Remote Memory References at Block Granularity

Gili Yavneh
Remote Memory References at Block Granularity

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Gili Yavneh

Submitted to the Senate of the Technion — Israel Institute of Technology
Iyar 5777 Haifa April 2017
This research was carried out under the supervision of Prof. Hagit Attiya, in the Faculty of Computer Science.

Acknowledgements

I would like to thank my advisor, my husband, my parents and my siblings.

The generous financial help of the Technion is gratefully acknowledged.
# Contents

List of Figures

Abstract ........................................ 1

1 Introduction .................................. 3

2 Related work .................................. 5

3 Model ......................................... 9
   3.1 RMRs ...................................... 9
   3.2 Block RMRs ................................ 12

4 $B$-Block Placement for Objects with Varying Sizes 15

5 Block RMRs in the CC model .................. 17
   5.1 A Polynomial Algorithm for 2-Block Placement 18
   5.2 Hardness Proof for $B$-Block placement, $B > 2$ 23

6 Block RMRs in the DSM model ............... 27
   6.1 DSM with Negligible Invalidation Cost ....... 27
   6.2 DSM Without Invalidation .................. 29

7 Conclusions and Future Directions .......... 33

Hebrew Abstract ................................ i
List of Figures

3.1 Models: Cache Coherence (left) and DSM (right) ......................... 10

5.1 Graph after the weights are calculated ..................................... 19

5.2 Graph with original weights (left) and after incorrect weight adjustment (right) ................................................................. 22
Abstract

The cost of accessing shared objects that are stored in remote memory, while neglecting accesses to shared objects that are cached in the local memory, can be evaluated by the number of remote memory references (RMRs) in an execution. Two flavours of this measure—cache-coherent (CC) and distributed shared memory (DSM)—model two popular shared-memory architectures. The number of RMRs, however, does not take into account the granularity of memory accesses, namely, the fact that accesses to the shared memory are performed in blocks.

This thesis proposes a new measure, called block RMRs, counting the number of remote memory references while taking into account the fact that shared objects can be grouped into blocks. On the one hand, this measure reflects the fact that the RMR incurred for bringing a shared object to the local memory might save another RMR for bringing another object placed at the same block. On the other hand, this measure accounts for false sharing: the fact that an RMR may be incurred when accessing an object due to a concurrent access to another object in the same block.

This thesis proves that in both the CC and the DSM models, finding an optimal placement is NP-hard when objects have different sizes, even for two processes. In the CC model, finding an optimal placement, i.e., grouping of objects into blocks, is NP-hard when a block can store three objects or more; the result holds even if the sequence of accesses is known in advance. In the DSM model, the answer depends on whether there is an efficient mechanism to inform processes that the data in their local memory is no longer valid, i.e., cache coherence is supported. If coherence is supported with cheap invalidation, then finding an optimal solution is NP-hard. If coherence is not supported, an optimal placement can be achieved by placing each object in the memory of the process that accesses it most often, if the sequence of accesses is known in advance.
Chapter 1

Introduction

In a typical multiprocessor, processes communicate by concurrently accessing objects in the shared memory. To reduce the high cost of access to the shared memory, a fast memory, local to each process, is used to cache recently-used objects. The cost of a cache hit—finding an object in the process’s local memory—is negligible relative to the cost of a cache miss, which requires an access to the shared memory. We assume that a block is not evicted from the local memory unless it is required by some other process; that is, the local memory is large and fully associative. A remote memory reference (RMR) [46] is incurred for every cache miss, according to one of two models: In the cache coherent (CC) model, the first time a process accesses an object it puts the object in its local memory; subsequent reads and writes to the same object by the same process are free, as long as no other process modifies the object between them; otherwise, the object must be updated in the local memory again, incurring an RMR. In the distributed shared memory (DSM) model, each object is created in the local memory of some process, before the first access to it occurs; an access to the local memories of other processes incurs one RMR, while an access to the local memory is free. If cache coherence and cheap data invalidation is supported then, like in the CC model, the object will be copied by other processes for read operations, while write operations invalidate the copies at other processes. Making a copy incurs an RMR, while accesses to an existing copy do not. If invalidation is not supported, the object remains in the local memory of the process that created it, and any access to it by a different process incurs an RMR.

Both models, however, ignore the fact that access to the shared memory is performed in blocks, namely, several objects are placed together and moved together between local memories and shared memory. When an object is read into the memory, all the objects in the same block are moved, so that later accesses to them incur a cache hit. Similarly, if one object is moved to another process, accesses to the other objects in the same block causes a cache miss, even if no other process has accessed them in between. This phenomenon is called false sharing [27,44].

Many works deal with cache-conscious organization of the memory, namely, the
placement of objects in blocks so as to increase the number of cache hits, e.g., [22, 23, 41]. However, as we discuss in Chapter 2, these works consider only single-process scenarios and do not take into account the effects of concurrent access to blocks. Algorithms and lower bounds on the number of RMRs, which capture the effects of concurrency, do not take into account the granularity of memory accesses, which are done in blocks.

**Our Contributions** We introduce the block RMRs complexity measure. In the CC model, when a process accesses a block, an RMR is incurred when the block is brought to its cache, and later accesses to the same block are free, as long as no other process writes an object in this block.

For this model, we prove that finding an optimal placement of objects into blocks, i.e., a placement with a minimal number of block RMRs, is NP-hard when blocks can hold three or more objects; the result holds even when the sequence of accesses is known and all objects have the same size (Section 5.2). The problematic access sequence is a natural one, in which two processes perform a traversal on a graph. When blocks can hold two objects of the same size and the access sequence is known, we present an efficient algorithm for placing objects in blocks for the block-based CC model (Section 5.1).

In the block-based DSM model, each block is created in a specific process when the execution starts, and each access to a block not in a process’s own cache incurs an RMR. If cache coherence is supported then an object may have several copies, in which case, consecutive accesses to the same block are free, as long as no other process writes an object in this block. If cache coherence is not maintained, then the object remains in the local memory of the process that created it, and accesses from other processes incurs an RMR, while accesses of the creating process are free.

For the block-based DSM model, we prove that the number of block RMRs depends on the cost of invalidation and existence of cache coherence. If invalidation cost is negligible and cache coherence is supported then finding a placement of objects into blocks is NP-hard, even if the access sequence is known and all objects have the same size. If cache coherence is not supported, we show that the number of block RMRs is indifferent to the order of accesses in the sequence, and use this result to design an algorithm that finds a placement with an optimal number of block RMRs, when the access sequence is known in advance. (The results for the DSM model appear in Chapter 6.)

For both models, we prove that handling objects with varying sizes makes the problem NP-hard (Section 4). This result is achieved with an access sequence in which two processes traverse a tree one after the other. Because of this result, the rest of the thesis concentrates on the case where all objects have the same size.
Chapter 2

Related work

Remote memory references have been proposed as a way to predict the scalability of shared-memory programs [46]. They have been mainly applied for evaluating the complexity of mutual exclusion algorithms. The first mutual exclusion algorithms were based on spin queues where \( N \) processes would use a Read-modify-write primitive to add themselves to the tail of the queue [8, 35, 39]. The algorithms by Anderson [8] and Graunke and Thakkar [35] have \( O(1) \) RMR complexity in the CC model and an unbounded RMR complexity in the DSM model. The algorithm of Mellor-Crummey and Scott [39] has an \( O(1) \) RMR complexity in both models. Newer mutual exclusion algorithms are based on reads and writes to shared objects, starting with Anderson [5] presenting an algorithm with \( O(N) \) RMR complexity and later with Anderson and Yang [47] presenting an algorithm with \( O(\log(N)) \) RMR complexity. In addition, adaptive mutual exclusion algorithms [2, 6] have time complexity that depends on the number of contending processes, i.e., the maximum number of processes that are active at the same time. For these algorithms the number of RMRs is unbounded in the DSM model. Other mutual exclusion algorithms appear in survey [7]. Other works investigated lower bounds on the RMR complexity of mutual exclusion problems [4, 21, 38, 48], as well as other related problems such as leader election [34] and renaming [3].

False sharing [16, 27, 44] occurs when different objects that are placed in the same block are accessed by different processes, and this causes cache misses that would not occur if the objects were in separate blocks. Bolosky and Scott [16] introduced models for false sharing and compared how well they align with the intuitive understanding of this phenomenon. Their interval definition says that the cost of false sharing is the difference in performance between a policy that makes optimal placement decisions, but enforces consistency on a whole-block basis, and one that enforces consistency only for real conflicts between accesses. Our definitions capture this formally by the difference between the number of block RMRs and the number of RMRs, and allow us to prove the NP-hardness result conjectured in [16]. Attempts to reduce false sharing include dynamically adjusting the block size [26] and restructuring shared data by hand tuning [27].
A large body of research studies memory locality for sequential computing. Both cache-conscious and cache-oblivious algorithms and data structures have been introduced.

Cache-conscious algorithms and data structure take into account the structure of the cache, i.e., cache size, block size, placement of objects in blocks and other parameters, in order to minimize cache misses. Petrank and Rawitz [41] showed that the problem of partitioning data into the blocks of a single cache of limited size is NP-hard, and in fact, it is hard to approximate. Our NP-hardness proof for the CC model employs multiple processes, and is achieved using traversals, rather than with an arbitrary access sequence as in their result. However, we do not show that approximating the optimal solution is hard; in fact, it is not, since the number of RMRs for the sequence divided by the size of the block is a lower bound on the number of block RMRs.

Lavaee [37] showed that the problem of data packing, using a fully associative, limited size cache and for a single process is NP-hard. By using multiple processes, our result is achieved with a more natural access sequence that does not require dummy objects in order to fill the cache and cause data to be evicted.

Calder et al. [22] first introduced cache-conscious data placement. Their framework enables a compiler to profile a program and then decide on the placement of the variables on the stack and heap according to temporal locality. Chilimbi, Hill and Larus [23] try to improve the layout of variables in the memory in order to increase spatial and temporal locality by designing a tool that reorganizes variables and the memory pointer structure by clustering together variables that are accessed at the same time and putting frequently accessed variables in non conflicting parts of the memory. In addition, they introduce an allocator that attempts to allocate together variables that are accessed at the same time. Afek et al. [1] introduce a memory allocation scheme that is cache index-aware, i.e., takes into account that objects are placed in a cache ”row” that coincides with their cache index, which is a consecutive subset of the bits in their memory address. Object placement in the memory with a non-uniform distribution of index values may induce many cache misses due to object being taken out of the memory to make place for other object with the same index. They change the allocator to space blocks in the memory in order to achieve a more uniform distribution of blocks over the indexes.

Cache-oblivious algorithms optimize the object layout in the memory and their design is indifferent to cache parameters such as size and block size. The cache-oblivious model was introduced by Frigo, Leiserson, Prokto and Ramachandran [32, 43]. For example, a partitioning of tree nodes into blocks that reduces the number of blocks accessed is given by van Emde Boas trees [45]. Bender, Demaine and Farach-Colton [12] generalized the van Emde Boas layout for complete trees for node degrees between 2 and a constant $\Delta \geq 2$, resulting in both static and dynamic cache oblivious B-trees. More simplified dynamic B-trees were discovered by Brodal, Fagerberg and Jacob [19], and simultaneously by Bender, Duan, Iacono and Wu [13]. A tight lower bound of
\[ \lg e \log_B N \] was proved on the number of memory transfers between every pair of cache levels [11].

Cache-oblivious algorithms were suggested for matrix transposition and FFT [32] and for priority queues [9]. Cache-oblivious sorting algorithms have been proposed in [17, 20, 28–30, 32]. Cache-oblivious dictionary structures appear in [10, 13, 14, 18, 31]. Chowdhury and Ramachandran [24] present cache-oblivious algorithms for several fundamental dynamic programming problems. These algorithms are designed for a single process and do not take into account multi-threading.

More recent work attempts to provide cache-oblivious algorithms in multi process environments. Blelloch, Fineman, Gibbons and Simhadri [15] introduced a distributed version of the cache-oblivious model first presented in [32]. In the parallel cache-oblivious model, a computation of a task such as an algorithm or a series of accesses to a data structure may fork into several sub-computations that later join in order to complete the computation. In the general case, the number of cache misses for algorithms in this model is higher than in the regular cache-oblivious model, due to restrictions needed to minimize false sharing and memory imbalances between sub-tasks. However, it is shown that for quicksort, sample sort, matrix multiplication, matrix inversion, sparse-matrix multiplication, and convex hulls, the asymptotic bounds are not affected. Additionally, they introduced a scheduler that can balance irregular memory usage by different sub-tasks. Chowdhury, Ramachandran, Silvestri and Blakeley [25] introduced the notion of cache-oblivious algorithms which are also multi-core oblivious, i.e., the cache parameters as well as the number of cores are not part of the algorithm. In these models, different processes cooperate in order to complete a single task and data placement is designed to avoid cache misses as much as possible. It is assumed that no other tasks are running at the same time and accessing the same objects. Our work focuses on multi-process environment as well, but takes into account possible contention between different tasks run by different processes which may access the same object, for example concurrent accesses to the same data structure.
Chapter 3

Model

We consider an asynchronous system in which a set of \( n \) processes, \( P = \{p_1, \ldots, p_n\} \), execute concurrently and communicate by accessing a set \( O \) of shared objects. Each process has a local cache memory associated with it. Objects placed in the local memory can be easily retrieved, and the cost of doing so is negligible compared with the cost of fetching objects from another process’s local cache, or from the main shared memory.

A process may read an object’s current value, or write a new value to an object.

Objects may be part of the same data structure, for example, vertexes of a graph. A common access pattern to such data structures is a traversal: a sequence of accesses by a single process to the vertexes of the graph, where each pair of consecutive accesses in the sequence \( \pi \), either access the same vertex or adjacent vertexes.

The local memory is partitioned into blocks of size \( B \), each of which can contain objects whose combined sizes is at most \( B \). The local memory can hold an unbounded number of blocks and therefore, blocks are not evicted due to lack of space. A \( B \)-block placement of \( O \) is a partition of the objects in \( O \) into disjoint sets (blocks), \( \tilde{O} = \{O_1, \ldots, O_\ell\} \), each containing objects with a combined size of at most \( B \). We assume each object can be placed in a single block, and is not spread across blocks.\(^1\)

If all objects have the same size, we will consider their size to be 1. In this case, a \( B \)-block placement of \( O \) is \( \tilde{O} = \{O_1, \ldots, O_\ell\} \) such that for each block \( |O_i| \leq B \).

An access sequence \( \pi \) is a sequence \( (p_{i_1}, a_{1_{j_1}}, o_{j_1}), \ldots, (p_{i_m}, a_{m_{j_m}}, o_{j_m}) \) such that \( p_{i_h} \in P \), \( a_{h} \in \{\text{read}, \text{write}\} \) and \( o_{j_h} \in O \), for every \( h, 1 \leq h \leq m \).

3.1 RMRs

There are two models for counting the number of remote memory references (RMRs) in an access sequence \( \pi \), the cache coherent (CC) model and the distributed shared memory (DSM) model (see Figure 3.1).

\(^1\) Objects with size larger than \( B \) need more than one block. By placing an object in the minimal number of blocks that can contain it (i.e., all parts of the object except perhaps one are placed in a block alone), our results hold in this model, by ignoring the parts of the object that fill full blocks and taking into account only the part of the object that remains and does not fill a full block.
In the CC model, a remote memory reference (RMR) is incurred either on the first access (whether it is a read or a write) to an object by a process or on the first access after a write to the same object by another process. Formally, for an access sequence \( \pi = (p_{i1}, a_1, o_{j1}), \ldots, (p_{im}, a_m, o_{jm}) \)

\[
\#rmr^{CC}(\pi) = |\{(p_{ih}, a_h, o_{jh}) \in \pi | (\forall k, 1 \leq k < h, (p_{ih} = p_{ik}) \implies (o_{jh} \neq o_{jk})) \text{ or } \\
(\exists k, 1 \leq k < h, (o_{jk} = o_{jh}, a_k = \text{write}, \text{ and } p_{ik} \neq p_{ih}) \text{ and } (\forall \ell, k < \ell < h, (p_{ih} = p_{i\ell}) \implies (o_{jh} \neq o_{j\ell}))\}|
\]

In the DSM model, an access to another process’ local memory incurs an RMR, depending on the specific characteristics of the implementation. In theoretical models, every read and write that are not to a process’ own local memory incur an RMR.

Most practical DSM applications provide some form of cache coherence, i.e., a mechanism that ensures processes only read or write data that is valid, and not data that was already overwritten by another process. There are three variants of DSM implementations, depending on whether invalidating an object in a remote memory is possible, and what is its cost of invalidation. The options are summarized in Table 3.1.

Each object \( o \) is created at a creator process, denoted \( creator(o) \), which is the owner of the local memory in which \( o \) originally resides. The creator process \( creator(o) \) is chosen before the first access to \( o \). An object is valid when created, and therefore, accesses to the object are free as long as they are just by the creator process. In all variants of the DSM model, we may choose \( creator(o) \) according to whatever information we have on the access sequence.

If invalidation cost is negligible (for example, see survey [40]), then each read or write to an object not in the local memory causes a copy of the object to appear in the local memory, incurring an RMR. If the action is a write then other copies of the object
<table>
<thead>
<tr>
<th></th>
<th>Reads</th>
<th>Writes</th>
<th>Invalidation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Free as long as data in local memory is valid. Costs 1 otherwise.</td>
<td>Costs 0 if data in local memory is valid, and 1 otherwise. Causes invalidation of object copies in other local memories.</td>
<td>Negligible</td>
</tr>
<tr>
<td>DSM - Negligible</td>
<td>Free as long as data in local memory is valid. Costs 1 otherwise.</td>
<td>Costs 0 if data in local memory is valid, and 1 otherwise. Causes invalidation of object copies in other local memories.</td>
<td>Negligible</td>
</tr>
<tr>
<td>Invalidation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSM - Expensive</td>
<td>Free - data is kept up to date.</td>
<td>Every write causes an update across the system in order to push the new values. Costs 1.</td>
<td>Expensive, usually replaced by an update</td>
</tr>
<tr>
<td>Invalidation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSM - No Coherence</td>
<td>Reads by object creator cost 0. Other reads cost 1..</td>
<td>Writes to an object by a non creator process cost of 1 RMR. Writes by the owner cost 0.</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

Table 3.1: Remote memory references per model
are invalidated, and the cost of doing so is negligible.

Formally,
\[ \#rmr^{DSM-Neg}(\pi) = |\{(p_{ih}, a_h, o_{jh}) \in \pi | (p_{ih} \neq creator(o_{jh})) \}
\quad \text{and} \quad (\forall k, 1 \leq k < h, (p_{ih} = p_{ik}) \implies (o_{jh} \neq o_{jk})) \]
\quad \text{or} \quad (\exists k, 1 \leq k < h, (o_{jk} = o_{jh}, a_k = write, \text{and} \ p_{ik} \neq p_{ih})
\quad \text{and} \quad (\forall \ell, k < \ell < i, (p_{ih} = p_{i\ell}) \implies (o_{jh} \neq o_{j\ell}))\} |.

On the other hand, invalidation cost can be expensive compared to the cost of an RMR. We present this model in order to complete the picture, however, we did not find a use case for this model. In this model we assume that writes make updates to the other local memories (since the cost of an invalidation is almost equal to the cost of an RMR). Therefore, a write always incurs an RMR and reads are always free, because they read valid information. Formally, the number of RMRs is:
\[ \#rmr^{DSM-w-CC}(\pi) = |\{(p_{ih}, a_h, o_{jh}) \in \pi | a_h = write\}| . \]

Finally, if coherence and invalidation are not supported [8, 39], then every access, whether a write or a read, to an object by a process other than its creator, incurs an RMR. Formally, the number of RMRs is:
\[ \#rmr^{DSM-No-CC}(\pi) = |\{(p_{ih}, a_h, o_{jh}) \in \pi | p_{ih} \neq creator(o_{jh})\}| . \]

### 3.2 Block RMRs

In the CC model, given a \(B\)-block placement \(\hat{O} = \{O_1, \ldots, O_\ell\}\), we count the number of block accesses in \(\pi\) that incur a cache miss, i.e., the first access (either a read or a write) to an object in one of the blocks in \(\hat{O}\), or an access after a write by another process to some object in the same block. Formally,
\[ \#brmr^{CC}(\pi, \hat{O}) = |\{(p_{ih}, a_h, o_{jh}) \in \pi | (o_{jh} \in O_\ell) \text{ and} \}
\quad (\forall k, 1 \leq k < h, (p_{ih} = p_{ik}) \implies (o_{jh} \notin O_\ell)) \text{ or}
\quad (\exists k, 1 \leq k < h, (o_{jk} \in O_\ell, a_k = write, \text{and} \ p_{ik} \neq p_{ih})
\quad \text{and} \quad (\forall \ell, k < \ell < i, (p_{ih} = p_{i\ell}) \implies (o_{jh} \notin O_\ell)))\} | . \]

In the DSM model, given a \(B\)-block placement \(\hat{O}\), each block \(O_i\) is associated with a creator process \(creator(O_i)\). All objects in the block \(O_i\) are created by \(creator(O_i)\) The cost of reads and writes is shown in Table 3.2, and explained below.

If coherence is supported with negligible invalidation cost, then each access to an object not in the local memory incurs an RMR; it causes a copy of the accessed block to appear in the local memory. If the access is a write then all other copies of the
<table>
<thead>
<tr>
<th></th>
<th>Reads</th>
<th>Writes</th>
<th>Invalidation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Free as long as data in local memory is valid. Costs 1 otherwise.</td>
<td>Cost 0 if data in local memory is valid, and 1 otherwise. Causes invalidation of other copies in other local memories.</td>
<td>Negligible</td>
</tr>
</tbody>
</table>

| DSM - Negligible Invalidation | Free as long as data in local memory is valid. Costs 1 otherwise. | Cost 0 if data in local memory is valid, and 1 otherwise. Causes invalidation of other copies in other local memories. | Negligible                  |

| DSM - Expensive Invalidation | Free - data is kept up to date. | Writes cost 1. If several consecutive writes are made to the same block, they are consolidated, into a single write. | Expensive, usually replaced by an update |

| DSM - No Coherence | Reads by the object’s creator process cost 0. Other reads cost 1. | Writes by owner process cost 0. Other writes cost 1. | Not supported |

Table 3.2: Block remote memory references in each model
block are invalidated. Accesses to objects whose blocks are in the local memory are free. Formally:

$$\# brmr_{DSM}^{CC}(\pi, \bar{O}) = \{|(p_{ih}, a_h, o_{jh}) \in \pi | (o_{jh} \in O_l) \text{ and }$$

$$(p_{ih} \neq creator(O_l)) \text{ and }$$

$$((\forall k, 1 \leq k < h, (p_{ih} = p_{ik}) \implies (o_{jk} \notin O_l))) \text{ or }$$

$$((\exists k, 1 \leq k < h, (o_{jk} \in O_l, a_k = write, \text{ and } p_{ih} \neq p_{ik})$$

$$\text{ and } ((\forall \ell, k < \ell < h, (p_{ih} = p_{i\ell}) \implies (o_{j\ell} \notin O_l))))\}.$$

On the other hand, if invalidation cost is expensive, then block writes are pushed to other local memories. Therefore a write incurs an RMR, unless it was immediately preceded by a write to the same block and reads are always free, because they read valid information. As in the cases of RMRs, we present this model in order to complete the picture. Formally, the number of block RMRs is:

$$\# brmr_{DSM-w}^{CC}(\pi) = \{|j | a_j = write, o_{ij} \in O_l \text{ and } j = 1 \text{ or } o_{i-1,j} \notin O_l \}.$$

Finally, coherence is not supported, each access to a block by a process other than its creator incurs an RMR; accesses by the creator are free. Formally:

$$\# brmr_{DSM-no-CC}(\pi) = \{|(p_{ih}, a_h, o_{jh}) \in \pi | o_{jh} \in O_l \text{ and } p_{ih} \neq creator(O_l)\}.$$
Chapter 4

*B*-Block Placement for Objects with Varying Sizes

We prove that if objects have varying sizes, then the B-block placement problem is \(NP\)-hard even for a simple traversal on a tree. Assume that each object \(o_i\) has a size \(a_i \in \mathbb{Z}^+\), \(a_i \leq B\), and each object is placed in a single block (and not across more than one block). A \(B\)-block placement is a partition of the objects in \(O\) into disjoint sets \(\tilde{O} = \{O_1, \ldots, O_n\}\) such that, for every \(1 \leq j < n\), \(\Sigma_{o_i \in O_j} a_i \leq B\).

For both the CC and the DSM models, there is a reduction from the bin packing problem, a well-known \(NP\)-complete problem [33].

**Definition 4.0.1 (Bin Packing).**

**Input:** An integer \(R\) (the bin size), and \(\ell\) items with sizes \(a_1, \ldots, a_\ell\).

**Question:** What is the minimum number \(M\) such that there is a partition of the \(\ell\) items into disjoint sets \(S_1, \ldots, S_M\) such that \(\Sigma_{i \in S_j} a_i \leq R\) for every \(j, 1 \leq j \leq M\)?

Fix \(R\) to be the block size \(B\), the number of objects to be \(\ell\) and the size of an object \(o_i\) be \(a_i\). We define a tree \(T = (V, E)\) whose vertexes are \(v_i\) for each object \(o_i\), and whose edges are \(E = \{(v_i, v_{i+1}) | 1 \leq i < \ell\}\). For processes \(p_1, p_2 \in P\), consider the access sequence \(\pi = (p_1, \text{write}, o_1), \ldots, (p_1, \text{write}, o_\ell), (p_2, \text{write}, o_1), \ldots, (p_2, \text{write}, o_\ell)\), in which \(p_1\) and \(p_2\) write to each object once.

An optimal algorithm for \(B\)-block placement packs the objects into the smallest number of blocks possible: Since the size of the local memory is unlimited, any block read into the local memory remains there until it is required by a different process in the CC and DSM with negligible invalidation cost models.

Therefore, the number of block RMRs is equal to twice the number of blocks in which the objects reside in the CC model and DSM with no invalidation models.

Thus, an algorithm that finds the optimal solution to the \(B\)-block placement problem also finds a partitioning of objects of sizes \(a_1, \ldots, a_\ell\) into blocks of size \(B = R\), which minimizes the number of blocks used. This yields an optimal solution to the bin packing problem.
problem. Since bin packing is NP-complete [33], so is $B$-blocking for objects of varying sizes.

Due to this result, the rest of the thesis concentrates on the case where all objects have the same size.
Chapter 5

Block RMRs in the CC model

In the cache coherence model, all objects are in the main memory before the access sequence is executed. Therefore, once the access sequence $\pi$ is fixed, no optimization can reduce the number of RMRs and it can be computed by a sequential pass over the access sequence $\pi$, while tracking the last process that modified each object. Specifically, for every access, if there was no previous access in the sequence to the same object by the process or if the previous access was a modification by a different process, we increase the number of RMRs by one and update the latest process to access the object.

Note that when each object is placed in a separate block, that is, in the assignment $\tilde{O} = \{O_1, \ldots, O_n\}$, with $|O_i| = 1$, for every $i$, $1 \leq i \leq n$, we have:

$$\#brmr^{CC}(\pi, \tilde{O}) = \#rmr^{CC}(\pi).$$

Therefore, $\#rmr^{CC}(\pi)$ is an upper bound on the minimal number of block RMRs for the sequence $\pi$.

Given a $B$-block placement of $O$, $\tilde{O} = \{O_1, \ldots, O_n\}$, let $\#rmr^{CC}(o_i, \pi)$ be the number of accesses to the object $o_i$ that incur an RMR, i.e., an access to $o_i$ that caused the block containing $o_i$ to be brought to the process’s local memory. For every access to $o_i$ that incurs an RMR, there is a block RMR that brings $o_i$ to the accessing process, incurred by either the access itself or a previous access to another object in the block. Therefore, the number of block RMRs for objects in $O_j$ is at least

$$\max_{o_i \in O_j} (\#rmr^{CC}(o_i, \pi)).$$

The average number of block RMRs per block (over all objects in the block) is smaller than the maximum for an object in that block. Since each block contains at most $B$ objects, the number of block RMRs is at least

$$\frac{\Sigma_{o_i \in O_j}(\#rmr^{CC}(o_i, \pi))}{B}.$$

The overall number of block RMRs is the sum of block RMRs for the objects in
each block $O_j$ and therefore, it is at least
\[ \sum_{O_j \in O} \frac{\sum_{o_i \in O_j} (\#rmr^{CC}(o_i, \pi))}{B} = \frac{\#rmr^{CC}(\pi)}{B}. \]

Therefore, for every access sequence $\pi$,
\[ \frac{\#rmr^{CC}(\pi)}{B} \leq \min_{\tilde{O}} \#brmr^{CC}(\pi, \tilde{O}) \leq \#rmr^{CC}(\pi) \]

We consider the question of optimally placing objects into blocks, in order to minimize the number of block RMRs. We prove that for $B > 2$, the problem is NP-hard, even if the sequence of accesses is known in advance, by showing a polynomial-time reduction from the graph partitioning problem [36]. But first, we prove that there is a polynomial-time algorithm for this problem when $B = 2$.

5.1 A Polynomial Algorithm for 2-Block Placement

A 2-block placement can be found with an algorithm for finding a maximum weighted matching for a graph, which can be done in $O(|V|^2|E|)$ using linear programming [42].

**Definition 5.1.1 (Maximum Weighted Matching).**

**Input:** Undirected graph $G = (V, E)$, weights $w(e) \in Q$ for each edge $e \in E$.

**Question:** Which matching, i.e., a set of pairwise non-adjacent edges, has maximum weight (the sum of the weights of the edges in the matching)?

Given an input to the 2-block placement problem, we take the complete graph $G = (V, E)$ with a vertex $v_i \in V$ for every object $o_i \in O$ and an edge between every pair of vertexes.

The edge weights are calculated according to the access sequence $\pi$. For each access by process $p_i$ to an object $o_1$ and every object $o_2$, we determine whether the access would incur an RMR if $o_1$ and $o_2$ are placed in the same block. To do this, we check if $o_2$ was previously accessed by $p_i$. If there is a previous access to $o_2$ by $p_i$, we check whether an access by another process in between could invalidate the block in the local memory and decrease the weight of the edge between $o_1$ and $o_2$. Otherwise, we check which previous access could save an RMR if $o_1$ and $o_2$ are placed in the same block and increase the weight of the edge between $o_1$ and $o_2$. For example, Figure 5.2 shows the final weights for the next access sequence on four objects $A, B, C, D$:

$$(p_1, \text{write}, A)(p_1, \text{write}, B)(p_2, \text{write}, C)(p_3, \text{write}, D)(p_2, \text{write}, C)$$

The weight of $(A, B)$ is 1 because placing them in the same block would save a block RMR on the access to $B$. The weight of $(C, D)$ is -1 because placing them in the same block would incur an extra block RMR on the second access to $C$ (which does not happen if $C$ is placed in a singleton block).
After the weights are calculated, we find a maximum matching in the weighted graph. A matching and a 2-block placement naturally correspond to each other: the endpoints of edges in the matching represent disjoint pairs of objects, and each pair can be placed in a single block in the memory. The remaining objects are placed in singleton blocks.

The weights are calculated according to the access sequence $\pi$ (Algorithm 5.1). We initialize the weight for each edge to zero. Next we go over the access sequence, and for each access to an object, check which previous accesses to other objects could either increase or decrease the number of RMRs if they are put in the same block as the current object. For each access $(p_{ih}, a_{jh}, o_{jh}) \in \pi$, let $(p_{ik}, a_k, o_{jh}) \in \pi$ be the previous access to the same object in $\pi$. If no such access exists, for example, if $(p_{ih}, a_{jh}, o_{jh})$ is the first access to the object, we look for previous accesses by the same process. For every object $o$, whose most recent access (prior to the $h$-th access) is by $p_{ih}$, we increase the weight of $(o, o_{jh})$ by 1, since placing them in the same block will avoid a block RMR on the access $(p_{ih}, a_{jh}, o_{jh})$.

Now we assume $(p_j, a_j, o_j)$ exists. If $p_j = p_i$, then for each object $o'$, if there is a process $p' \neq p_i$ such that there is an index $k$ such that $j < k < i$ and $p_k = p'$ and $o_k = o'$, and this is the most recent write to $o'$ prior to the $i$-th access, we decrease the weight of the edge $(o', o_i)$ by 1. For example, for three processes, objects $O = \{A, B, C, D\}$, and the sequence:

$$(p_1, \text{write}, A), (p_2, \text{write}, B), (p_1, \text{write}, C), (p_3, \text{write}, B), (p_1, \text{write}, A),$$

the weight of the edge $(A, B)$ will be decreased by 1, since there is a write to $B$ by a process other than $p_1$ between the two accesses to $A$. Intuitively, if $A$ and $B$ are placed in the same block then we will incur an extra RMR for this part of the sequence, which would not be incurred had they been placed in different blocks.

If, on the other hand, $p_j \neq p_i$, then for each object $o'$ such that there is an index $k$, $j < k < i$, $p_k = p_j$ and $o_k = o'$, and this is the most recent write to $o'$ prior to the $i$-th access, we increase the weight of the edge $(o', o_i)$ by 1. For example, for the sequence:

$$(p_4, \text{write}, D), (p_2, \text{write}, B), (p_1, \text{write}, C), (p_2, \text{write}, B), (p_2, \text{write}, D)$$
Algorithm 5.1 Sequential pseudocode for calculating the weights.

```plaintext
calc_weights(G = (V, E), w, acc)
1: init w to zeros
   // latest and latest_write are arrays of size |V| representing,
   // respectively, the most recent access and write to an object
2: init latest to nulls
3: init latest_write
4: for i = 1...|acc|
5:   (curr_proc, curr_act, curr_obj) = acc[i]
6:   prev_acc = latest[curr_obj]
7:   if (prev_acc is null):
8:      // Find which objects were previously accessed by the
8:      // same process, and increase the corresponding weight
9:      for obj = 1...|V|:
10:     if (obj ≠ curr_obj and latest[obj] ≠ null):
11:        (proc, temp1, temp2) = acc[latest[obj]]
12:           if (proc == curr_proc):
13:              w((obj, curr_obj)) += 1
14:        else:
15:           prev_proc = acc[prev_acc][0]
16:       for obj = 1...|V|:
17:          // Check if the obj was accessed between the previous access
18:          // to curr_obj and the current access
19:          if (obj ≠ curr_obj and latest[obj] ≠ null
20:             and latest[obj] > prev_acc):
21:             (proc, temp1, temp2) = acc[latest[obj]]
22:                // Putting obj and current_obj together can save a block RMR
23:             if (prev_proc ≠ curr_proc
24:                 and proc = curr_proc):
25:                w((obj, curr_obj)) += 1
26:          else if (latest_write[obj] ≠ null
27:             and latest_write[obj] > prev_acc
28:             and prev_proc = curr_proc
29:             and acc[latest_write[obj]][0] ≠ curr_proc):
30:                w((obj, curr_obj)) -= 1
31:       // Update latest and latest_write according to the access
32:      latest[curr_obj] = i
33:      if curr_act is a write:
34:         latest_write[curr_obj] = i
```

20
the weight of the edge \((B, D)\) will be increased by 1, since there is a write to \(B\) by process \(p_2\) between the two writes to \(D\) by \(p_2\). Intuitively, if \(D\) and \(B\) are placed in the same block then we will incur one less RMR for this part of the sequence than would have been incurred had they been placed in different blocks.

Figure 5.1 shows the final weights of the graph for the sequence that is the concatenation of the two previous sequences (all actions are writes):

\[
(p_1, A), (p_2, B), (p_1, C), (p_3, B), (p_1, A), (p_4, D), (p_2, B), (p_1, C), (p_2, B), (p_2, D)
\]

**Lemma 5.1.2.** If the matching \(M_{\tilde{O}} \subseteq E\) corresponds to a 2-block placement \(\tilde{O} = \{O_1, \ldots, O_{\ell}\}\), then \(#\text{rmr}^{CC}(\pi) - \sum_{e \in M_{\tilde{O}}} w(e) = \#\text{brmr}^{CC}(\pi, \tilde{O})\).

*Proof.* Consider some block \(O_j \in \tilde{O}\). If \(O_j = \{a\}\) is a singleton block, then the number of RMRs incurred by access to \(a\) is \(#\text{rmr}^{CC}(a)\), since accesses to other objects do not cause it to move to a different cache. On the other hand, if \(O_j\) is mapped to some edge \(e_{O_j} \in M_{\tilde{O}}\) it contains two objects, \(a\) and \(b\). We count the block RMRs incurred by accesses to \(a\) and \(b\). For every access to \(a\) (and similarly \(b\)) there are three cases:

1. The access is counted as an RMR in \(#\text{rmr}^{CC}(\pi)\) and a block RMR in \(#\text{brmr}^{CC}(\pi, \tilde{O})\).

2. The access is counted as an RMR in \(#\text{rmr}^{CC}(\pi)\), but not in \(#\text{brmr}^{CC}(\pi, \tilde{O})\). This happens only if the access to \(a\) was preceded by an access to \(b\), that brought \(O_j\) to the accessing process’ cache. This can happen in either the first access to \(a\) or in subsequent ones. We note that for each such occurrence we increased \(w(e_{O_j})\) by 1.

3. The access is counted as an RMR \(#\text{brmr}^{CC}(\pi, \tilde{O})\), but not in \(#\text{rmr}^{CC}(\pi)\). This happens only if the access to \(a\) was preceded by an access to \(b\), which moved the block \(O_j\) from the process’ cache. We note that for each such occurrence we decreased \(w(e_{O_j})\) by 1. \(\square\)

Therefore, the number of block RMRs incurred by \(a\) and \(b\) is exactly

\[#\text{rmr}^{CC}(a) + \#\text{rmr}^{CC}(b) - w(e_{O_j}).\]

Summing over all blocks, we get:

\[\#\text{brmr}^{CC}(\pi, \tilde{O}) = \sum_{O_j \in \tilde{O}, O_j = \{a\}} (#\text{rmr}^{CC}(a)) + \sum_{O_j \in \tilde{O}, O_j = \{a, b\}} (#\text{rmr}^{CC}(a) + #\text{rmr}^{CC}(b) - w(e_{O_j})).\]

And thus:

\[#\text{rmr}^{CC}(\pi) - \sum_{e \in M_{\tilde{O}}} w(e) = \#\text{brmr}^{CC}(\pi, \tilde{O}).\]

Given this lemma, it is easy to see that if \(\sum_{e \in M_{\tilde{O}}} w(e)\) is maximized, then our target function \(#\text{brmr}^{CC}(\pi, \tilde{O})\) is minimized. Finding the maximum for \(\sum_{e \in M_{\tilde{O}}} w(e)\) is finding a maximum weight matching in a weighted graph, which can be solved in
Figure 5.2: Graph with original weights (left) and after incorrect weight adjustment (right)

$O(|V|^2 \cdot |E|)$ steps [42]. Since $|V| = |O|$ and the graph is complete, this is in $O(|O|^3)$. The total complexity of the algorithm depends on the calculation of edge weights. Using a straightforward approach, for every access we must go over all previous accesses and update an edge to every object, resulting in an $O(|\pi|^2 + |\pi| \cdot |O| + |O|^3)$ time complexity. This can be improved by using a hash table to remember the last read and write for each object while going over the access sequence. This alleviates the need to go over all previous accesses; instead, we go over the objects and find which were accessed in the relevant part of the sequence. This results in $O(|\pi| \cdot |O| + |O|^3)$ time complexity.

The algorithm does not guarantee that between two solutions with the same minimal number of block RMRs, it chooses the one with the smallest number of blocks. This can be done by maximizing the number of blocks containing two objects, or equivalently, maximizing the number of edges in the matching. Therefore, we look for a maximum-weight matching with as many edges as possible.

We note that any such solution cannot contain negative-weighted edges, since removing them would increase the total weight. Note also that removing 0-weighted edges still leaves a maximum-weight matching. Therefore, the maximum-weight matching with the maximal number of edges is comprised of a maximum-weight matching with as many 0-weighted edges added to it as possible. By adding a small positive weight to the 0-weighted edges before running the maximum-weight matching algorithm, we can ensure that as many such edges as possible will be added to the solution, since adding them will only increase the total weight. This weight must be the same for all edges, since we do not prefer one 0-weighted edge over the other. In addition, the weight must be small enough to ensure that the new solution contains some original maximum-weight matching as a subset. For example, every maximum-weight matching for the graph of the sequence:

$$(1, A)(1, B)(2, C)(3, D)(2, C)$$

must contain the edge $(A, B)$, since it is the only edge with positive weight. If the weight added to 0-weighted edges is 1, as shown in Figure 5.2, then a maximum-weight
matching on this graph is either the edges \((A, D), (B, C)\) or \((A, C), (B, D)\), neither of which contain the edge \((A, B)\). However, if we give all the 0-weighted edges a weight \(< \frac{1}{|E|}\) (for example \(\frac{1}{|V|}\)), the total weight of all the 0-weighted edges is smaller than 1. Since all the positive-weighted edges weigh at least 1, any combination of originally 0-weighted edges weighs less than the weight of those edges. Therefore, the solution found by the algorithm must contain an original maximum-weight matching as a subset, otherwise, such a matching will produce a higher total weight, contradicting the algorithm’s optimality in finding a maximum-weight matching. This implies the next theorem:

**Theorem 5.1.** In the CC model, there is a polynomial algorithm for finding a 2-block placement, with the minimal number of block RMRs for a given access sequence, while minimizing the total number of blocks used.

### 5.2 Hardness Proof for \(B\)-Block placement, \(B > 2\)

We now prove a hardness result for \(B\)-block placement with \(B > 2\), by showing a polynomial-time reduction from the graph partitioning problem, known to be \(NP\)-complete, even for a fixed \(K \geq 3\) and even if all vertex and edge weights are 1 \([36]\).

**Definition 5.2.1 (Graph Partitioning).**

**Input:** Undirected graph \(G = (V, E)\), weights \(w(v) \in \mathbb{Z}^+\) for each vertex \(v \in V\) and \(l(e) \in \mathbb{Z}^+\) for each edge \(e \in E\), positive integers \(K\) and \(J\).

**Question:** Is there a partition of \(V\) into disjoint sets \(V_1, \ldots, V_m\), such that \(\Sigma_{v \in V_i} w(v) \leq K\) for \(1 \leq i \leq m\) and such that if \(E' \subseteq E\) is the set of edges that have their two endpoints in two different sets \(V_i\) then \(\Sigma_{e \in E'} l(e) \leq J\)?

We redefine \(B\)-block placement as a decision problem.

**Definition 5.2.2 (B-Block Placement Decision).**

**Input:** Two positive integers \(B\) and \(R\), a set of \(n\) processes \(P = \{p_1, \ldots, p_n\}\), a set of memory objects \(O\), a sequence of accesses \(\pi = (p_1, o_1), \ldots, (p_m, o_m)\) such that for every \(i\), \(1 \leq i \leq m\), \(p_i \in P\) and \(o_i \in O\).

**Question:** Is there a partition of the objects in \(O\) into disjoint sets \(\hat{O} = \{O_1, \ldots, O_k\}\) such that \(|O_i| \leq B\) and \(#brm^{CC}(\pi, \hat{O}) < R\)?

Given that it is easy to calculate the number of RMRs, and it is an upper bound on the number of block RMRs, a polynomial time algorithm for the decision problem can be used in conjunction with a binary search to solve the optimization problem.\(^1\). Therefore, if we prove that the decision problem is \(NP\)-hard, the minimization problem is also \(NP\)-hard.

\(^1\) That is, what is the minimal number of block RMRs for the sequence \(\pi\). An optimal placement of objects to blocks is not found in this way.
We will use the input graph for which partitioning must be found as the underlying data structure that two processes access simultaneously. Each process performs a traversal and accesses the edges in DFS order. Processes access the edges in round-robin order, each accessing the endpoints of an edge one after the other, before control is passed to the next process. This access pattern ensures that putting the two endpoints of an edge in the same block results in fewer block RMRs than if they are in different blocks. Therefore, given a placement with fewer block RMRs, the partition into blocks induces a partition of the graph into disjoint sets of vertexes, which gives a good solution to the graph partitioning problem. Conversely, we prove that if there is no such placement, then there is no valid solution to the graph partitioning problem.

In more detail, the input is an undirected graph \( G = (V, E) \), in which all edge and vertex weights are 1, and positive integers \( K \geq 3 \) and \( J \).

We take two processes \( p_1 \) and \( p_2 \) and an object \( o \) for each vertex \( v \in V \). For each edge \( e \in E, e = (v_i, v_j) \) and process \( p \), we define a subsequence

\[
\pi_{(p, e)} = (p, \text{read}, o_i)(p, \text{write}, o_i), (p, \text{read}, o_j)(p, \text{write}, o_j).
\]

Consider the following traversal in DFS order of the edges of the graph: The traversal starts at an arbitrary vertex. When the traversal reaches a node, it either immediately retreats through the same edge, if the node was already visited, or it continues to visit the node’s neighbors, and then retreats through the same edge. Therefore, each edge is visited twice during the traversal, and neighboring vertexes are accessed one after the other. Let \( e_1, \ldots, e_{2|E|} \) be the sequence of edges in this traversal.

We define \( \pi = \bigcirc_{1 \leq j \leq 2|E|} \pi_{e_{ij}} \). For each process \( p \), the sequence of accesses is \( \bigcirc_{1 \leq j \leq 2|E|} \pi_{e_{ij}} \), which is a traversal due to the choice of the order of the edges. The length of the access sequence \( \pi \) is in \( O(|E|) \), since it is a concatenation of \( 2|E| \) constant size sequences, and therefore, the reduction is polynomial in the size of the input. Let \( B = K \) and \( R = 4(|E| + J) \).

**Lemma 5.2.3.** There is a \( B \)-block placement of \( O \) for sequence \( \pi \) with \( R \) or fewer block RMRs if and only if there is a partitioning of the graph \( G \) under the \( K \) and \( J \) weight sum constraints.

**Proof.** Given a graph and a partition \( V_1, \ldots, V_m \) of the vertexes such that \( \Sigma_{v \in V_i} w(v) \leq K \), we define a \( B \)-block placement for \( B = K \) where \( O_i \) contains all the objects that correspond to the vertexes in \( V_i \). Since all weights are 1, the number of vertexes in \( V_i \) is at most \( K \), and so is the number of objects in \( O_i \). Similarly, a \( B \)-block placement of the objects induces a partition of the corresponding vertexes that satisfies the \( K = B \) constraint on the graph.

**Claim 5.2.4.** Given a \( B \)-block placement, and the corresponding graph partitioning, if \( E' \) is the set of edges with endpoints in different sets \( V_i \), then the number of block RMRs is \( 4(|E| + |E'|) \).
Proof. After each subsequence $\pi_e$, each block is either still in the main memory or in the cache of process $p_2$, since $p_2$ modifies each block after $p_1$ modifies it. Therefore, if $e = (v_i, v_j)$ and $o_i$ and $o_j$, the corresponding objects, are in the same block, then the number of block RMRs for $\pi_e$ is exactly 2, since each process must bring the block into its cache (including the first process) in order to access $o_i$, and then proceed to access $o_j$. Otherwise, if $o_i$ and $o_j$ are in different blocks, the number of block RMRs for $\pi_e$ is exactly 4, since each process must access both blocks in turn. Therefore, since every edge is traversed twice, the total number of block RMRs is $4(|E \setminus E'| + 2 \cdot 4|E'| = 4(|E| + |E'|).

It remains to show that a solution to the $B$-block placement problem implies a solution to the graph partitioning problem. If there is no $B$-block placement for $B = K$ and the sequence $\pi$ with $R = 4(|E| + J)$ or fewer block RMRs, then for every placement and corresponding graph partitioning, the number of block RMRs is $4(|E| + |E'|) > R = 4(|E| + J)$ and therefore, $|E'| > J$. Hence, there is no graph partitioning such that the size of every group is $K$ or less and the number of edges between different groups is $J$ or less.

Conversely, if there is a $B$-block placement for $B = K$ and the sequence $\pi$, with fewer than $R = 4(|E| + J)$ RMRs, then $4(|E| + |E'|) \leq R = 4(|E| + J)$, and $|E'| \leq J$. Therefore, the corresponding partitioning of $G$ satisfies the constraints on $K$ and $J$.

This proves the next theorem:

**Theorem 5.2.** In the CC model, the $B$-block placement decision problem, for $B \geq 3$, is NP-hard.
Chapter 6

Block RMRs in the DSM model

This chapter focuses on the different DSM models of counting RMRs, where all objects are of the same size. In the DSM model, the object is created in the local memory of one of the processes. Therefore, in the DSM model, as opposed to the CC model, the number of RMRs does not depend solely on the sequence $\pi$, but also on the decision which object is created by which process. In addition, we must take into account the different options for the support and cost of invalidating data.

6.1 DSM with Negligible Invalidation Cost

We note that the only difference between this model and the CC model is the number of RMRs incurred by the first access to every block: If the first access is done by process $p$ that accesses $o$ and $p = creator(o)$, then the step causes one RMR, and 0 otherwise. Therefore the number of RMRs depends on the choice of $creator(o)$ for these accesses, while other accesses are not affected.

We would like to show that the same results for the CC model applies to this model. The algorithm for blocks which may contain at most 2 objects can be adjusted to this model by changing the edge weight to account for the objects being created in the process’ local memory instead of the main memory. For the hardness proof we will create a different problematic access sequence.

Given the parameters to the graph partitioning problem, the parameters to the $B$-Block mapping are very similar: The input is an undirected graph $G = (V, E)$, weights $w(v) \in Z^+$ for each vertex $v \in V$ and $l(e) \in Z^+$ for each edge $e \in E$, and positive integers $K$ and $J$. We assume that all weights are 1, and $K \geq 3$.

The access sequence is somewhat different: We take two processes $p_1$ and $p_2$, and an object $o$ for each vertex $v \in V$. For each edge $e \in E, e = (v_i, v_j)$ and process $p$, we define a subsequence

$$\pi_{(p,e)} = (p, \text{read}, o_i)(p, \text{write}, o_i), (p, \text{read}, o_j)(p, \text{write}, o_j)).$$
Let \( \pi_e = (\pi_{(p_1, e)}, \pi_{(p_2, e)})^{11} \). We define \( \pi = \bigcirc_{e \in E} \pi_e \). The length of \( \pi \) is \( O(|E|^2) \) and therefore, the reduction is in polynomial time. Let \( B = K \) and \( R = 4|E|(|E| + J) - |V| \).

Given the part of the access sequence corresponding to an edge \( e = (v_i, v_j) \),

\[
\pi_e = ((p_1, \text{write}, o_i), (p_1, \text{write}, o_j), (p_2, \text{write}, o_i), (p_2, \text{write}, o_j))^{11}, \]

the number of block RMRs is as follows: If both endpoints of \( e \) are in the same block, then every access by \( p_1 \), except perhaps the first one, incurs a block RMR. This is because at the end of \( \pi_{(p_2, e)} = (p_2, \text{write}, o_i), (p_2, \text{write}, o_j) \), the block is in \( p_2 \)'s local memory. The total number of block RMRs is \( 2|E| \), from which we reduce 1 if this is the first access to the block containing \( o_i \) and \( o_j \).

On the other hand, if the endpoints of \( e \) are in different blocks, then every access by either \( p_1 \) or \( p_2 \) must move a block into its local memory, except perhaps the first two accesses of \( p_1 \). The total number of block RMRs is \( 4|E| \), minus 1 or 2, depending on whether or not this sequence accessed blocks for the first time.

Since at the end of each subsequence \( \pi_e \) all blocks that were already accessed are in the local memory of process \( p_2 \), if \( |E'| \) is the number of edges with endpoints in different blocks, the total number of RMRs is at most

\[
2|E| \cdot 2|E \setminus E'| + 4|E| \cdot 2|E'| = 4|E|(|E| + |E'|) \]

and at least

\[
2|E| \cdot 2|E \setminus E'| + 4|E| \cdot 2|E'| - |V| = 4|E|(|E| + |E'|) - |V|, \]

where \( |V| \) is the maximum possible number of blocks.

If there is no \( B \)-block ent for \( B = K \) and the sequence \( \pi \) with \( R = 4|E|(|E| + J) - |V| \) or fewer block RMRs exists, then for every placement and corresponding graph partitioning the number of block RMRs is at least

\[
4|E|(|E| + |E'|) - |V| > R = 4|E|(|E| + J) - |V| \]

and therefore \( |E'| > J \). Hence, there is no graph partitioning such that the size of every group is \( K \) or less and the number of edges between different groups is \( J \) or less.

Conversely, if there is a \( B \)-block placement for \( B = K \) with \( S \) blocks, and the sequence \( \pi \), induces fewer than \( R = 4|E|(|E| + J) \) block RMRs, then

\[
4|E|(|E| + |E'|) - S \leq R = 4|E|(|E| + J) - |V|, \]

and therefore

\[
4|E| \cdot |E'| \leq 4|E| \cdot J - (|V| - S). \]

Since \( |V| - S > 0 \), we have \( |E'| \leq J \). Therefore, the corresponding partitioning of \( G \)
holds under the constraints on $K$ and $J$.

This proves the following theorem:

**Theorem 6.1.** In the DSM model with cache coherence and negligible invalidation, the $B$-block placement decision problem is NP-hard, for $B \geq 3$.

For $B = 2$, there is a polynomial time algorithm that finds an optimal $B$-block placement, which is very similar to the algorithm for the CC model (section 5.1) with a small change in the computation of the edge weights to account for the creator process.

### 6.2 DSM Without Invalidation

In the DSM model without invalidation, when each object is accessed separately, it is best to create each object by the process that accesses it the most, since one RMR is incurred every access from the processes that do not hold the object in its local memory. As in the CC model, the solution of putting one object in each block and choosing the process that accesses the object the most to create it is legal, and therefore the minimum number of block RMRs over all placements is at most the minimum number of RMRs.

The next lemma relies on the fact that the minimal number of RMRs for an access sequence depends only on the number of times each process accesses each object and not on the order of accesses. Denote by $w_{ij}(\pi)$, the number of times an object $o_i$ is accessed by process $p_j$ during an access sequence $\pi$.

**Lemma 6.2.1.** Let $\tilde{O} = \{O_1, \ldots, O_n\}$ be a $B$-block placement and let $(O_i)$ be the creator process of $O_i$ for each $1 \leq i \leq n$. If $O_i$ is placed in the cache of process $p$ then $o_i$ contributes $\sum_{j \neq p} w_{ij}(\pi)$ block RMRs during $\pi$.

**Proof.** Let $\pi = (p_{i_1}, a_{i_1}, o_{i_1}), \ldots, (p_{i_m}, a_{i_m}, o_{i_m})$ and let $\pi' = (p_{j_1}, a_{j_1}, o_{j_1}), \ldots, (p_{j_m}, a_{j_m}, o_{j_m})$ be a reordering of $\pi$. We argue that the number of RMRs for both sequences is the same, i.e.,

$$\#brmr_{\text{DSM-No-CC}}(\pi, \tilde{O}) = \#brmr_{\text{DSM-No-CC}}(\pi', \tilde{O}).$$

By definition,

$$\#brmr_{\text{DSM-No-CC}}(\pi) = |\{(p_{i_h}, a_{i_h}, o_{i_h}) \in \pi | o_{i_h} \in O_\ell \text{ and } p_{i_h} \neq \text{creator}(O_\ell)\}|.$$

and

$$\#brmr_{\text{DSM-No-CC}}(\pi') = |\{(p_{j_h}, j_{h}, o_{j_h}) \in \pi | o_{j_h} \in O_\ell \text{ and } p_{j_h} \neq \text{creator}(O_\ell)\}|.$$

Since $\pi'$ is a reordering of $\pi$, every access in $\pi$ has a corresponding access to the same object by the same process in $\pi'$. Therefore, each object $o$ is accessed by each process
Let \( \tilde{C} = \{C_1, \ldots, C_\ell\} \) be a partitioning of the objects in \( O \) into \( n \) disjoint sets. Let \( \tilde{O} = \{O_1, \ldots, O_n\} \) be a B-block placement and \( \text{creator} \) the associated owner processes. If every block \( O_i \) is fully contained in some \( C_j \) and \( \text{creator}(O_i) = p_j \) we say the B-block placement \((\tilde{O}, \text{creator})\) induces the partitioning \( \tilde{C} \). In other words, a partitioning \( \tilde{C} \) refers to which object reside in which local memory, regardless of blocks.

We now show that given \( \tilde{C} \), all cached B-block placements induce the same number of block RMRs.

Lemma 6.2.2. Consider two cached B-block placement \((\tilde{O}, \text{creator})\) and \((\tilde{O}', \text{creator}')\) that induce \( \tilde{C} \). For every access sequence \( \pi \),

\[
\#\text{brmr}^{\text{DSM-no-CC}}(\pi, \tilde{O}, \text{creator}) = \#\text{brmr}^{\text{DSM-no-CC}}(\pi, \tilde{O}', \text{creator}').
\]

Proof. By definition,

\[
\#\text{brmr}^{\text{DSM-no-CC}}(\pi, \tilde{O}, \text{creator}) = \left| \{(p_{j_h}, j_h, o_{j_h}) \in \pi \mid o_{j_h} \in O_\ell \text{ and } p_{j_h} \neq \text{creator}(O_\ell)\} \right|.
\]

Similarly

\[
\#\text{brmr}^{\text{DSM-no-CC}}(\pi, \tilde{O}', \text{creator}') = \left| \{(p_{j_h}, j_h, o_{j_h}) \in \pi \mid o_{j_h} \in O'_\ell \text{ and } p_{j_h} \neq \text{creator}(O'_\ell)\} \right|.
\]

Given an object \( o_i \in C_j \), both \( \tilde{O} \) and \( \tilde{O}' \) place \( o_i \) in a block that is created by process \( p_j \). Lemma 6.2.1 implies that each object \( o_i \) contributes \( \Sigma_{h \neq j} w_{ih}^{\pi} \) block RMRs to the total number of block RMRs, both for \((\tilde{O}, \text{creator})\) and for \((\tilde{O}', \text{creator}')\). Therefore, the number of block RMRs is equal.

We conclude that the minimal number of RMRs does not depend on the partitioning into blocks, just on the choice of creator processes.

Given a partitioning \( \tilde{C} \) we define \((\tilde{O}_{\tilde{C}}, \text{creator}_{\tilde{C}})\) as some cached B block placement that induces \( \tilde{C} \). Calculating \((\tilde{O}_{\tilde{C}}, \text{creator}_{\tilde{C}})\) given \( \tilde{C} \) is trivial.

We now show an algorithm that finds a cached B-block mapping with a minimal number of block RMRs. Let \( \tilde{C} = \{C_1, \ldots, C_n\} \) be a partitioning of the objects in \( O \),
into $n$ disjoint sets, such that $o_i \in C_j$ if $w_{ij}(\pi) \geq w_{ih}(\pi)$, for $1 \leq h \leq n$. Namely, $o_i$ is placed in $C_j$ if $j$ reads $o_i$ the maximal number of times out of all the processes. If more than one process accesses $o_i$ $w_{ij}(\pi)$ times, choose the process with the smallest identifier (to make the algorithm deterministic). Let $\tilde{O}_{\text{greedy}} = \tilde{O}_c$ and $\text{creator}_{\text{greedy}} = \text{creator}_c$.

Lemma 6.2.3.

$$\#\text{brmr}^{\text{DSM-\text{No-CC}}} (\pi, \tilde{O}_{\text{greedy}}, \text{creator}_{\text{greedy}}) = \#\text{brmr}^{\text{DSM-\text{No-CC}}} (\pi, \tilde{O}_{\text{opt}}, \text{creator}_{\text{opt}}).$$

Proof. We consider the block RMRs from the access sequence. By Lemma 6.2.1, the number of block RMRs contributed by every object $o_i$, if $o_i$ is cached by process $p$ in $(\tilde{O}_{\text{greedy}}, \text{creator}_{\text{greedy}})$, is $\Sigma_{j \neq p} w_{ij}(\pi)$. Since $w_{ip}(\pi)$ is maximal by the definition of $(\tilde{O}_{\text{greedy}}, \text{creator}_{\text{greedy}})$, this is the minimal possible number of block RMRs for $o_i$. Therefore the total number of block RMRs in the access sequence is minimized by $(\tilde{O}_{\text{greedy}}, \text{creator}_{\text{greedy}})$. Therefore, the number of block RMRs given the cached $B$ block $(\tilde{O}_{\text{opt}}, \text{creator}_{\text{opt}})$ during the access sequence is the same number of block RMRs if $(\tilde{O}_{\text{greedy}}, \text{creator}_{\text{greedy}})$ is used. Thus,

$$\#\text{brmr}^{\text{DSM-\text{No-CC}}} (\pi, \tilde{O}_{\text{greedy}}, \text{creator}_{\text{greedy}}) = \#\text{brmr}^{\text{DSM-\text{No-CC}}} (\pi, \tilde{O}_{\text{opt}}, \text{creator}_{\text{opt}}).$$

Let us now compare the minimal number of RMRs,

$$\min_{\text{creator}} \{ \#\text{rmr}^{\text{DSM-\text{No-CC}}} (\pi, \text{creator}) \},$$

to the minimal number of block RMRs,

$$\min_{\tilde{O}, \text{creator}} \{ \#\text{brmr}^{\text{DSM-\text{No-CC}}} (\pi, \tilde{O}, \text{creator}) \}.$$

As discussed in the beginning of this section,

$$\min_{\tilde{O}, \text{creator}} \{ \#\text{brmr}^{\text{DSM-\text{No-CC}}} (\pi, \tilde{O}, \text{creator}) \} \leq \min_{\text{creator}} \{ \#\text{rmr}^{\text{DSM-\text{No-CC}}} (\pi, \text{creator}) \}$$

since we can always put each object in a different, singleton block and send it to the process that accesses it the most, which will result in the same number of block RMRs as the optimal solution for RMRs. We denote this solution by $(\tilde{O}_{\text{opt}}, \text{creator}_{\text{opt}}^s)$. Note that $(\tilde{O}_{\text{opt}}, \text{creator}_{\text{opt}}^s)$ and the greedy algorithm solution $(\tilde{O}_{\text{greedy}}, \text{creator}_{\text{greedy}})$ induce the same cached block placement. Therefore, by Lemma 6.2.2 the number of block RMRs induced by the greedy $B$-block placement is equal to the number of block RMRs induced by the optimal placement.

We conclude the following theorem:
Theorem 6.2. \( \#brm^{DSM-No-CC}(\pi, \tilde{O}_{opt}^s, creator_{opt}^s) = \#brm^{DSM}(\pi, \tilde{O}_{opt}, creator_{opt}) \).
Chapter 7

Conclusions and Future Directions

This thesis introduces a framework for studying the cost of accessing remote memory (whether shared memory or data stored at another process), which takes into account the fact that shared objects can be placed and moved together in larger blocks. We introduce a formal complexity measure, called block RMRs, for both the CC and the DSM models. Our main result shows that it is NP-hard to place objects into blocks in a way that minimizes the number of block RMRs, even for a fixed access sequence. The result holds for both the CC model and the DSM model with coherence and negligible invalidation cost, when a block can contain three objects or more.

In contrast, in the DSM model without invalidation, for a given access sequence it is simple to find an optimal placement, and to find the number of block RMRs.

In the common situation, however, the access sequence is not known in advance. Instead, we know it is from a family of sequences, typically, those generated by a particular concurrent algorithm, for example, interleavings of (partial) traversals or searches by a subset of the processes. It would be interesting to find block placement methodologies for such families, in a way that exploits the benefits of moving several objects together. Taking this a step further, it is interesting to look at probabilistic models where the access sequences are chosen from a family of sequences with a known distribution. The goal is to choose a $B$-block placement such that the expected number of block RMRs is minimized. In the DSM model without coherence, these problems may have simple solutions in the form of choosing the object’s creator to be the process that is expected to access it the most.

Our results assume that the block size $B$ is known, but in realistic applications, it is important to design algorithms that can work well regardless of $B$. As mentioned in Chapter 2, cache-oblivious algorithms [32] lay out objects in the memory in a way that ensures that a partitioning into blocks will have good locality, regardless of the block size and the size of the local memory. Although not designed with concurrency in mind, cache-oblivious algorithms may offer interesting insights for situations with low
contention among processes. When the contention between processes is high, it does not seem that the memory layout of cache-oblivious algorithms decreases the number of block RMRs. Studying this question is left for future research.

Our results assume that object sizes are smaller than $B$. Objects with size larger than $B$ would have to be placed in more than one block. If each object is placed in the minimal number of blocks that can contain it (i.e., all parts of the object except perhaps one are placed in a block alone), then our results can be replicated for this model as well by ignoring the parts of the object that fill full blocks and taking into account only the part of the object that remains and is smaller than a full block.

It would also be interesting to study the effects of bounding the local memory size, so it can hold a bounded number of blocks, and limiting the cache associativity. It is likely that in general, the problem is $NP$-hard due to our results as well as $[37, 41]$. For specific access sequences or sequence families it may be possible to find optimal $B$-block placements, where the DSM without invalidation is most likely to yield results.

It is interesting to consider approximations for optimal mappings. In the CC model, for a given sequence, the ratio between the number of block RMRs in the optimal solution and an arbitrary mapping may be $O(|\pi|)$. For example, for the sequence:

$$(p_1, write, o_1), (p_2, write, o_2), (p_1, write, o_1), (p_2, write, o_2), ...$$

the optimal solution which places each object in a singleton block incurrers two block RMRs and a solution that puts both objects in the same block costs $|\pi|$ block RMRs. Similarly, the optimal solution for one sequence can be far from optimal for a different sequence. For example the optimal solution for this sequence requires $o_1$ and $o_2$ be placed in the same block, which would cause $O(|\pi|)$ block RMRs for the sequence from the previous example:

$$(p_1, write, o_1), (p_1, write, o_2), (p_2, write, o_1), (p_2, write, o_2), ...$$

This is true even if the sequences are different interleavings of the same set of accesses, due to false sharing.
Bibliography


The document appears to be in Hebrew and contains mathematical and technical content. It seems to be discussing computer science topics, possibly related to computer architecture or algorithm design.

The text is not fully legible due to the quality of the image, but it appears to be discussing hardware or software design, with mentions of components like "block cache," "cache coherence," and "scalability." There is a reference to a thesis from the Technion - Computer Science Department, indicating that this document is an academic work.
תקציר

בעזרת מרובה ומגביה טיפוסיים, הכלים מתכשיטים ובאמעטת הגישה presets, מקיימות משך זמן אינטיקס שעונים מתכשיטים אינטיקס גיוון החללים אינטיקס גיוון החללים, המitorio על פיטיו. 

כליים פיתוי מאיצים ומיקרים המוקמים והמאיצים יכוונים בключа להחזרה על אינטיקס הколоורית, ואינו גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

ש.butは何 gypsum אינטיקס שיקוף בelfare הגישה למקורות הколоורית בעושה אינטיקס האינטיקס, תמיכה למגבים בqueda וברבורה גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

יש עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

ש.butは何 gypsum אינטיקס שיקוף ב Welfare הגישה למקורות הколоורית בעושה אינטיקס האינטיקס, תמיכה למגבים בqueda וברבורה גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

יש עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

ש.butは何 gypsum אינטיקס שיקוף ב Welfare הגישה למקורות הколоורית בעושה אינטיקס האינטיקס, תמיכה למגבים בqueda וברבורה גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

יש עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

ש.butは何 gypsum אינטיקס שיקוף ב Welfare הגישה למקורות הколоורית בעושה אינטיקס האינטיקס, תמיכה למגבים בqueda וברבורה גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

יש עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

ש.butは何 gypsum אינטיקס שיקוף ב Welfare הגישה למקורות הколоורית בעושה אינטיקס האינטיקס, תמיכה למגבים בqueda וברבורה גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

יש עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

ש.butは何 gypsum אינטיקס שיקוף ב Welfare הגישה למקורות הколоורית בעושה אינטיקס האינטיקס, תמיכה למגבים בqueda וברבורה גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

יש עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

עיבודים רחוב שאר ישיק פעולות שאר אינטיקס יבוכי עץ ופשיטה למגבים אינטיקס גיוון החללים אינטיקס גיוון החללים, המבירים את האינטיקס. 

ש.butは何 gypsum אינטיקס שיקוף ב Welfare הגישה למקורות הколоורית בעושה אינטיקס האינטיקס, תמיכה لمג

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.

un清凉איאווככב, אוזיבי ייאפיע ארלי כהירה ביינור המקומן של הרצל יפר ארצות.
תודה
ברצוני להודות駈צהתוים לימיםפים:

אני מודה ל粜נויי על התרומתה הב StringField בזידנה המของเขาולים.
גיסות לידכון משותף ביהדות של בלקים

היבר על מחקר

ולש מחאת חלקי שלachers בקטלט יהודה
מדעי ה phêס

גיל יבנה

רוחש לסנטה טוסט - מחנכת טכנולוגיה לישראל
אמיר רכשלא - חפץ יוני 2017
ניתוח ליזיכרון משותף בינידות של בלוקים

גילי יבנה