On the Theory of Spatial and Temporal Locality *

Marc Snir and Jing Yu

Computer Science Department

University of Illinois ar Urbana-Champaign

Urbana, IL 61801 USA

{snir,jingyu}@uiuc.edu

July 21, 2005

Abstract

This paper studies the theory of caching and temporal and spatial locality. We show the following results: (1) hashing can be used to guarantee that caches with limited associativity behave as well as fully associative cache; (2) temporal locality cannot be characterized using one, or few parameters; (3) temporal locality and spatial locality cannot be studied separately; and (4) unlike temporal locality, spatial locality cannot be managed efficiently online.

1 Introduction

The performance of modern microprocessors is increasingly constrained by the performance of the memory subsystem. Memory access speed has increased much more slowly than processor speed, leading to the well known "memory wall" problem [17]. To palliate this problem, an increasing fraction of a microprocessor chip is devoted to caches and to buses. Caches are effective for codes that exhibit good locality in their memory accesses.

The theoretical underpinnings of caches and of locality of accessed have been studied in a large number of papers – see, e.g., the survey in [2, Chapter 3.9]. Nevertheless, it turns out that many

*This work was partially supported by DARPA Contract NBCH30390004, and DOE contract DE-FG02-03ER25560X

fundamental issues are still unanswered. It has been known since long that *reuse distance* (or *stack depth*) provides a "correct" characterization of temporal locality in sequences of accesses, in that it predicts cache misses for fully associative, *LRU* caches (We discuss terminology and previous results in Section 2). The seminal paper of Sleator and Tarjan [14] has shown that a fully associative, *LRU* cache is a best possible design for a cache, in the sense that no other design can guarantee a higher hit rate, for the same cache size, and even full knowledge of future memory accesses cannot improve cache performance much. On the other hand, there is less understanding of the proper way to characterize spatial locality.

This paper explores the following issues:

- Are the theoretical results discussed above relevant to caches with limited associativity? We answer affirmatively by showing that by hashing addresses in memory one can insure that a direct mapped cache provides the same performance guarantees as a fully associative cache.
- Is it possible to characterize temporal locality with one or a few parameters? There have been recent attempts to do so, and to characterize applications using a two dimensional temporal/spatial locality space [10]. One approach [16, 15] is to define a parameterized family of synthetic probabilistic streams of references, and to show that the stream of references generated by an application can be approximated by a suitable chosen member of this family: both will generate about the same number of cache misses on any cache system. We show that such an approach cannot possibly succeed: The "dimensionality" of temporal locality is unbounded (proportional to total memory size).
- Is it possible to study temporal locality and spatial locality in isolation? We answer in the negative, by showing that a stream of references can exhibit different temporal locality for different cache line sizes and different spatial locality for different cache sizes. Thus, one cannot speak, in general, of applications with good or bad temporal locality or applications with good or bad spatial locality: the statement on spatial locality needs to be restricted to a given cache size, and the statement on temporal locality need to be restricted to a given line size.
- Can spatial locality be managed online, the same way that temporal locality can be managed online, using suitable cache replacement policies such as *LRU*? We answer in the negative, showing that in a model where data can be rearranged so as to improve spatial locality offline algorithms gain additional power in proportion to the line size.

We introduce in the next section terminology and survey known results. We discuss in Section 3 the effect of limited associativity; address the parametrization of temporal locality in Section 4;

address the dependence of temporal and spatial locality in Section 5; and discuss rearrangement in Section 6.

2 Terminology

We provide in this section a quick survey of cache organization and terminology.

Caches are small but fast stores that hold a copy of parts of the content of memory; they result in improved performance if a large fraction of memory accesses can be satisfied by the cache. A cache is typically organized into multi-word *cache lines*. Whenever a memory location is accessed, the entire line containing this location is brought into cache, thus amortizing the large memory latency against multiple word accesses. Caches are divided into *sets*, each containing a fixed number of cache line frames. The address of a cache line determines the set that can hold it; it can occupy any line frame within this set. Thus, a cache is associated with the following parameters:

Size M The total size of the cache (in words)

Line size L The size of each cache line (in words)

Associativity A The number of cache line frames within each associativity set

A cache is *fully associative* if it consists of only one set ($A = M/L$), so that a line can be stored anywhere in the cache; it is *direct mapped* if each set contains only one cache line ($A = 1$).

A memory reference is a *hit* if it is satisfied from the cache, a *miss*, otherwise. The *hit rate* is the fraction of references that are hits, and the *miss rate* is the fraction that are misses. When a miss occur, if the accessed set is full, then a line is evicted from the set to make room for the newly accessed line. Various policies can be used to decide which line to evict from the set:

LRU The least recently referenced cache line is evicted.

FIFO The cache line that was least recently brought into the cache is evicted.

RANDOM A randomly chosen cache line is evicted.

OPT The cache line that is next accessed farthest in the future is evicted.

The first three policies are *online* algorithms: the decision on which line to evict depends only on the memory accesses performed up to the miss. The OPT policy minimizes the number of cache

misses (hence its name). However, unlike the other three policies, it is an *offline* algorithm, since the eviction decision depends on future accesses. An online algorithm can be implemented by hardware that observes memory accesses as they occur. An offline algorithm requires prior knowledge of the sequence of memory accesses to be performed. It cannot be implemented by a hardware controller that merely observes the accesses performed by the executing program. However, the OPT policy is a useful reference point as it achieves the best possible hit rate.

Caches are effective in hiding the memory latency when most of the memory accesses are cache hits. There two main reasons for this:

Temporal locality If a memory location is accessed then it is likely to be accessed again in the near future. This results from two effects: (i) some memory locations are accessed more frequently than others and (ii) for the same memory location, accesses are clustered in time. As a result, when a word is brought into the cache, there is a good likelihood that it will be accessed again before it is evicted.

Spatial locality If a memory location is accessed than nearby locations in memory are likely to be accessed in the near future. This results from the fact that "related" values, such as fields of the same record, or neighbor elements of a matrix, are often stored in close proximity to each other. As a result when a cache miss causes a cache line to be brought into the cache, there is a good likelihood that words in the line other than the one that caused the miss will be accessed as well.

2.1 Temporal Locality and Stack Depth

The sequence of memory references performed by a processor during a computation is represented by a sequence of addresses $\mathcal{A} = a_1, a_2, \ldots, a_n$. We first assume that $L = 1$: each cache line contains one word and all accesses are word sized. We assume that the cache is fully associative and LRU.

Consider and unbounded stack containing the referenced words and assume that each word is moved to the top of the stack when it is referenced. We define the *Stack Depth* of reference j, $Depth(j)$, to be the depth in the stack of the word referenced at step j; $Depth(j) = \infty$ if j is the first reference to a_j. It is easy to see that the M top words on the stack at step j are the same as the M words held in a fully associative LRU cache at step j. Thus, the j-th access is a hit if $Depth(j) \leq M$, a miss otherwise. Thus, if we define the *Miss* function as

$$Miss^{\mathcal{A}}(m) = \frac{|\{j : Depth(j) > m\}|}{|\mathcal{A}|}$$

then $Miss^{\mathcal{A}}(m)$ is the miss rate of a fully associative, LRU cache of size m on the stream of

references $\mathcal{A}$.

The *Reuse Distance* of reference j, $ReuseDist(j)$, is defined to be the number of distinct memory locations accessed since the last access to a_j, or ∞, if there is no previous access to a_j. Note that if $Depth(j) = k < \infty$ then there have been exactly $k-1$ distinct words accessed since the last preceding access to a_j – the words that are above a_j in the stack, so that the $ReuseDist(j) = Depth(j) - 1$. Thus, the terms "stack depth" and "reuse distance" are often used interchangeably.

The association between stack depth and LRU caches shows that $Miss()$ is monotonic non-increasing: larger caches means fewer misses. The same holds true for the OPT algorithm: this algorithm can do no worse with a cache of size $M+1$ than a cache of size M since it can always choose to ignore the extra location. Monotonicity does not necessarily hold for other cache management algorithms.

More generally, we denote by $Miss^{\mathcal{A}}_{ALG}(m)$ the miss rate that occur with the reference stream $\mathcal{A}$ when a cache of size m is managed using algorithm ALG. In particular, $Miss^{\mathcal{A}}_{OPT}(m)$ denotes the number of misses when the optimal offline algorithm is used (this algorithm evict the word that is accessed furthest in the future).

Definition 2.1. A cache management algorithm ALG is $c(M,m)$-*competitive* if, for any stream of references $\mathcal{A}$, $Miss^{\mathcal{A}}_{ALG}(M) \leq c(M,m) Miss^{\mathcal{A}}_{OPT}(m)$.

Theorem 2.1. *[14]*

1. *LRU is $M/(M-m+1)$ competitive.*
2. *If an online algorithm is c-competitive then $c \geq M/(M-m+1)$.*

If we take $M = 2m$, then the theorem claims that the number of cache misses with an LRU managed cache will be within a factor of 2 of the best possible for a half sized cache, even assuming a cache management algorithm that has full knowledge of the future.

The upper bound holds not only for LRU, but for a wide variety of cache management algorithms, including $FIFO$ and $RANDOM$ [2, 11]. While LRU often results in fewer cache misses than $FIFO$ or $RANDOM$, it does not provide a better guaranteed performance than these other algorithms.

The last result implies that online cache management is adequate: an LRU managed cache will take advantage of the temporal locality in the stream of references in order to achieve a cache hit rate that is competitive; a simply managed larger cache is more effective than a smaller cache with more elaborate software management.

3 Cache Associativity

It is not practical to build large, fully associative caches, because of the complexity of performing many simultaneous address comparisons. In practice, caches are built to be set-associative, with relatively small set sizes (1–8). Because of the limited associativity, one encounters *conflict misses*: misses that occur because multiple lines are mapped to the same set, but that would be avoided by a fully associative cache. In the worst case, all addresses in a reference stream may fall within the same associativity set: the effective capacity of the cache will be the the number of lines within the associativity set, not the total number of cache lines.

In practice, it is relatively easy to avoid a frequent incidence of conflict misses, by padding arrays and avoiding "bad" distances between references that occur in close proximity in a loop [6, 13, 1]. We provide below theoretical results that make the same point: compiler and run-time transformations can be used to ensure that caches with limited associativity essentially generate the same miss rates as fully associative caches.

Let $\mathcal{H}$ be a class of functions from A to B. $\mathcal{H}$ is a *universal hash function*, if, for randomly chosen $h \in \mathcal{H}$, for any $a, a' \in A$, $\Pr(h(a) = h(a')) = 1/|B|$. Easy to compute universal hash functions from $\{0,1\}^n$ to $\{0,1\}^m$ are defined in [4]. For example, the family of functions $h_{r,s}(x) = rx + s \mod 2^m$ is a universal hash function.

We assume a programming model where a fixed amount of memory (a fixed address range) is allocated at the beginning of program execution. The program executes using a direct mapped cache of size M; addresses are mapped onto cache locations using a hash function h randomly chosen from a class of universal hash functions, before program execution.

This model can be implemented in a variety of ways: Hashing can be done in hardware; programs generate unhashed memory references that are hashed before they are used to address caches. Hardware hashing of memory addresses has been used in the past to avoid conflicts in multibanked memory systems; see, e.g., [8, 12]. The obvious downside of such a design is the added delay for cache access; this could be circumvented by using hashing only for second and third level caches. Hashing could reduce the efficiency of prefetching, but this can be circumvented by using the unhashed addresses in the prefetch predictor logic. Finally, hashing could reduce locality at the next level of the memory hierarchy, e.g., at the level of pages; this can be circumvented by hashing only the displacement within each page.

Alternatively, one can imagine that "hashing" is achieved by a remapping of variables that is performed by the compiler; a simple example is to pad arrays so as to avoid "bad" extents [6, 13, 1].

Let $Miss^{\mathcal{A}}_{Hash}(M)$ be the number of misses generated by the reference stream $\mathcal{A} = a_1, \ldots, a_n$

when these references are hashed, and a *direct mapped* cache of size M is used

Theorem 3.1.

$$E\left(Miss^{\mathcal{A}}_{Hash}(M)\right) \leq \frac{M}{M-m+1}\left(Miss^{\mathcal{A}}_{OPT}(m)+m\right)$$

for any sequence of references $\mathcal{A} = a_1, \ldots, a_n$.

The theorem shows that a direct mapped cache, when coupled with memory hashing, provides the same performance guarantee as a fully associative cache.

Proof. We use the same potential function used in [11] to analyze $RANDOM$ cache management. Let S^i_{Hash} be the (random) set of words in the (size M) cache maintained by $Hash$ after step i, and let S^i_{OPT} be the (deterministic) set of words in the (size m) cache maintained by OPT after step i; $S^0_{Hash} = S^0_{OPT} = \emptyset$; let $\Phi^i = |S^i_{OPT} \cap S^i_{Hash}|$ and let $\Delta\Phi^i = \Phi^i - \Phi^{i-1}$. Let δ^i_{OPT} be the indicator function that is equal to 1 if OPT has a miss at step i, 0 otherwise. Let δ^i_{Hash} be similarly defined, for the $Hash$ algorithm. Finally, let

$$X^i = \delta^i_{Hash} - \frac{M}{M-m+1}\delta^i_{OPT} - \frac{M}{M-m+1}\Delta\Phi^i.$$

Then

$$\sum_{i=1}^{n} X^i = Miss^{\mathcal{A}}_{Hash}(M) - \frac{M}{M-m+1}Miss^{\mathcal{A}}_{OPT}(m) - \frac{M}{M-m+1}(\Phi_n - \Phi_0) \geq$$
$$Miss^{\mathcal{A}}_{Hash}(M) - \frac{M}{M-m+1}Miss^{\mathcal{A}}_{OPT}(m) - \frac{M}{M-m+1}m.$$

Thus, it suffices to show that $E(\sum X^i)) = \sum E(X^i) \leq 0$.

There are four cases to consider.

Case 1. Neither OPT nor $Hash$ have a miss at step i. Then $X^i = 0$.

Case 2. OPT has a miss, but $Hash$ has a hit. Then $\delta^i_{Hash} = 0$, $\delta^i_{OPT} = 1$ and $\Delta\Phi^i = 0$ if OPT evicts an word from $S^{i-1}_{OPT} \cap S^{i-1}_{Hash}$, 1, otherwise. Thus

$$X^i \leq -\frac{M}{M-m+1} < 0.$$

Case 3. $Hash$ has a miss but OPT has a hit. Then $\delta^i_{OPT} = 0$ and $\delta^i_{Hash} = 1$. $\Delta\Phi^i = 0$ if $Hash$ evicts an word from $S^{i-1}_{OPT} \cap S^{i-1}_{Hash}$, 1, otherwise. But $S^{i-1}_{OPT} \cap S^{i-1}_{Hash} \subset S^{i-1}_{OPT} - \{a_i\}$, so that $\Delta\Phi^i = 0$ only if a_i maps to the same cache location as some word in $S^{i-1}_{OPT} - \{a_i\}$. Since

$|S_{OPT}^{i-1} - \{a_i\}| = m - 1$, this happens with probability of at most $(m-1)/M$. Thus

$$\mathrm{E}(X^i|\text{Case 3}) \leq 1 - \frac{M}{M-m+1}(1 - \frac{m-1}{M}) = 0.$$

Case 4. Both Opt and $Hash$ have a miss. Then $\delta_{OPT}^i = \delta_{Hash}^i = 1$. Let b be the word evicted by OPT. We have $\Delta\Phi^i \geq 0$ unless OPT evicts an word that is not in S_{Hash}^{i-1}, while $Hash$ evicts an word from $S_{Hash}^{i-1} \cap S_{OPT}^{i-1}$, in which case $\Delta\Phi^i = -1$. This can happen only if a_i conflicts with an word in $S_{OPT}^{i-1} - \{b\}$, so happens with probability $\leq (m-1)/M$. Thus,

$$E(X^i|\text{Case 4}) \leq 1 - \frac{M}{M-m+1} - \frac{M}{M-m+1}(-\frac{m-1}{M}) = 0.$$

□

The proof assumes and *oblivious adversary*: the sequence of references is independent from the hashing function chosen by the algorithm.

The last theorem, while showing that a direct mapped cache will provide the same performance guarantee when used with hashed addresses as a fully associative cache does, does not imply that such a cache will perform as well as a fully associative cache on each sequence of accesses. To prove such a result, we need a stronger independence property from the hashing function: The class of hashing functions $\mathcal{H}$ is *strongly independent* if, for any $a_1, \ldots a_k \in A$, $b_1, \ldots b_k \in B$

$$\Pr(h(a_1) = b_1 \& \ldots \& h(a_k) = b_k) = \frac{1}{|B|^k}.$$

Note that usual constructions for universal hash functions provide pairwise independence, or k-wise independence, for fixed k [5]; we are not aware of constructions that work for arbitrary k. On the other hand, results proved under stronger independence conditions are likely to hold, in practice.

Suppose that a system uses a direct mapped cache of size M, where addresses are hashed using a strongly independent hash function. Suppose that reference i has a stack depth of $Depth(i) = d < \infty$. Then reference j is a cache hit if none of the $d-1$ distinct addresses accessed since the previous reference to a_i mapped into the same set as a_i. This happens with probability

$$(1 - \frac{1}{M})^{d-1} \approx (\frac{1}{e})^{\frac{d-1}{M}}.$$

If the cache were fully associative of size m, we would have a hit if $d \leq m$, a miss, otherwise. We now have a high probability of a hit if d is significantly smaller than M. Thus, by picking M to be

large relative to m we can ensure that the miss rate of the direct mapped cache is not much larger than the miss rate of the fully associative cache. This is expressed by the following theorem.

Theorem 3.2. *Assume that a reference stream $\mathcal{A}$ generates a miss rate of r when executed with a fully associative cache of size m. Then, if addresses are hashed using a strongly independent universal hash function, a direct cache of size*

$$M \geq 1 - \frac{m-1}{\ln(1-r)}$$

will generate an expected miss rate of $\leq 2r$.

Proof. We use the inequality

$$1 - \frac{1}{M} \geq e^{-\frac{1}{M-1}}$$

Let $Depth(i) = d$. If $d > m$ then reference i is a miss when executed with the fully associative cache. On the other hand, if $d \leq m$ then the probability that reference i is a hit when executed with the direct mapped cache is at most

$$(1 - \frac{1}{M})^{d-1} \geq (1 - \frac{1}{M})^{m-1} \geq e^{\frac{1-m}{M-1}} \geq 1 - r.$$

Thus, hits of the execution with the fully associative cache become misses on the direct mapped cache with probability $\leq r$. The claim follows. □

Note that for small r we have

$$1 - \frac{m-1}{\ln(1-r)} \approx \frac{m}{r}$$

so that the condition of the theorem becomes $M \geq m/r$.

4 On the Dimensionality of Temporal Locality

The Miss function provides a complete characterization of temporal locality in a sequence of memory accesses, in the sense that it predict cache misses for caches of any size. The function is defined by an unbounded sequence of parameters, one for each possible cache size. It would be desirable to provide a more concise characterization of temporal locality – one might want to have only one parameter that characterizes locality on $[0-1]$ scale (where 0 stands for no locality and 1 stands for most locality). Unfortunately, such information compression is not possible, as indicated by the following theorem.

Theorem 4.1. *Let $0 < m_1 < m_2 < \ldots < m_k$ be a monotonic increasing sequence of positive integers; let $1 \geq p_1 \geq p_2 \geq \ldots \geq p_k \geq 0$ be a monotonic nonincreasing sequence and let ϵ be a positive constant. One can design a sequence of accesses $\mathcal{A}$ so that $\forall i \;\; |Miss^{\mathcal{A}}(m_i) - p_i| < \epsilon$.*

Proof. Let $q_{k+1} = p_k$, $q_i = p_{i-1} - p_i$, for $i = 2, \ldots, k$, and $q_1 = 1 - p_1$. Then $p_i = \sum_{j>i} q_j$, for $i = 1, \ldots, k$, and $\sum q_i = 1$. Let $m_{k+1} > m_k$ be an integer. We are going to build a sequence of accesses so that a fraction of approximately q_i of the accesses hit in a cache of size m_i, but not in a smaller cache.

Let r_i, s_i be positive integers chosen so that $|q_i - r_i/s_i| < \epsilon/(2k)$, and $\sum r_i/s_i = 1$. Let n be a common multiple of $m_1, \ldots, m_{k+1}, s_1, \ldots, s_{k+1}$, chosen so that $m_i/n < \epsilon/(2k)$, for $i = 1, \ldots, k+1$. Finally, let $n_i = (r_i n)/(s_i m_i)$, for $i = 0, \ldots, k+1$. Consider the sequence of accesses

$$\mathcal{A} = (a_1^1 \ldots a_{m_1}^1)^{n_1} \ldots (a_1^{k+1} \ldots a_{m_{k+1}}^{k+1})^{n_{k+1}}$$

where all locations a_j^i are distinct. It is easy to see that, for each $i = 1, \ldots, k+1$, stack depth m_i occurs exactly $m_i(n_i - 1)$ times; no other stack depth occurs, except ∞. The total sequence length is $\sum_{i=1}^{k+1} m_i n_i$. Therefore

$$\Pr(Depth(j) = m_i) = \frac{m_i(n_i - 1)}{m_1 n_1 + \cdots + m_{k+1} n_{k+1}} = \frac{r_i n/s_i - m_i}{n \sum_{j=1}^{k+1} r_j/s_j} = \frac{r_i}{s_i} - \frac{m_i}{n}$$

Consequently

$$|\Pr(Depth(j) = m_i) - q_i| < \epsilon/k$$

and

$$|Miss(m_i) - p_i| = |\sum_{d>m_i} \Pr(Depth(j) = d) - p_i| = |\sum_{s>i} \Pr(Depth(j) = m_s) - \sum_{s>i} q_s| < \epsilon.$$

□

Informally, the theorem states that reference streams can exhibit arbitrary cache miss rates for increasing cache sizes – the only constraint is that cache miss rates decrease with increasing cache size. Thus, if a measure of locality is supposed to predict cache miss rates for caches of arbitrary sizes, then a fixed number of "locality parameters" cannot provide such a measure. Of course, if one is interested only in specific cache sizes, then one can compute miss rates only for those sizes. Similarly, information compression might be possible if one considers not all possible reference streams, but only those occurring in a small number of applications. However, one must be careful with such an approach: the miss rates for various cache sizes are often related to the size of the

problem solved, and may depend on the input values. A successful characterization of temporal locality for a set of benchmarks using one or few parameters may reflect the limited dimensionality of the benchmark set, and may not capture anything meaningful for the application set represented by these benchmarks.

5 Spatial Locality

We turn now to discuss spatial locality and caches with longer cache lines. The usual model used, which correspond to how hardware caches are organized, is one where memory is organized in frames of size L, the size of a cache line; frame j contains words with addresses $jL, jL+1, \ldots, (j+1)L-1$. When a cache miss occurs the frame containing the missed word is brought into a cache line, and the frame previously contained in this line is evicted.

We define $Miss^{\mathcal{A}}_{ALG}(M, L)$ to be the miss rate of a cache of size M with cache lines of size L that is managed by algorithm ALG on the sequence of accesses $\mathcal{A}$ (we always assume that cache size is a multiple of cache line size). An additional useful measure is provided by the *miss bandwidth*, defined as

$$MissBw^{\mathcal{A}}(M, L) = L \times Miss^{\mathcal{A}}(M, L).$$

$MissBw^{\mathcal{A}}(M, L)$ is the amount of memory traffic generated by misses in a (fully associative, LRU) cache of size M, with line size L.

The results of the previous sections can be easily extended by replacing "words" everywhere with "frames". We thus have

Theorem 5.1. *1. LRU is $(M/L)/((M/L) - (m/L) + 1) = M/(M - m + L)$ competitive.*

2. If an online algorithm is c-competitive, then $c \geq M/(M - m + L)$.

In order to study temporal locality we studied the behavior of the function $Miss(M, L)$ as a function of M, with L fixed: A sequence of references has good temporal locality if the the miss rate decreases fast as M increases. In order to study spatial locality, we need to study the behavior of the function $Miss(M, L)$ as a function of L, with M fixed: A sequence of references has good spatial locality if the miss rate decreases fast when L increases.

Unfortunately it turns out that "temporal locality" depends on cache line size: a sequence of accesses can have good temporal locality for small cache lines and bad temporal locality for long cache lines; and vice-versa. Similarly, spatial locality depends on cache size: a sequence of accesses can have good spatial locality for small caches and bad spatial locality for large caches; and vice-versa.

	m	M
1	$m\min(\alpha,1)+M(\beta+\gamma)+(M+1)\delta)$	$m\min(\alpha,1)+M\min(\beta,1)+M\min(\gamma,1)+(M+1)\delta$
L	$m\alpha+\frac{M}{L}(\beta+\delta)+M\gamma+\delta$	$m\min(\alpha,1)+M\min(\beta,1)+M\gamma+(\frac{M}{L}+1)\delta$

Table 1: Number of misses for sequence $\mathcal{A}$ for small and large caches and short and long lines

To illustrate this, let us consider two cache sizes m and $M > m$ and two line sizes 1 and $L > 1$; we assume that $mL < M$ and L divides m and M. Consider the sequence of accesses

$$\mathcal{A} = (a, a{+}L, \ldots, a{+}(m{-}1)L)^{\alpha}(b, b{+}1, \ldots, b{+}M{-}1)^{\beta}(c, c{+}L, \ldots, c{+}L(M{-}1))^{\gamma}(d, d{+}1, \ldots, d{+}M)^{\delta}.$$

The displacements a, b, c and d are multiple of L chosen so that all accessed addresses are distinct. The total number of accesses is

$$\sigma = m\alpha + M(\beta + \gamma + \delta) + \delta.$$

The number of misses for the four possible combinations of cache sizes and line sizes is shown in Table 1.

Assume that $\alpha = \beta = \delta = 0$, while $\gamma \gg 0$. Then $\sigma = M\gamma$ and

$$Miss(m, 1) = 1 \quad Miss(M, 1) = \tfrac{1}{\gamma}$$
$$Miss(m, L) = 1 \quad Miss(M, L) = 1$$

The sequence of accesses exhibits good temporal locality for line size 1 and bad temporal locality for line size L.

Conversely, let $\beta = \gamma = \delta = 0$ while $\alpha \gg 0$. Then $\sigma = m\alpha$ and

$$Miss(m, 1) = \tfrac{1}{\alpha} \quad Miss(M, 1) = \tfrac{1}{\alpha}$$
$$Miss(m, L) = 1 \quad Miss(M, L) = \tfrac{1}{\alpha}$$

The sequence exhibits bad temporal locality for line size 1 and good temporal locality for line size L.

Let $\alpha = \delta = 0$, while $\beta \gg \gamma \gg 0$. Then $\sigma = M(\beta + \gamma)$ and

$$Miss(m,1) = 1 \qquad Miss(M,1) = \frac{2}{\beta+\gamma}$$
$$Miss(m,L) = \frac{\beta/L+\gamma}{\beta+\gamma} \quad Miss(M,L) = \frac{1+\gamma}{\beta+\gamma}$$

The sequence exhibits good spatial locality for cache size m, as increasing line size to L reduces the number of cache misses by a factor of approximately L; it has bad spatial locality for cache size m as increasing line size to L significantly increases the miss rate.

Finally, let $\beta = \gamma = 0$, while $\alpha \gg \delta \gg 0$. Then $\sigma = m\alpha + (M+1)\delta$ and

$$Miss(m,1) = \frac{m+(M+1)\delta}{m\alpha+(M+1)\delta} \qquad Miss(M,1) = \frac{m+(M+1)\delta}{m\alpha+(M+1)\delta}$$
$$Miss(m,L) = \frac{m\alpha+(M/L+1)\delta}{m\alpha+(M+1)\delta} \quad Miss(M,L) = \frac{m+(M/L+1)\delta}{m\alpha+(M+1)\delta}$$

The sequence exhibits bad spatial locality for cache size m, with a significant increase in miss rate when line size increases, and good spatial locality for cache size M, with a reduction in cache miss rate that is approximately proportional to L.

Thus, one cannot speak, in general, of applications with good or bad temporal locality or applications with good or bad spatial locality: one can speak of spatial locality for a given given cache size, and of temporal locality for a given line size. This behavior has been observed by us in actual codes, in particular in the HPC Challenge benchmarks [18].

Changing the cache line length, while leaving the total cache size and the replacement algorithm unchanged may have more extreme effects. On one hand, it is possible to exhibit sequences where the miss bandwidth actually decreases with increasing line length: the line length has a superlinear effect on the miss rate. Consider a repetitive of accesses of the form $a_0 a_1, \ldots, a_m, a_0, a_1, \ldots$. On a cache of even size m with word size lines this sequence will generate 100% misses (cold misses ignored). Assume now that the cache has two words per line. If consecutive locations in the sequence belong to the same line, so that (a_{2i-1}, a_{2i}) are on the same line, for $i = 1, \ldots, m/2$, then the miss rate is 50%; every second access is a miss, and the miss bandwidth has not changed. If, in the other hand, the locations are paired in lines as $(a_0, a_m), (a_1, a_{m-1}), (a_2, a_{m-2}), \ldots$, then the miss rate is $2/m$ and the miss bandwidth is $4/m$ – a very significant reduction.

6 Spatial Locality with Rearrangement

The model we used in the last section to study spatial locality does not capture the possibility of dynamically rearranging variables in memory during execution. Such rearrangement is done as part of garbage collection, in languages such as Java, and can be used to reduce cache misses [7]. Compilers may also modify the layout of data in memory in order to reduce cache misses [9, 3]. To address this, we define an alternative model, which we call *Caching with Rearrangement*. In this model the location of variables in memory can be dynamically changed. We assume that memory is partitioned into disjoint frames each consisting of L consecutive memory locations. When a word that is not in cache is accessed, the frame containing the word is brought into the cache. If the cache is full, then a line consisting of L words is evicted from the cache and stored in a memory frame. The cache controller can choose an arbitrary subset of L words in cache for eviction.

The difference between this formalism and the previous formalism is that one can change the layout of variables in memory during execution. This model is more general than what can be supported efficiently with current hardware mechanisms, but is very appropriate for disk caches, since disk blocks are addressed indirectly. The model captures changes in memory layout done by compiler and runtime. Note that this model has no additional power if $L = 1$; rearrangements only help with spatial locality, when cache lines contain more than one element.

We shall assume w.l.o.g. that when a frame is evicted it replaces in memory the frame that was just loaded. It is easy to see that this does not change the power of the model.

We denote by $RMiss^{\mathcal{A}}_{ALG}(M, L)$ the miss rates under this model when a cache of size M with lines of size L is managed by algorithm ALG (R stands for "Rearrangement"). The LRU algorithm is defined to be the (online) algorithm that always evicts the L least recently used words from the cache and stores them in one memory frame. The OPT algorithm is the (offline) algorithm that minimizes the number of cache misses. It is easy to see that the algorithm that always evict the L entries that are accessed further in the future is not optimal. We do not have a simple characterization of the optimal algorithm.

Theorem 6.1. *1. LRU is $L(M-L)/(M-m)$ competitive in the rearrangeable model, for $L > 1$.*

2. Assume that $m > L^2 - L$. If an online algorithm is c-competitive, then $c \geq L(M - L^2 + 2L)/(M - m + L)$

Proof. The proof is a refinement of the original proof of Theorem 2.1 (see also [2, Section 3.5]). Let $\mathcal{A}$ be a sequence of accesses. We divide the accesses into successive phases, where at each phase (with the possible exception of the last) LRU has exactly $M - L$ misses. None of locations where a miss occurred is evicted from the LRU cache during a phase, so that each miss occurred at a

distinct address. When a phase starts then the last address accessed at the previous phase is in the cache of both LRU and OPT; this location stays in the LRU cache up to the last LRU miss in the phase, so that this address does not cause an LRU miss during the phase. It follows that at least $(M-L)-(m-1) = M-m-L+1$ of the addresses accessed during the phase are not in the cache of OPT at the beginning of the phase, so that OPT has at least $\lceil (M-m-L+1)/L \rceil = (M-m)/L$ misses. Therefore, LRU is $(M-L)/((M-m)/L) = L(M-L)/(M-m)$ competitive.

Let $\mathcal{F} = \{F_1, \ldots, F_r\}$ and $\mathcal{F}' = \{F'_1, \ldots, F'_s\}$ be two sets of memory frames. We say that $\mathcal{F}$ and $\mathcal{F}'$ are *orthogonal* if $|F_i \cap F'_j| \leq 1$, for all i, j.

Let ALG be an online algorithm. We define by induction a sequence of addresses $\mathcal{A}_0, \mathcal{A}_1, \ldots$ and define an offline caching algorithm ADV so that in each phase $\mathcal{A}_i$, $i > 0$, ADV has $(M-m+L)/L$ misses, while ALG has $M - L^2 + 2L$ misses; the lower bound follows.

The algorithm ADV uses $L^2 - L$ locations in the cache as *write buffer* and uses $m - L^2 + L$ locations as a *core cache*. When a location needs to be evicted from the core cache then it is moved to the write buffer. If the write buffer is full then a frame is written back to memory. The L entries of the frame are chosen from among the $L(L-1)$ entries in the write buffer and the one entry being evicted from the core cache. If $\mathcal{F} = F_1, \ldots, F_r$ is a set of memory frames, then one can always choose the evicted frame to be orthogonal to $\mathcal{F}$.

We assume that the memory used by ADV and ALG has $m + (M+1)(M-m+L)$ locations. The memory of ADV is divided into a segment S_0 of m locations and $M+1$ segments $S_1, \ldots, S_{M+1}$ of $M - m + L$ locations each. In phase zero ADV accesses sequentially all locations in memory.

In the first part of each phase $j > 0$ a set A_i of $M - m + L$ locations is accessed. The set A_i has the following properties:

1. The elements of A_i are all contained in one segment $S_{f(i)}$ of the memory of ADV at the beginning of the phase ($1 \leq f(i) \leq M+1$).
2. No element of A_i is in the cache of ALG at the beginning of the phase.
3. A_i is orthogonal to the frames in the memory of ALG at the beginning of the phase.

If the set A_i fulfills these three properties, then the $M - m + L$ accesses to the elements of A_i can be ordered so that they will cause $(M - m + L)/L$ misses of ADV and $M - m + L$ misses of ALG.

let B_i be the set of M elements that where either in the cache of ADV at the beginning of phase i, or were accessed during the first part of phase i. Since $|B_i| = M + L$ then B_i contains an

element that is not in the cache of ALG at the end of the first part of phase i. We can continue and define a sequence of elements $b_1, \ldots, b_{m-L^2+L} \in B_i$ so that if b_j is accessed at the j-th step following the first part of phase i then b_j is not in the cache of ALG. ALG will suffer $m - L^2 + L$ misses, while ADV can avoid any miss in this second part by keeping all these elements in its core cache. Overall during phase i ADV has $(M-m+L)/L$ misses, while ALG has $M-L^2+2L$ misses

We show now that a set A_i with the desired properties can be found. The first two properties can be satisfied since that are more than M segments. Assume that segment $S_{f(i)}$ was last accessed (and updated) by ADV at phase $j < i$. Let $\mathcal{F} = F_1, \ldots, F_r$ be the frames in the memory of ALG at the beginning of phase i. We require that when ADV writes back into $S_{f(i)}$ at phase j than all frames written back are orthogonal to $\mathcal{F}$. This is well-defined, since the accesses performed at phases $j, j+1, \ldots, i-1$ and hence the value of $\mathcal{F}$ does not depend on the layout of the elements written back by ADV at phase j. □

If $M > m \gg L$ then both the upper bound and the lower bound are $\approx LM/(M-m)$. If we compare this to the result in Theorem 5.1, we see that the ability to rearrange data increases the gap between online and offline algorithms by a factor of L. Note that $L(M-L^2+2L)/(M-m+L) \geq L$ when $m > L^2 - L$. Even if the online algorithm uses a cache that is arbitrarily larger than the cache of the offline algorithm, there are sequence of accesses where it will have L times more cache misses than the offline algorithm. This result show that spatial locality cannot, in general, be managed by online policies.

7 Conclusion

This paper discussed measures of temporal and spatial locality. The results in Sections 2 and 3 indicate that stack depth captures well temporal locality: it predicts cache misses for fully associative caches and, with some compiler or runtime help, it predicts well misses for realistic caches. The approach proposed, namely hashing addresses to prevent conflict misses, may not be practical to implement as described. However, caches with limited associativity are known to perform in practice nearly as well as fully associative caches, and compiler transformations have been effectively used to limit conflict misses in caches with low associativity [6, 13, 1]. Thus, the theoretical results are indicative of practice.

The well-known results of [14] shows that online cache management for temporal locality is satisfactory in that one can compensate for the inferior performance of online algorithms, as compared to offline algorithms, by using larger caches.

The results in Sections 4 and 5 show that temporal locality cannot be characterized by one or

few parameters, and that spatial locality and temporal locality cannot be studied in isolation of each other. While we have not pursued this direction in this paper, a similar approach can be used to show that spatial locality cannot be characterized by a fixed number of parameters.

Finally, the results in Section 6 show that online cache management for spatial locality is not possible, in general: it is not possible to compensate for the inferior performance of online algorithms by using larger caches; full prior knowledge of the sequence of accesses can reduce the miss rate by factor that equals the line size.

This last result provides a theoretical justification to the view that spatial locality must be handled, to some extent, as part of algorithm design, and cannot be left to the hardware and run-time. Note that this view runs counter current trends in language design: In modern languages such as Java data layout is totally opaque and cannot be controlled by the programmer. Even older languages such a C or Fortran provide only partial control of data layout, as one cannot easily align variables at cache line boundaries; data layout is not part of the programming model. Yet our analysis shows that some level of control on data layout is desirable. Furthermore, a change in the layout of a data structure such as an array requires "name" changes, since the names (indices) of elements in an array determines their displacement. Thus, if a user transposes an array in the middle of an FFT computation to improve locality, then variable `a[i,j]` becomes variable `a[j,i]`. A more satisfactory design would allow the data layout to change, without changing the names of the variables. Rather than transposing the array and using new indices to access array elements, the user could simply specify that the array layout should be modified from row-major to column-major. This would allow to optimize for spatial locality without obfuscating the computation logic.

Our paper leaves several open problems: We do not have a simple characterization of the OPT cache management algorithm in the model of a rearrangeable cache. Also, we have a gap between the upper bound and the lower bound on the competitiveness of online algorithms in the rearrangeable model.

The LRU algorithm does not seem, intuitively, to be a very good algorithm for the management of caches with long cache lines: This algorithm will evict entries not accessed in a line that was just brought from memory as soon as a new line is brought from memory. For example, if the computation has two interleaved streams of sequential accesses, then LRU will take no advantage of spatial locality. It would be interesting to define an online algorithm for the rearrangeable model that take better advantage of spatial locality than LRU, while providing the same guarantees as LRU in the worse case.

Finally, while our terminology is borrowed from the that of hardware caches, the theory applies to other situations where caches are used, such as algorithms for virtual memory management or

algorithms for managing disk or database caches.

References

[1] David F. Bacon, Jyh-Herng Chow, Dz ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 3. IBM Press, 1994.

[2] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[3] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 252–262, New York, NY, USA, 1994. ACM Press.

[4] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112. ACM Press, 1977.

[5] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *STACS '96: Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, pages 569–580. Springer-Verlag, 1996.

[6] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999.

[7] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[8] BBN Laboratories. Butterfly parallel processor overview. Technical Report 6148, Version 1, BBN Laboratories, Cambridge, 1986.

[9] Monica S. Lam and Michael E. Wolf. A data locality optimizing algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[10] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC challenge benchmark suite. Technical report, UTK, 2005.

[11] Prabhakar Raghavan and Marc Snir. Memory versus randomization in on-line algorithms. *IBM J. of Research and Development*, 38(6):683–708, 1994.

[12] Randall D. Rettberg, William R. Crowther, Philip P. Carvey, and Raymond S. Tomlinson. The Monarch parallel processor hardware design. *Computer*, 23(4):18–28, 30, 1990.

[13] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. *SIGPLAN Not.*, 33(5):38–49, 1998.

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

[15] E. Strohmaier and Hongzhang Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2004)*, pages 467 – 474, 2004.

[16] Joel L. Wolf, Harold S. Stone, and Dominique Thiebaut. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. Comput.*, 41(4):388–410, 1992.

[17] W. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[18] Jing Yu, Sara Baghsorkhi, and Marc Snir. A new locality metric and case studies for hpcs benchmarks. Technical Report UIUCDCS-R-2005-2564, Computer Science, UIUC, 2005.