Effectively Sharing a Cache Among Threads

Guy E. Blelloch*
Carnegie Mellon University
guyb@cs.cmu.edu

Phillip B. Gibbons
Intel Research Pittsburgh
phillip.b.gibbons@intel.com

ABSTRACT

We compare the number of cache misses M_1 for running a computation on a single processor with cache size C_1 to the total number of misses M_p for the same computation when using p processors or threads and a shared cache of size C_p. We show that for any computation, and with an appropriate (greedy) parallel schedule, if $C_p \geq C_1 + pd$ then $M_p \leq M_1$. The depth d of the computation is the length of the critical path of dependences. This gives the perhaps surprising result that for sufficiently parallel computations the shared cache need only be an additive size larger than the single-processor cache, and gives some theoretical justification for designing machines with shared caches.

We model a computation as a DAG and the sequential execution as a depth first schedule of the DAG. The parallel schedule we study is a parallel depth-first schedule (PDF-schedule) based on the sequential one. The schedule is greedy and therefore work-efficient. Our main results assume the *Ideal Cache* model, but we also present results for other more realistic cache models.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous; C.1.2 [**Processor Architecture**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms

Algorithms,Theory

Keywords

Scheduling algorithms, shared cache, multithreaded architectures, chip multiprocessors

*This work was supported in part by the National Science Foundation as part of the Aladdin Center (`www.aladdin.cs.cmu.edu`) under grants CCR-0086093, CCR-0085982, and CCR-0122581.

SPAA'04, June 27–30, 2004, Barcelona, Spain

1. INTRODUCTION

In a shared-cache machine, multiple processors or concurrent threads share a single cache. Such machines include those that support simultaneous multithreading (SMT) [31] or that contain single chip multi-processors (CMP) with a shared cache [15, 14, 4]. Many current processors are either SMT or CMP [30, 19, 18, 27]. When a shared physical memory is used as a cache for disk memory, as is supported by virtual memory, standard shared-memory multiprocessors can also be treated as shared-cache machines.

Previous research has studied the performance of shared caches for independent processes. Much of the research is motivated by the need to analyze cache performance for time-shared processes [28, 2, 29, 25]. In this scenario processes are interleaved on a single processor and must share the processor's cache—a process, for example, can evict the contents of a cache line previously loaded and later needed by another process. Other work considered multiple independent processors sharing a cache [23, 11, 5, 26]. All of this work, however, assumes the processes are independent.

In this paper we are interested in analyzing cache performance when running a single shared-memory computation on a shared-cache machine. In this context, we can take advantage of the potential overlap in memory references among the threads. We compare the number of cache misses for running a computation on a single processor to the total number of cache misses for running the same computation in parallel on p processors or threads that share a single, possibly larger, cache. We show that by using an appropriate schedule, the shared cache need only be *an additive amount* $p \cdot d$ larger than the sequential cache in order to guarantee *no additional misses for any computation.* Here d is the depth of the computation, *i.e.*, the longest critical path of dependences. This result assumes the Ideal Cache model [6], which is fully associative and has the optimal replacement policy. We also consider various more realistic cache models.

To better understand how large $p \cdot d$ is, we note that in practice p is small (*e.g.* 2-4 on current architectures, and up to 256 on research prototypes [20]). Likewise, often the depth is small. Many problems can be solved in polylogarithmic depth. This includes all problems in the complexity class NC, many of which can be solved work efficiently. Moreover, if a computation includes synchronization barriers, the effective depth d is just the maximum length of a critical path of dependences between consecutive barriers. In contrast, shared cache sizes are currently on the order of MBs, and growing.

Cache Model	Shared Cache Size	Miss Bound	Parallel Steps Bound
Shared Ideal Cache	$C_1 + \frac{(p-1)d_p}{\delta+1}$	M_I	$\frac{W_I+M_I\delta}{p} + d_p$
Shared LRU Cache	$2\left(C_1 + \frac{(p-1)d_p}{\delta+1}\right)$	$2M_I$	$\frac{W_I+M_I(m+2\delta)}{p} + d_p$
Shared LRU Cache	$C_1 + \frac{(\alpha+1)(p-1)d_p}{\delta+1}$	$W_L/\alpha + M_L$	$\frac{(1+(m+\delta)/\alpha)W_L+M_L\delta}{p} + d_p$
Shared Seq-LRU Cache	$C_1 + \frac{(p-1)d_p}{\delta+1}$	M_L	$\frac{W_L+M_L\delta}{p} + d_p$

Table 1: Summary of results (notation in Table 2).

G	computation DAG
w	weight function on actions of G
p	number of processors
m	latency for a cache miss
C_1	sequential cache size
C_p	shared cache size
$\mathcal{S}_1$	sequential schedule
$\mathcal{S}'_1$	prefix of $\mathcal{S}_1$
$\mathcal{S}_p$	parallel schedule for p processors or threads
$\mathcal{S}_\ell$	linearized parallel schedule
$\mathcal{S}'_\ell$	prefix of $\mathcal{S}_\ell$
M_1	number of misses with a sequential schedule
M_I	number of misses with Sequential Ideal Cache
M_L	number of misses with Sequential LRU Cache
W_1	work with a sequential schedule
W_I	work with Sequential Ideal Cache
W_L	work with Sequential LRU Cache
W_p	work with a parallel schedule
d_p	depth with a parallel schedule
δ	(≥ 0) number of dummy nodes before a miss
α	(≥ 0) cache size parameter for LRU bounds

Table 2: Notation used in this paper.

As in previous work [10, 7], we analyze computations in terms of their dependence graphs. The dependence graph might unfold dynamically so we do not assume the structure is known ahead of time, but rather just use it as an analysis technique. A standard sequential execution of the computation corresponds to a particular depth-first schedule (1DF-schedule) of the dependence graph. We assume the programmer is free to choose the particular 1DF-schedule to optimize cache performance (*e.g.*, if there is a parallel loop or fork-and-join, the programmer can order the subcalls to minimize cache misses). Our bounds are relative to such a programmer-optimized sequential ordering.

We consider parallel schedules that are prioritized based on a given sequential schedule—*i.e.*, if there are multiple tasks ready to execute at a given time step, the schedule will preferentially pick the tasks that are earliest in the sequential schedule. A parallel schedule based on a 1DF-schedule is called a PDF-schedule [7]. In previous work we described how to maintain a PDF-schedule online for various types of computations [7, 8]. In the case of computations with parallel loops and fork-and-join constructs (ones that lead to series-parallel dependence graphs), maintaining such a schedule is particularly simple [7]. For computations with synchronization variables, it is somewhat more difficult [8].

In this paper, we define various models for shared caching which differ primarily in their replacement policies. Table 1 summarizes the main results in this paper. These results (described in detail in the remainder of the paper) show that over a variety of shared caching models, only a modestly larger shared cache suffices to have a low miss, work-efficient schedule. We also present an asymptotically tight lower bound on the shared cache size needed to incur no extra misses, as well as trade-offs between misses and shared cache size and between work and shared cache size.

```
for k = 0 to (n/b)-1
  for i = 0 to (n/b)-1
    for j = 0 to (n/b)-1
      for ii = i*b to (i+1)*b - 1
        for jj = j*b to (j+1)*b - 1
          for kk = k*b to (k+1)*b - 1
            C[ii,jj] += A[ii,kk] * B[kk,jj]
```

Figure 1: A block matrix multiply for multiplying two $n \times n$ matrices `A` and `B` into a result matrix `C`.

1.1 Example: Block Matrix-Multiply

As an example, consider the block matrix-multiply pseudocode shown in Figure 1 for $n \times n$ matrices using a block size of $b \times b$. All of the six loops in the code can be parallelized. The `i`, `j`, `ii`, and `jj` loops can be parallelized by forking the iterations to run in parallel using a tree of depth $O(\log n)$. In addition to forking, the `k` and `kk` loops require a summation, which can also be done on a tree of depth $O(\log n)$. To reduce overheads, the block size b is selected so that three blocks can reside in the (sequential) cache simultaneously, *i.e.*, $C_1 \approx 3b^2$. This ensures that the working set defined by any instance of the three inner loops of the code fits in the cache. However, a p processor schedule that executes any of the three outer-most loops in parallel has a working set of $2p$ blocks ($2pb^2$ locations). Therefore, in order to avoid significant thrashing, the shared cache must be of size at least $\frac{2}{3}p \cdot C_1$, *i.e.*, $\Omega(p)$ times larger than the sequential cache size.[1] A standard work-stealing scheduler, for example, would tend to schedule the outer loop in parallel.

We note, however, that our bounds (based on a PDF-schedule) are much better. Consider using p processors. If we parallelize all the loops, the depth is $O(\log n)$ and the shared cache need only be of size $C_1 + O(p \log n)$, as compared to the size $\frac{2}{3}pC_1$ needed above. Alternatively, a more coarse-grained parallelization might not parallelize the `k` and `kk` loops, in order to avoid the fine-grained parallel summations. Here, the effective depth of the computation is $O(b + \log n)$ because there is a synchronization barrier after each outer `k` iteration and the inner k iteration has depth $O(b)$. Thus the shared cache need only be of size $C_1 + O(p(b + \log n))$. Intuitively, these improved bounds arise

[1] Note that techniques to improve shared cache performance by modifying the program, *e.g.*, reducing the block size, are beyond the scope of this paper. We are interested in proving general results valid for any program.

because the PDF-schedule tends to schedule the iterations of the innermost loops concurrently, so that the working set is only a few blocks.

1.2 Related Work

As mentioned earlier there has been various work on modeling or experimenting with independent concurrent threads running on a shared cache [28, 2, 29, 23, 11, 5, 25, 26].

Blumofe *et al.* [9] studied a distributed shared-memory model, showing that, for the important class of "fully strict" multithreaded computations, if each processor has its own cache of C_1 pages then the number of cache misses (page faults) is only $O(C_1 pd)$ larger than a sequential execution with C_1 pages. The total cache size is pC_1. Later results showed that the bounds hold for shared-memory models with standard cache policies and that the bounds are tight [1]. None of this work considered shared caches.

There is a large body of literature on the competitive analysis of caching/paging algorithms (see [16] for a recent survey). This work compares online to offline algorithms, different replacement policies, and different cache sizes. Most of this work is not for a shared cache setting. Some of this work (*e.g.*, [5]) studied online algorithms for multiple processes sharing a fixed-sized page cache, but only considered the case where each process executes an independent program.

In earlier work we showed that if a computation with depth d uses M_1 total memory on a single processor it can run with $M_1 + O(pd)$ memory using p processors, by using a parallel schedule $\mathcal{S}_p$ prioritized based on the sequential schedule [7]. This work has been extended in various ways [8, 22, 21, 12]. These results leverage a key lemma from [7] that bounds the number of tasks that can be executing "prematurely" during any step of $\mathcal{S}_p$, when compared to the ordering of tasks in the sequential schedule. We note, however, that these results on bounding total memory do not immediately carry over to the context of shared caches. In particular the results rely on the fact that the maximum number of premature tasks at any point during the computation is less than pd, and therefore the maximum additional memory needed is $O(pd)$ (under certain assumptions).

In the context of shared caches, however, any premature task can cause a cache miss that was not present in the sequential computation, even when the shared cache has $O(pd)$ extra space. Therefore the number of additional cache misses could potentially be proportional to the total number of tasks that are scheduled prematurely across the entire computation, which has no bound with respect to pd. Furthermore even if a premature task does not result in a miss, its early execution may change the replacement priority of its accessed block resulting in some later task to miss. Our results for the Ideal Cache model rely on showing that every additional miss at one task is compensated by an addition hit at another task.

2. SCHEDULES AND MODELS

We begin by defining the models and other terminology used in this paper.

2.1 Computation DAGs and Schedules

Following standard terminology, we model a computation as a directed acyclic graph (DAG) G. Each node in G represents the execution of a single non-preemptable task, called an *action*. The edges represent any ordering dependences among the actions—a path from a node u to a node v implies that the action for u must complete before the action for v starts. We also assume an integer weight function $w(a)$ over the actions which indicates the number of time steps taken by the action. Note that we will use node and action interchangeably in this paper.

We assume the computation $C = (G, w)$ generated by a program may depend on the input and is revealed online, *i.e.*, we do not assume it is known before running the program. We assume that w may also depend on the particular schedule (*e.g.*, to account for the difference in time between a cache hit and a cache miss), but that G depends only on the input. We can thus talk about the DAG independently of the particular schedule.

A *schedule* of a computation (G, w) consists of a sequence of steps $(1, \ldots, \tau)$ in which each action a *covers* a contiguous block of $w(a)$ steps $(i, \ldots, i + w(a) - 1)$ for some i, $1 \leq i \leq \tau - w(a) + 1$. We say the action *starts* on step i and *finishes* on step $i + w(a) - 1$. An action is *ready* on step i if all its ancestors in G have finished before step i, and is *waiting* on step i if it is ready on step i but starts after step i. In all schedules an action must be ready on the step it starts.

A *p-schedule*, for $p \geq 1$, is a schedule such that each step is covered by at most p actions. A schedule is *sequential* if $p = 1$, and *parallel* otherwise. A *greedy p-schedule* is a p-schedule with the property that an action only waits on step i if step i is covered by p actions.

The *work* of a computation (G, w) is defined to be the sum of the weights of the actions. The *depth* is defined to be the maximum, among all the paths in G, of the sum of the weights along the path.[2] The *number of steps* of a schedule S, denoted $|S|$ is the maximum step on which an action finishes. Note that these definitions preserve an important well-known property of greedy p-schedules:

LEMMA 1. *Let W_p be the work and d_p be the depth of a computation when run with a greedy p-schedule $\mathcal{S}_p$. Then $|\mathcal{S}_p| < W_p/p + d_p$.*

Given a priority order on the actions, a *prioritized p-schedule* $\mathcal{S}_p$ is a schedule in which an action can only start on step i if no action with higher priority is waiting on step i. A prioritized p-schedule is *based on* a 1-schedule for the same DAG if the priorities are defined by the order of actions in the 1-schedule (highest priority first).

A *depth-first 1-schedule* (1DF-schedule) is a prioritized 1-schedule in which the priority is given by the maximum step number on which any of the action's parents finished. This is the schedule executed by a "standard" sequential implementation. Note that multiple nodes can have the same parent (*e.g.*, when the parent is a fork node) so there are typically many 1DF-schedules for the same DAG. The user can decide which one is best to optimize cache behavior. A *depth-first p-schedule* (PDF-schedule) of G is a p-schedule based on a 1DF-schedule of G. An example is given in Figure 2.

To simplify the exposition, we will map a parallel schedule to a sequential schedule that preserves the parallel order, as follows. A *linearization* of a p-schedule $\mathcal{S}_p$ of (G, w) is a sequential schedule $\mathcal{S}_\ell$ of (G, w) such that if an action a_1 finishes before an action a_2 starts in $\mathcal{S}_p$, then it appears before

[2]These measures are called *total work* and *total critical-path length*, respectively, in the distributed shared-memory model of [9].

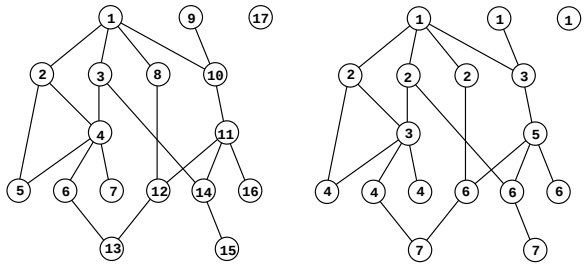

Figure 2: A greedy PDF-schedule of a DAG G, for $p = 3$. All nodes have weight 1. On the left, the nodes of G are numbered in order of a 1DF-schedule, $\mathcal{S}_1$, of G. On the right, G is labeled according to the greedy PDF-schedule, $\mathcal{S}_p$, based on $\mathcal{S}_1$; $\mathcal{S}_p = V_1, \ldots, V_7$, where for $i = 1, \ldots, 7$, V_i, the set of nodes scheduled in step i, is the set of nodes labeled i in the figure.

a_1 in $\mathcal{S}_\ell$. For the example in Figure 2, one linearization of the PDF-schedule is

$$9, 17, 1, 8, 3, 2, 4, 10, 5, 6, 7, 11, 14, 12, 16, 15, 13.$$

We define the number of parallel steps of a linearization $\mathcal{S}_\ell$ of $\mathcal{S}_p$ to be the same as the number of steps of $\mathcal{S}_p$, *e.g.*, 7 in the example. We will consider *worst case* linearizations, *i.e.*, the adversary can select the linearization during execution time. Note that what the adversary controls is the "ordering" among the actions that cover the same parallel step.

Following [7], we define the premature nodes of a p-schedule. Consider any prefix $\mathcal{S}'_\ell$ of a linearized p-schedule $\mathcal{S}_\ell$ based on a sequential schedule $\mathcal{S}_1$. Let $\mathcal{S}'_1$ be the longest prefix of $\mathcal{S}_1$ such that all the nodes in $\mathcal{S}'_1$ are in $\mathcal{S}'_\ell$. The nodes in $\mathcal{S}'_\ell$ can be partitioned into two groups: those not in $\mathcal{S}'_1$, called *premature* nodes, and those in $\mathcal{S}'_1$, called *mature* nodes. $\mathcal{S}'_1$ is also called the *mature-prefix* of $\mathcal{S}'_\ell$. The following lemma is an immediate consequence of Theorem 2.3 of [7].

LEMMA 2. *Let $\mathcal{S}_\ell$ be a linearized p-schedule based on a sequential schedule $\mathcal{S}_1$ of a DAG. Then any prefix of $\mathcal{S}_\ell$ has fewer than $(p-1)d_p$ premature nodes, where d_p is the depth of the DAG under the schedule $\mathcal{S}_\ell$.*

Note that this lemma does *not* provide a bound on the number of nodes that are ever premature—which can be as bad as $\Omega(n)$ for a DAG of n nodes—only a bound on how many nodes can be simultaneously premature.

2.2 Caching Models

In this paper we are interested in modeling multiprocessors with a shared memory and a smaller, but faster shared data cache. In the following definitions we stick to standard terminology as much as possible.

We assume the cache and the memory each consists of a set of fixed-size blocks that hold multiple memory words, and that there is a fixed mapping C from memory blocks to cache blocks. We say a memory block is associated with the cache blocks it maps to. If every memory block maps to all cache blocks the cache is *fully associative*. If every memory block maps to one cache block, the cache is *direct mapped*.

The *residency* $R \subset C$ of memory blocks in the cache is a partial one-to-one function from memory blocks to cache blocks. A memory access of block m can update the residency. If m is already in the cache at some cache block c (*i.e.* $(m, c) \in R$) we call the access a *cache hit* and R is unchanged. If m is not in the cache, then for some c such that $(m, c) \in C$, (m, c) will be added to R (*loaded*) and if there is an $(m', c) \in R$, then it is removed from R (*evicted*). We call this a *cache miss*.

The *replacement policy* for a cache dictates which memory block is removed from R on eviction. On a cache miss for m, any m' such that $(m, c) \in C$ and $(m', c) \in R$ is a *candidate* for eviction. In the *LRU (least recently used)* replacement policy, the evicted memory block is the candidate memory block whose most recent access is the furthest in the past. In the *ideal* replacement policy, the evicted memory block is the candidate memory block whose next access is the furthest in the future. This policy is known to minimize the number of misses [6].

Frigo *et al.* [13] introduced the *Ideal Cache* model, which is fully associative and uses the ideal replacement policy. Although idealized, the model can be emulated on more realistic cache models at the cost of only constant factors [13]. Many *cache-oblivious* algorithms have been designed using this model (*e.g.*, see [3] and the references therein).

In this paper, we make the natural extension of the Ideal Cache model to a multiprocessor setting. For a given p-schedule, a parallel step may be covered by up to p memory accesses. Note that there may be overlap even among the memory blocks accessed in a step.[3] Multiprocessors have different policies for resolving (serializing) concurrent accesses to the same memory block. In this paper, we (pessimistically) consider the serialization to be under the control of the adversary: our bounds consider all possible linearizations of the p-schedule. By serializing concurrent accesses to the cache, a linearization serves two purposes: it determines what the cache's consistent state is for each access and it totally orders the cache misses/hits so that sequential replacement policies can be directly applied. Note that linearization is for the purposes of cache analysis only: the actions in a parallel step are assumed to occur in parallel.

2.3 DAGs with Caching

For analyzing costs for computations with caching we assume each memory access is its own action, and has weight 1 if it is a cache hit and weight m if it is a cache miss. Typical values for m on current processors are in the hundreds, and increasing with each passing year. The weight of a memory access can depend on the schedule, since whether it is a hit or a miss can depend on the order in which actions are scheduled. For example, if node 7 and node 9 in Figure 2 are to the same memory block m, and node 8 does not access memory, then node 7 is a miss and node 9 is a hit in the 1DF-schedule. In the PDF-schedule, on the other hand, node 9 is a miss (because node 9 is scheduled before node 7) and node 7 may be either a hit or a miss depending on whether m gets evicted before node 7 is scheduled. We assume the weights of all other actions do not depend on the schedule.

We will refer to a 1-schedule under the Ideal Cache model as the *Sequential Ideal Cache* model, and a linearized p-schedule under the Ideal Cache model as the *Shared Ideal*

[3] In the case of *false sharing*, the processors may be interested in different memory words that reside in the same memory block; such actions are rightfully unordered in the DAG.

Cache model. Similarly, we will refer to a 1-schedule under a variant of the Ideal Cache model that uses LRU instead of ideal replacement as the *Sequential LRU Cache* model, and a linearized p-schedule under the same variant as the *Shared LRU Cache* model. Finally, observe that a replacement policy need not dictate an eviction priority based solely on the ordering of accesses in its schedule. We will also consider a replacement policy for p-schedules called Seq-LRU, which is LRU according to a given 1-schedule (and not the current p-schedule).[4] We will refer to a linearized p-schedule under a variant of the Ideal Cache model that uses Seq-LRU instead of ideal replacement as the *Shared Seq-LRU Cache* model.

3. LOW MISS CACHE SHARING

This section presents our main results, as summarized by the bounds in Table 1. To simplify the presentation of the results, we will show only the case where the parameter δ in the bounds is 0, leaving to Section 4 the generalization to $\delta > 0$. The section concludes with our lower bound on the shared cache size required to have no extra misses.

3.1 Shared Caching with No Extra Misses

In this section, we compare the number of cache misses M_I on the Sequential Ideal Cache model with cache size C_1 to the total number of misses M_p on the Shared Ideal Cache model with p processors and cache size C_p. Our main result shows that for any computation, if $C_p \geq C_1 + pd_p$, then $M_p \leq M_I$ when using a p-schedule based on the sequential schedule. This implies that although the set of miss nodes differs between the two schedules, overall there are *no extra misses*.

THEOREM 1. *Consider a computation with M_I misses and W_I work on the Sequential Ideal Cache model with cache size C_1 under a sequential schedule $\mathcal{S}_1$. Any linearized greedy p-schedule $\mathcal{S}_\ell$ based on $\mathcal{S}_1$ will have at most M_I misses and $\frac{W_I}{p} + d_p$ parallel steps on the Shared Ideal Cache model with p processors and any cache size $C_p \geq C_1 + (p-1)d_p$, where d_p is the depth of the computation* DAG *under the schedule $\mathcal{S}_\ell$.*

PROOF. We denote as a *bad* node any node that incurred a hit in $\mathcal{S}_1$ but a miss in $\mathcal{S}_\ell$, and a *good* node any node that incurred a miss in $\mathcal{S}_1$ but a hit in $\mathcal{S}_\ell$. We will show that any bad node has a 1-1 correspondence with a good node accessing the same memory block.

Let v be a bad node, and let b be the memory block accessed by v. Let $\mathcal{S}'_\ell$ be the prefix of $\mathcal{S}_\ell$ up to but not including v, and let $\mathcal{S}'_1$ be the mature-prefix of $\mathcal{S}'_\ell$ (*i.e.*, the longest prefix of $\mathcal{S}_1$ such that all its nodes are in $\mathcal{S}'_\ell$).

To facilitate the proof, we consider an abstract Shared Cache Model that is identical to the Shared Ideal Cache model except that it has a replacement policy that we define below. Because the ideal replacement policy necessarily incurs no more misses than any other replacement policy, showing that the abstract model incurs at most M_I misses implies that the Shared Ideal Cache model incurs at most M_I misses.

In the abstract model, we use C_1 cache blocks to mimic the cache residency for $\mathcal{S}_1$ under the Sequential Ideal Cache model and use the rest to hold memory blocks accessed by premature nodes. Specifically, we maintain the invariants that (I1) every cache resident memory block in the sequential cache immediately after $\mathcal{S}'_1$ is also cache resident in the shared cache immediately after $\mathcal{S}'_\ell$, and (I2) the shared cache holds any memory block accessed by a premature node. By lemma 2, there are always fewer than $(p-1)d_p$ premature nodes, and each such node can access at most one memory block. It follows that all the needed blocks fit within a shared cache of size $C_1 + (p-1)d_p$.

Because the bad node v is a hit in $\mathcal{S}_1$, there must a node earlier in $\mathcal{S}_1$ that incurred a miss in $\mathcal{S}_1$ and brought b into the sequential cache. Let v_1 be the latest node among any such nodes. Note that b resides in the sequential cache from immediately after v_1 to immediately after v (and possibly longer). Let $v_2, \ldots, v_j$ be any nodes between v_1 and v in $\mathcal{S}_1$ that also access b. These accesses are all hits in $\mathcal{S}_1$. We denote the nodes $v_1, \ldots, v_j$ as the *run* for v. We will show that these accesses are all hits in $\mathcal{S}_\ell$, and hence v_1 is a good node, and the rest are neither good nor bad.

Each node v_i, $i \in [1..j]$, is scheduled either before or after v in $\mathcal{S}_\ell$. We consider each case in turn.

Suppose v_i is scheduled after v in $\mathcal{S}_\ell$. Node v is premature until all of $v_1, \ldots, v_j$ are scheduled. Thus, immediately before v_i is scheduled in $\mathcal{S}_\ell$, v is premature. By invariant I2, b is in the shared cache at that point, so v_i is a hit.

Next, suppose at least one node v_i, $i \in [1..j]$, were scheduled before v in $\mathcal{S}_\ell$. We will show that v would be a hit in $\mathcal{S}_\ell$, a contradiction. If v_i were premature in $\mathcal{S}'_\ell$, then b is in the shared cache immediately before v (by invariant I2), and v would be a hit, a contradiction. Thus, any v_i scheduled before v is mature in $\mathcal{S}'_\ell$, *i.e.*, v_i is in $\mathcal{S}'_1$. As noted above, b resides in the sequential cache from v_i to v, so it is in the sequential cache immediately after $\mathcal{S}'_1$. By invariant I1, v would be a hit, a contradiction.

Summarizing, we have shown that any node in the run for a bad node v is scheduled after v in $\mathcal{S}_\ell$, and is a hit in $\mathcal{S}_\ell$. This includes v_1, the node at the start of the run, which was a miss in $\mathcal{S}_1$. Thus v_1 is a good node.

Therefore, for each bad node there is a good node. To see that no other bad node is associated with this same good node, note that a bad node ensures that all nodes in its run are hits in $\mathcal{S}_\ell$. If two bad nodes were to share the same good node, the bad node with the lower sequence number would be included in the run of the other bad node and hence would be a hit in $\mathcal{S}_\ell$, a contradiction.

It follows that $\mathcal{S}_\ell$ incurs at most M_I misses.

Finally, because $\mathcal{S}_\ell$ has at most as many miss nodes as $\mathcal{S}_1$, its work is at most W_I. By Lemma 1, $\mathcal{S}_\ell$ has fewer than $W_I/p + d_p$ parallel steps. □

A Sequential Ideal Cache of size C_1 can be simulated by a Sequential LRU Cache of size $2C_1$ incurring at most twice the misses [24], yielding the following corollary.

COROLLARY 1. *Consider a computation with M_I misses and W_I work on the Sequential Ideal Cache model with cache size C_1 under a sequential schedule $\mathcal{S}_1$. Any linearized greedy p-schedule $\mathcal{S}_\ell$ based on $\mathcal{S}_1$ will have at most $2M_I$ misses and $\frac{W_I + mM_I}{p} + d_p$ parallel steps on the Shared Ideal Cache model with p processors and any cache size $C_p \geq 2(C_1 + pd_p)$, where d_p is the depth of the* DAG *under the schedule $\mathcal{S}_\ell$.*

3.2 Shared LRU Caching with No Extra Misses

In this section, we show that the optimal replacement policy is not required to obtain good cache behavior—simply

[4] We discuss in Section 4 how this might be implemented.

matching the replacement priorities of the sequential cache can suffice. In particular, we consider the Shared Seq-LRU Cache model, a weaker variant of the Shared Ideal Cache model that uses an LRU replacement policy according to a given 1-schedule. Specifically, the model associates with each memory access its sequence number in the 1-schedule. At any step in the p-schedule, we can associate these sequence numbers with the set of memory accesses in the p-schedule so far. For each memory block currently resident in the shared cache, we let the maximum sequence number among these accesses be the block's priority. We evict the memory block with lowest priority.

The following theorem shows that a result similar to Theorem 1 can be obtained for the Shared Seq-LRU Cache model.

THEOREM 2. *Consider a computation with M_L misses and W_L work on the Sequential LRU Cache model with cache size C_1 under a sequential schedule $\mathcal{S}_1$. Any linearized greedy p-schedule $\mathcal{S}_\ell$ based on $\mathcal{S}_1$ will have at most M_L misses and $\frac{W_L}{p} + d_p$ parallel steps on the Shared Seq-LRU Cache model with p processors and any cache size $C_p \geq C_1 + (p-1)d_p$, where d_p is the depth of the computation* DAG *under the schedule $\mathcal{S}_\ell$.*

PROOF. The proof has a similar high level structure as the proof of Theorem 1 but is complicated by not being able to leverage an ideal replacement policy.

Let *bad* nodes, *good* nodes, the bad node v, its memory block b, the prefix $\mathcal{S}'_\ell$, its *mature-prefix* $\mathcal{S}'_1$, and the *run* $v_1, \ldots, v_j$ for v all be defined as in the proof of Theorem 1.

We will show that all the accesses in the run for v are hits in $\mathcal{S}_\ell$. Each node v_i, $i \in [1..j]$, in the run is scheduled either before or after v in $\mathcal{S}_\ell$. We consider each case in turn.

Suppose v_i is scheduled after v in $\mathcal{S}_\ell$. Node v is premature until all of $v_1, \ldots, v_j$ are scheduled. Thus, immediately before v_i is scheduled in $\mathcal{S}_\ell$, v is premature. Hence b is among the $(p-1)d_p$ highest priority memory blocks. The shared cache always holds the C_p highest priority memory blocks, so b is in the shared cache immediately before v_i, and v_i is a hit.

Next, suppose at least one node v_i, $i \in [1..j]$, were scheduled before v in $\mathcal{S}_\ell$. We will show that v would be a hit in $\mathcal{S}_\ell$, a contradiction. If v_i were premature in $\mathcal{S}'_\ell$, then b is among the $(p-1)d_p$ highest priority memory blocks, so b is in the shared cache immediately before v, and v would be a hit, a contradiction. Thus, any v_i scheduled before v is mature in $\mathcal{S}'_\ell$. Let $v_{i'}$ have the largest sequence number among these nodes. Immediately after $v_{i'}$ is scheduled in $\mathcal{S}_\ell$, memory block b is in the shared cache.

We claim that the number of memory blocks in $\mathcal{S}'_\ell$ with higher priority than b is less than the shared cache size, and hence b is in the shared cache when v is scheduled, contradicting v being a miss in $\mathcal{S}_\ell$. To simplify the exposition, we define v_{j+1} to be v in what follows. Because $v_{i'+1}, \ldots, v_{j+1}$ are hits in $\mathcal{S}_1$, the nodes in $\mathcal{S}_1$ between each consecutive pair in this sequence can access up to $C_1 - 1$ distinct memory blocks. Thus there can be far more than C_p distinct memory blocks accessed between $v_{i'+1}$ and v. As the p-schedule may schedule these intervening nodes in a different order, a more careful argument is needed than simply counting the number distinct memory blocks between $v_{i'+1}$ and v. Instead, we argue that because v is scheduled no later than $v_{i'+1}$ (by the definition of $v_{i'}$), the only *mature* nodes with larger sequence numbers than $v_{i'}$ in $\mathcal{S}'_\ell$ lie between $v_{i'}$ and $v_{i'+1}$. Moreover, by Lemma 2 there are fewer than $(p-1)d_p$ premature nodes, each of which can access at most one distinct memory block. The claim follows.

Finally, it follows as argued in the previous proof that v_1 is a good node, we have a 1-1 correspondence from the bad nodes to the good nodes, and hence $\mathcal{S}_\ell$ incurs at most M_L misses. The theorem follows by applying Lemma 1. □

3.3 Miss vs. Cache Size Trade-offs for Shared LRU Caching

In this section, we consider the Shared LRU Cache model; this model uses standard LRU replacement (unlike the Shared Seq-LRU Cache model considered in Section 3.2). We present a trade-off between the shared cache size and the number of misses. Unlike the result in Corollary 1, the shared cache is only an additive term larger than the sequential cache.

THEOREM 3. *Let $\alpha \geq 1$ be an integer parameter. Consider a computation with M_L misses and W_L work on the Sequential LRU Cache model with cache size C_1 under a sequential schedule $\mathcal{S}_1$. Any linearized greedy p-schedule $\mathcal{S}_\ell$ based on $\mathcal{S}_1$ will have at most $W_L/\alpha + M_L$ misses and at most $\frac{(1+m/\alpha)W_L}{p} + d_p$ parallel steps on the Shared LRU Cache model with p processors and any cache size $C_p \geq C_1 + (\alpha+1)(p-1)d_p$, where d_p is the depth of the computation* DAG *under the schedule $\mathcal{S}_\ell$.*

PROOF. We use the definitions for good node, bad node, and run from Theorem 1. Consider a bad node v and its run $v_1, \ldots, v_j$ in $\mathcal{S}_1$ (this is the sequence of accesses to the same memory block such that v_1 is a miss and $v_2, \ldots, v_j, v$ are hits in $\mathcal{S}_1$). We call each pair (v_j, v) a *support pair*. We consider two cases relative to each support pair (v_a, v_b): when v_a appears before v_b in $\mathcal{S}_\ell$ (*i.e.*, as in $\mathcal{S}_1$) and when it appears after. We refer to nodes (actions) that access memory as *access nodes*. We say that a node v is *premature for n access nodes* if during n accesses in $\mathcal{S}_\ell$ v is premature.

Consider the first case. By definition v_b is a hit in $\mathcal{S}_1$ and a miss in $\mathcal{S}_\ell$. For v_b to become a miss in $\mathcal{S}_\ell$ it must be separated from v_a by at least C_p access nodes. In $\mathcal{S}_1$ v_b is separated from v_a by at most C_1 access nodes. To force a miss we therefore need to insert at least $C_p - C_1 = (\alpha+1)(p-1)d_p$ new access nodes between v_a and v_b in $\mathcal{S}_\ell$. Such nodes can be inserted in two ways. First, access nodes that come after v_b in $\mathcal{S}_1$ can be executed prematurely relative to v_b. By Lemma 2 there can be at most $(p-1)d_p$ such nodes. Second v_a can be executed prematurely in $\mathcal{S}_\ell$. To force a miss at least $(\alpha+1)(p-1)d_p - (p-1)d_p = \alpha(p-1)d_p$ access nodes that appear before v_a in $\mathcal{S}_1$ must appear after v_a in $\mathcal{S}_\ell$. Thus v_a must be premature for at least $\alpha(p-1)d_p$ access nodes.

To analyze how many accesses in $\mathcal{S}_\ell$ can be premature for $\alpha(p-1)d_p$ access nodes we note that for each access node in $\mathcal{S}_\ell$ at most $(p-1)d_p$ loads are premature (Lemma 2), and that there are a total of at most W_L access nodes. Therefore we can have at most $((p-1)d_pW_L)/(\alpha(p-1)d_p) = W_L/\alpha$ accesses that are premature for $\alpha(p-1)d_p$ accesses nodes. These are the only possible additional misses caused by the first case, v_a before v_b. These will add mW_L/α steps to W_p.

We now consider the second case, v_b before v_a in $\mathcal{S}_\ell$. Consider the run $v_1, \ldots, v_j$ for v_b in $\mathcal{S}_1$ and the step at which each node in the run becomes mature in $\mathcal{S}_\ell$. Call these the maturing points. Then in $\mathcal{S}_\ell$ v_b can either be scheduled before all the maturing points or between two maturing points

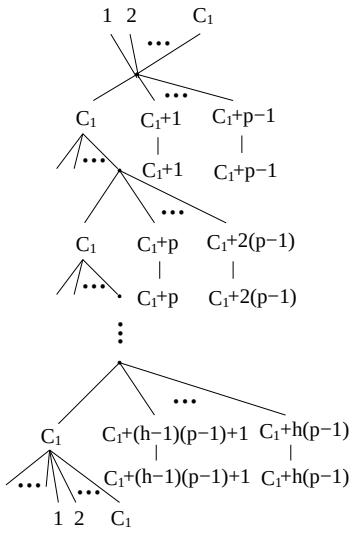

Figure 3: Computation DAG for showing the lower bound in Theorem 4.

v_i and v_{i+1}. In the latter case it will either be a cache hit because v_i supports it, or we can count the fact it is a cache miss against v_i as discussed above (because v_i must premature for $\alpha(p-1)d_p$ access nodes). In the prior case one of v_1 or v_b will support the other as long as neither is premature for more than $C_1+(\alpha+1)(p-1)d_p$ access nodes. Therefore to create a net additional miss one of them has to be premature for $C_1+(\alpha+1)(p-1)d_p$ access nodes. Since this is greater than $\alpha(p-1)d_p$ we have already counted these in the first case. We need to show, however, that such a node can't net two additional misses: it's own due to the second case and another node's due to the first case.

Consider a node v. To net two additional misses, v needs to support another node v_+ in $\mathcal{S}_1$ (which will be a bad node), and needs to be supported by a node v_j in $\mathcal{S}_1$ (so v can be a bad node). The node v_j must be scheduled in $\mathcal{S}_\ell$ at least $\alpha(p-1)d_p$ access nodes before v matures otherwise v_j will support v_+ $\mathcal{S}_\ell$ and v_+ won't be a bad node. For the second case, v has to be scheduled $C_1+(\alpha+1)(p-1)d_p$ access nodes before v_j in $\mathcal{S}_\ell$. Therefore if v falls into both cases it has to be premature for $(C_1+(\alpha+1)(p-1)d_p)+(\alpha(p-1)d_p)$ access nodes. This means each span of $\alpha(p-1)d_p$ of memory accesses for which a node is premature is responsible for at most one additional miss. The same bound on additional misses as the first case follows. □

3.4 A Lower Bound on Shared Caching with No Extra Misses

Theorems 1 and 2 show that a shared cache of size $C_1+(p-1)d_p$ suffices to have no extra misses. In this section, we show that a shared cache of size $C_1+o((p-1)d_p)$ is not sufficient.

THEOREM 4. *For all number of processors $p \geq 2$, sequential cache size $C_1 \geq 2$, miss cost $m \geq 2$, sequential depth $d_L > 6m$, and shared cache size $C_p \leq C_1 + \frac{(p-1)d_p}{3}$, there exists a computation* DAG *G such that:*
(1) a 1DF*-schedule $\mathcal{S}_1$ of G incurs M_L misses and d_L depth on the Sequential LRU Cache model with cache size C_1, and (2) any linearized greedy p-schedule $\mathcal{S}_\ell$ based on $\mathcal{S}_1$ incurs greater than M_L misses on the Shared Ideal Cache model with cache size C_p, where d_p is the depth of the computation* DAG *under the schedule $\mathcal{S}_\ell$.*

PROOF. Consider the computation DAG depicted in Figure 3. Nodes are labeled with the memory block they access. Unlabeled nodes do not access memory. Each node labeled C_1 has $m-1$ children, except that the top such node has one child, the bottom such node has no children, and the next-to-bottom such node has $m-1+C_1$ children. All nodes have weight 1, except that nodes that are misses in a given schedule have weight m. The construction works when d_L is an even number, although it is trivially adapted to handle odd d_L. The middle section of the DAG repeats itself a total of h times, where h is determined by the analysis below.

Consider a 1DF-schedule $\mathcal{S}_1$ of the DAG that schedules the children of a parent node in left-to-right order. $\mathcal{S}_1$ accesses memory blocks $1, 2, \ldots, C_1$, which are all misses, followed by h additional accesses of block C_1, which are all hits, followed by accesses to blocks $1, 2, \ldots, C_1$, which are all hits. After that, $\mathcal{S}_1$ accesses the remaining $h(p-1)$ blocks twice each in succession, with the first access a miss and the second a hit. Thus $\mathcal{S}_1$ incurs only one miss for each memory block, which is the minimum number possible: $M_L = C_1 + h(p-1)$.

The sequential depth d_L is $m+2h-1+m+1 = 2m+2h$. Thus setting $h = (d_L - 2m)/2$ gives the desired sequential depth. Also, because $d_L > 6m$, we have that $h > 2m$, *i.e.*,

$$m < h/2 \tag{1}$$

Now consider the p-schedule $\mathcal{S}_\ell$ and assume that it also incurs only one miss for each memory block. That is, any time an access to a memory block is scheduled after the first such time, it must be a hit. We will show this leads to a contradiction.

Given the assumption, $\mathcal{S}_\ell$ accesses memory blocks $1, 2, \ldots, C_1$, which are all misses, in an order determined by the linearization. This is followed by h rounds of one processor accessing C_1 (a hit) and $p-1$ processors accessing new blocks (misses). While these latter processors are incurring cost m to service the miss, the former processor executes the $m-1$ children of the hit node. Immediately after the last of these h rounds, $\mathcal{S}_\ell$ has accessed all $C_1+h(p-1)$ blocks and each block still needs to be accessed one more time. By the assumption, all the blocks must currently reside in the shared cache, otherwise there would be subsequent misses. Thus $C_p \geq C_1 + h(p-1)$. Moreover, $d_p = d_L = 2m+2h$. But also, $C_p \leq C_1 + \frac{(p-1)d_p}{3} = C_1 + \frac{(p-1)(2m+2h)}{3}$, which by equation 1 is less than $C_1+h(p-1)$, a contradiction.

Thus $\mathcal{S}_\ell$ incurs more misses than $\mathcal{S}_1$. □

Note that although Theorem 4 is stated in terms of the Sequential LRU Cache model, it can be generalized to other sequential cache models. The only properties of the sequential cache that are needed for the proof are (1) that we can determine a priori C_1 memory blocks that map to distinct cache blocks (we need the memory blocks labeled $1, 2 \ldots, C_1$ to each map to distinct cache blocks), and (2) only referenced blocks are brought into the cache (so that evictions occur only as planned). Thus, for example, the Shared Ideal Cache with cache size $C_1 + \frac{(p-1)d_p}{3}$ will incur more misses than even a *direct-mapped* sequential cache with cache size C_1. As the proof shows, an extra miss occurs because the p-schedule has too large of a working set—clever caching strategies can not overcome this problem.

4. FURTHER RESULTS AND DISCUSSION

4.1 Trading Off Work for Cache Size

For the ideal and shared seq-LRU cache models the bounds on cache size is due to the fact that there can be $(p-1)d_I$ premature cache misses. Here we show how the number of premature cache misses can be reduced by adding a delay δ to each of the cache misses. This delay is executed before the cache miss is processed and will reduce the number of premature cache misses from $(p-1)d_I$ to $(p-1)d_I/(1+\delta)$. For the bounds, it is important that a preemption can happen any time during the delay.

To deal with the delay in the DAG model of computations we allow for the insertion of new actions into a computation. This violates our assumption that the DAG structure is independent of the schedule, but only in a limited way. In particular for certain actions v (cache misses in our case) we add a chain of δ new *associated* actions immediately preceding v, each with weight 1. We say that v is *δ-extended*, and say that a DAGs G' is an an extension of G if when any associated actions are removed from G' the DAGs are the same. A prioritized p-schedule $\mathcal{S}_\ell$ of a DAG G' is *an extension of* a 1-schedule $\mathcal{S}_1$ of a graph G, if G' is an extension of G, the priorities for actions $v \in G$ are given by their order in $\mathcal{S}_1$ (highest priority first), and the priority of any of the associated actions of a δ-extended action $v \in G$ are assigned starting just above the priority for v.

THEOREM 5. *Let $\mathcal{S}'_\ell$ be a prefix of a linearized p-schedule $\mathcal{S}_\ell$ that is an extension of a sequential schedule $\mathcal{S}_1$. The number of δ-extended premature actions in $\mathcal{S}'_\ell$ is less than $(p-1)d_p/(\delta+1)$.*

PROOF. For a δ-extended action v to be premature, all of its associated actions need to be premature since their priorities fall immediately higher than that of v. Therefore if there can be at most $(p-1)d_p$ premature actions (Lemma 2) then there can be at most $(p-1)d_p/(\delta+1)$ δ-extended actions, along with their δ associated actions, that are premature. □

Based on this theorem, and assuming every cache miss is δ-extended, adjusting the proof of Theorems 1 and 2 to generate the bounds given in Table 1 is not hard. For example the proof of Theorem 1 goes through even if (I2) states that the shared cache holds any memory block accessed *and missed* by a premature action. The stricter bound on premature misses then gives the required bounds. Given M_p misses in $\mathcal{S}_\ell$ the work increases by δM_p since every miss adds δ actions.

We note that these bounds do not violate the lower bound of Theorem 4 since by adding the delay actions we are changing the structure of the DAG.

4.2 Coarse-Grained Parallelism

Our results also apply to multithreaded computation models (*e.g.*, [10]). In such models the actions are properly partitioned into a set of paths in the DAG, which are called *threads*. In practice it is best to avoid preempting threads between actions since it might involve a context switch (the assumption is the actions within a thread share state such as the register set). Formally we say a thread t is *preempted* between actions a_1 and a_2 of t if a_1 finishes on some step i, and a_2 is ready but does not start on step $i+1$, *e.g.*, because an action from another thread is scheduled instead.

In general a prioritized p-schedule often needs to preempt a thread to schedule a higher priority action from another thread. This ability is important in our results so that the number of premature actions of certain types can be limited, *e.g.*, cache misses. Since we do not care about other actions, however, we can looses the definition of a prioritized schedule to only prioritize certain actions. This means that preemption is never necessary before any other action. Formally we say a *$p(a)$-prioritized p-schedule* is one in which only actions satisfying a predicate $p(a)$ are prioritized. In Table 1 the bounds of rows 1, 2 and 4 are valid when $p(a)$ is only true for the action immediately following each cache miss in a thread. This is because all we care about are the premature nodes that are misses—by only preempting after cache misses we can get more than $(p-1)d_p$ premature nodes, but only $(p-1)d_p$ of them can be cache misses. When combined with delay (Section 4.1) we still require that a thread can be preempted any time during its delay steps. The lower bound of Theorem 4 is valid for any $p(a)$ which is *at least* true for the action immediately following each cache miss in a thread. Finally, in Table 1, the bounds of row 3 are valid when $p(a)$ is true for all memory references, but not other instructions. This is because even a premature cache hit can cause an extra miss.

4.3 Maintaining the Priority Order

Section 3.2 considered a replacement policy for p-schedules called Seq-LRU, which is LRU according to a given 1-schedule. Implementing such a replacement policy is actually not much more difficult than implementing a standard LRU policy, at least if the 1-schedule is a 1DF-schedule. The standard implementation of an LRU policy uses a doubly linked list to maintain the priority order. Executing a cache hit on memory block b will splice b out of where it currently appears in the list and put it at the head of the list. Executing a cache miss on b will add b to the head of the list, and evict the tail of the list.

To extend this to maintain the Seq-LRU policy for a 1DF-schedule we have each ready node v maintain an additional finger element in the list which corresponds to its priority; *i.e.*, all elements in the list after the finger correspond to accesses that occur after v in the 1DF-schedule, and elements before the finger correspond to accesses before v in the 1DF-schedule. Here we outline how these fingers can be maintained. When v executes a memory access the memory block is spliced in immediately before the finger element (instead of at the head of the list). Any nodes that become ready after v is executed, and are not dependent on any other nodes, are given fingers immediately following v (if more than one, they are put in their sequential order). If the ready child is dependent on other nodes, the child inherits the latest finger from its parents.

The implementation as is described is sequential. To implement a parallel version would likely require some form of tree. We do not study such a parallel version here, which could likely use the priority queue scheduler presented in [8].

4.4 Direct-Mapped Shared Caches

Many machines provide direct-mapped caches. We are currently exploring two ways to extend our results to direct-mapped shared caches. One approach is to study randomized direct-mapped caches, in which memory blocks are assigned to cache blocks using a random hash function. Such

a cache has similar characteristics to a fully associative LRU cache in that the more memory accesses between two consecutive accesses to the same block b, the greater the likelihood that b will be evicted between the two. In the LRU this is a 0-1 step function, while in the randomized direct-mapped case it is a smoothly increasing function. We conjecture that bounds similar to Theorem 3 can be shown on expectation. A second approach is to consider a machine with a direct-mapped cache augmented by a fully associative victim cache [17]. We conjecture that a small ($O(pd_p)$ size) specially-designed victim cache may suffice, because it need only hold blocks accessed by the currently premature nodes.

5. CONCLUSION

This paper presented upper and lower bounds for the number of cache misses of a computation when run on p processors or threads with a shared cache as compared to the number of misses for the same computation on a single processor. The user is free to optimize the single processor ordering (*e.g.*, by reordering loops, or the order of function calls), and our bounds are then relative to that ordering. We showed that for the Ideal Cache and a variant of the LRU cache the p processor version need suffer no additional misses when the shared cache is pd_p larger than the single processor cache. We showed that these bounds are tight for greedy schedules. For the true LRU cache the p processor version might require additional misses, but these are bounded. All our bounds are for prioritized schedules based on the sequential schedule.

Our results give some theoretical justification for sharing caches among processors since they show that under certain conditions it is adequate to support a total shared cache size that is only slightly larger than the single processor cache size. The results also indicate that perhaps prioritized schedules, and in particular PDF-schedules, are the most appropriate for machines with shared caches.

6. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] A. Agarwal, M. Horowitz, and J. L. Hennessy. An analytical cache model. *ACM Trans. on Computer Systems*, 7(2):184–215, 1989.

[3] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th ACM Symp. on Theory of Computing (STOC)*, pages 268–276, May 2002.

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proc. 27th ACM International Symp. on Computer Architecture (ISCA)*, pages 282–293, June 2000.

[5] 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.

[6] L. A. Belady. A study of replacment algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[7] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[8] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 12–23, June 1997.

[9] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, June 1996.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[11] Y.-Y. Chen, J.-K. Peir, and C.-T. King. Performance of shared caches on multithreaded architectures. *Journal of Information Science and Engineering*, 14(2):499–514, 1998.

[12] P. Fatourou. Low-contention depth-first scheduling of parallel computations with write-once synchronization variables. In *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 189–198, July 2001.

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 285–298, Oct. 1999.

[14] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[15] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

[16] S. Irani. Competitive analysis of paging. In *Online Algorithms*. Springer, 1998. LNCS, 1442:52–73.

[17] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th ACM International Symp. on Computer Architecture (ISCA)*, pages 364–373, May 1990.

[18] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous multi-threading implementation in POWER5. In *15th IEEE Hot Chips*, Aug. 2003.

[19] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture, white paper. *Intel Technical Journal*, 6(1), Feb. 2002.

[20] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 93–102, July 2001.

[21] G. J. Narlikar. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems*, 35(2):151–187, 2002.

[22] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Trans. on Programming Languages and Systems*, 21(1):138–173, 1999.

[23] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 169–178, July 1990.

[24] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[25] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with application to cache partitioning. In *Proc. 2001 ACM International Conference on Supercomputing*, pages 1–12, June 2001.

[26] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[27] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture, technical white paper. Technical Report 20, IBM Server Group, Oct. 2001.

[28] D. Thibaut and H. S. Stone. Footprints in the cache. *ACM Trans. on Computer Systems*, 5(4):305–329, 1987.

[29] D. Thibaut and H. S. Stone. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.

[30] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[31] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. 22nd ACM International Symp. on Computer Architecture (ISCA)*, pages 392–403, June 1995.