LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-465

MEMORY ASSIGNMENT FOR MULTIPROCESSOR CACHES THROUGH GRAPH COLORING

Anant Agarwal
John Guttag
Marios Papaefthymiou

February 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Memory Assignment for Multiprocessor Caches through Graph Coloring

(Extended Abstract)

Anant Agarwal
John Guttag
Marios Papaefthymiou

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

February 20, 1992

Abstract

It has become apparent that the achieved performance of multiprocessors is heavily dependent upon the quality of the available compilers. In this paper we are concerned with compile-time techniques that can be used to achieve better performance by improving cache utilization. Specifically, we investigate the problem of assigning data chunks to memory in a way that will minimize collisions in direct-mapped multiprocessor caches. We show that while this problem is computationally intractable, there are interesting special cases that can be solved in polynomial time. We also present several techniques that can be used when conflict-free assignment is not possible, or when finding a conflict-free assignment is computationally infeasible. These techniques include uniform decaching, which involves not caching specific data blocks, and data replication, which involves making multiple copies of read-only data. Finally, we present a memory assignment technique, grey coloring, that reduces latency in the presence of collisions by distributing cache misses among processors in a way that minimizes the total number of cache misses in any specific cache.

Keywords: multiprocessors, compilers, cache coherence, direct-mapped caches, data partitioning, graph-coloring.

Authors' e-mail: agarwal@mit.edu, guttag@mit.edu, marios@theory.lcs.mit.edu.

1 Introduction

Multiprocessors that support the shared-memory programming model provide the abstraction of a single coherent memory that is equally easily and equally efficiently accessible by multiple processors. Processors communicate with each other by reading and writing the common memory. Typically, the shared-memory abstraction is implemented by a large amount of physical memory, accessed through a network. In bus-based machines, the memory is implemented as a single module accessed over the bus, while in most large-scale machines the memory is physically distributed among all the processing nodes. However, virtually all machines that support the shared-memory programming abstraction provide local caches at each processor [18, 23, 19, 13, 16, 2]. Caches automatically replicate memory locations close to the processor and avoid expensive network traversals for most memory accesses.

The existence of caches does not change the memory abstraction. The hardware is responsible for maintaining the illusion that all reads and writes take place directly to main memory. This illusion is called *cache transparency.* Maintaining it involves detecting and handling attempts to read or write from a location not contained in the cache. Such attempts are called *misses.* When a miss occurs, the hardware reads a *line* from main memory into the cache. A line typically contains four or eight consecutive words of main memory.

Caches can be either fully associative, direct-mapped, or n-way set associative. They differ in the way in which the hardware decides which (if any) cache entry corresponds to a particular location in the shared memory. In an associative cache, tag bits are associated with each entry. These bits indicate which part of main memory the entry corresponds to. When a processor attempts to access a location in main memory the hardware checks all the tag fields. In a fully associative cache, these fields are all checked simultaneously. The problem with fully associative caches is that they are extremely expensive to build; so expensive, that they are rarely used in practice.

In a direct-mapped cache, each line in the main memory corresponds to a unique entry in the cache memory. If the direct-mapped cache has a total of S lines, this entry is specified by the $log_2(S)$ low order bits of the line's address in the main memory. The problem with a direct-mapped cache is that when a processor accesses two or more locations in the main memory that correspond to the same location in the cache, there is a conflict that leads to cache misses and performance degradation.

An n-way set-associative cache is most conveniently thought of as n direct-mapped caches operating in parallel. Experience indicates that the performance of a cache of size m with associativity n will be roughly comparable in performance to a direct-mapped cache of size $m \times n$ [3]. The set-associative cache, however, is likely to have a longer access time [14, 20]. Throughout this paper we will deal explicitly only with direct-mapped caches. However, our results can be easily translated to deal with n-way set associative caches.

Accessing data in a local cache is many times faster than accessing the shared memory. Therefore, the cycles per instruction (*CPI*) achieved is governed largely by the number of times a processor is forced to wait while data is fetched from the common memory and by the ratio *processor speed/memory speed.* Since processor speeds are increasing faster than memory speeds, the cost of a cache miss is growing.

1.1 Data Blocks versus Cache Lines

This paper deals with the problem of reducing cache misses by avoiding cache collisions between *blocks* of data or code accessed by the processor. In considering cache conflicts, we will deal with conflicts between blocks or chunks of data as opposed to conflicts between individual cache lines. Since most programs exhibit considerable spatial locality, compilers attempt to organize memory so that locations that are often accessed together (e.g., a row of a matrix) lie in a contiguous *chunk* of memory.[1] Because contiguous lines in memory do not conflict with each other, a large fraction of the conflicts between cache lines will fall into equivalence classes of conflicts between corresponding data segments. Dealing with conflicts between chunks instead of individual cache-line conflicts simplifies greatly the combinatorial complexity of finding conflict-free assignments.

Accordingly, we will specify cache size in terms of the maximum number of data blocks a cache can hold. Let l be the size of each data block measured in lines, and let us assume for simplicity that all data blocks are of the same size. Let k be the number of *sets* in each cache, that is, the maximum number of data blocks that can be stored in it. We emphasize that k, the number of cache sets, is measured in terms of the number of data blocks a cache can store rather than in terms of the number of cache lines. For example, suppose that the cache size is 64K words. If data blocks are 2K words, then our analysis will use $k = 64/2 = 32$.

1.2 The Memory Assignment Problem

When two or more data blocks in a processor's working set map to the same location in the processor's direct-mapped cache, processor performance degrades due to the resulting conflict misses. The extent of this degradation grows in severity with the number of processors for two reasons.

First, the probability of a conflict increases with the number of processors. In fact it is easy to show (see Appendix A) that if the assignment of data blocks to memory is oblivious to cache conflicts, the probability that the entire machine suffers no conflicts diminishes exponentially with the number of processors. We further show in Appendix A that, for reasonable parameters, at least one conflict will almost certainly occur in large machines.

Second, the relative performance impact of such collisions is more serious as the number of processors increases. For many parallel algorithms, if one processor is running slowly, the whole computation is disrupted—leading to long latency. This phenomenon is illustrated by the graph of Figure 1. This graph gives the activity profile of a matrix multiply program obtained on a simulator of the Alewife machine [2]. After a start-up transient (not shown in the graph), the number of operations going on in parallel becomes high. Subsequently, however, at around 1.1 Mcycles this number decreases drastically and remains at low levels for most of the rest of the computation. When we first observed this behavior we were mystified as to its cause. Subsequent investigation revealed that the cause was persistent collisions in the cache of one of the processors (see Figure 2). Since the other processors had to wait for the slow processor to complete, the whole computation was stalled.

The frequency of collisions in multiprocessor caches is governed largely by the way processing and data is partitioned among the processors. There are two fundamentally different

[1] These chunks are sometimes called data blocks, data segments, or data tiles.

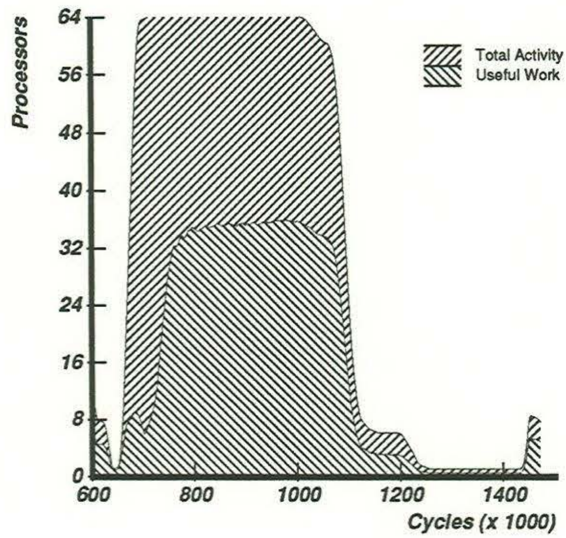

Figure 1: Activity profile for a parallel matrix multiply.

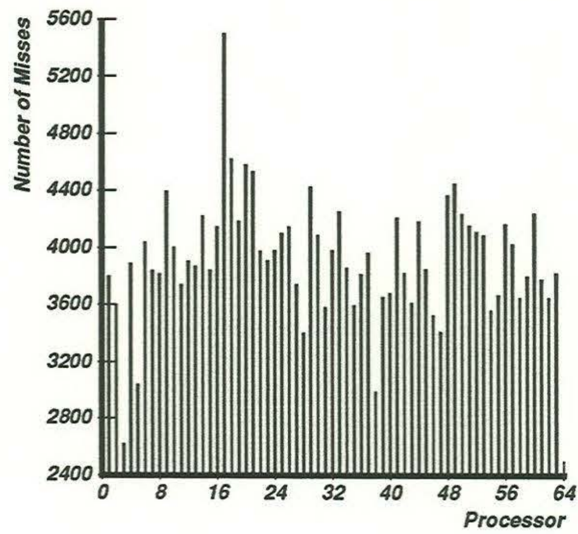

Figure 2: Histogram of miss rates from the individual processors.

approaches to performing this partitioning. It can be made the responsibility of either the application programmer or of the compiler and run time system. In either case, there are two goals to strive for, an equitable distribution of work among the processors and maximization of locality of reference, so that cache misses and invalidations are minimized.

Leaving this partitioning to the application programmer flies in the face of the main rationale for building shared-memory systems, namely, the relatively simple programming model that they present. As soon as programmers are forced to think about exactly how the computation and data are to be distributed, they lose the advantage of the shared-memory abstraction. On the other hand, leaving it to the compiler and runtime system poses significant technical difficulties.

An intermediate approach is to have the programmer or compiler partition the computation, and let the data "fall where it may" as the program runs. For some applications this works quite well. Techniques to achieve *communication-efficient partitioning* [22, 25, 1, 26, 21] typically focus on *tiling* the program to maximize *reuse* of data blocks. Block-structured algorithms which result in optimal tiles are also becoming commonplace. Tiled programs, whether written by a programmer or produced automatically by a compiler, result in clustered accesses into arrays and other data structures.

For some applications, like the matrix multiplication program discussed above, ignoring the initial placement of data can lead to extremely poor performance. This paper discusses the problem of partitioning data for such applications. The specific problem we address is: given compile-time information about the patterns of access to main memory, how does a compiler find an assignment of the data in the main memory such that there are no conflicts in the (direct-mapped) caches? We call this the *memory assignment problem.*

For sequential machines, the memory assignment problem is not difficult. Provided the uniprocessor cache is large enough to hold the working set of data blocks accessed by the processor, allocating these data blocks contiguously in memory will avoid conflicts in the cache.[2] For parallel machines, the difficulty lies in finding an assignment that avoids collisions in each cache simultaneously. This problem is illustrated by Figures 3 and 4. Figure 3 shows a conflict-free assignment of the shared code segment x and the data chunks a, b, c, and d, under the assumption that the size of each cache is three ($k = 3$), and that processor one accesses a and b, processor two accesses b and c, and processor three accesses c and d. Suppose, however, that processor three accesses c and a. In that case there is no conflict-free assignment possible, as depicted in Figure 4.

1.3 Overview of the Paper

The remainder of this paper is structured as follows. Section 2 begins by showing that given perfect information about which processors will access which chunks of memory, finding a memory assignment that avoids collisions is $\mathcal{NP}$-complete. This is done by reduction to the graph coloring problem. Section 3 then discusses important special cases for which there are polynomial-time algorithms.

In Section 4, we discuss approaches to dealing with the general memory assignment problem which attempt to *minimize the expected running time* for finding a conflict-free assignment. These are similar to algorithms currently used for register allocation. We close the sec-

[2]See [17] for further work in compile-time optimization of single-processor caches.

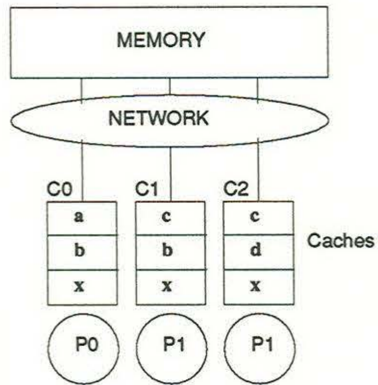

Figure 3: A conflict-free assignment.

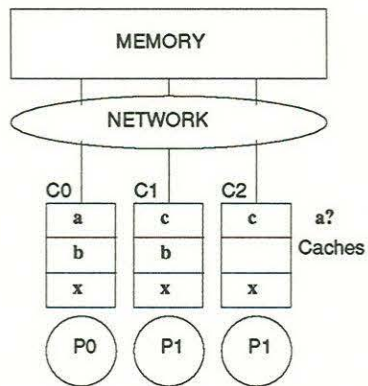

Figure 4: An access pattern for which no conflict-free assignment is possible.

tion by presenting techniques that can be used when conflict-free assignment is not possible, or when finding a conflict-free assignment is computationally infeasible. *Uniform decaching* is a method that forces a data block to be noncachable. Placing the uncached data block in a memory module physically close to the processors accessing it reduces the performance degradation. *Data replication* can be used at the software level for read-only data. The basic idea is to put copies of the conflicting data at several memory locations, so that it can be mapped to different locations in the cache. Finally, *rehashing* is a method implemented in hardware, in which a block that conflicts with another block already in the cache is mapped to another location in the cache by means of a separate hash function.

Section 5 proposes an approach, *grey coloring*, to *reducing the impact of the cache misses* caused by collisions. Misses are expensive not only because they slow down the processor taking the miss, but also because other processors may be idled while waiting for the straggling processor. If misses are concentrated in one processor, that processor becomes a bottleneck that can greatly increase latency. Grey coloring is a memory assignment technique that spreads the misses among processors by partially overlapping data chunks in caches, and attempts to minimize the maximum number of misses in any single processor.

Section 6 of this extended abstract summarizes the earlier sections and describes future work. In the final paper, we expect to include experimental data derived from implementations of the various memory assignment techniques.

2 The Memory Assignment Problem is $\mathcal{NP}$-Complete

In this section we prove that the memory assignment problem is $\mathcal{NP}$-complete by showing that it is as difficult as graph k-colorability.

First, we briefly describe the graph k-colorability problem. In this problem we are given a graph G and k colors, and we are asked whether it is possible to color the vertics of the graph in such a way that no adjacent vertices have the same color. More formally, the graph k-colorability problem is defined as follows [11].

GRAPH k-COLORABILITY: Given an undirected graph $G = (V, E)$ and an integer k, find an assignment $f : V \rightarrow \{1, \ldots, k\}$ such that $f(u) \neq f(v)$, for every edge $(u, v) \in E$, or infer that no such assignment f exists.

Graph k-colorability is among the most difficult intractable problems. It has been proven that, unless $\mathcal{P} = \mathcal{NP}$, there exists no polynomial approximation scheme for graph k-colorability with a performance guarantee smaller than 2, that is, a scheme that guarantees to color any k-colorable graph with no more than $2k$ colors [11]. In recent years, several approximation schemes have been proposed [24, 4, 5, 6], but none of them gives a performance guarantee that is bound by a constant.

In the memory assignment problem we are given a set of memory-resident data blocks and a set of caches. In each cache (of size k) we must store a given subset of the data blocks. A given data block may be stored in multiple caches. The caches are assumed to be direct-mapped, that is, a data block is always stored in the same location in any one of the caches. This location is determined unambiguously by the low order bits of the data block's address in the main memory. The memory assignment problem asks for an assignment of the data blocks to the main memory such that no conflicts occur in the caches, and more formally, it

is defined as follows.

MEMORY ASSIGNMENT: Let D be a set of data blocks, let P be a set of direct-mapped caches in a multiprocessor, and let $g : P \rightarrow 2^D$ be an assignment of data blocks to caches [3], where $|g(p_i)| \leq k$, for every cache $p_i \in P$. Find an assignment $f : D \rightarrow \{1, \ldots, k\}$ of storage locations for the data blocks in D, such that $f(d_i) \neq f(d_j)$ for all data blocks d_i, d_j in the same cache, or infer that no such assignment f exists.

Theorem 1 MEMORY ASSIGNMENT *is $\mathcal{NP}$-complete.*

Proof: We prove that graph k-colorability is polynomial-time reducible to memory assignment. Given a graph $G = (V, E)$ and k colors, we construct an instance of the memory assignment problem with $|V|$ data blocks and $|E|$ caches. Each vertex $v \in V$ corresponds to a data block $d_v \in D$ and each edge $(u, v) \in E$ corresponds to a cache $p_{uv} \in P$. Each cache p_{uv} has size k and is required to store the two data blocks d_u, d_v. Since no two data blocks in a cache can be stored in the same location, an assignment of locations in the range $\{1, \ldots, k\}$ to the data blocks in D yields a coloring of the vertices in V with k colors. Conversely, a coloring of the vertices in V with k colors yields an assignment of locations to the data blocks in D. □

This theorem, in conjunction with the fact that graph colorability is among the most difficult intractable problems [11], implies that it is very unlikely to find a provably good algorithm for the memory assignment problem.

3 Efficient Solutions to the Memory Assignment Problem

Despite the difficulty of the general memory assignment problem, several important classes of computations permit efficient computation of a conflict-free assignment by relying on the special structure of their memory access patterns. This section describes some of these computations as well as the corresponding algorithms for memory assignment. Specifically, we give a simple linear-time algorithm that yields an optimal conflict-free assignment for programs with special structure such as matrix-vector multiply and matrix-matrix multiply. We also show that for other classes of computations, such as tree computations and grid computations, we can efficiently compute a conflict-free memory assignment. Finally, we argue that the complexity of memory assignment can be reduced for single-program multiple-data computations.

The basic construct behind the algorithms in this section is an encoding of the dependencies among the data blocks in the form of a *conflict* graph $G = (V, E)$. Each vertex $u \in V$ represents a data block d_u, and each edge $(u, v) \in E$ signifies that the data blocks d_u and d_v must share at least one of the caches. Note that the degree of each vertex $u \in V$ equals the number of data blocks with which d_u shares a cache. This number may *not* equal the number of data blocks that must be stored in some particular cache together with d_u, or the number of caches in which d_u must be stored. The conflict graph G can be computed in linear time, with respect to the number of conflicts among the data blocks. Our algorithms search for a particular structure in G that allows us to color its vertices efficiently. Because the colors and

[3] We denote by 2^D the set of all subsets of D, the powerset of D.

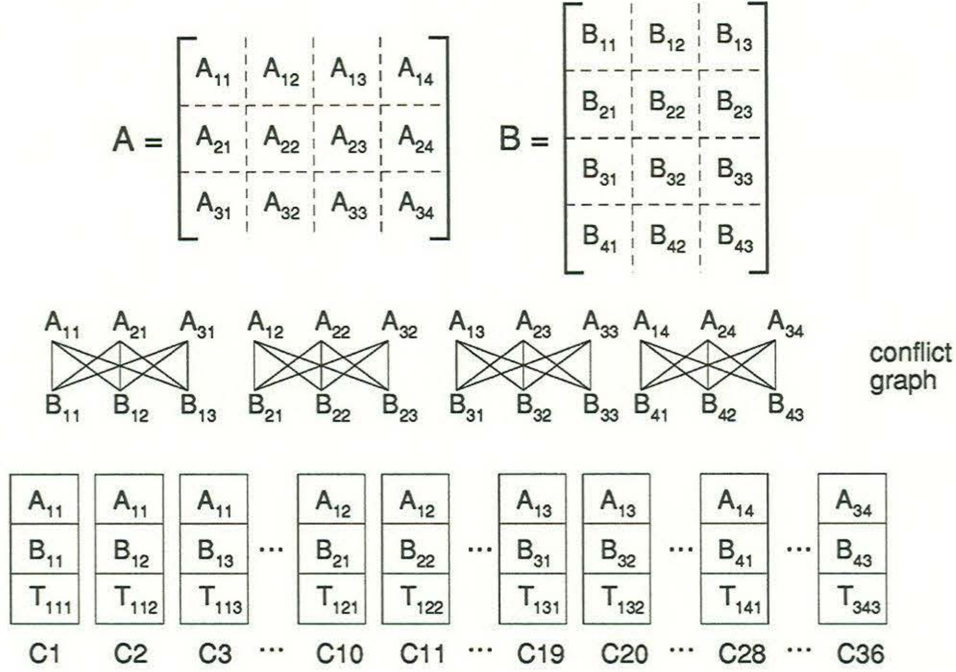

Figure 5: Multiplication of a matrix A by a matrix B for $M = 3$, $N = 4$, and $|P| = 36$. The conflict graph G is bipartite and each one of the 36 processors is responsible for an edge in G.

the cache locations correspond one-to-one, the coloring yields automatically an assignment of memory locations to the data blocks.

Our first algorithm relies on the fact that when the conflict graph G is bipartite, *i.e.*, of the form $G = (V_1 \cup V_2, E)$, and for every $(u, v) \in E$ we have $u \in V_1$ and $v \in V_2$, we can always find a 2-coloring in $O(|V_1| + |V_2|)$ time: we simply assign color i to all vertices in V_i, for $i = 1, 2$. We can verify efficiently whether G is bipartite by an $O(|E|)$-time breadth-first search. Therefore, given any computation with bipartite conflict graph G we can find a conflict-free memory assignment in $O(|E| + |V_1| + |V_2|)$ steps.

An example of a computation with a bipartite conflict graph G is multiplication of an $m \times n$ matrix A by an $n \times m$ matrix B. A blocked decomposition of this computation on $|P|$ processors partitions A into $M \times N$ square submatrices A_{ij} and B into $N \times M$ square submatrices B_{ij}, where $M^2 N = |P|$ (see Figure 5)[4]. Each processor is responsible for computing a temporary submatrix $T_{ijk} = A_{ij} \times B_{jk}$, for $i, k = 1, \ldots, M$ and $j = 1, \ldots, N$. This part of the computation exhibits locality and extensive sharing of data by the processors. The conflict graph G has N disconnected components, each one being a bipartite graph with $2M$ vertices and M^2 edges. The temporaries T_{ijk} are not shown in the conflict graph because we wish to keep the figure simple, and because computing the resulting submatrices $C_{ik} = \sum_{j=1}^{N} T_{ijk}$ does not involve extensive reuse of the data words within these blocks.

Consider another interesting class of computations in which every data block competes for a cache location with at most 3 other data blocks. Tree computations, for example, exhibit this structure. The degree of the resulting conflict graph is 3 or less, thereby admitting a

[4]It is straightforward to verify that this can be achieved for $M = (m|P|/n)^{1/3}$ and $N = (n^2|P|/m^2)^{1/3}$.

polynomial time algorithm which can color G with the minimum number of colors [11].

In the previous two algorithms, the resulting memory assignments use the minimum possible number of sets in the caches. Several computations, however, have conflict graphs of degree no greater than the number of cache sets k, in which case any conflict-free assignment that fits in the available cache space will suffice. Highly localized access patterns, such as in grid computations, should be expected to posess such a property for sufficiently large caches, since they access only a limited number of data blocks for each point in the grid. Since any connected, non-complete graph G with degree at least 3 but no greater than k can be colored with k colors in polynomial time [12, 7], we can compute efficiently a feasible assignment for such G. If G is complete, a fact that can be detected in $O(|E|)$ time, then the problem is infeasible.

Finally, we can reduce the size of the initial coloring problem for some computations. In SPMD (Simple-Program Multiple-Data) computations [10], for example, we can remove the code block from the conflict graph, since this block exists in every cache and it is guaranteed to utilize one color, thereby reducing the original k-coloring problem into a $(k-1)$-coloring problem.

4 Heuristics for the Memory Assignment Problem

This section presents heuristics for the memory assignment problem. Our basic algorithm colors the conflict graph by recursively removing all vertices of degree less than k. Usually, this operation is chained, and, eventually, the entire graph is eliminated and a coloring is obtained. Sometimes, however, the algorithm cannot eliminate the entire graph, either because the original assignment problem is infeasible, or because of the order in which previous vertices were removed. In these cases, we apply one of two heuristics called *uniform decaching* and *data replication* to allow further coloring of the conflict graph. We also discuss a hardware-level solution for the problem, when no conflict-free assignment is possible, based on *rehashing*. Finally, and most importantly, Section 5 presents a novel memory assignment method called *grey coloring*, which attempts to spread the misses among processors by partially overlapping data chunks in caches so that the number of misses in any single processor is minimized.

Our basic algorithm for k-coloring the conflict graph $G=(V,E)$ is the same as the general graph-coloring scheme in [8, 9] that is used in the context of register allocation for sequential processors. If we wish to find a k-coloring of G having a vertex u of degree less than k, then G is k-colorable if and only if the graph G' with $V' = V - \{u\}$ and $E' = E - \{(u,v) \in E\}$ is k-colorable. If a k-coloring can be found for the graph G', then we can reinsert vertex u and assign it a color that is not used by the vertices with which it shares an edge. Our basic algorithm reduces the size of the conflict graph by progressively removing from it all vertices of degree less than k. Our approach differs from the previous ones in the way vertices are selected for removal, both when vertices with degree less than k exist, and when they don't. At each iteration, our algorithm removes a vertex u with degree less that k, which has the most vertices of degree k in its adjacency list. A more computationally intensive approach is to perform a breadth-first search of depth d, in order to identify a vertex u with the most vertices of degree less than $k+d-1$ in its vicinity.

Most of the time, if caches are much larger than the total size of the data blocks accessed by any individual processor, our basic algorithm will eventually eliminate the entire graph.

Vertices are then put back in the reverse order that they were removed. As each vertex is reinserted, an appropriate color is picked for it, and at the end a coloring is obtained.

Sometimes, however, we are left with a graph in which every vertex has degree at least k, so no vertex can be removed. At this point we apply one of several schemes to allow the algorithm to continue. An important common element of these schemes is that one or more data blocks are chosen to be treated in a special way. Two such schemes, data replication and rehashing, are discussed in Appendix B.

We focus here on a third scheme, *uniform decaching*, because this scheme, in conjunction with the novel memory asignment scheme that we describe in the next section, can provably decrease the latency of non-bandwidth-limited computations. In uniform decaching, one or more vertices are removed from the graph until a vertex of degree less than k is found. The corresponding data blocks, called *decached blocks*, are placed in the main memory at a location in a memory module physically close to all processors that need them, and in the case of a single processor accessing a block, in its local memory module. Uniform decaching strives to select data blocks so that the total number of misses is spread uniformly across the processors. This is done in a greedy fachion, by keeping track of the number of misses in each cache due to decaching at previous stages, and selecting to decache blocks so that the maximum number of misses over all processors is kept minimal. Other ways for selecting blocks to decache are the subject of current research. In future work we will address this issue by comparing various implementations.

5 Grey Coloring

The previous section described an algorithm that recursively removed all vertices of degree less than k from the graph. When a vertex with degree less than k could no longer be found, a vertex with degree greater than k was selected for removal, so the algorithm could proceed. While the vertices with degree less than k are easily reinserted in the graph during the coloring phase, we need to find a way of inserting the vertices that have degree greater than k. This section proposes a method called *grey coloring* that inserts these vertices by allowing partial overlap of the data chunks in the caches, thereby relaxing the constraints of the memory assignment problem. The name "grey coloring" refers to the mixed color in lines that overlap.

Specifically, we investigate the following problem: given a conflict-free assignment that leaves some of the data chunks uncached (either because the original assignment problem was infeasible, or because our heuristic algorithm did not find a solution), find a partially overlapping assignment of the data chunks that reduces the impact of the cache misses caused by collisions. Grey coloring, our technique for solving this problem, is motivated by the fact that some data blocks may be left uncached in a conflict-free assignment, even though free cache slots exist. Grey coloring exploits this free space to spread the misses across the caches and minimize the maximum number of misses in any single cache. By minimizing the maximum number of misses in any single cache, we expect to reduce the overall latency of non-bandwidth-limited computations.

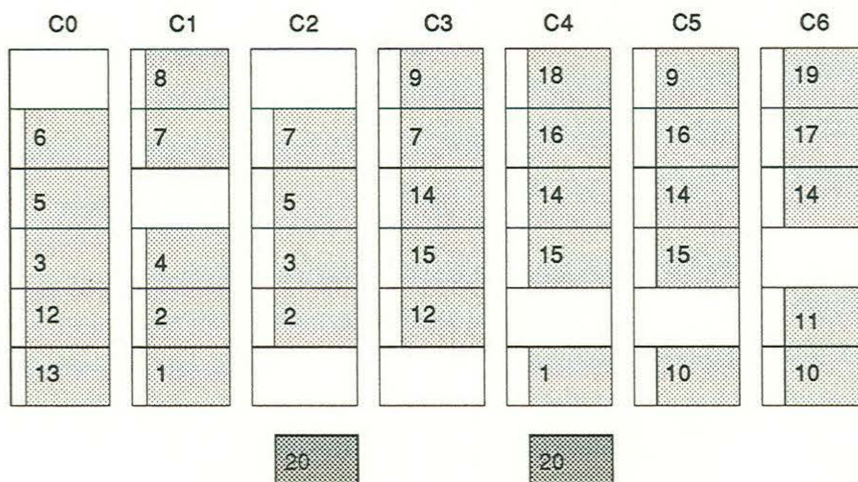

Figure 6: A memory assignment of 20 data blocks accessed by seven caches according to some given access pattern. Data block 20 is not cached and the cost of cache misses is proportional to the number of lines l in the data block.

An Example

Let us demonstrate grey coloring by means of an example. Figure 6 shows a memory assignment of 20 blocks accessed by a set of seven caches according to some given pattern. Each cache can hold up to six data blocks ($k = 6$). Data blocks 1 through 19 are cached. Data block 20 is left uncached, however, because both C2 and C4 do not have any free slots with the same index. If the block is placed in the free slot of C4, then the cost of cache misses due to conflicts in C2 is proportional to the time required to fetch l lines from the main memory, where l is the number of lines in each data block. Figure 7 illustrates the conflicts created by attempting to slide data block 20 in the first slot of the caches. This slot is empty in C2, but it is occupied by data block 1 in C4. The arrows in this figure denote the set of data blocks whose motions must be orchestrated, in order to fit data block 20 in the caches. We refer to this network of dependencies as the *marionette* of data block 20, with respect to the given memory assignment and the first slot in C2 and C4 (see Figure 8). The motivation behind this name is that when block 20 is inserted in the caches, the blocks in its network of dependencies rooted at the conflicting data block move in a manner resembling a marionette – blocks closer to the root block move greater distances.

The final arrangement of the data chunks in the caches is illustrated in Figure 9. Any two overlapping blocks share a fraction $\theta = 1/8$ of their l lines. Observe that $1/\theta = 8$ is the number of layers d in the marionette, and that each node i at layer d_i has been shifted by $(d - d_i)\, l/8$ words. As a result of this shift some data blocks may wrap around the caches. Note that the cost of the cache misses has decreased by more than 60%: it is proportional to the time required to fetch only $(3/8)\, l$ lines from the main memory, since there are at most 3 overlapping regions of size $l/8$ lines in any one of the caches.

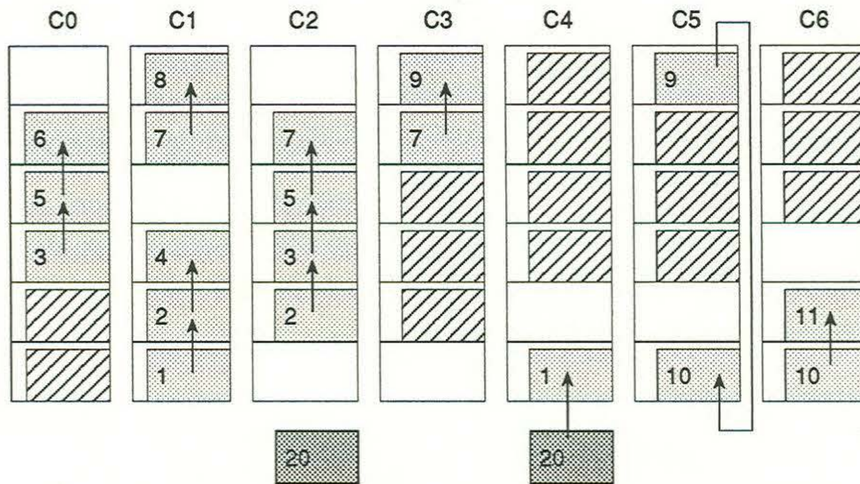

Figure 7: Dependencies among data blocks when sliding data block 20 into the first slot of the caches C2 and C4. For simplicity, data blocks that have no dependencies are shaded by diagonal lines and their numbers are omitted. Data blocks are drawn shorter than their real size, in order to indicate the possibility of overlap.

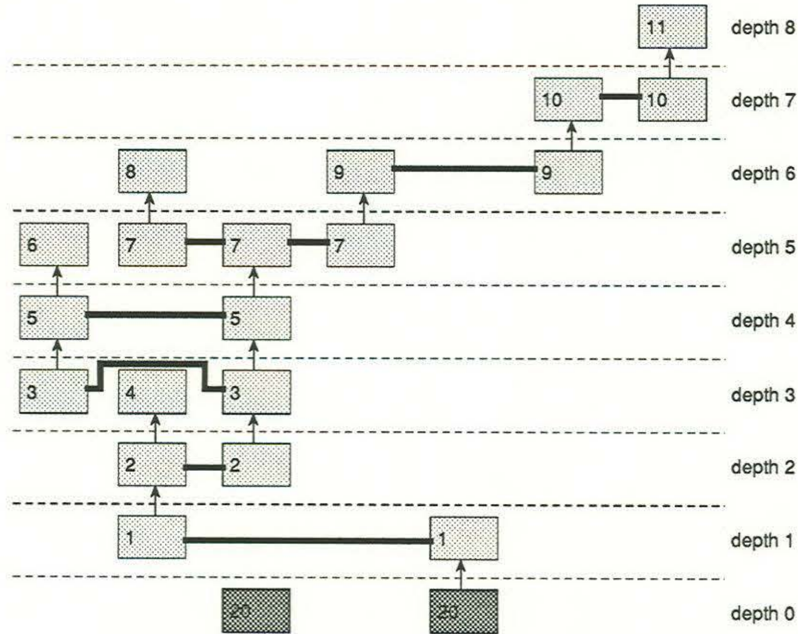

Figure 8: The marionette of data block 20, with respect to the given memory assignment and the first slot in caches C2 and C4.

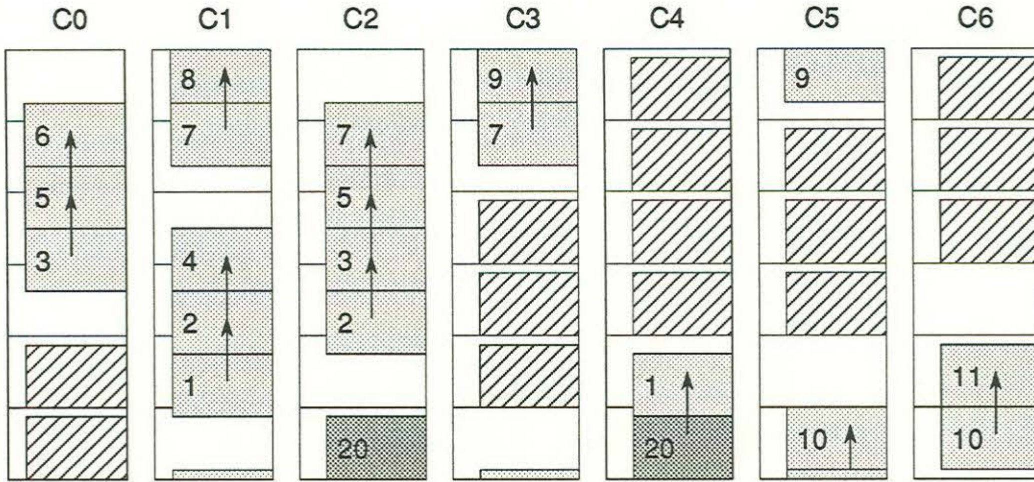

Figure 9: An assignment after grey coloring. Note that the cost of cache misses is proportional to $(3/8)\,l$.

Construction of a Marionette

The basic construct required for grey coloring is the marionette of the data block w that is inserted in the caches. The marionette of w, with respect to

- a memory assignment f, and
- the cache slot s into which we try to insert w

is a layered graph $G_m = (V_m, E_m)$. The vertices V_m of the graph are data blocks. An edge (u, v) in E_m signifies that data blocks u and v are adjacent in a cache, and that they will overlap when w is inserted. Each layer of G_m corresponds to a set (with a common index) in the caches. The total number of layers in G_m is its depth.

A marionette can be constructed recursively as described by the pseudocode in Figure 10, where for clarity the slot s into which we try to insert w is taken to be 0. All manipulations of the indices are assumed modulo k. Therefore, the starting cache slot $q = 0$ can be chosen at will, and each conflicting data block admits up to k marionettes. Let $C_i[q]$ denote the copy of a data block in slot q of cache C_i. $C_i[q]$ is NIL if slot q in cache C_i is empty. Initially, for each cache C_i in the cache set P we introduce an edge between a copy of w and the block $C_i[0]$ in the first slot of C_i, if $C_i[0]$ conflicts with w. Then, we recursively traverse the k slots in the caches, introducing an edge (x, y) between blocks $x = C_i[q]$ and $y = C_i[q+1]$ whenever the following three requirements are met:

1. There exists an edge (u, v) between u and v in the slots $q-1$ and q of some cache C_j,
2. x, v are copies of the same data block in memory, and
3. y is not yet in the marionette.

The running time of the algorithm is $O(k|P|)$, where $|P|$ is the number of caches in the system.

```
CONSTRUCT-MARIONETTE(w, f, P)

V_m ← {w}
E_m ← ∅
for each C_i ∈ C
    do if C_i[0] ≠NIL and C_i[0] conflicts with w
        then V_m ← V_m ∪ {C_i[0]}
             E_m ← E_m ∪ {(w, C_i[0])}
for q = 0 to k|P| − 1
    do for i = 0 to |P| − 1
        do if C_i[q] = C_j[q] and C_j[q] ∈ V_m for some j ∈ {0, ..., |P| − 1}
            then V_m ← V_m ∪ {C_i[q]}
            if C_i[q + 1] ≠NIL and C_i[q + 1] ∉ V_m
            then V_m ← V_m ∪ {C_i[q + 1]}
                 E_m ← E_m ∪ {(C_i[q], C_i[q + 1])}
```

Figure 10: A recursive procedure for constructing the marionette of w, where f is a given memory assignment and P is the set of caches.

Once the marionette G_m has been constructed, we can grey color the graph straightforwardly. First, we compute the depth d of G_m and we set $\theta = 1/d$. Then, starting from the root of G_m we shift each data block u at depth d_u in G_m by $(d - d_u)\,l\,\theta$ words. This algorithm terminates in $O(k|P|)$ steps.

As mentioned previously, in a non-bandwidth-limited system, the latency of computation is related to the maximum number of misses in any single cache. The following lemma, whose proof will be included in the final version of the paper, states that the maximum number of conflicting lines in any cache is never more than l. Note that l conflicts would result if the block is placed without allowing partial overlap.

Lemma 1 *Let l be the number of lines in any data block, and let θ be the fraction of l that any two overlapping blocks share after grey coloring. Moreover, let l_g be the maximum number of conflicting cache lines in any cache after grey coloring. Then*

$$l_g = \max_{0 \leq i \leq |P|-1} \sum_{\substack{u,v \in C_i \\ (u,v) \in E_m}} l\theta$$

and

$$l_g \leq l. \qquad \square$$

In other words, the number of misses introduced in any cache is proportional to the number of edges in the marionette G_m that connect blocks in that cache. Since each cache is accessed independently, the maximum among these numbers determines by how much the computation degrades due to these misses. Intuitively, the cost of the misses is spread through the marionette over the entire set of caches. The contribution of each cache to the final cost is dictated by how many of the original misses are spread on that particular cache. We should

note that as a result of grey coloring, the traffic in the communication network may increase but it is spread over all caches in the marionette.

An important property of grey coloring is that it guarantees to decrease the maximum number of misses in any cache, if there exists some marionette of sufficiently large depth d.

Corollary 1 *Let G_m be the marionette of depth d of a conflicting data block w, and let l_g be the maximum number of conflicting cache lines in any cache after grey coloring. If $d > k$, then $l_g < l$.* □

Intuitively, the deeper the marionette, the better we can spread the cache misses among the caches. In practice, by setting $\theta = \min\{1/k, 1/d\}$, we expect to almost always decrease the maximum number of cache misses in any single cache.

The same basic grey coloring procedure is repeated for every decached block, that was removed from the conflict graph according to the uniform decaching heuristic in Section 4, and that we want to insert into the caches. As more decached blocks are reinserted the free spaces in the caches become smaller, but this does not change the way marionettes are constructed for each block, or the way blocks move. Overlapping blocks are treated as one solid block. When all blocks have been inserted and there is still free space in the caches, it is possible to further reduce the number of cache misses in a single cache by trying to minimize the number of overlapping blocks in each one of them.

6 Conclusion

We investigated techniques for reducing cache misses in multiprocessors with direct-mapped caches by avoiding cache collisions. Collisions are avoided by judiciously assigning data blocks to main memory locations such that there are no conflicts in the caches. We proved that the resulting memory assignment problem is $\mathcal{NP}$-complete when no overlap is allowed between the blocks in the caches. We showed, however, that several interesting classes of computations, such as matrix multiply, admit an efficient solution to this problem. We proposed heuristic approaches based on graph coloring for assigning memory in the general case. When a conflict-free memory assignment is infeasible, or when it is computationally expensive to derive a conflict-free memory assignment, we suggest two software-level heuristics, decaching and data replication, and a hardware-level heuristic based on rehashing. Finally, we proposed a technique called grey coloring for spreading the cache misses over the entire set of processors. Grey coloring minimizes the maximum number of misses in any single cache, thus decreasing the overall latency of computation in non-bandwidth-limited situations.

We are currently implementing and evaluating these memory assignment techniques. In the final paper we expect to report experimental data derived from these implementations.

Acknowledgements

We thank Norm Rubin, Venkat Natarajan for discussions on register allocation through graph coloring. Dan Nussbaum and Kirk Johnson observed the problem with the straggling processor, and provided the graphs from simulations of Alewife. This research was supported in part by NSF under grant MIP 9012773 and in part by the Defense Advanced Research Projects Agency under Grant N00014-91-J-1698.

References

[1] S. G. Abraham and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2:318–328, July 1991.

[2] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. Also appears as MIT/LCS Memo TM-454, 1991.

[3] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating Systems and Multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[4] B. Berger and J. Rompel. A better performance guarantee for approximate graph coloring. *Algorithmica*, 1988.

[5] A. Blum. An $\tilde{O}(n^{0.4})$-approximation algorithm for 3-coloring (and improved approximation algorithms for k-coloring). *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 535–542, May 1989.

[6] A. Blum. Some tools for approximate 3-coloring. *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 554–562, October 1990.

[7] R. L. Brooks. On colouring the nodes of a network. *Proc. Cambridge Philos. Society*, 37:194–197, 1941.

[8] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM Sigplan '82 Symposium on Compiler Construction*, pages 22–31, June 1982.

[9] F. C. Chow and J. L. Hennessy. The prioriry-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4), October 1990.

[10] F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. Technical Report RC 11552 (55212), IBM T. J. Watson Research Center, Yorktown Heights, November 1986.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., San Francisco, 1979.

[12] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[13] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, Hawaii, June 1988.

[14] Mark D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25 – 40, December 1988.

[15] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings, International Symposium on Computer Architecture '90*, pages 364–373, June 1990.

[16] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of the Stanford DASH Multiprocessor. Computer systems laboratory tr 89-403, Stanford University, December 1989.

[17] Scott McFarling. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[18] Encore Multimax. Encore, Marlboro, Massachusetts.

[19] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings ICPP*, pages 764–771, August 1985.

[20] Steven A. Przybylski. Performance-directed memory hierarchy design. Technical Report CSL-TR-88-366, Stanford University, Stanford, CA, September 1988.

[21] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[22] V. Sarkar and J. Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. In *Proceedings of the ACM Sigplan '86 Symposium on Compiler Construction*, pages 17–26, 1986.

[23] Sequent Symmetry. Sequent, Portland, Oregon.

[24] A. Wigderson. Improving thr performance guarantee for approximate graph coloring. *JACM*, 30(4):729–735, 1983.

[25] M. Wolfe. More Iteration Space Tiling. In *Proceedings of Supercomputing '89*, pages 655–664, November 1989.

[26] M. E. Wolfe and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

A Probabilistic Analysis

This section presents a simple probabilistic analysis of the memory assignment problem. Assuming that data blocks are stored in the main memory in a uniformly random manner,

the analysis indicates that at least one cache conflict is a virtual certainty as the number of processors increases. Specifically, the probability of conflict-free assignment in all caches with random initial placement drops exponentially with the number of processors. As, discussed earlier, even one straggler suffering from cache conflicts out of the $|P|$ processors can dramatically reduce overall performance.

Real systems, however, rarely assign random memory locations to data blocks. Rather, they commonly assign blocks to contiguous memory locations. Contiguous assignment of blocks in main memory obviates conflicts between neighboring blocks, but does not preclude conflicts between blocks displaced by distances greater than the cache size. Because conflicts are related to the relative placement of blocks in the cache, which in turn are related to the access patterns of processors, random placement of blocks in the cache is a reasonable assumption. Furthermore, for our purposes, random placement of data blocks in the cache is tantamount to random placement in memory.

Let us assume that a multiprocessor has $|P|$ processors, and that each processor has its own cache. As stated before, let k be the number of *sets* in a cache, that is, the maximum number of data blocks that can be stored in it. We assume for simplicity that each processor accesses the same number of data blocks, $n \leq k$.

If every data block is stored in the main memory randomly, each one of the k sets of a cache is equally likely to be the target of a data block. Therefore, the probability there are no conflicts in cache i, denoted $P(nc_i)$, is simply the probability of having no more than one data block assigned to the same set in the cache. Thus

$$P(nc_i) = \frac{\prod_{j=0}^{n-1}(k-j)}{k^n}.$$

The numerator of this expression is the total number of ways of selecting n sets from the cache without selecting any one of them more than once. The denominator is the total number of ways of selecting n sets from the cache. We can rewrite $P(nc_i)$ as follows.

$$\begin{aligned} P(nc_i) &= \prod_{j=0}^{n-1}\left(1-\frac{j}{k}\right) \\ &\leq \prod_{j=0}^{n-1} e^{-j/k} \\ &\leq e^{-n(n-1)/2k} \end{aligned}$$

since $1 - x \leq e^{-x}$.

If the mapping of data blocks to cache sets for a given processor is independent of their mapping in other processors, the probability $P(nc)$ of having no conflicts in any of the $|P|$ caches is given by

$$\begin{aligned} P(nc) &= P(nc_i)^{|P|} \\ &\leq e^{-n(n-1)|P|/2k}. \end{aligned}$$

However, as demonstrated by the example in the introduction, we know that when multiple processors access the same data block, finding a conflict-free memory assignment may be

impossible. Thus, the lack of independence in mappings in real systems makes the probability of conflict even greater than indicated by this analysis.

Thus we see that the probability of having no conflict in any of the multiprocessor's caches drops exponentially with the number of processors $|P|$. To get a sense of the numbers involved, let us compute the probability of conflict-free assignment in a system with $|P| = 100$ processors and caches of size 64K words. Let the size of each data block be $2K$ words, implying that the size of a cache in terms of the number of data blocks it can hold is $k = 64K/2K = 32$. Let the number of blocks n accessed by each processor be 8, corresponding to a cache loading of 25%. Thus, for $k = 32$ and $n = 8$, the probability of having no conflicts in the system is smaller than 10^{-40}.

B Data Replication and Rehashing

In this part of the appendix, we discuss software-level and hardware-level approaches that allow us to color the conflict graph G of a computation when our basic heuristic cannot remove any more vertices from G.

The first scheme is called *data-replication* and can be used at the software level for read-only data. In this scheme we split one or more vertices of degree at least k into vertices of smaller degree. The sum of the degrees of the replicated vertices equals the degree of the original vertex. This operation is tantamount to replicating in the main memory the data chunks corresponding to the split vertices. The total number of additional copies of a particular data chunk equals the total number of splits of the original vertex during the execution of the algorithm.

A hardware solution for the memory assignment problem when a conflict-free assignment is not possible is rehashing [3]. With rehashing, a processor first accesses the cache in the normal way with the low order bits of the location. On a miss, instead of directly going to main memory, the processor accesses the cache with a different hash function. Because each memory location can effectively reside in two cache (or more) locations, the probability of conflicts is reduced, although it cannot be completely eliminated.

One other hardware solution is to use a victim cache [15]. A victim cache is a small associative cache used in conjunction with the primary direct-mapped cache. Unfortunately, victim caches require data area proportional to the size of the conflicting chunks, which in general can be as large as the primary cache itself. However, if most of the conflicts can be resolved at compile time, the victim cache may be able to store a small number of conflicting chunks of data.