cup \{z\}$$

as required.

A.2 x_i is a miss in A

This is the most involved scenario. We can assume wlog that x_i is also a miss in B (otherwise - just hit on it and stall).

As A started taking care of x_i at time $t+T$, it finishes in time $t+2T-1$. Similarly, B stated at time $t+1$, and therefore it finishes in time $t+T$. We now need to go over the content of the cache. We need to consider three cases

case 1: For any $T-1 \geq j \geq 1$ we have $k(t+j) \geq |B(t)|$, and $k(t+T) > |B(t)|$. In this case we have $B(t+T) = B(t) \cup x_i$. Size is not an issue here (Throughout stages $t, t+1, \ldots t+T$), and

$$A(t+2T-1) \subset A(t+T-1) \cup \{x_i\} \subset B(t) \cup \{z\} \cup \{x_i\} \subset B(t+T) \cup \{z\}$$

and we are done.

To define cases $2, 3$, we let τ denote the index j which minimizes $\min_{T-1 \geq j \geq 1} k(t+j)$.

case 2: $k(t+\tau) < |B(t)|$, and $k(t+T) \geq k(t+\tau)+1$. In this case, we again use the fact that $|A(t+T)| \leq k(t+\tau)+1$. Denoting $\kappa = k(t+\tau)$, we have $A(t+T) = \{\alpha_1, \dots \alpha_\kappa, z\}$, and therefore we can choose $B(t+T) = \{\alpha_1, \dots \alpha_\kappa, x_i\}$. This means that whenever we needed to throw away elements from time t to time $t+T$ we kept $\{\alpha_1, \dots \alpha_\kappa\}$ and acted arbitrarily on the rest. This is possible, because $k(t+\tau) = \kappa$ is the minimum.

case 3: $k(t+\tau) < |B(t)|$, and $k(t+T) < k(t+\tau)+1$. the same argument holds as in case 2, only that the limiting factor is different.