MASTER

Data-cache conflict-miss reduction by high-level data-layout transformations

van Meeuwen, T.P.

Award date:
2002

Link to publication
Data-cache Conflict-miss Reduction by High-level Data-layout Transformations

Tycho van Meeuwen

Coaches: dr.ir. M. Miranda (IMEC Leuven, België)  
ir. E. Brockmeyer (IMEC Leuven, België)  
dr.ir. A.C. Verschueren (TU/e)
Supervisor: prof.dr. H. Corporaal (TU/e / IMEC)
Period: September 2001 – June 2002
Master Thesis Report

Data-cache Conflict-miss Reduction by High-level Data-layout Transformations

By:
Tycho van Meeuwen

Advisors: Dr. Ir. M. Miranda (IMEC)
Ir. E. Brockmeyer (IMEC)
Dr. Ir. A.C. Verschueren (TUE)
Supervisor: Prof. Dr. H. Corporaal (TUE/IMEC)
Period: September 2001 – June 2002
Abstract

In order to reduce the power consumption of data dominant applications, a multi-level memory hierarchy is often introduced containing both memories and caches. The cache memories used in this hierarchy must have simple controller mechanisms and tag-comparison overheads to reduce the energy per cache access as much as possible. However, the simplest cache architecture, a direct mapped cache, potentially also causes power overhead: because of the limited placing freedom, extra cache misses (conflict misses) will occur and each extra miss results in extra costly transfers to (bigger) memories in the next hierarchy layer.

An effective way to reduce the extra misses is a careful placing of the data in the main memory. But this can only be interesting if not too much overhead is involved with this placement of the data (e.g. memory overhead or address computation overhead).

Therefore we use a memory data-layout scheme that has a relative small amount of overhead to reduce the amount of conflict misses. This scheme interleaves arrays in the main memory to assign different arrays to own locations in the cache. This method is integrated with an array padding technique to fine-tune the placing of arrays even more.

In order to know where to place the data, we studied the nature of the conflict misses in detail. We extended an existing miss model and developed a mathematical property to predict the amount of conflict misses. We combined the knowledge from the model with the observations we obtained from simulations to optimize some data dominant drivers.

We showed how we could significantly reduce the miss rate. We tried to come as close as possible to the theoretical lowest rate (the miss rate of a fully associative cache with an optimal replacement policy). The last percentages of misses we could not solve within reasonable increase of data-layout complexity and memory overhead. However, with a good data layout we reached with a direct mapped cache a miss rate equal to the rate of a 4-way associative cache, without significant memory or addressing overhead. Hereby saving the cost of the more expensive 4-way associative cache.
Acknowledgements

During a period of ten months I have been working to accomplish this thesis. There have been many people who contributed to the realization of this work. I want to thank all of them for their cooperation, helpful advices, interesting discussions and support.

Some people I would like to mention in particular.

First of all I want to thank Miguel Miranda and Erik Brockmeyer, my advisers at Imec. They both reserved a lot of time to help me finding the right tracks to follow. This required energy and patience, since the path we had to follow was twisting and steep from time to time. But on this long journey they taught me many aspects of the subject.

Thanks to my supervisor Prof. Stevens and my advisor Ad Verschueren from the university of Eindhoven, for enabling me to graduate externally and for supporting my work.

I want to express my gratitude to Prof. Francky Catthoor for the assistance in the guidance of my work. I especially thank Prof. Henk Corporaal for taking care of my supervision in the final stage of my thesis. His knowledge of my thesis subject and the connection to the university of Eindhoven could be combined effectively for the evaluation of the theses.

A word of thanks to Cedric Ghez for the intensive cooperation and the discussions we had. My gratitude to Martin Palkovic, Arnout Vandecappelle and all other colleagues for their good advices and pleasant atmosphere.

My final thanks are to my supporters in the background: my family, my parents and my brother who encouraged me and helped whenever they could, my Heleenjie for the warmth and inspiration she gave me, and also all my friends who made this year in Leuven so special for me.

Tycho van Meeuwen
Leuven, June 2002
Contents

1. INTRODUCTION .................................................................................................................................. 1
   1.1 CACHES IN THE CONTEXT OF THE 'DATA TRANSFER AND STORAGE EXPLORATION' OPTIMIZATION ...
   1.2 PROBLEM STATEMENT .................................................................................................................. 1
   1.3 MAIN CONTRIBUTIONS .................................................................................................................. 4
   1.4 STRUCTURE OF THIS THESIS ....................................................................................................... 5

2. COMPILER CENTRIC CACHE MISS DEFINITION AND MEASUREMENT .......................................................... 6
   2.1 CACHE FUNDAMENTALS ................................................................................................................. 6
      2.1.1 Cache in the Memory Hierarchy ............................................................................................... 6
      2.1.2 Mapping Organization ............................................................................................................ 6
      2.1.3 How does a cache work ........................................................................................................... 7
      2.1.4 Replacement Policies ............................................................................................................. 7
      2.1.5 Memory update policy ........................................................................................................... 8
      2.1.6 Block allocate policy .............................................................................................................. 8
      2.1.7 Data-level transfers versus block-level misses ........................................................................ 9
   2.2 COMPILER ORIENTED CACHE MISS CLASSIFICATION ...................................................................... 9
      2.2.1 Platform independent miss types ............................................................................................. 10
      2.2.2 Controller policy independent miss types ................................................................................ 11
      2.2.3 Controller policy dependent miss types .................................................................................. 11
   2.3 FRAMEWORK FOR THE MEASUREMENT OF THE DIFFERENT CACHE MISS TYPES ....................... 13
      2.3.1 Measuring miss types one by one .......................................................................................... 13
      2.3.2 The simulation environment .................................................................................................. 14

3. APPLICATION DRIVERS ........................................................................................................................ 16
   3.1 WAVELET ...................................................................................................................................... 16
   3.2 VOICECODER ALGORITHM .......................................................................................................... 16
   3.3 CAVITY DETECTION ..................................................................................................................... 16
   3.4 QSDPCM ..................................................................................................................................... 17

4. DATA LAYOUT TRANSFORMATIONS FOR CACHE MISS REDUCTION ......................................................... 19
   4.1 THE OPTIMIZATIONS FOR CAPACITY MISSES EXPRESSED IN THE DATA-LAYOUT .................... 19
   4.2 THE NATURE OF CONFLICT MISSES IN RELATION TO DATA-LAYOUT ....................................... 20
   4.3 THE PRINCIPLE OF TILING .......................................................................................................... 20
      4.3.1 Single interleaving-data-layout ............................................................................................... 21
      4.3.2 Multiple interleaving-data-layouts .......................................................................................... 22
      4.3.3 Memory overhead .................................................................................................................... 22
      4.3.4 In-place transformations ......................................................................................................... 24
      4.3.5 Source code transformations for the tiling approach .............................................................. 25
      4.3.6 CPU specific overhead .......................................................................................................... 27
   4.4 LIMITATIONS OF THE TILING APPROACH FOR CONFLICT MISS REMOVAL ......................... 28
      4.4.1 What are real associativity misses ........................................................................................ 28
      4.4.2 One decision per array .......................................................................................................... 30
      4.4.3 Exotic shapes of working sets .............................................................................................. 31

5. OPTIMIZATIONS FOR APPLICATIONS WITHOUT DATA REUSE .............................................................. 32
   5.1 MHLA DIRECTED APPROACH ..................................................................................................... 32
      5.2 TILE BASED DATA-LAYOUT OPTIMIZATION OF CACHE MISSES .......................................... 32
      5.2.1 Existing model ....................................................................................................................... 33
      5.2.2 Development of a new cache miss model .............................................................................. 33
      5.2.3 Experiments to test the miss model for individual arrays .................................................... 38
      5.2.4 Optimizing a complete driver ............................................................................................... 39
      5.2.5 Limitations of the approach and conclusions ....................................................................... 40

6. OPTIMIZATIONS FOR APPLICATIONS WITH DATA REUSE ..................................................................... 42
CONTENTS

6.1 DEFINITION OF DATA-REUSE ......................................................................................................... 42
6.2 RELATION TO MEMORY HIERARCHY DESIGN ................................................................................. 43
6.3 PROBLEM DEFINITION ............................................................................................................... 43
6.4 REDUCTION OF DATA-LAYOUT CONFLICT MISSES BY TILE-SIZE REFINEMENT ANDPadding

TECHNIQUE ......................................................................................................................................... 44
  6.4.1 Mathematical properties for predicting conflicts .................................................................. 44
  6.4.2 Optimizations for specific access patterns ......................................................................... 45
  6.4.3 Designing the right data-layout .......................................................................................... 49
6.5 REDUCING OVERHEAD........................................................................................................... 51
  6.5.1 Constrained tile-size selection ............................................................................................. 51
  6.5.2 Cache in-place of copy candidates ....................................................................................... 51
6.6 EXPERIMENTS AND RESULTS FOR THE CAVITY DETECTION-DRIVER........................................... 51
  6.6.1 Introduction to the experiments ............................................................................................ 51
  6.6.2 Analyses ............................................................................................................................... 52
  6.6.3 Optimization .......................................................................................................................... 55
  6.6.4 Conclusions ............................................................................................................................ 58
6.7 EXPERIMENTS AND RESULTS FOR THE QSDPCM-DRIVER ............................................................. 59
  6.7.1 Introduction to the experiments ............................................................................................ 59
  6.7.2 Optimization of the 'frame' array ......................................................................................... 60
  6.7.3 Optimization of the 'prev_frame' array ............................................................................... 62
  6.7.4 Optimization of 'prev_sub2_frame' array .......................................................................... 65
  6.7.5 Optimization of 'prev_sub4_frame' array .......................................................................... 67
  6.7.6 Global optimization results for the different arrays together ................................................. 67
  6.7.7 Simulating the complete QSDPCM driver (data-dependent behavior and full functionality included) ........................................................................................................ 70

7. CONCLUSIONS AND SCOPE FOR FUTURE WORK ........................................................................ 75
  7.1 CONCLUSIONS ........................................................................................................................... 75
  7.2 POSSIBILITIES FOR FUTURE WORK ....................................................................................... 75

BIBLIOGRAPHY .................................................................................................................................. 77
1. Introduction

For the reduction of power in data dominated applications we focus on the usage of energy efficient memory hierarchies that contain hardware controlled cache memories. For energy efficiency reasons it would be good to have caches with a minimal tag overhead and a simple controller, as long as the overhead in cache misses does not undo the benefits. This can be achieved by software steered miss elimination strategies. This thesis focuses on the reduction of cache misses for energy efficiency. In this section first the context of the work is presented followed by the problem statement and an overview of the main contributions.

1.1 Caches in the Context of the ‘Data Transfer and Storage Exploration’ Optimization Script

Since the transfer of data from memory to the processor is the main bottleneck in many current embedded applications, a cache hierarchy is often added between the processor and the background memory. The motivation for the design of memory hierarchies is driven by the ever-increasing gap between processor and memory speeds (see Figure 1.1). Fetched data is retained in a memory layer close to the processor, hoping that it will be accessed from that layer again. These subsequent accesses can then be performed much faster and more energy-efficient. The optimization for caches as presented in this thesis is part of a much wider memory-data-transfer optimization script. This optimization script is introduced first.

![Figure 1.1 The increasing performance gap between CPU and memory](image)

For most advanced real-time communication and information processing applications (video, imaging, telecommunication), the manipulation of complex data has a major or even dominant effect on the energy cost of the global system. To eliminate this implementation bottleneck a Data Transfer and Storage Exploration (DTSE) approach has been developed at IMEC for which both robust and prototype tools are being obtained. This DTSE methodology consists of an almost platform-independent stage followed by a more memory platform-dependent one. Both stages include the systematic application of well-decoupled optimization steps.

The different steps in the Data transfer and Storage Exploration methodology are shown in figure 1.2. The main input for DTSE is a program-language code of a specific data-dominated system. The output is an optimized code together with optimized platform specifications. The final goal is to have tool support for all the different steps: a pre-compiler that optimizes high-level source code.

Not all steps are directly related to specific cache optimizations. Therefore only some steps will be explained in more detail. For a complete overview, reference [5] should be consulted. The first step that is related to our cache optimizations (related to memory hierarchy exploitation) is the Data-reuse step. Based on memory-bandwidth constraints and estimations of the storage size, the background memory transfers are partitioned over several (or the available) hierarchical memory levels in order to reduce
the power and/or area cost or to speed up the processing. Every original array is assigned to a memory partition. In order to get the array-data to the data-path intermediate copies are created which can be stored in intermediate memory layers.

Storage Bandwidth Optimization (SBO) is executed as a part of the Storage Cycle Budget Distribution (SCBD) step to determine which arrayss should be made simultaneously accessible in the memory architecture so that the real-time constraints can be met with minimal memory bandwidth related costs. Because of stringent cycle budget constraints, a number of data accesses are executed in the same cycle: these data accesses are in conflict. After the parallel accesses are ordered and SBO is performed, an Extended Conflict Graph (ECG) is given as output. This ECG represents the minimum amount of access conflicts between arrays, in such a way that it is still possible to find a schedule later on that fits within the cycle budget and the available bandwidth (port constraints).

The memory allocation and assignment step uses this ECG as input to come up with a background memory architecture that is optimal in terms of area and/or power cost and that can handle the conflicts between arrays due to the parallel accesses.

After memory allocation and assignment, the data layout optimization step determines where and how the data elements of arrays are stored in the memory. One goal of this step is to reduce the size of
the memories. The principle is simple: if the lifetimes of arrays are not overlapping (or only partly overlapping, see Figure 1.3), the space reserved in the memory for them can be shared. In practice, this becomes more complex for multi-dimensional arrays. The used technique is called in-place optimization. The in-place optimization step has two major sub-steps: intra-in-place and inter-in-place. Intra-in-place optimizes the internal organization of each array separately (see Figure 1.4), whereas inter-in-place optimizes the reuse of memory-space between different arrays (see Figure 1.5).

Steps for optimizing caches present in the memory hierarchy fit within the data layout optimization step because the applied transformations aim at organizing the layout of data within the memory to optimize cache behaviour. The optimizations focus on the reduction of misses because each cache miss results in an extra (costly) access to higher memory layers. The data-layout optimization step determines the locations of data in memory in order to minimize conflict-misses in the cache. The locations that are chosen for the different arrays in memory have to match with the mapping-organization of the cache. The organization is based on the usage of (parts of) arrays in different parts of the application. Arrays that are simultaneously used shouldn’t use the same lines in the cache (see Figure 1.6). This is avoided by arranging the data in the memory in such a way that different parts of the cache are used by different arrays. The organization of elements within arrays is also considered. Different elements from one array that should be simultaneously present in the cache are placed in memory in such a way that they are not using the same cache lines. Many tradeoffs between different costs are involved when an optimal data layout organization is constructed.

For the conflict miss optimizations we consider static, data dominated applications. The flexibility of cache-memories can perfectly be used for dynamic applications too, however the analyses that can be done in advance are more restricted because of the unpredictable behavior. The optimization of static applications for caches is useful since often a cache is available in the memory hierarchy. Not using the full advantage of the available hardware would be very energy inefficient. Especially in platforms intended for applications with mixed dynamic and static behavior, the presence of the cache is very likely.

## 1.2 Problem Statement

On the one hand we want to use a simple cache-mapping organization policy (a direct mapped cache) to reduce overhead in tag memory and controller complexity. On the other hand this simplifications potentially increase the amount of conflict misses. Therefore we need techniques to reduce conflict misses when using a simple cache-mapping organization.

If the processor requests data that is not present in the cache, this is called a cache miss. As result of this miss, data is moved from the background memory to the cache. Those data transfers are often non-optimal transfers since they could have been avoided with a more advanced cache controller, but the more advanced, the more power it consumes. In order to optimize the cache for power, tradeoffs between extra misses (each extra transfer consumes energy) and the introduction complexity of the
cache should be made. E.g. a cache with more associativity has more freedom to avoid cache misses but also more hardware is needed to provide the freedom.

In a cache with limited associativity, the placing of data in the cache is more or less dictated by the layout of data in the main memory. The particular miss-type related to non-optimal placing of data in the main memory is called data layout conflict misses. So the techniques we need to avoid those misses have to find a good data layout in memory adapted to the mapping function.

1.3 Main Contributions

At Imec, a technique has been developed for reducing the conflict misses in the cache by exploiting the freedom of the layout of the data in the memory. This technique (called ‘tiling’) reserves particular cache locations for each array in such a way that no interference between arrays is present. However, assigning specific sizes to arrays in the cache is not only conflict-miss-related, it also embodies capacity-miss-related optimizations. Capacity miss optimizations have to cooperate with conflict miss optimizations. Therefore the relation to capacity-miss optimizations needs to become clear.

We extended existing data-layout techniques for conflict miss removal, we have applied this to relevant drivers and showed that we are able to remove a significant amount of data-layout conflict misses where previous approaches failed to do so. In order to minimize the energy consumption in an embedded system and to achieve a low cache miss rate, the reduction of conflict misses is combined with the removal of capacity misses. An integration of the two techniques is presented.

The main contributions of this thesis are listed below.

1. Reconsideration of the compiler centric miss classification:
The definitions of compiler centric miss types have been updated and extended. The order of optimizations for different miss types is refined and we set up a simulation environment to support cache experiments.

2. The possibilities of the existing ‘tiling’ technique are clarified and extended:
The existing idea of tiling is extended to a ‘multiple’ tiling structure. With still a controllable amount of addressing overhead, the interleaving of arrays in memory can be varied. Multiple interleaving-patterns that are successively placed in memory can be used to implement the idea of cache in-place. We showed how to deal with the bypassing of arrays within the tiling approach and still avoiding memory overhead. We showed that it is difficult for the tiling approach to beat the scalar-level placing as possible in a fully associative cache.

3. We extended an existing miss prediction model:
For applications not dominated by data reuse this model can be used to steer a global cache-miss minimization technique (conflict misses and capacity misses are simultaneously taken into account). The technique is based on a weighted approach and can help to take optimization-decisions in situations where tradeoffs are involved to overcome the limitations of the tiling approach.

4. We introduced an effective conflict elimination technique for ‘reuse-dominated’ drivers:
For drivers that are dominated by reuse, the tiling approach is used for both the optimization of conflicts misses and for capacity misses. It is shown that the optimizations for both types of misses can co-operate. The decisions are orthogonal and can be expressed both in the data-layout. For a better integration, the technique of tiling is extended with the technique of ‘array padding’ to remove conflict misses. A mathematical property is developed that predicts situations where the data-layout removes the conflict misses most efficiently.

5. Optimized drivers:
We have demonstrated that the optimization techniques work well on drivers representative of both data-reuse and non data-reuse dominated applications. A large reduction of cache miss overhead can be reduced if optimization techniques for capacity misses and conflict misses are combined.
1.4 Structure of this Thesis

The rest of this thesis is organized as follows:

Section 2 introduces the fundamentals of the cache and defines a compiler centric miss type definition needed to support compiler optimization for cache misses.

Section 3 presents drivers that are used to do the different cache optimization experiments.

Section 4 presents data-layout transformations to avoid data-layout conflict misses. The existing tiling approach is described, the extensions are presented and the limitations are discussed.

Section 5 presents optimizations for drivers without data-reuse. The experiments are directly related to the extension of the cache miss model that existed at Imec. A new cache miss model is constructed based on the observed cache behavior. Also the limitations of this model are presented.

Section 6 presents optimizations for drivers with data-reuse. This part contains the main contribution of the thesis. It presents a conflict miss optimization technique that is based on a mathematical property. The technique is extensively applied to drivers to show its effectiveness.

Section 7 summarizes and concludes the main contributions of this thesis and provides directions for future work.
2. Compiler Centric Cache Miss Definition and Measurement

Our systematic and compiler centric miss type definitions preceded by an introduction to the cache fundamentals are presented in this section. The miss definitions are needed to support well-decoupled compiler optimization steps. We also introduce our framework to measure the different miss types.

2.1 Cache Fundamentals

Some definitions given here are more refined from what is seen in computer architecture [12]. This is necessary to give an unambiguous way of characterizing (from a compiler point of view) the various sources of overhead for the transfer of data along the full memory hierarchy and also to support a systematic methodology for the optimization of the cache operation.

2.1.1 Cache in the Memory Hierarchy

Cache is a local memory type typically found in the first layer(s) of the memory hierarchy between the (CPU) foreground memory and the (multiprocessor) shared memory. Other types of local memories can also be present in these layer(s), namely scratch-pad memories. The difference between a scratch-pad memory and a cache is that the latter can determine whether a required data is present in its data memory area. This mechanism allows the cache to take autonomous decisions independently of the CPU. This autonomy leverages the compiler to find an optimal scheduling of the different memory related operations along the complete memory hierarchy.

When the actions taken by the cache memory are fully autonomous (in this case the compiler issues load/store operations considering the memory hierarchy between the cache and the shared memory as a black box), that cache is of the HW controllable type. Otherwise when the read/write operations between the cache and the shared memory can be (partly) steered by the software via compiler directives (e.g. locking specific data in the cache, decisions to bypass data), that cache is of the software controllable type. In the pure SW controlled type, the HW is totally lacking but that corresponds what is then typically called an on-chip RAM or a scratch-pad memory, and not a cache any more.

2.1.2 Mapping Organization

The mapping organization defines the location policy for the different memory blocks in the cache (see figure 2.1). A block of data is considered to be the main unit of data transfer between the two levels of the memory hierarchy.

- Direct Mapped: If each block has only one place it can appear in the cache, the cache is said to be direct mapped. The cache implementing this mapping organization is very attractive for low-power applications, it provides the lowest energy per access due to the simplicity of its
controller and minimum amount of (parallel) tag comparisons (these are needed to decide whether a particular data is present in the cache).

Cache index = (Block address) MOD (Number of blocks in cache)

- Set Associative: If a block can be placed in a **restricted set of places** in the cache, the cache is said to be **set associative**. A set is a group of blocks in the cache. A block is first mapped on to a set and then the block can be placed anywhere within that set. The larger the associativity of the cache, the larger the energy per access required to know whether a particular data is present in the cache.

Cache index = (Block address) MOD (Number of sets in cache)

- Fully Associative: If a block can be placed anywhere in the cache, the cache is said to be **fully associative**. A cache with this mapping organization is impracticable to implement, even if so it will require the largest amount of energy per access.

### 2.1.3 How does a cache work

The cache is organized in blocks with a certain size in bytes. Therefore, every cache block may contain one or more data elements depending on their size. Caches have an address identifier on each block that identifies every single data element contained on it. The address identifier is divided in the block-address and the block-offset. The block address is further divided into the **tag field** and the **index-field**. The block-offset selects the desired data from the block, the index-field selects the set, and the tag is compared against it for a hit. A **valid bit** is added to the tag to specify whether this contains a valid address. If the bit is not set, there cannot be a match on this address.

From the whole address-identifier only the tag needs to be compared to see if it matches the block address of the CPU since: (1) checking the index will be redundant (it is already used to select the set to be checked); (2) checking the offset is unnecessary (the goal is only to know whether the entire block is present or not). For speed efficiency, all possible tags are normally searched in parallel.

### 2.1.4 Replacement Policies

The **replacement policy** of an associative cache refers to the type of protocol used to identify a line in the cache which will be replaced on a cache miss. Note for a direct mapped cache there is no replacement choice since it is dictated by the mapping organization.

Typically a **least recently used** (LRU) [7] type of policy is found in most processors. This is because it has acceptable results and is relatively easy to implement. Other replacement policies include **first in first out** (FIFO) [7] and Belady's **optimal** (OPT) replacement policy [2] which presumes a full knowledge of all the future accesses.

More information is provided now:

- **Random**: This policy is especially suited for uniformly spreading the allocation of blocks in cache. Some systems select the blocks in a pseudo-random way to get reproducible behavior. This policy completely ignores block life-time issues hence it may replace blocks which are needed at the short term.

- **First In First Out** (FIFO): This is a very simple and obvious policy, where the block that has been in cache the longest is the one that is replaced. In practice, with most applications, this turns out to be a bad replacement policy. The fact that a block has been in memory for a long time does not always imply that it will not be needed again.

- **Least Recently Used** (LRU): This is usually the best choice. This policy selects for replacement the block in cache that has not been used for the longest time. If a block has not been recently used, it is reasonably likely it will not be needed anytime soon. Unfortunately, LRU replacement requires that the controller performs some fairly complex record keeping (although still easy for a 2-way associative cache), for example, keeping a constant record of when each block was accessed [12]. Still, this is the most commonly implemented policy in hardware controlled caches.

- **Belady's Algorithm** (OPT): Here the policy is to replace the block that will not be needed for the longest time into the future. This strategy, called MIN by Belady, only optimizes the fetch traffic and does not optimize the write-back traffic. This policy, even though conceptually simple, is **not implementable** in hardware controlled caches because it requires predicting the future. It is possible to implement, either by the designer or the compiler, in software controlled caches if the execution order is **not** dependent on the input (i.e. it is manifest). But
it is very expensive in terms of hardware cost. Moreover, analysis of execution traces of real programs shows that LRU replacement usually comes surprisingly close to Belady's policy. That is, how recently a block has been used is generally a good predictor of how likely it is to be needed again.

Current state-of-the-art processors have hardware controlled replacement policies that is they have hardware counters to monitor the least recently used data in cache. Still, a policy based on compile-time analysis can potentially lead to better results than fixed statistical decisions for programs which are manifest. Indeed for processors like TM1000 [15] with a partly software controlled cache allow the programmer or the compiler to identify which cache line has to be replaced on a cache miss.

2.1.5 Memory update policy

In order to keep consistent the values of the data kept in cache with the values of the data kept in shared memory, the cache controller will eventually have to copy the data value back in shared memory. There are two type of update policies:

- **Write-through** Every time the CPU writes a value in the cache a copy of the corresponding cache block is made in shared memory.

- **Write-back** Only when a cache block is to be flushed out of the cache a copy of the block is made in the memory or in general in the "higher" layer of memory. At that time the block is considered to be "dirty", hence its valid bit is unset.

2.1.6 Block allocate policy

When a data element has to be written in the cache via the CPU and the data is not yet present, a *write miss* will happen. In this situation there are three possible alternatives for deciding how to allocate blocks in cache (policies):

- **Write-around** also called "no allocate on write miss": In this policy, the block (irrespective of the number of data elements contained in the actual block size) is not written in cache but directly in memory (some sort of cache bypass for the memory write operation is assumed here). This policy is typically used together with the write-through memory update policy since subsequent writes to that block will still have to go to memory. From both, an energy and system-bus load, points of view this is not an efficient policy to follow since the advantage of having a L1 memory hierarchy is not exploited on a write miss.

- **Fetch-on-write** also called "allocate on write miss": The memory block where the data element is present is first loaded in the cache on a write miss, followed by the cache store operation (via the CPU). This policy is typically used together with the write-back memory update policy assuming that subsequent writes to that block will be captured by the cache. There are two situations depending on the number of data elements fitting in a block size:
  - **One data element per cache block**: In case only one element can be allocated in the actual cache block (e.g., due to the size of the data element), there is no need to perform the fetch operation preceding the write miss (see the optimal allocation policy). Still, some (non-application specific) HW controlled caches will perform the fetch operation since they do not exploit the application data type sizes (this requires sub-blocking bits). This is the policy assumed in this case and followed by most simulators.
  - **Several data elements per cache block**: When more than one data element is allocated in the cache block, a fetch operation will precede the write miss. This is on the one hand side due to the fact that the transfers between the background memory and the cache are done in blocks of data elements. On the other hand, the cache controller cannot fully predict in advance when all data elements of that block will be written before the block needs to be replaced. A conservative policy is therefore to load the missing block in the cache before any of its data elements needs to be updated. In this case, the copies of the block's data elements, both in cache as in memory, will remain consistent. No matter when the missing block will have to be eventually replaced (hence written back in memory)

- **Optimal-allocation**: The memory block where the data element is present is only loaded in the cache on a write miss when not all data elements of that block will be updated in cache before the block has to be replaced. Note this miss type exist even when an optimal replacement policy is begin used. There are two situations depending on the number of data elements fitting in a block size:
- **One data element per cache block**: In case only one element can be allocated in the actual cache block (e.g., due to the size of the data element), there is no need to perform the fetch operation preceding the write miss (see the optimal allocation policy). Therefore, the optimal allocation policy here is simply not to allocate the block.

- **Several data elements per cache block**: Here the policy is not to perform the fetch operation when not needed. Similarly to what happens for the optimal replacement policy, this policy is **not implementable** in hardware controlled caches because it requires predicting the future. It is possible to implement it, either by the designer or the compiler, in software controlled caches if the execution order is **not** dependent on the input (i.e. it is manifest). However, in this case it is not yet clear whether the fetch-on-write policy will perform almost as good as the optimal allocation policy as it is the case of the LRU replacement policy with respect to the optimal replacement policy.

### 2.1.7 Data-level transfers versus block-level misses

Given that in an application specific context it is possible to exploit data type information coming from the application during the exploration mapping process, it is necessary in our context to differentiate between data level misses and block level misses. Note here a data element is the minimum granularity assumed for the data transfer (one byte).

- **Data-level misses**: data level transfers between the next level of the memory hierarchy and the cache (= memory read miss traffic). This miss granularity is used in the definitions of the compulsory, minimal capacity and block pre-fetch misses in the next paragraph.

- **Block-level misses**: transfers of data level blocks between the next level of the memory hierarchy and the cache. This miss granularity is used in the definitions of the block-allocation, associativity conflict, replacement and data-layout conflict misses in the next paragraph.

In case the block size is bigger than one data element and in case a relative comparison of a data level miss contribution against a block level miss contribution is needed, the number of data level misses measured should be divided by the number of data elements per block.

### 2.2 Compiler oriented Cache Miss classification

In this section we will focus on giving an appropriate classification for the miss types of data transfer overhead. Writes are not considered here. The miss type classification presents the different sources of overhead from a compiler point of view. The conventional cache related terminology is re-considered in the context of hardware but also software controlled caches. The focus here will be on the hardware controlled caches.

![Figure 2.2 Transfers from/to the cache](image)

Four sources of data transfer overhead exist from a non-optimal amount of indexed transfers (in this document we only focus on DTSE relevant transfers, hence array transfers): read and write operations from the CPU/RF (Register File) side, and cache misses and write-backs from the higher layers of the memory hierarchy (see Figure 2.2). Scalar level transfers and/or foreground memory related transfers are typically directly controlled by the low-level compiler and not considered.
The most general classification for a miss is a "non optimal transfer" which results from any move of data from the (shared) memory to the cache memory which is different from the minimally required to keep the copy of the data values in the cache. In our definition set we start with an ideal cache of infinite size with a single data element per block implementing a fully associative mapping organization with an optimal replacement policy and following an block allocation policy. Our target cache is of finite size with a particular block size (it could be just one element) implementing a particular mapping organization (must not be fully associative for energy/area cost reasons) with a particular replacement policy (including potentially the optimal one if it is obtainable) and a particular block allocate policy (excluding the write-around policy which is detrimental for energy).

The miss types defined in our classification are clustered into three categories related to their nature:

- **Platform independent misses**: these are miss types purely application specific and independent on any physical characteristics of the cache memory. This is a valid miss type for purely HW and SW controlled caches and mixes.

- **Controller-policy independent misses**: these are miss types dependent on the physical characteristics of the cache like its geometry (e.g. number of lines, block size). But these miss types are independent on the policies of the cache. These are valid miss types for purely HW and SW controlled caches and mixes when prefetching is exploited in the purely SW controlled case.

- **Controller-policy dependent misses**: All the miss types clustered in this category are purely dealing with cache controlling issues. These are valid miss types for purely HW controlled caches, less relevant for mixes and irrelevant for purely SW controlled caches (pure compiler controlled policies).

### 2.2.1 Platform independent miss types

- **Compulsory misses**

These misses are due to the fact that a data value is not present in the ideal cache the first time it is required by the CPU. A miss will take place to transfer the value to the cache. These transfers are mostly due to transfers of data from the test-bench (we assume here the data to be initially available in main memory) toward the algorithm. Any data written from the algorithm back to memory (back to the test bench) can be considered a compulsory write-back. Compulsory misses are the only miss type that requires a change in the execution order of the application in order to be eliminated. All the other miss types (see further in this section) are data-layout related for a given execution order.

There are two main classes of compulsory misses: avoidables and unavoidable. The avoidable type can be further classified in four subtypes. Unavoidable misses are the remaining ones and only depend of the application itself irrespective of the execution order of the read/write operations. The sub-classification for avoidable misses is:

- **Redundant data transfers**: To avoid them, this requires to apply data-flow transformations that changes the functionality of the application (although the I/O behavior is kept). An example are advance array substitution and propagation techniques that removes redundant accesses in the code.

- **Background/foreground memory related**: To avoid them, this typically requires to change the loop structure of the application to change the life-time of the arrays involved in background memory such that they can be stored now in foreground memory.

- **Bypassable transfers**: these are data transfers for values which are read only once and therefore there is no advantage to store them on the cache.

- **Block prefetch misses**: even for an ideal cache (as defined in this section), there is a subclass in the block prefetch misses (see definition further in this section) that could be avoided by changing the life-time of the data elements within the blocks (i.e., by means of SBO type of transformations).

Since the definition of this miss type is at the data transfer level (single data element), the misses measured here are data-level misses.

**Definition:**

\[
\text{compulsory misses} = \# \text{ of different data element values read through the ideal cache}
\]
CHAPTER 2. COMPILER CENTRIC CACHE MISS DEFINITION AND MEASUREMENT

Note 1: We differentiate the data values accessed (read) through the cache from those not accessed through the cache (i.e.: via cache bypass or via local memories). We only consider the first ones.

Note 2: At this level no concept of address space exist yet so we don't differentiate between (different) values coming from the same memory address from those coming from different memory addresses.

2.2.2 Controller-policy independent miss types

- **Minimal capacity misses**
  In a fully associative finite cache, some data values are discarded and retrieved later because of size limitations. If the cache is full the data value to discard is the furthest used in the future (optimal replacement policy). Since the definition of this miss type is at the data transfer level (single data element), the misses measured here are data-level misses.

  **Definition:**

  \[
  \text{minimal capacity misses} = \left( \text{#misses in ideal cache of the same size as the target cache} \right) - \text{compulsory misses}
  \]

- **Block prefetch misses**
  This miss type is related to the block size of the cache. It characterizes the miss occurring from a data not present in a loaded block. Indeed when a miss occurs, a number of data corresponding to the block size is fetched from the main memory to the cache and fills an entire cache block. In case the other data loaded to the same block are flushed before being read, there are block prefetch misses. This amount of misses gives an indication on how efficient is the layout in memory to adapt the ideal situation where all prefetched data are the required ones. In case the block size corresponds to only one data element, there is no block prefetch miss because there is no prefetch. Since the definition of this miss type is at the block transfer level (multiple data elements per cache block), the misses measured here are block-level misses.

  **Definition:**

  \[
  \text{block prefetch misses} = \left( \text{#misses in ideal cache of the same size, same #blocks and same block size as the target cache} \right) - \text{compulsory misses} - \text{minimal capacity misses}
  \]

2.2.3 Controller policy dependent miss types

- **Block allocation misses**
  At this point the dimensions of the cache are supposed to be fixed and we measure the effects of the selected block allocation policy. Some caches (especially the HW controlled types) fetch the block from main memory on a block write miss (fetch-on-write block allocate policy). This is done to ensure the consistency between the copy of the data in the cache and the value stored in memory. This extra read operation (block allocation miss) is needed in case the block is replaced before all its data elements have been updated. Since the definition of this miss type is at the block transfer level (multiple data elements per cache block), the misses measured here are block-level misses.

  **Definition:**

  \[
  \text{Block Allocation misses} = \left( \text{#misses in ideal cache with same size, same #blocks, same block size and same block allocate policy as the target cache} \right) - \text{compulsory misses} - \text{minimal capacity misses} - \text{block prefetch misses}
  \]
CHAPTER 2. COMPILER CENTRIC CACHE MISS DEFINITION AND MEASUREMENT

Note 1: There are two subclasses of block allocation misses: (1) those that require a change in the execution order in order to be avoided (we consider these as unavoidable from a data layout point of view) and; (2) those which can be avoided in a SW controlled context.

Note 2: In case the block size is one data element, the optimal allocation policy is to have no allocation of a block.

Note 3: The Block prefetch and Block allocation miss types could be merged together in a common miss type called "Block miss" in case no concern is given to the effects of the allocation policy.

- **Associativity conflict misses**
  A cache which is not fully associative will limit the replacement freedom, even if we exploit the mapping by optimal data layout. For instance, if the same block has to be loaded in the cache several times (due to any of the previous miss types), the block may be allocated in different places against of what the chose replacement policy will select. Hence, a particular data-layout could eventually avoid that the first of the allocations of the target block will conflict with an existing one in the cache, but not for the subsequent allocations of the same block. Since the definition of this miss type is at the block transfer level (multiple data elements per cache block), the misses measured here are block-level misses.

  Definition:
  \[\text{associativity conflict misses} = (\text{#misses in cache with same associativity, same block allocate policy, same #blocks and same block size as target cache using an optimal data-layout})\]
  - compulsory misses
  - minimal capacity misses
  - block prefetch misses
  - block allocation misses

- **Replacement misses**
  An optimal replacement strategy optimally uses the capacity of the cache and misses can be avoided because this policy exploits knowledge of the future accesses. For instance data required in a near future will not be replaced. However, with a non-optimal replacement policy it can perfectly happen that data are flushed just before being required, which is source of extra misses. Therefore, a suboptimal replacement strategy increases the replacement misses. In advanced multi-media caches, the replacement policy is software controlled. Since the definition of this miss type is at the block transfer level, the misses measured here are block-level misses.

  Definition:
  \[\text{replacement misses} = (\text{#misses in cache with same associativity, same block allocate policy, same #blocks, same block size and same replacement policy as target cache using an optimal data-layout})\]
  - compulsory misses
  - minimal capacity misses
  - block prefetch misses
  - block allocation misses
  - associativity conflict misses

In case the target cache is direct mapped. The measurements of the replacement misses for that particular cache can be skipped due to the lack of replacement policy in such cache (due to its fixed mapping organization policy).

- **Data layout conflict misses**
  Data layout conflict misses are by definition the conflict misses that can be eliminated by choosing an optimal conflict oriented data layout (by means of array interleaving techniques, array padding, selective gap placement and/or any combination of all these). The (total) conflict miss count = # associativity conflict misses + # data layout conflict misses.
Definition:

\[
data\ \text{layout conflict misses} = (\#\text{misses in cache with same associativity, same replacement policy, same block allocate policy, same \#blocks and same block size as target cache using real (non-optimal) data-layout})
- compulsory misses
- minimal capacity misses
- block prefetch misses
- block allocation misses
- associativity conflict misses
- replacement misses
\]

2.3 Framework for the measurement of the different cache miss types

This paragraph describes how to measure the effects of the different types of data cache misses according to the introduced classification. Both from conceptual point of view as from practical point of view. This is necessary to support our experiments. Currently some miss types are not the target to optimize (we focus on the data-layout conflict misses) or no optimization methods are available, therefore it will be indicated how we can eliminated the influences of those type of misses when other misses are measured.

Current cache simulators are well-suited for measuring the miss types defined in the eXlstmg architecture centric [12]. However, our classification is compiler centric, therefore currently no simulator is available for measuring the distinguished miss types in a straight-forward way. Still, we need to use a cache simulator to measure the different types of cache miss. As it has been defined, each miss type is related to a particular non-ideality of the controller. Hence, this is very handy to use a parametrisable simulator.

According to our cache miss, measurements can start from the parameters associated to an ideal cache. Cache non-idealities are successively introduced in the simulations in order to reveal the type of miss related to a specific non-ideality. We assume that the different types of miss are accumulative, this means that for each non-ideality introduced, only one new type of miss appears. For that, we will follow the definition for the different miss-rate types: data-level vs. block-level with focus on the first one. Therefore, the basic principle for measurement is to subtract two consecutive simulation results.

2.3.1 Measuring miss types one by one

Compulsory misses

According to the definition, a compulsory miss occurs when a data value is not in the cache the first time it is required by the CPU. In theory, a simulation with an ideal cache would provide the amount of compulsory miss. However, it is currently not possible to make such experiments with our simulators. Because in the drivers (or code examples) to experiment, array initializations related to the testbench should not be simulated. Arrays from the testbench are supposed to be already initialized before the execution of the algorithm and should therefore not be responsible for extra compulsory misses. We assume here the actual implementation of the system will allow a direct communication between the I/O and the memory system. Therefore, the current method is to count manually from the code the amount of data that is accessed for the first time (the 'input' data from the test bench to the application; this can be more then the amount of distinct addresses counted by the simulator). No distinction will be done here between the different subtypes of compulsory miss and the block size is assumed to be one data element to ignore the block effects. This is feasible for medium size types of C codes.

Minimal capacity misses

Assuming the number of compulsory misses is known, a simulation with an ideal cache is done but now in a range of finite sizes. As long as all data alive (at the same time) fits in the cache, no extra misses should be measured. However, for smaller caches, minimal capacity misses are present. Since not all alive data may fit in the cache, some data will be flushed out and when needed again later on a capacity miss occurs. Therefore, the number of extra misses measured for finite cache size represents the amount of minimal capacity misses assuming the rest of the cache parameters to be ideal.
Block prefetch misses
Those misses are currently neglected. This is possible by setting the cache block-size equal to the size of one data element. This implies that the block level miss-rate is equal to the data-level miss rate and that there is no prefetch.

Block allocation misses
Also those misses are currently neglected. This is possible by setting the cache block-size equal to the size of one data element and disabling the allocation of blocks. Therefore the simulator is adapted so the ‘fetch on write’ block allocation policy could be disabled.

Associativity conflict misses
To measure the amount of associativity conflict misses, only doing cache simulations with a fully associative and a limited set associative cache is not sufficient because data layout conflict misses (see next section) will also be measured at the same time. It is therefore necessary that the optimal data-layout is considered when simulating for the limited set associative cache. In this case, our most advanced data layout optimization is assumed to be optimal and the minimal associativity miss can be measured. Since determining the optimal data layout is hard (if not infeasible) for complex algorithms, in many cases this number cannot be exactly computed. Therefore simply the number of associativity misses is measured instead of the number of minimal associativity misses.

Replacement misses
Until now the optimal replacement policy is assumed [2]. According to the definition, it is possible to measure the number of replacement misses from simulations when changing the replacement policy from the optimal one to a non-optimal one (like LRU). We can consider the extra misses introduced when using the non-optimal replacement policy as being the replacement miss count. Note that those type of misses are not present when the associativity is reduced to a direct mapped cache, since the placing of the elements is completely fixed, there is no choice which element can be replaced.

Data layout conflict misses
The amount of data layout conflict misses can be measured by simulating the codes with and without data layout organization. Once again, the assumption will be made that our most advanced data layout technique is optimal. Since the other types of misses are supposed to be measured at this point of time, the simulations are done with the target cache parameters (non-fully associative and a non-optimal replacement policy).

2.3.2 The simulation environment
In order to measure the miss types and to study the influence of optimizations, a simulation environment is needed. The measurements of the different miss types require a lot of flexibility on the simulator. Assuming an architecture with a main memory and a data-cache, we have to be able to set the following parameters:
- cache size,
- line (block) size allowing small sizes in order to store one data element only (e.g. 1 integer = 4 bytes),
- possibility to disable the block allocate policy (on write misses)
- different associativities (fully associative, set associative, direct mapped),
- replacement policies (OPTIMAL and LRU)

A simulator that can fulfill many of those whishes we found in ‘The SimpleScalar Toolset’ [4]. We used the functional cache simulator called ‘Sim-Cheetah’. It is able to generate simulation results for multiple cache configurations with one single simulation. The Cheetah engine simulates fully associative caches effectively as well as simulating an optimal replacement policy (optimal for read only streams). This simulator didn’t support the disabling of the block allocation policy, but since the source code was available, we could implement this in the simulator. The Simplescalar architecture is a close derivative of the MIPS architecture [13].
We encountered also problems in using this environment. The first problem is the fact that this architecture doesn’t support cache bypass, however we incorporated this in our experiments. But the main problem is related to scalars. Since we want to measure the effect of array-signals being stored in the cache, we had to eliminate the effect of scalars in the cache. For small drivers this was no problem,
since the architecture used by the Simplescalar toolset supports a register file of 32 integer registers. However, for the simulations with bigger applications, this was not enough. We clearly noticed that the amount of accesses to the cache was higher than the expected amount of accesses, very likely caused by 'register spilling'. The problem of scalars we solved by using a different environment called Trimaran [10] to compile our drivers. Trimaran is an integrated compilation and performance-monitoring infrastructure. The architecture space that Trimaran covers is characterized by HPL-PD, a parameterized processor architecture. This allowed us to increase the size of the general-purpose register file up to 512 registers. The Trimaran environment is used to compile the code of our drivers and to produce a trace file that contains all memory accesses. Afterwards an adapted version of the SimCheetah simulator is used to read the trace file and generate the cache statistics. This combination of Trimaran and Simplescalar was the best environment we had available.

We verified the correctness of the produced trace files. We constructed an instrumented version of our driver code that produced a trace file 'on the fly', i.e. while running the application, each memory access is recorded and the accessed address is added to the trace file. The simulation results using this trace file where compared to the simulations on the trace files made by Trimaran and the miss-rates turned out to be equal.
3. Application drivers

This section introduces data-dominated drivers that are used to do our experiments related to conflict misses and to show the effectiveness of the miss-removal approaches. We distinguish two types of drivers: with and without data-reuse optimization opportunities. The focus of the conflict-miss optimization technique is different for both types of drivers.

3.1 Wavelet

For our first experiments we used the kernel of a 2-dimensional wavelet compression algorithm that can be used for image compression. Wavelets are mathematical functions that cut up data into different frequency components and then study each component with a resolution matched to its scale. Wavelet compression is an efficient and flexible method. Our kernel contains several small arrays. Some arrays are accessed more frequently than other arrays. The references to the arrays are contained in different loopnests. We used this driver to test our cache optimization technique for non-data-reuse dominated applications.

3.2 Voicecoder algorithm

Most of the communication applications like GSM require digital coding of voice for efficient and secure storage and transmission. This algorithm is a Linear Predictive Coding (LPC) vocoder which provides an analysis synthesis approach for speech signals. In this algorithm, speech characteristics are extracted from a frame of 240 consecutive samples of speech. Consecutive frames share 60 overlapping samples. All the information to reproduce the speech signal with an acceptable quality is coded in 54 bits. The LPC analyses use auto-correlation to obtain the best matching predictor values. Further details about the algorithm are left out, since they are not crucial for the understanding of the optimizations. More interesting is the fact that the algorithm consists of 30 small loop nests. Many small arrays are used. Most of those arrays are accessed quite regular, i.e. all the different data elements are accessed and there are no smaller groups of elements being used more often than other groups. Therefore we used this driver for the experiments we did with the applications that are not reuse-dominated.

3.3 Cavity Detection

The Cavity Detection algorithm [3] is a medical image processing algorithm which extracts contours from images such that they can be interpreted more easily by physicians to detect brain tumors. The initial algorithm consists of four functions, each of which has an image frame as input and one as output as shown in Figure 3.1. In the first function, a horizontal and vertical gauss-blurring step is performed in which each pixel is replaced by a weighted average of itself and its neighbours. In the

![Figure 3.1 The Cavity Detection algorithm](image)
second function (compute edges), the difference with all eight neighbours is computed for each pixel, and this pixel is replaced by the maximum of those differences. In the last function (detect roots), the image is first reversed, then the maximum value of the image is computed and each pixel is replaced by the difference between this maximum value and itself. Next, we look for each pixel whether a neighbour pixel is larger than itself. If this is the case, the output pixel is false, otherwise it is true.

Frequent accesses to the intermediate frames in the background memory results in very high power costs. However after applying different DTSE steps, the total required data storage is significantly reduced. In this algorithm, more than once a pixel is accessed together with all its neighbours. If the complete image is scanned using this access pattern, there is a lot of reuse of elements. The optimization for this reuse proposes a small memory to store elements in the foreground of the memory architecture. For this memory a direct mapped cache is used. This makes the driver interesting to optimize for data layout conflict misses in the context of data reuse since we have to ensure that the mapping of particular set of elements from a big array to a small cache does not result in conflicts between individual elements of the set.

3.4 QSDPCM

The Quadtree Structured Difference Pulse Code Modulation (QSDPCM) algorithm is an interframe compression technique for video images [14]. It involves a hierarchical motion estimation step, and a quadtree based encoding of the motion compensated frame-to-frame difference signal.

A global view of the QSDPCM algorithm with its main signals is given in Figure 3.2. Most of the submodules of the algorithm operate in cycles with an iteration period of one frame. For the first step of the algorithm (motion estimation in the 4x4 subsampled frames), the previous frame must already be reconstructed and both the current and the previous frame must be 4x4 subsampled. The 4x4 motion estimation produces motion vectors that are used to indicate the regions where the more detailed 2x2 motion estimation is done. This estimation produces vectors for the motion estimation with one pixel accuracy.

The motion estimation step is the dominant part of the driver in terms of data-storage and memory optimization. Therefore the signals to optimize are those related to the motion estimation. The version of the QSDPCM we used, is optimized by different DTSE optimization steps. Loop merges are applied

![Figure 3.2 The QSDPCM algorithm and its main signals](image-url)
in order to bring the production and the consumption of the data elements close to each other. Because of this locality, less storage size is needed for the signals. In this driver we distinguish relatively big signals from which only a selective amount of elements is heavily accessed simultaneously. It is worth noting that the algorithm spans several pages of complex C code.
4. Data Layout Transformations for Cache Miss Reduction

In this section it is explained what is the influence of the data-layout on cache misses. The relation between capacity misses and data layout is indicated. It is explained what is the nature of conflict misses and how the amount of conflict misses can be influenced by changing the data-layout in main memory. The freedom in changing the data-layout is discussed in relation to different types of overhead that is introduced. The existing tiling approach is described, its extensions are presented and the limitations are discussed.

Data-layout describes the placing of data in memory in order to influence the behaviour of the cache. Related to caches we basically distinguish two types of data layout. The first focuses on ordering data in main memory on the granularity of blocks equal to the line size of the cache. The second data layout focuses on rearranging data within those blocks. The latter is related to block prefetch misses (data-elements with equal lifetimes should be grouped in a block in order to prevent that elements are brought to the cache that are fetched at distant times). This data layout is not related to our goal of eliminating conflict misses. Therefore the data-layout considered here (simply referred to as 'data-layout') is of the first type. The effect of the second type of data layout is eliminated by selecting a cache block-size (= line size) that contains one data-element only.

A data-layout does not influence the behaviour of a fully associative cache since for a fully associative cache the location of blocks in main memory (with a size equal to a cache-line) does not influence the location of blocks in the cache. This is because of the fact that each block in main memory has the possibility to be placed at any cache-location. At the moment a block is brought to cache, it is placed at any location determined by the replacement policy (regardless of its location in main memory). A fully associative cache is very costly to implement. Therefore in real caches the associativity is limited. A block that is brought from main memory to cache cannot be placed at any cache-location anymore. For an n-way associative cache, the amount of different locations is limited to n. The associativity is limited maximally if a block in main memory has only one location in cache to be stored (the direct-mapped cache). When a fully associative cache is replaced by a direct mapped cache, two important things change:

- For a direct mapped cache, the replacement policy is not present anymore because there is no choice to make which element to replace. For a direct mapped cache, the locations of blocks in main memory do completely determine the location of blocks in the cache. So the benefits of the intelligence of the replacement policy are gone; the cache controller has no influence on the cache performance anymore.
- The placement of data in main memory (data-layout) influences the cache-performance.

4.1 The Optimizations for Capacity Misses expressed in the Data-layout

This paragraph briefly indicates the relation between data layout and capacity misses. Minimal capacity misses are those misses we can measure with a fully associative cache with an optimal replacement policy. The optimizations we do for capacity misses target to have not more than this minimal amount of capacity misses if caches are used that are less ideal than this fully associative cache with its optimal replacement policy (e.g. our target cache, the direct mapped cache). Basically this optimization decides which elements should be stored in the cache taking into account the limited storage-capacity of the cache. This decision is taken by the optimization step called Memory Hierarchy Layer Assignment. To the memory hierarchy layer containing a cache, specific arrays (or part of arrays) are assigned.

For a direct mapped cache, the decisions of the array sizes to be stored must be expressed in the data-layout. This is done by reserving specific cache regions for specific arrays. Although the available space is guaranteed, the placing of the different elements within each array is not taken into account, because those issues are related to conflict misses.
4.2 The Nature of Conflict Misses in relation to Data-layout

This paragraph describes how it can be that data layout conflict misses are the result of non-optimal placing of data-elements in the cache, so elements are competing for the same cache locations because their mapping-freedom is constrained.

Data layout conflict misses are those misses that could have been avoided if data-elements are placed differently in the main memory. The nature of those misses is that a set of elements (the ‘working set’) that is planned to be stored in the cache (there is enough space reserved for this set) cannot be present in the cache as a whole simultaneously. This is because elements within the set are mapped to equal cache locations. This indirectly means that other cache-locations are not used at this time. So a conflict miss optimization has to ensure that each element that is planned to be stored in the cache, is mapped to a free cache location. Therefore a global way to describe data-layout conflict miss elimination is: avoid that elements belonging to the same working set are stored in the main memory on locations that map to the same cache lines. For a direct mapped cache the mapping organisation is such that elements from the main memory that are placed on distances equal to the cache size map to the same cache address, so those locations shouldn’t be allocated for elements within the same working set.

The freedom of data-layout is big because theoretically each block of data may be placed at any free location in main memory in order to obtain a good cache-performance. However, this freedom has practical problems, since the addressing of the data becomes impossible if each block has its own location (a lookup-table will be needed to locate the blocks and since we try to reduce memory-accesses, the extra accesses to such a lookup-table are not acceptable nor is the extra overhead in memory usage).

Although the freedom in data-layout is big, it has more restrictions than the freedom that is present with a fully associative cache. This restriction is caused by the fact that by a data-layout the location of blocks can be decided only once. This directly fixes the location where this element is going to be mapped in the cache and also fixes which elements it will replace in the cache. Blocks are typically brought to the cache more than once (e.g. due to capacity issues, blocks are discarded and fetched again later). In the case of a fully associative cache, the locations where the block is stored in cache will likely to be different for each time the block is stored again. When the location of a block in cache is determined by the location of a block in memory (data-layout), each time the block is brought to cache its location is determined by the location in memory and will be equal each time. The same is true for n-way associative caches. The place of blocks in cache is limited to ‘n’ locations; one of those locations is not necessary the one that would have been chosen by the cache-controller of the fully associative cache. The misses caused by this restriction of placing possibilities are the misses defined as associativity misses.

If considered well, there is no situation ‘without’ data-layout. Always a location is chosen for arrays in memory. However, if arrays are simply allocated linearly after each other in memory, no influences on the cache are taken into account. This doesn’t mean that the cache-behaviour is always bad, but no guarantee can be given either. The data considered in the data-layout are those signals that are stored as arrays. Scalars are not taken into account. Scalars could potentially interfere with other data in the cache, but this problem is not approached here. This can be solved by a register-file that is big enough to store all registers so no register spilling is present (as these found in VLIW processors e.g. Trimedia). Other solution are: to reserving a particular part of the cache for scalars only, or using a separate cache to store the scalars.

4.3 The Principle of Tiling

This paragraph introduces a specific data layout scheme called tiling (not related with loop tiling!). We’ll show that tiling is an effective way to get control over control data-layout conflict misses. First, the existing idea of using a single layout is presented [8] [9], then this is extended to the use of multiple layouts. The different types of overhead caused by tiling are discussed and the practical implementation of tiling is explained.
4.3.1 Single interleaving-data-layout

The tiling approach aims to change the placing of arrays in main memory, so that it is known beforehand which cache lines are used by which arrays. In order to do so, the main memory is layouted in repeated patterns with a size equal to the cache size. The space within each block is divided in tiles. Each tile is used by one specific array.

During the execution of an application, different data is brought to the cache. In general there is not enough space in the cache to store all the data that is used in the part of the application that is currently executed, since the cache space is limited e.g. because of power savings. Therefore smaller working sets will be selected to be stored in the cache and capacity misses will occur. However we want to ensure that the selected working set stays in the cache, it should not be overwritten. Therefore tiling assigns different arrays to different (unique) parts of the cache by breaking up the array in pieces and a repetitive placing of those pieces in memory on distances equal to the cache size (figure 4.1). The size allocated for one array in cache is indicated as the tile-size of the array. If all arrays present in the application get their own place in the cache, we speak about the situation with one 'single interleaving-data-layout' because in order to assign arrays to different parts of the cache, data in main memory should be interleaved.

A single interleaving data-layout has several consequences that will be discussed. The first remark is that as soon as each array is assigned to its own locations in cache, there will be no influences from one array to the other anymore. The so-called 'inter-array-conflict-misses' are completely eliminated. However, the limited space reserved for one array can cause extra conflicts within the array, the so-called 'intra-array-conflict-misses'. The distribution of the cache-space over the different arrays has to be determined. The first goal of this distribution is to trade-off intra-array-misses of the different arrays in order to minimise the sum of misses. The practical realisation of this division of the cache possible results in extra memory-usage. Therefore the trade-off is not only a trade-off between intra-array-misses of different arrays, but also a trade-off between intra-array-misses and memory-overhead. A third type of overhead is the overhead in addressing. But the type of data-layout that is considered (allocating parts of the cache for parts of arrays) is realisable within reasonable margins of addressing-overhead (see section 4.3.6).

Only arrays that are alive simultaneously should be packed in one interleaving layout. Arrays that are not alive, should not occupy a part of the cache and make it 'unusable' for other arrays. This can be the case if the layout is restricted to a single interleaving data-layout only. Therefore more freedom is needed in the layout.
4.3.2 Multiple interleaving-data-layouts

An interleaving-data-layout can be seen as a group of arrays that are assigned to unique partitions of the cache. This is extended to multiple groups of arrays. Within each group the arrays are assigned to unique partitions of the cache. This is a multiple interleaving-data-layout structure. In principle it is possible now that there are conflicts between arrays from different interleaving layouts because only within one layout the cache-lines are assigned to one unique array.

If a multiple interleaving data-layout structure is used in stead of a single interleaving layout structure, arrays that are present in an application should be grouped. Each group of arrays is layouted together, so the total cache-space is distributed over the different arrays of the group. The space per array in the cache in the multiple-layout case is typically bigger than in the single layout situation, because there are less arrays per layout that need to be divided over the total cache space. This is good for the elimination of intra-array-conflict-misses. The drawback is that it is not ensured any longer that all inter-array-conflict-misses are eliminated, because only if arrays that map to the same cache lines don't have overlapping lifetimes, the inter-array-conflict-misses are eliminated (if two arrays that are alive simultaneously use the same cache-lines they will very likely cause conflict misses because of alternately occupying one location).

The amount of conflict misses (because of the common use of cache lines by two arrays) can be big if two overlapping arrays are simultaneously alive in a dominant loop-nest, but can also be small if the usage of the arrays in the overlapping period is small. Here a difference between caches and memories becomes clear: arrays in memory may not reuse their locations as long as the lifetimes of the arrays do overlap (the application becomes incorrect if overlap is enforced, or at least extra control mechanisms are needed), whereas arrays in the cache can reuse locations if their lifetime does overlap without harming the application's functionality (the cache controller will correct the situation). The extra inter-array-conflict-misses should be smaller than the benefit in intra-array-conflict misses to have advantage of the introduction of a multiple interleaving data-layout. So the grouping of the arrays should be based on the overlap in lifetimes. Preferably arrays with overlapping lifetimes are combined in one layout.

Extra freedom that can be explored in the multiple-layout case (that was not present in the single layout case) is the order of arrays in an interleaving-data-layout. Since the order determines which arrays are actually using which cache-lines, there is a possibility to minimise the inter-array conflicts. Within one interleaving-data-layout the sum of all the tiles typically fills the complete cache (otherwise a part of the cache is wasted). In the multiple-layout case it can be interesting to assign sizes to arrays whose sum not fully fills the cache. The free place that is left in the cache can be filled by an array of on other interleaving-pattern.

4.3.3 Memory overhead

The filling of the cache is determined by the placing of the arrays in main memory. Particular memory locations should be used by particular arrays and other locations definitely not. This results in unused memory locations.

Assume a direct mapped cache with $C$ lines (1 element per line). The aim of tiling is to reserve a typical part of the cache (e.g. $TS$ lines at position $P$) for one array. This can be obtained by breaking up
the array in pieces of \(TS\) elements and place the first elements of those pieces at memory-locations \(P, C+P, 2C+P\) ... until all pieces are placed in main memory. The aim is to create in each memory-part with size \(C\) the same pattern as the pattern that is desired in cache. If more than one interleaving-data-layout is used, the different layouts are stored consecutively in the memory. In the most simple case, each layout starts where the previous layout finishes (see figure 4.2), however there will be situations where layouts can be partly merged (the new layout fills the free locations that are left at the end of the next layout).

The interleaving-pattern of one layout is repeated in memory until the last piece of each array gets its location in memory. If an array gets a small tile in cache, it is divided in more pieces and has to be repeated more often in order to store all elements compared to the case where a bigger tile is assigned. Memory-overhead is the result of differences in the amount of repetitions needed to store all elements of an array. The part of the memory used by one layout is as long as the array that is repeated most. Other arrays having fewer repetitions leave holes at the end in the memory. Those holes won't be used anymore by other arrays and therefore are wasted space. The amount of wasted locations can be calculated. The maximum range of memory needed to store a single-interleaving-layout is determined by the array that has the maximum number of repeated pieces. The memory-range \(M\) needed by array \(x\) is calculated by the amount of needed repetitions multiplied by the cache size:

\[
M_x = \left\lceil \frac{\text{declared size}}{\text{cache tile size}} \right\rceil \times \text{cache-size}
\] (4.1)

The total memory usage is obtained by finding the maximal range ever needed:

\[
tot = \max(M_x)
\] (4.2)

The amount of wasted memory is:

\[
\text{wasted mem} = tot - \text{sum declared sizes}
\] (4.3)

![Memory layout](image)

Figure 4.3 Unused memory locations because of different amount of repetitions

The amount of wasted memory can be used as a cost when deciding the right data layout. Another way of working is to give the memory as a constraint. The constraint on memory directly determines a lower bound on the tile size that can be assigned to particular arrays.
4.3.4 In-place transformations

In the introduction it is stated that the data-layout transformations also involve in-place optimizations. In-place analyses the life-times of data elements in order to determine if memory locations can be reused. Elements with non-overlapping lifetimes can be mapped to the same memory locations [5]. Obviously the same is true for cache locations. In this paragraph we will see how in-place transformations can reduce memory overhead related to data-layout conflict miss optimizations in the cache.

The type of in-place optimization that is most decoupled from other data-layout transformations is the **intra-array-inplace**. This optimization reduces the storage size for each array independently. A minimal window size is determined by the maximal amount of elements alive at the same time. The storage size can be reduced to the size of this window. The reduction of the storage size of a particular array does not directly change decisions for the tiling approach, however, less tile repetitions are needed in the main memory to store the complete array. Only if the size of the array with the most repetitions is reduced, the memory overhead will decrease.

The second type of in-place optimizations is **inter-array-inplace**. This optimization determines if different arrays have fully disjoint lifetimes and can share the same memory locations. This type of optimization is much more coupled with the cache optimizations, because it changes the locations in main memory just as the tiling approach. We distinguish two types of inter-array-inplace. **Cache inplace** and **memory inplace**. We speak about cache inplace if arrays share the same cache lines. We speak about memory inplace if arrays share the same locations in main memory. Due to the mapping function of direct mapped caches (that projects memory locations on cache locations), memory inplaced arrays will automatically be cache inplaced, but cache inplaced arrays don’t need to be memory inplaced.

---

**Figure 4.4 Memory and cache layouts for different inter-array inplace situations:**

- a. initial situation, no inplace;
- b. memory and cache inplace;
- c. only cache inplace
If two arrays are memory replaced, this potentially reduces the main memory size since less memory locations are used. However, the main memory overhead due to the tiling approach is not changed as long as the proportion (declared size / tile size) does not change. The effect of memory inplace on memory overhead is indirect, because memory inplace implies cache inplace and cache inplace can affect memory overhead.

Cache inplace can be done either by inplaceing arrays in main memory (Figure 4.4b) or by using the multiple interleaving layout (Figure 4.4c). It is possible that arrays cannot be inplaceed in memory but can be inplaceed in the cache, because the small part of an array stored in the cache can have a more limited life-time than the complete array. When using the multiple interleaving layout to inplace arrays in the cache, the arrays that have to be inplaceed must be part of different interleaving layouts. In case multiple interleaving layouts are used, the layouts have to be placed after each other in the memory. This increases the main memory usage, therefore it is preferred that cache inplace is combined with memory inplace.

Inplace of arrays in the cache creates more room in the cache because tiles that initially used separate cache lines can now overlap in the cache and use the same cache lines. So compared to the non-inplaceed situation, there is room to extend tiles. This can be used to reduce the memory overhead (now the proportion declared size / tile size can be changed).

4.3.5 Source code transformations for the tiling approach

When using the tiling approach, elements of one array stay in the same order and the pieces are still parts of the original array. Therefore the new addressing can be obtained by small modifications to the original address-expression. It will be shown how this can be practically implemented.

The performed data-layout transformations are high-level transformations (i.e. transformations in source code of e.g. Ansi-C). The big advantage is that on this level, the structure of access patterns is much better known than with low level code. In order to show that the proposed data layout is realistic and feasible, the change of addressing is illustrated. Before transforming array-indices it is assumed that the arrays are single dimension only. In other words: we assume the storage order to be fixed. To break up the original array in pieces, a modulo-operation is used. To allocate a different location for each piece at positions $P$, $C+P$, $2C+P$, the integer-division is used and a multiplication by cache size. Finally an offset $P$ is added. Without loss of generality the examples are expressed in C-syntax. Assume that originally there is an array $A$ accessing its elements at locations determined by an arbitrary expression:

```
... = A[expr];
```

After the address-transformation the new addressing will look like:

```
... = A[expr % TS + expr / TS * C + P];
```

The elements 0 to $TS$ are placed at location $P + 0*C$, the elements $TS$ to $2*TS$ are placed at location $P+1*C$ etc. Next to the change in addressing all arrays are going to be stored in one new array that is representing the main-memory. When the code is compiled and the linker allocates space in memory for the data, the new array is continuously allocated in memory and the mapping to the cache is going to be like assumed. In C++ the different arrays can be brought into one array by the following change of code:

- the declaration of the original array is removed.
- the name of the array is used to declare a pointer to the new ‘MainMemory’ array (this pointer can directly be used to incorporate the offset $P$ and should not appear in the address anymore).

The original declaration

```
int a[10];
```

changes into

```
int * a = (int *) ((void *) &MainMemory[P]);
```
Obviously the size of the 'MainMemory' array should be large enough to hold all the data of the different arrays at the locations they are placed. Extra attention has to be paid to cases where different data-types with different sizes are used: elements in the MainMemory array are of the char-type (1 byte), matching the basic granularity used to express the cache-size. If for example arrays of integers are mapped to the MainMemory array, it has to be taken into account that the integers have a size of 4 bytes. When calculating tile-sizes and offsets, this should be taken into account.

We made a tool that can automatically apply the source code transformations to support our experiments. Since for experiments it is needed to be able to apply transformations quickly and they are quite sensitive to manual errors. The input to the tool is the cache size, and for all arrays whose addressing should be transformed: a tile-size and the offset in main memory. The declarations of the arrays change to pointers to one common array. The new size of common array is calculated and check is done if arrays in memory are assigned to mutual exclusive locations to check if provided input is correct. This tool uses libraries that are available within IMEC that support parsing and dumping c-code. The program code is expressed as an 'abstract syntax tree' in which transformations can be applied.

```c
int a[100];
short b[100];
int p;

int main(void)
{
  int k;
  for(k = 0; k < 100; k++)
  {
    a[k] = k;
    b[k] = k % 10;
  }

  for(k = 0; k < 50; k++)
  {
    p += a[k] + a[k + 50] + b[k] + b[2 * k];
  }
}
```

**Figure 4.5** Piece of C-code before address transformations

```c
/* Generated by address transformation tool on Fri June 7 16:13:28 2002 */

char MainMemoryArray[664];
int *a = (int *) ((void *) (&MainMemoryArray[0]));
short *b = (short *) ((void *) (&MainMemoryArray[12]));
int p;

int main()
{
  int k;

  for (k = 0; k < 100; k++)
  {
    a[k % 3 + k / 3 * 5] = k;
    b[(k & 3) + (k >> 2) * 10] = k % 10;
  }

  for (k = 0; k < 50; k++)
  {
    p += a[k % 3 + k / 3 * 5] +
        a[(k + 50) % 3 + (k + 50) / 3 * 5] +
        b[(k & 3) + (k >> 2) * 10] +
        b[(2 * k & 3) + (2 * k >> 2) * 10];
  }
}
```

**Figure 4.6** C-code of Figure 4.5 after address transformations (automated)
Figure 4.5 shows a small example program that is going to be transformed. The input for the tool contains the following information:
- tile-size array a: 12 bytes
- tile-size array b: 8 bytes
- offset array a: 0 bytes
- offset array b: 12 bytes
- cache-size: 20 bytes

Figure 4.6 shows the result after applying the transformation. Note that the transformation took into account that array a consists of integers (4 bytes) and array b consists of shorts (2 bytes). If possible the modulo and division operations are replaced by cheaper mask and shift operations.

4.3.6 CPU specific overhead

The extra addressing that has been introduced includes modulo-operations and integer-division operations. Those operations are costly instructions (in terms of cycle overhead) if no hardware-unit is available on the target-platform. A general C-compiler is not able to optimize those instructions unless the operands are power of two values. Unfortunately tile-sizes are not necessarily a power of two.

```c
for(y=0;y<9;y++)
  for(x=0;x<11;x++)
    for(vy=0;vy<5;vy++)
      for(vx=0;vx<5;vx++)
        for(y2=0;y2<8;y2++)
          for(x2=0;x2<8;x2++)
            ... = TheArray[((y*8 + vy + y2)*(88 + 20) + x*8 + vx + x2) % 96 + ((y*8 + vy + y2)*(88 + 20) + x*8 + vx + x2) / 96 * 1024];
```

Figure 4.7 Piece of C-code before address optimizations

```c
for (y=0;y<9;y++) {
    mnx=-8; dx+=9216;
    for (x=0;x<11;x++) {
        mnx+=8;
        if(mnx >= 96){mnx -= 96; dx+=1024;}
        mvy=m_x-12; dvy=dx-1024;
        for (vy=0;vy<5;vy++) {
            mvy+=12; dvy+=1024;
            if(mvy >= 96){mvy -= 96; dvy+=1024;}
            mvx=m_vy-1; dvx=dvy;
            for (vx=0;vx<5;vx++) {
                mvx++;
                if(mvx >= 96){mvx -= 96; dvx+=1024;}
                my2=mvx-12; dy2=dvx-1024;
                for (y2=0;y2<8;y2++) {
                    my2+=12; dy2+=1024;
                    mx2=mvx-1; dx2=dy2;
                    for (x2=0;x2<8;x2++) {
                        mx2++;
                        if(mx2 >= 96){mx2 -= 96; dx2+=1024;}
                        ... = TheArray[mx2 + dx2];
                    }
                }
            }
        }
    }
}
```

Figure 4.8 C-code of Figure 4.7 after address optimizations

Therefore an address-optimization stage that has been developed at IMEC is applied after the address transformation stage. This stage is referred to as ADOPT. It is a systematic methodology for reducing the piece-wise linear indexing to linear pointer algorithmic [6]. Instead of modulo, division and
multiplication operators, counters and conditions are introduced. This maps the code much better to a common accumulator architecture. Experiments have proved that the overhead of the complex addressing caused by tiling can be removed again. Figure 4.7 and Figure 4.8 show a piece of C-code before and after ADOPT optimizations. A loop-nest is shown with inside the access to an array. The transformed code looks more complex, but can easier be executed. The same loop structure is visible, but counters and conditional are inserted. A big advantage for CPU overhead is the fact that only a small amount of simple instructions is needed in the most critical inner loop.

4.4 Limitations of the Tiling Approach for Conflict miss Removal

This paragraph explains why tiling is not always the optimal way of data-layout for conflict miss removal. However, an optimal layout for conflict misses will cause much more overhead. Taking this into account, we can consider tiling as a good heuristic approach.

4.4.1 What are real associativity misses

The remaining conflict misses that are not removed by the tiling approach are for simplicity reasons classified as 'associativity misses'. However, an example is given that shows which misses are really associativity misses and which misses are due to the heuristic character of the tiling approach.

The extra misses that occur when the associativity of the cache is changed from full associative to e.g. a direct mapped cache (1-way associative) assuming an 'ideal data-layout' we simply call associativity-misses. But, we'll hardly be able to prove for real-life applications that the constructed data-layout is ideal since so many different layouts are possible. Therefore it is difficult to prove that particular misses cannot be solved by any complex data-layout in order to show that those misses are pure associativity misses. In general we'll not be able to find the amount of 'minimal-associativity' misses (see paragraph 6.7).

To find real associativity-misses, a situation should be found where a specific element definitely has to be stored at different locations in the cache for each time the element is actually stored in the cache. A small situation with real associativity misses is shown in figure 4.9. Assume that two elements that are stored in memory are heavily accessed simultaneously. The intention is to store those two elements in a 'mini-direct-mapped cache' of 2 elements. Three different pairs of elements are accessed. First A1 and A2 are often accessed together (situation 1), then A1 and A3 together (situation 2) and finally A2 and A3 together (situation 3). There is no mapping of A1, A2 and A3 to the two available cache locations in such a way that in the situations 1, 2 and 3 the pairs of elements are assigned to different cache-locations (unless we make a copy of an element that can be stored freely). The constraint that one
memory-location can be mapped to one cache-location only is therefore too restrictive. A cache with higher associativity could solve those misses. It is clear that this problem is the result of small 'circular' dependencies between the pairs of elements being accessed (A2 - A1 - A3 - A2).

We can make plausible that e.g. the addressing becomes extremely complex in order to solve particular misses by a data-layout (if elements have to be placed according to a non-regular pattern: no 'function' can be defined anymore to address the elements, but a lookup table is needed in stead). Those misses cannot be solved by the applied 'heuristic' data-layout. Therefore we'll have to accept that the amount of misses measured when limiting the associativity of the cache are not the 'minimal-associativity' misses but simply associativity-misses (accepting that misses are left that could be solved by a better layout).

\[
\begin{align*}
\text{for}(y=0;y<N;y++) \\
\text{for}(x=0;x<M;x++) \\
\text{for}(y=0;y<2;y++) \\
\text{for}(x=0;x<2;x++) \\
\text{for}(i=0;i<2;i++) \\
\text{for}(k=0;k<2;k++) \\
tmp+=\text{mem}[2*x+vx+k] \cdot [y+vy+l];
\end{align*}
\]

Figure 4.10 Artificial example of memory elements being accessed

\[
\begin{align*}
\text{Memory to-cache-mappings / data-layouts (tile-size = 6 elements only, cache-size = CS)} \\
\text{Conflict (situation 1)} \\
\text{Conflict (situation 2)} \\
\text{"Advanced vertical mapping"} \\
\text{conflicts for situations 1, 2, 3, 4, 6, 7.} \\
\text{interleave and pad rightly => not trivial} \\
\text{if(access to column "1,2")} \\
tmp+=\text{mem}(((2*x+vx+k)+(y+vy+l)+(N+pad2)\%4 + ((2*x+vx+k)+(y+vy+l)+(N+pad2)+2*CS2); \\
\text{else} \\
tmp+=\text{mem}(((2*x+vx+k)+(N+pad2)+(y+vy+l))\%4 + ((2*x+vx+k)+(y+vy+l)+2*CS2);}
\end{align*}
\]

Figure 4.11 Different data-layouts with different complexity to solve conflict misses
For instance, in the memory access patterns used in the motion-estimation part of the QSDPCM no clear circular dependencies are visible. It is likely that with a very detailed analysis and complex data-layout (at scalar-level), the miss-rate of a fully associative cache can be obtained with a direct-mapped cache. But by using a small example it will be shown that the data-layout becomes complex. Figure 4.10 shows an artificial example in which elements are read from a 2-dimensional array (comparable to motion-estimation). Within a block of 3x3 elements, a block of 2x2 elements is accessed 4 times at different locations. Afterwards the 3x3 block shifts 2 positions in the x-direction. As soon as the end of the x-dimension is reached, the 3x3 block is shifted 1 position in the y-direction. In a direct-mapped cache, a tile of 6 elements is reserved in order to have as few as possible accesses to the memory itself (cache misses). A circle in the figure indicates the 4x4 block that is currently accessed. The elements that would be stored between the different iterations by a fully-associative cache with optimal replacement policy, are indicated by a gray field. A direct mapped cache could do as good as the optimal cache if the gray memory locations in a specific situation are never assigned to equal cache-locations.

Three different data-layout possibilities are shown in Figure 4.11. The memory locations are drawn again and the number inside indicates to which cache address (1..6) the memory locations are mapped. Also a piece of C-code is given that shows how the addressing looks to make the desired mapping. The two first mapping-situations are the result of assigning a tile to the memory-array with size 6. Only the storage order is different. Six horizontal or vertical memory locations map to six consecutive cache addresses. Those layouts do not solve all conflict misses. For both layouts one situation is indicated where two elements that should be stored simultaneously in the cache are mapped to the same cache-location (resulting in a conflict miss). The maximal amount of gray fields per situation is 6, so in principle there is no need for more cache-locations. The third memory-to-cache mapping does indeed show a situation without any conflicts. This directly proves that there are no real associativity misses in this case. But it also shows that the addressing is not trivial anymore.

Fortunately there is still a regularity visible: The odd columns in the memory are mapped to cache-locations 1 and 2. The even columns are mapped to the cache locations 3 to 6 only. In order to make the addressing feasible, the memory-signal should be split into two signals. Each array gets its own tile-size. Moreover, careful padding is needed to adjust the relative placement of the elements in the odd and even columns. With this example it is shown that extra conflict-situations can be avoided by a very detailed data-layout, but at the expense of complex addressing. Moreover, to find out which layout should be made is no trivial task. For this detailed-layout, an analysis at the scalar-level has been made. If complete arrays are considered atomic-objects, only the first two layouts can be made. Therefore we can distinguish different limits for data-layout optimization. As long as complete arrays are the ‘basic groups’ (the structure of arrays is not touched) we’ll be able to reach an ‘array-level’ limit. If the elements within the array are considered separately it is very likely that a better data-layout can be made until the ‘scalar-level’ limit is reached. In between there is a range of possibilities (e.g. splitting arrays in smaller basic groups). If groups of 2 adjacent elements are considered as the basic-groups, there would still be the possibility to construct the 3rd data-layout. If the basic-groups are groups of 3 adjacent elements, this becomes impossible because there is no possibility anymore to spread the elements over the columns in groups of 4 and 2 elements.

4.4.2 One decision per array

A limitation of the tiling approach that is inherent to data-layout transformations is that for each array only one decision can be taken about the tile size and its offset in memory. This decision holds for all references to this same array in the code. If different references ask for complete different tile-sizes (they are accessed in different loop-nests), still only one size can be applied. So a trade-off should be made. In order to avoid as much conflicts as possible, the layout should be adapted to the reference with the most potential misses.

In case there is a need for selecting different tile-sizes for one array, this can be done by making explicit copies in the source-code of parts of the array or by duplicating the array. However, the extra overhead in memory and in keeping both arrays coherent should be considered as well. Since this is outside the scope of data-layout transformations, this is not further considered here.
4.4.3 Exotic shapes of working sets

Tiling is used to keep elements belonging to the same working set together in the cache. However, if elements belonging to one working set are spread in a very wild pattern through the memory, tiling cannot always map the elements to the desired cache locations. Tiling can deal for example with working sets that are spread through the memory in pieces at constant distances (as we'll find back in e.g. the QSDPCM driver). If the working set is spread randomly in a restricted compact region in memory, optimization can always use the smallest enveloping rectangle. Since also elements not belonging to the working set are considered, this will result in wasted cache-lines, but this can still be considered. If tiling does not really manage in mapping the elements of an array to the cache, it still has the advantage that it avoids the interference of one array with the others arrays. So if no improvement for conflict misses for one array is achieved, it will not cause degradation of the miss rates for other arrays either. Which is the case if no tiling is applied.
CHAPTER 5. OPTIMIZATIONS FOR APPLICATIONS WITHOUT DATA REUSE

5. Optimizations for applications without data reuse

This section describes conflict miss optimizations for drivers that are not data-reuse dominated. 'Not dominated by reuse' does not mean that data-elements are only used once. It means that no smaller parts of arrays are reused locally in time, or: elements from an array are accessed again only before all other elements have been accessed too.

We use the tiling approach to eliminate conflict misses. Decisions have to be made about the sizes of the different tiles. We have two different approaches to decide on those sizes. The first approach, described in paragraph 5.1, uses sizes determined by Memory Hierarchy Layer Assignment to construct a data layout. The second approach, described in paragraph 5.2 decides on tile-sizes by a weighted cost, estimated by a cache miss model.

The MHLA approach has the problem that the recommended tile-sizes cannot always be translated in a data-layout because for one array multiple tile-sizes (possible for each array reference a different size) are prescribed. The second approach can deal with trading-off different sizes, but has the problem that the model cannot deal with reuse in arrays. This is a quite severe limitation because most applications have a kind of local reuse (e.g. two references in one loop-nest that both access the same data-elements).

5.1 MHLA directed approach

For applications without data-reuse, MHLA can assign arrays with a specific size to memory layers to minimize for capacity misses. MHLA will have to decide for complete arrays as a whole if they are assigned to the cache or not, because the effect of storing only parts of arrays if there is no local reuse within the arrays is not taken into account. This decision on the assignment of arrays to layers based on minimal capacity-miss contributions. By choosing the right combinations of arrays, this miss-rate will be minimized.

If a complete array is assigned to an own place in the cache (as is done by the tiling approach), there will be no conflict misses: all inter array conflict misses are eliminated and also all intra array conflict misses are eliminated because each data element has an unique cache location. But the situation is not always so simple.

If for different parts of the application one array has different sizes, this is a problem in the data-layout, since only one size can be selected. It must be decided which size to take (the average, the extreme?). A possible way to tackle this problem is to duplicate the array. For each duplication of the array an other tile-size can be chosen. However this causes overhead in copying the arrays.

MHLA will decide not to store particular arrays in the cache. They will have to be bypassed. If platforms do not support bypass, those arrays will have to be assigned to the cache too. Then a small tile will be reserved for those arrays and it is assumed that all accesses to those arrays will cause misses (for each access the main memory is accessed). But those tiles can not be too small, because this causes much memory overhead.

So it is clear that the proposed array sized by MHLA cannot directly be translated in an optimal cache layout, but it could be used to provide an initial tile size that would have to be refined at the data layout level. These possibilities for the integration with MHLA in the case of non-data-reuse dominated applications will have to be worked out in more detail (the case with reuse is described in chapter 6).

5.2 Tile based data-layout optimization of cache misses

This approach aims at dividing the cache space into different parts, based on the miss-rate prediction for each array. This approach does not take into account initial estimations of array sizes to be stored in the cache. The tile-size decisions are steered by a miss model. The reason for describing the optimizations for applications without data-reuse separated from data-reuse dominated applications is
related to this miss estimation model. This model only works fine in situations where no local reuse is present within the accessed arrays.

5.2.1 Existing model

The PHD-work done at IMEC by Chidamber Kulkarni is the starting point of this research. In his thesis “Cache optimization for multimedia applications” [8], an extensive description is given of many different cache optimization steps. One of the parts describes the conflict-miss optimization in a direct mapped cache by arranging data-layout in main-memory. A methodology is given to optimize this layout. The idea of tiling is introduced and a cache miss model is presented. We'll briefly describe this model and indicate why we had to modify the model, for details reference [8] should be consulted.

The idea behind the model is to estimate the amount of misses for a given placing of arrays in the memory (different tile-sizes, different offsets). Per loop-nest an estimation is made about the size of the working set of an array and the amount of accesses to the array. Misses are counted if two arrays that are accessed in the same loopnest are (partly) overlapping in the cache (depending on the amount of overlap and the amount of accesses, see Figure 5.1a). Misses are also counted for situations where the tile-size is smaller then the working set of an array (the array is overlapping with itself, see Figure 5.1b). The total miss rate is the sum of the misses for the individual arrays over all loop-nests of the complete application.

![Figure 5.1 Miss-estimation based on overlapping cache areas](image)

The tile-sizes are determined by one decision based on the 'importance' of the array (size and amount of accesses), the offset is determined by trying all possible offsets and evaluating the miss-cost. The scope of this model is not very clear. Moreover, the outcome (an 'optimal tile-size and placing for different arrays in the cache') does not directly correspond to a feasible tiling data-layout. For instance if two arrays are assigned to the cache but they (partly) overlap in the cache, this cannot be realized by a single interleaving pattern in memory, because the arrays cannot be stored at overlapping locations in the memory.

Another problem is that this model cannot deal with local reuse within the arrays because it has no knowledge about the actual placing of the elements in the cache. It assumes the different elements being used from an array to map in the cache consecutively.

5.2.2 Development of a new cache miss model

We needed to develop a new model that, first of all can help to generate a data-layout that can be implemented by the tiling concept. This model is derived based on observations we have done when simulating different arrays. Still it is a very global model that does not take into account the detailed addressing of array elements. It also makes strong assumptions about the access patterns of the arrays (no local reuse). But taking into account those restrictions the model can predict the miss rate quite well and steer the decisions for the tiling approach. This tiling approach then optimizes for capacity misses and in the meantime eliminates inter array conflict misses and minimizes intra array conflict misses.

The model does not distinguish between capacity and conflict misses though. It simply aims at minimizing the total miss-rate of an application. Combining the optimizations for both miss types in a direct mapped cache is not very strange, since the decisions for both optimizations are expressed in the
same data-layout. An extra cache parameter that is taken into account in this model is the influence of a
higher associativity of the cache.

We have tested the accuracy of the model using different arrays from different drivers (Vocoder,
Wavelet, Cavity). We also have combined the different arrays from the Wavelet in one layout using the
predictions made by the model to see how we could reduce the miss rate for the complete driver. The
results of those experiments are presented in this chapter.

**Incorporating the effects of higher associativity in the tiling approach**

The tiling approach aims at assigning unique parts of the cache to each array so all inter array conflicts
are eliminated. We can achieve this goal for a direct mapped cache, but to achieve this goal for a 2 way
associative cache, the tiling approach should be slightly adapted.

![Figure 5.2 Relation memory-tile and cache-tile](image)

When assigning a tile in cache to an array, it is assumed that the tile-size equals the amount of cache-
locations that can be occupied by this array. For a direct mapped cache (associativity = 1) this tile is
one continuous range of elements in the cache. When a 2-way set-associative cache is used, the aim is
still to assign unique parts of the cache to one array. This can only be done if complete cache sets are
assigned to one array. A 2-way set-associative cache has an amount of sets that is half the amount of
cache-locations. Within each set there are 2 banks (locations to be selected by the cache-controller).
The mapping function from memory addresses to cache addresses of a set associative cache is:

\[
\text{address of cache set} = \text{memory address} \mod \#\text{sets}
\] (5.1)

The modulo value is not ‘cache size’ like for a direct mapped cache but ‘\# sets’ (= cache-size divided by
the associativity). In order to fill each set of a 2-way associative cache with one array only, a two
times finer interleaving-pattern in memory is needed compared to the direct mapped cache. In order to
indicate ‘the amount of locations reserved for one array in the cache’ a the term ‘cache-tile’ is
introduced. A ‘cache-tile’ is in case of a 2-way associative cache the sum of the tiles in both banks. For
the continuous ranges of elements in memory the term ‘memory-tile’ is introduced. The sizes of
memory- and cache-tiles are related: cache-tile = \(a\) * memory-tile, with ‘\(a\)’ is the associativity of the
cache (see Figure 5.2).

**Input information for the model**

The aim of this model is to predict the amount of misses that are the result of self-interference of an
array (elements of the same array do overwrite each other in the cache). The model can deal with
applications in which arrays are accessed within different loop-nests. For an array in a loop-nest a
global estimation is made for the amount of misses. If an array is accessed in more than one loop-nest,
the estimations per loop-nest are combined to obtain the complete model for this array. The information
used by the model is the following:

- declared size of the array: the total amount of array elements
**CHAPTER 5. OPTIMIZATIONS FOR APPLICATIONS WITHOUT DATA REUSE**

- Effective size of the array in each loop-nest: amount of different elements of an array used in the loop-nest (= size of the 'working set')
- Access-count: amount of times the array is accessed in each loop-nest
- Amount of array-instances: a static read or write in the loop-nest

This model does not take into account detailed information about the actual addresses pointed to by the index-expressions.

**Deriving the model for one array reference**

Based on the observations we have done when simulating arrays and varying the tile-size, we derived a relation between the tile-size of one array and its miss-rate.

Since the model has no knowledge about the array elements that are actually accessed within a loop-nest, the model is assuming a particular access pattern. It assumes that for an array A with an effective-size of M elements the access-pattern is so that all M elements of the array are accessed before an element is accessed for the second time (no reuse of smaller blocks of elements within the M elements).

Assuming that there are N locations available in the cache for M elements of array A (cache-tile = N), it is evaluated how this access-pattern causes misses or hits. If the cache-tile equals the effective-size, each data-element has its own location in the cache and no conflict misses will occur. Now the case is

**Figure 5.3 One location missing for direct mapped cache**

**Figure 5.4 One location missing in set: domino effect**
studied where \( M = N + 1 \) (there is one location missing in the cache to store the \( M \) elements of the array: surplus = 1).

For a direct mapped cache the mapping-pattern in the cache is as indicated in Figure 5.3: the \( N^{th} \) element is always mapped on cache-location 0. Since location 0 is alternately occupied by two different elements, each access to this location has become a miss. For a fully associative cache with a least recently used replacement policy, the mapping-pattern in the cache is as indicated in Figure 5.4: the first time the \( N^{th} \) element is stored at location 0, but then element 0 is stored at location 1 (replacing the least recently used element) etc. This causes a destructive effect: each element is flushed-out just before it is needed again (referred to as ‘domino-effect’). All accesses to the cache become misses. For a 2-way associative cache (also with least recently used replacement policy) the effect is less extreme but still visible.

![Image](image-url)  
**Figure 5.5** Saturation point for direct mapped cache and 2-way associative cache (considered for one loop-nest)

The general conclusion from this is: ‘all accesses within one set become misses if there is a surplus of one element in a set’. The bigger the sets, the more extreme the domino effect will be.

In order to model this effect, the tile-size \( N \) is going to be decreased until the point is reached that the cache-behaviour is completely destroyed (all the accesses to the array are misses). The tile-size for which this happens for the first time is called ‘saturation-point’. The saturation point can be derived from the effective-size and the associativity. Saturation is reached if all the accesses to each set become misses. The accesses to one set become misses if there is a surplus of 1 element in this set. In order to affect all sets, a surplus is needed that equals the amount of sets. For a direct-mapped cache and a 2-way associative cache this is indicated in Figure 5.5. This results in the following relation to obtain the cache-tile size that is the saturation-point (with \( a \) is the associativity of the cache):

\[
cache\_tile\_saturation = \frac{a}{a + 1} \times eff\_size
\]  
(5.2)

The amount of misses at this saturation-point equals the amount of accesses to the array. Also for tile sizes smaller than the saturation point, all accesses to the array are misses. The miss rate linearly decreases for bigger tile-sizes to the point where the complete effective size fits in the cache tile. On this point only the compulsory misses are left (since each element has to be brought to the cache at
least once). This results in the miss-model shown in Figure 5.6 that indicates the miss rate for one array that is mapped to a cache-tile with a particular size.

Let us assume for a moment that the replacement is optimal and free to decide if elements are replaced or not. Then the extra elements that cannot be stored in the cache ('the surplus') will not destroy the cache-usage for the remaining elements. They will be 'bypassed' (e.g. by mapping them all to one single cache location). In this case the benefit of the cache is only reduced to zero for tile-sizes equal to zero. For bigger sizes the miss-rate decreases linearly (see Figure 5.7). The mapping needed to obtain this behaviour is not possible by the tiling approach because all elements in the 'surplus-set' must be mapped to one single cache line. Therefore the extra misses due to the tiling approach cannot be removed. They are related to restricted replacement possibilities when using the tiling approach.

![Figure 5.7](image)

**Figure 5.7** Extra misses due to restricted replacement possibilities when using the tiling approach

**Extension to more references and different loop-nests**

Arrays mostly have more than one reference and are present in different loop-nests. An attempt is made to incorporate this in the model too.

Different references in different loop-nests can have different effective sizes. For each loop-nest a separate model can be constructed. The models are combined to one graph that relates the selected tile-size to the expected miss rate (remember that for each array only one tile size can be chosen). Combining is done by adding the different models (see Figure 5.8). This is again based on assumptions. The sum of miss rates is only valid if there is no reuse of elements between different references. For example in the case that two references exactly access the same data-elements, the second reference will never have any miss, since the required data element is just accessed by the first reference. This is not possible if reuse is excluded, therefore all misses for all references may be added to the total miss model.

![Figure 5.8](image)

**Figure 5.8** Combining miss models from different loop-nests
5.2.3 Experiments to test the miss model for individual arrays

The accuracy of the model in different situations is shown using arrays from the Cavity and Vocoder drivers.

We extracted the code from our drivers that contained all references to one single array to test the accuracy of the model. The miss rates for different tile-sizes are combined in one graph. This graph is compared with the prediction of the model.

Figure 5.9 shows results for one array from the cavity detection algorithm and one array from the Vocoder algorithm. The estimation follows the simulation, but there is still some variation. This is because there is local reuse between references, this quickly happens when an array has more than one reference in a loop-nest. We see that for bigger tile-sizes the miss rates saturate. Since the complete array fits in the cache and no other misses than compulsory misses occur. The problem is that since the model has no knowledge at all about the reuse behaviour, it is also not possible to predict for these applications how big the deviations between the simulations and the model will be.
5.2.4 Optimizing a complete driver

The results provided by the model can be used to trade-off costs from different arrays in order to divide the available cache space optimally between different arrays. Next to the estimated miss-rate, also the memory-overhead cost per array should be incorporated. Based on those costs per array, a set of tile sizes can be determined that fills the cache with the lowest costs.

```
For (all arrays j) tile_size[j] = effective_size[j];

While (∑ tiles > cache) {
    min_cost = ∞;
    shrink_candidate = unknown;
    For (all arrays k) {
        cost = compute_shrink_cost(k);
        If (cost < min_cost) {
            min_cost = cost;
            shrink_candidate = k;
        }
    }
    tile_size[ shrink_candidate ] = step;
}
```

**Figure 5.10 Iterative approach to fill the cache with arrays**

A heuristic way to divide the cache-space optimally between the different arrays in an application is the following iterative approach:

1. Initialise the tile sizes for each array on the effective size.
2. Determine for each array the cost to decrease its tile-size with one step (extra miss-rate and memory overhead)
3. Reduce the tile-size of the array with the lowest cost
4. Repeat steps 2 and 3 until the sum of tile-sizes fits in the cache. (see Figure 5.10).

Since not all arrays are alive through the complete algorithm, we should not divide the cache-space among all the arrays of an application. This would result in situations where only parts of the cache are used at the same time. For good optimizations we have to cluster the arrays in groups. The arrays in each group are going to be placed in a separate interleaving data-layout. Therefore all inter array conflicts between the arrays in one group are eliminated. In order to minimize the amount of inter array conflicts between arrays in different groups, the grouping decision is based on the lifetimes of the arrays. Arrays with overlapping lifetimes are preferably grouped together.

The decision about the amount of groups that are required and which array should be classified in which group is currently made manually, based on the amount of accesses to each array in each loop-nest and on the combinations of arrays being accessed in each loop-nest. Within each group, arrays are assigned to a part of the cache using the iterative approach described above.

**Experiments for optimizing the Wavelet driver**

We have experimented with the data layout possibilities using the Wavelet kernel. First we have applied inter array inplace transformations to reduce the storage size of the different arrays. Then, we
have analyzed the amount of accesses to each array in each loop-nest and the size of the working set of each array in each loop-nest. This information is combined to construct a data-layout for specific cache sizes in order to optimize the miss-rate for direct mapped caches.

For each cache size we have simulated the miss-rate before and after our data-layout transformations. The results are shown in Figure 5.11. We see that for small cache sizes we are able to reduce the amount of misses with at least a factor 3 by applying a suited data-layout. For bigger cache-sizes there is no improvement visible, here the arrays are apparently placed optimally in the non-optimized case (the linker simply allocates consecutive locations in memory for each array, this can be good if all data starts fitting in the cache).

We also have a look at the memory overhead. Figure 5.12 shows the amount of memory usage for different situations: before intra array inplace, after intra array inplace and after data-layout transformations for cache miss reduction on the intra-inplaced code. The first observation is that the extra memory needed for our data-layout transformations is very small in comparison to the amount of memory we saved by the intra-inplace data-layout transformations. The memory sizes needed after applying the data-layout transformations for cache miss reduction show a slight variation, but none of them show a significant increase. So by the combination of inplace transformations and the tiling approach for reducing cache-misses, we could both reduce the memory usage and the miss rate (at least for the small cache-sizes below 256 bytes).

5.2.5 Limitations of the approach and conclusions

The experiments done with different application kernels have showed that in some situations the miss-model is quite exact and conflict misses can be eliminated. However, the proposed cost-model uses very global information and makes strong assumptions about the application to optimize. The predictions of the model match reality best if arrays are accessed in a very regular way (if all elements are read consecutively and if there is no reuse of local groups of elements within the array). Because of the assumed regular behaviour, the mapping to the cache is ‘idealized’ and intra-array misses can be
predicted. These misses could be more related to capacity misses than to conflict misses (for instance, for relative small tiles a high miss-rate is predicted because of ‘lack of space in the tile’).

As soon as particular elements within an array are reused, the model will very likely over-estimate the amount of misses, since a smaller tile of the cache is able to hold the elements being reused and accesses counted as misses by the model can in reality be hits. However, it is not guaranteed that the elements being reused can actually stay in the cache simultaneously. If the elements have mutual conflicts in the cache, the accesses will be misses anyway.

The model under-estimates the advantages of caches with higher associativity. Although the described ‘domino’ effect has a negative influence for the considered access pattern, less regular access patterns will profit from the flexible placing possibilities of higher associativities.

The idea was to refine the existing model and try to increase the accuracy. But only if much more detailed knowledge of the access pattern is used, the predictions can come closer to reality because so many different parameters do influence the cache behaviour. We should introduce a full knowledge of the addressing expressions and loop structures, however the way this model is built is not suited for such extensions. The assumptions are the basics of the model. So this way of miss-prediction doesn’t seem an extendable one with much future possibilities.

The job of dividing the cache space in order to minimize capacity misses (and therefore energy) can also be done by MHLA. It selects sets of elements that should be stored simultaneously in the cache. Currently, MHLA only considers complete arrays and data-reuse copies (if they are present). But if there is no data-reuse in the array, it can still be interesting to store a smaller part of the array in a tile. To make an estimation about the amount of misses that will occur if only a part of an array is stored in the cache, the presented model could be used. In those cases we know that no reuse is present anyway and the misses expressed in this model are related to capacity misses, the same misses that are the target of the MHLA optimizations. So an integration of this model with the MHLA based approach is a possible extension.
6. Optimizations for applications with data reuse

This section describes our approach for conflict miss optimizations for applications that are data reuse dominated. The conflict miss optimizations are expressed in the data-layout by using the tiling approach but this has to co-operate with the optimizations for capacity misses (closely related to data-reuse) since they are also expressed in the data-layout. The reuse-related explorations and optimizations will be outlined first, then the data-layout optimizations will be explained. At the end of this chapter optimization examples are worked out.

6.1 Definition of data-reuse

To explain the concept of data reuse, we consider a reference to an array in a loop. We look to the relative time-order of the accesses to this array. Different data elements will be accessed. Over a long time frame all data values will be read from memory. However when we look at smaller time frames we will see that only a part of the data is actually needed in each time frame (if this is an array with reuse). Therefore a part of the array is accessed more than once locally in time.

Assume that the complete array is stored in the background memory. In this case, each access to the array is an access to main memory. This is not very power efficient, since in general, accesses to bigger memories are more power consuming. Therefore, we can decide to make copies of smaller parts of the array and store these copies in smaller, less power consuming memories between the main memory and the CPU. Depending on the size of a copy we can avoid accesses to the main memory since we’ll be able to explore more or less the data-reuse behaviour. In this way the bandwidth requirements of the large background memory are reduced. The more elements we can store in the copy, the more we’ll be able to benefit from reuse (if it is present). This principle is not restricted to one copy only. More intermediate copies can be made. Each introduced copy that explores reuse can avoid accesses to higher, more expensive memory-layers.

If we study the relation between copy-size and the explored reuse (expressed in a data-reuse factor: total # reads to the array / # reads to the higher layer), we see that this is not a smooth curve (see Figure 6.1). It shows clear bending points. This is because the reference to the array is within different levels of loops and each level is related to a specific amount of reuse. For each loop-level a copy can be proposed with an optimal size. The possible reuse-copies for one array reference is called a ‘copy candidate chain’. For each array reference a chain can be constructed. Depending on the reuse between different references, chains can be combined. This results in a ‘reuse tree’ with copy candidates (see Figure 6.2).

![Figure 6.1 Data reuse factor](image1.png)

![Figure 6.2 Copy candidate tree for one array](image2.png)
When deciding to implement a specific copy selected from the tree, the amount of explored data reuse is known. Depending on the memory size where this copy is stored in, the selection is also related to a specific amount of power since the amount of accesses to the copy-signal and the background memory are known. Also a combination of different intermediate copies can be selected and stored in different intermediate memories between the main memory and the CPU. For each combination also the amount of accesses to each memory can be calculated. More details about this reuse exploration methodology can be found in [1].

6.2 Relation to memory hierarchy design

The data-reuse information is used to decide how to use the available memory hierarchy in an optimal way in terms of power, area and timing. Therefore an exploration is necessary to decide where which arrays or copy candidates are stored in the different memory hierarchy layers.

Assume that from an application the complete reuse-trees for all arrays are known and that a platform is available with a specific amount of memory hierarchy layers with a specific size each. Then an exploration step called Memory Hierarchy Layer Assignment (MHLA) is used to decide which copy candidates from each array are stored in which layer. This exploration uses memory libraries that relate the size of the layers with the power per access to the layers. Also the latency to the memories is taken into account to meet certain timing constraints. For each different assignment the total power can be computed, since for each selected copy candidate the amount of accesses to higher (more background) layers is known. The assignment to layers is also constrained. First of all: the sum of the assigned copy candidates may not exceed the layer size, otherwise there will be no possibility to explore the amount of reuse related to the selected copy candidates. Secondly the array itself (referred to as top level array) must be stored somewhere in the memory hierarchy whereas the storage of the copy candidates is optional.

If we have the possibility to define the complete memory hierarchy in a custom way, MHLA is able to decide the optimal size per layer (given the amount of layers). The decision on the amount of intermediate layers between the CPU and the background memory can be made based on the reuse-trees. Per array there is an optimal mapping of the copies to a specific amount of layers. Based on the outcome for all arrays, one decision is made for the amount of layers. Assuming that the amount of layers is fixed, we can decide about the optimal size per layer.

Typically, a size of a layer is a power of two value, so the amount of possibilities are restricted. For each size, a selection of copy candidates is made (from each tree at most one candidate is selected) and assigned to the layer. The selection is so chosen that the layer size is not exceeded and that the amount of accesses to the higher layer is minimized. The amount of accesses to each layer related to the size of each layer results in the total power of the selected layer size. An optimal layer size is selected based on this power estimation. There is an optimal layer size since small layers are power consuming because many accesses to the next layer are needed and big layers are power consuming because the energy per access to the layer increases.

6.3 Problem definition

Memory hierarchy layers can contain normal (software controlled) memories or caches. In the case of memories, the copies selected by MHLA should be explicitly present in the code. However, in the case of caches, the cache controller will make the copies of arrays at the moment they are accessed (and the copy is not present in the cache yet). The controller also decides about the mapping of the copies. It is ensured that there is place for the copy in the cache, but we must avoid that elements of one copy do overwrite elements from other copies or from itself. When using a fully associative cache with optimal replacement policy, we should measure exactly the misses that are predicted by MHLA in the ideal case. However if we use a direct mapped cache, elements from the ‘current working sets’ of the arrays cannot be stored in the cache simultaneously, since the restricted mapping organization is causing conflict misses. Here, the data-layout organization must be optimized to ensure that copies of a specific size can co-exist in the cache.
6.4 Reduction of data-layout conflict misses by tile-size refinement and padding technique

The size of the copies suggested by MHLA can be enforced in the cache by the tile based data-layout. The size of each tile should be the size of a copy. This directly removes the inter array conflict misses. However, we should also consider the intra-array conflict misses, since elements from the same array may not conflict either. Therefore the tiling approach is followed by a fine-tune phase in which the tile-sizes are adjusted and an additional data-layout optimization of array padding is integrated to optimize for intra array conflict misses. The introduction of padding adds an extra dimension of freedom through which better trade-offs are possible.

In the fine-tune phase we'll have a detailed look at the addressing of an array and try to understand the access pattern (taking into account all references to the array, the storage order, conditionals etc.). Based on this knowledge we will try to predict (without simulations) how the mapping of the elements belonging to the selected copy (the working set) is going to be in the cache. We'll see that for a particular type of access patterns we can predict and eliminate the conflict misses reasonably well.

A working set is changing continuously in time. The next set can be a partly update of the current set. For all the different sets in time, the different elements should be mapped to the cache without conflicts. There are situations in which there is no problem to ensure that the mapping to the cache causes no single conflict miss. This is for example the case when all the elements of the working sets are always stored at consecutive locations in main memory. Typical cases that do potentially cause conflicts misses are working sets that are spread throughout the array. In those situations there can be more array elements from one working set that map to the same location in the cache.

Mapping of different elements to the same locations should be avoided. So elements from working sets should be distributed over the available cache locations within a tile. The mapping will be influenced by adjusting tile-sizes and changing dimension-sizes of arrays (padding). Assume that the concept of tiling is applied and for a particular array the working set (smaller than the array itself) is spread through the array. For this array data layout conflict misses are expected. If the data-layout-conflict-miss-rate is studied for different tile-sizes, a random behaviour is observed: very high and very low miss rates for only small changes in the tile-size. The locations of those 'peaks' have to be predicted however. Only this way a tile-size can be chosen with a low miss-rate. It is also possible for a given tile-size to change the conflict-miss-rate by changing the location of the peaks. This can be realised by padding.

6.4.1 Mathematical properties for predicting conflicts

In order to predict a good mapping of elements to the cache, a mathematical property is used. This property can ensure that cache-lines are equally used when a set of elements is mapped from the memory to a direct mapped cache using the modulo operator. A theorem is used that use co-prime numbers to predict the mapping to the cache. First some background is provided to prove the required theorems.

Let \( n \) be a positive integer with prime factorisation

\[
n = p_1^{a_1} p_2^{a_2} \ldots p_r^{a_r}, \tag{6.1}
\]

where \( p_i \) are distinct primes and \( a_i \) are positive integers. Let \( d \) be a divisor of \( n \). Then \( n = dm \), for some \( m \). Since the prime factorisation of \( n \) is unique, \( d \) cannot have in its prime factorisation a prime which is not among the primes \( p_1, p_2, \ldots, p_r \). Also a prime \( p_i \) in the prime factorisation of \( d \) cannot have an exponent greater than \( a_i \).

Let \( a \) and \( b \) be two positive integers. If \( d \) is a divisor of \( a \) and also a divisor of \( b \), then we say that \( d \) is a common divisor of \( a \) and \( b \). As there are only a finite number of common divisors, there is a greatest common divisor, denoted by \( \gcd(a, b) \). The number \( m \) is said to be a common multiple of \( a \) and \( b \) if \( m \)
is a multiple of \(a\) and also a multiple of \(b\). Among all common multiples there is a minimal one. It is called the least common multiple and it is denoted by \(\text{lcm}(a, b)\).

**Theorem 1.**

Let \(a \in \mathbb{Z}\) and \(b \in \mathbb{Z}\), then

\[
\text{gcd}(a, b) \cdot \text{lcm}(a, b) = a \cdot b
\]

**Proof.** In the decomposition (1) we had all exponents positive. However, sometimes it is convenient to allow some exponents to be 0. This is especially convenient, when we consider prime factorisations of two numbers \(a\) and \(b\), looking for \(\text{gcd}(a, b)\) and \(\text{lcm}(a, b)\), since we may assume that both \(a\) and \(b\) have the same set of primes in their prime factorisations. Let

\[
a = p_1^{a_1} p_2^{a_2} ... p_r^{a_r}, \quad b = p_1^{\beta_1} p_2^{\beta_2} ... p_r^{\beta_r}
\]

where \(a_i \geq 0\) and \(\beta_i \geq 0\), be two arbitrary positive integers. Then

\[
\text{gcd}(a, b) = p_1^{\min(a_1, \beta_1)} p_2^{\min(a_2, \beta_2)} ... p_r^{\min(a_r, \beta_r)}
\]

and

\[
\text{lcm}(a, b) = p_1^{\max(a_1, \beta_1)} p_2^{\max(a_2, \beta_2)} ... p_r^{\max(a_r, \beta_r)}
\]

moreover

\[
\text{gcd}(a, b) \cdot \text{lcm}(a, b) = p_1^{\max(a_1, \beta_1) + \min(a_1, \beta_1)} p_2^{\max(a_2, \beta_2) + \min(a_2, \beta_2)} ... p_r^{\max(a_r, \beta_r) + \min(a_r, \beta_r)} = a \cdot b
\]

To prove theorem 1 and (6.5) we have to notice that \(\max(a_1, \beta_1) + \min(a_1, \beta_1) = a_1 + \beta_1\).

**Theorem 2.**

Let \(\mathbb{Z}/b\mathbb{Z}\) be the group of equivalence classes modulo \(b\), if \(a, b \in \mathbb{N}\) and \(\text{gcd}(a, b) = 1\) (the integers \(a\) and \(b\) are co-prime), then \(a\) is an additive generator of \(\mathbb{Z}/b\mathbb{Z}\), hence \(0, a, 2a, ..., (b-1)a\) covers \(\mathbb{Z}/b\mathbb{Z}\) completely.

Theorem 2 assumes that \(\text{gcd}(a, b) = 1\). Then it follows from theorem 1 that \(\text{lcm}(a, b) = a \cdot b\). To prove that \(a\) is an additive generator of \(\mathbb{Z}/b\mathbb{Z}\), consecutive multiples of \(a\) starting at 0 are considered. The first multiple of \(a\) that is also a multiple of \(b\) is \(a \cdot b\) (since \(\text{lcm}(a, b) = a \cdot b\)).

If we can prove that all multiples of \(a\) in-between 0 and \(a \cdot b\) are members of distinct equivalence classes modulo \(b\), then \(a\) is clearly an additive generator of \(\mathbb{Z}/b\mathbb{Z}\).

Assume \(k \cdot a\) and \(l \cdot a\) are two different multiples of \(a\) between 0 and \(a \cdot b\). So \(k \cdot b < l \cdot b\). Those multiples \(k \cdot a\) and \(l \cdot a\) would be members of the same equivalence classes modulo \(b\) if their difference is a multiple of \(b\):

\[
|k \cdot a - l \cdot a| = |k - l| \cdot n \cdot b = 0 \pmod{b}, \text{ with } k, l, n \in \mathbb{Z}
\]

Since \(k \cdot b < l \cdot b\) it follows that \(|k - l| < b\). This would mean that \(|k - l| \cdot \cdot a\) is a multiple of \(a\) and \(b\) that is smaller than \(\text{lcm}(a, b)\). This leads to a contradiction. Therefore all multiples cover \(\mathbb{Z}/b\mathbb{Z}\) completely and \(a\) is an additive generator.

**6.4.2 Optimizations for specific access patterns**

The mathematical property of the additive generator is used to predict the mapping of elements from a working set to the cache. For a specific type of access patterns the property can be used. Those types of patterns will be described.

The property of an additive generator is translated to the ‘cache domain’. The controller of a direct mapped cache calculates the addresses of elements in the cache like:
When using the tiling approach, elements from one array are mapped to a limited part of the cache with 'tile size' elements. Chunks of the array are spread throughout the memory so that each array maps to its own tile as if it maps to a cache with 'tile size' elements. The modulo operator of the cache controller now translates 'element addresses within the array' to 'locations within the tile of the cache':

\[
\text{address within cache-tile} = \text{array element address} \mod \text{tile size}
\]  
(6.7)

The different addresses within the cache-tile can be interpreted as a set of integers representing the different equivalence classes modulo tile size. A selected set of array elements should be stored in the cache simultaneously, i.e., their addresses should all be member of distinct equivalence classes (of course this can only be the case as long as the size of this set does not exceed the tile size). If different elements from one set are members of the same equivalence classes ('map to the same cache address'), this very likely results in conflict misses. For different distributions of the elements within the set, the coverage of the cache-tile addresses is discussed. These are rather regular distributions, since for very wild distributions it won't be possible to predict an optimal memory to cache mapping.

1. The size of the array equals the size of the reserved tile in cache. Hence, the array element address will never exceed the tile size. Then the mapping as described by equation 6.7 ensures that never two elements can use the same cache line (see Figure 6.3.1). Using theorem 2: the amount of equivalence classes b equals tile size; the additive generator a = 1 (the distance between repeated elements within the array). Since a = 1, the gcd(a, b) = 1. Hence the tile is completely covered.

2. The size of the array is bigger than the tile size and the selected working set is a continuing range of addresses (not bigger than the tile size). Since the modulo-mapping does not change the continuity, also the occupied addresses in the cache will be a continuous range (if the addresses 0 and 'tile size' - 1 are considered to be adjacent addresses). Each cache address is occupied by exactly one memory element and no conflicts are expected (see Figure 6.3.2). An explanation using theorem 2 is equal to the distribution of situation 1.

3. The size of the array is bigger than the tile size and the selected working set is spread through the array in such a way that the distance between single elements of the set is constant (see Figure 6.3.3). Assume this distance to be a. The relative array element addresses are: 0, a, 2a, ..., (tile size - 1)a. Theorem 2 says that a good mapping is ensured as long as gcd(a, tile size) = 1.

![Figure 6.3 Different mapping situations](image-url)
4. The size of the array is bigger than the tile size and the selected working set is spread through the array. Elements are grouped in groups of constant length \( L \) and those groups are repeated at distances \( M \). This distribution is often found in image processing applications. In order to be able to predict a good mapping of this set of elements to the cache, extra assumptions are needed. The first assumption is that the tile size is a multiple of the group length \( L \). The second assumption is that the distance \( M \) is also a multiple of the group length \( L \). Now the following transformation is applied to create again the situation as the distribution of situation 3: Consider each group as a single element. Those new elements are repeated at distances \( M/L \) and are mapped to a cache with size \((tile size)/L\). Due to the extra assumptions, the division by \( L \) is possible. Now a good mapping is ensured as long as \( \text{gcd}(M/L, (tile size)/L) = 1 \). This ensures basically that the distance between cache addresses of the first elements of each group is constantly \( L \). Therefore not only the first elements of a group can be stored in the cache, but there is also room for \((L-1)\) consecutive elements. Stating that \( \text{gcd}(M/L, (tile size)/L) = 1 \) is the same as saying that \( \text{gcd}(M, (tile size)) = L \); if the greatest common divider equals the block size, we are in the right situation.

The figures 6.4 and 6.5 show graphically what is described in situations 3 and 4. Within a 2-dimensional image a block of elements is considered. This is the 'working set' that should be stored in the cache. This image is stored linear in the memory. This results in a working set that is spread through the memory. The mapping to the cache is shown for the situation that the tile-size \( t \) and the distance \( a \) are co-prime. The cache is equally filled.

Elements of working set in memory at distances 'a' mapped to tile 't'

- Steps in memory of size 'a' \( \rightarrow \) ensure that 0*a mod t, 1*a mod t, 2*a mod t ... covers the cache completely
- Coverage ensured if \( a \) and \( t \) are co-prime \( \Leftrightarrow \text{GCD} (a, t) = 1 \)

Figure 6.4 Single elements on equal distances in memory mapped to the cache

Groups of \( L \) elements in memory at distances \( a \)

- Jumps in memory of \( a \) \( \rightarrow \) ensure that first elements of groups have a minimal mutual distance of \( L \) if mapped to the cache

Figure 6.5 Groups of elements on equal cache-lines in memory mapped to the cache
This approach ensures for some combinations of padding and tile size that the mapping to the cache is good. For all the intermediate combinations there is no real prediction. However, the following observation shows promising results also for the intermediate combinations. Figure 6.6 shows the conflict miss rate (extra misses when simulating with a direct mapped cache compared to the simulation with an fully associative cache with optimal replacement policy) for an array (the ‘frame’ array from the QSDPCM driver) with the following characteristics: group length $L = 16$; group distance $a = 176$; and the tile size varying between 235 and 275. Figure 6.7 shows for the same range of tile sizes the absolute distance between the gcd$(a, \text{tile size})$ and the group length. This distance is expressed in the following expression:

$$y = \left| L - \gcd(a, \text{tile size}) \right| = \left| 16 - \gcd(176, \text{tile size}) \right|$$

(6.8)

If this distance is 0 we expect a good miss rate. This is well visible in both figures, however there are more similarities. Especially the points with ‘very good’ and ‘very bad’ miss rates are well predictable. Note that the absolute values of both figures are not related, only the relative behaviour is compared. Also interesting to see is that there are actually no other optimal points than predicted by our co-prime property.

In figures 6.6 and 6.7 the same is repeated for the ‘gauss_image’ array from the cavity detection driver. The applied formula for the prediction is the following (group-length = 4; array dimension = 176):

$$y = \left| L - \gcd(a, \text{tile size}) \right| = \left| 4 - \gcd(176, \text{tile size}) \right|$$

(6.9)

Again we see similarities but also differences. Where the simulation shows good miss rates for the tile-sizes 12, 15 and 17 to 20, the mathematical approach only discovers a tile-size of 12 and 20 being optimal. This approach should be seen as an initiation of a model that can be extended in the future.
6.4.3 Designing the right data-layout

We stated that a good cache mapping is ensured if the gcd of the tile size and the image dimension equals the block length. In order to improve the cache behaviour it is necessary that the mapping situation can be influenced. Given the constraint that \( \gcd(a, \text{tile size}) = L \), both \( a \) and \( \text{tile size} \) are parameters that can change the mapping. Reserving a slightly different amount of cache lines for this array in the cache can change the tile size. The parameter \( a \), representing the distance between elements in the memory, can also be influenced. The transformation needed to do this is array-padding. Array padding introduces holes within the array. The distance \( a \) is often related to the fact that a two-dimensional array is linearised; either a row-major or column-major storage order is chosen. The distance \( a \) equals then one of the dimensions of the original array. In this situation, padding is nothing else than extending this dimension.

The process of optimization for one array is finding combinations of tile sizes and padding distances for which hold that \( \gcd(a, \text{tile size}) = L \). There are many combinations of \( a \) and \( \text{tile-size} \) possible. However, those parameters are also related to costs. Globally considered, we can say that changing the tile-size is related to cache size usage and that changing the padding is related to main memory usage. Because a bigger tile size for one array decreases the amount of cache lines available for other arrays in the cache. A bigger padding introduces bigger holes in the array and those holes are locations in main memory that will not be used anymore, hence the memory is less optimally used. It is desirable that both memory and cache costs are minimised. Since the choice for a tile size for one array influences the choice for other arrays, different optimal possibilities will have to be considered if all arrays are taken into account. Therefore the optimal padding and tile size combinations per array are placed in a pareto plot. Only the pareto points are interesting to consider, since they minimise one cost in relation to the other (see Figure 6.10).

When we look more in detail to the different costs involved when tile size and padding are changed, the situation is more complex. The change in tile size can also influence the main memory usage. This is the case when the considered array is the one determining the maximal memory usage (has the most repetitions in main memory). Increasing the tile reduces the amount of repetitions and therefore can reduce memory overhead. Besides that, the change in tile size also influences the amount of capacity misses (since we deviate from the copy size that MHLA proposed). In case the capacity miss rate is very sensitive to changes in the available cache lines (the copy size), the difference in capacity misses should be taken into account.

The amount of eliminated conflict misses by changing the tile, should be bigger than an eventual increase in capacity misses (see Figure 6.11). Padding is related to main memory usage, but extra padding is not necessarily increasing main memory. Introducing holes within the array does increase
the size of an array. But if the complete picture of the tiling approach is considered, extra padding does not always increase main memory overhead. Only if the amount of repeated pieces of the padded array exceeds the amount of repetitions of the dominant array, extra memory is needed. Otherwise padding only results in a different distribution of unused memory locations (see Figure 6.12).

Per array a pareto plot is given. For each array one pareto point should be chosen in order to come up with a data-layout. However not all combinations of points are possible since the sum of the tile sizes may not exceed the cache size. Also related to main memory overhead there are influences possible. Preferably no extra padding is introduced for the array that currently determines the memory usage (with the most repetitions in main memory). So here an exploration is needed to find a solution that satisfies the data-layout constrains. As soon as one solution is found, the parameters are known to interleave the arrays in a data layout.

![Figure 6.11](image1.png)  
**Figure 6.11** Changing tile size influences both data-layout conflict misses and minimal capacity misses

![Figure 6.12](image2.png)  
**Figure 6.12** Influences of the combination of padding and tiling on memory overhead
6.5 Reducing overhead

The data-layout optimizations are basically related to three types of costs: power, performance and area. The power has been our main goal to optimize, since our focus has been to avoid as many cache misses as possible. Extra misses are directly related to extra power. We didn't really worry about performance, since the extra address computations that we introduce can be removed if we apply the address optimization technique Adopt. However, the cost of area is an element we should take into account, because we cannot allow that for cache optimizations the amount of required memory space is increased with factors of ten. Therefore we'll focus in this paragraph on issues related to memory overhead that are specific for the optimizations in relation to reuse-dominated applications.

6.5.1 Constrained tile-size selection

Already at the moment a copy candidate from an array is selected by MHLA, it can be taken into account what will be the influence on memory usage. The amount of memory needed to store an array when the tiling is applied is directly related to the declared size of the array, the size of the selected copy and the cache size. Big arrays with small tiles will be broken in many pieces that have to be repeated. If an upper-bound is given for the memory size, the copy candidate trees can be pruned beforehand, since for some small copy candidates it will be impossible to apply tiling within the constrained memory space.

Also related to memory-overhead is the bypass of arrays. Bypassing is needed if the top-level array is assigned to a layer that is not directly above the CPU and there is no copy candidate assigned to the intermediate layer. If the platform supports bypass no special action is needed. If no bypass is implemented, all arrays will have to go via the cache and therefore a specific place for those arrays should be reserved in the cache. From the power point of view, reserving only one line in the cache is sufficient, since every access to those arrays will be a cache miss anyway. But if also memory overhead is taken into account, a tile-size of one line can have undesirable consequences: in order to store all array elements in the memory, too many repeated blocks (with a size equal to the cache) must be repeated in the memory. Also for those arrays the upper-bound of the main memory defines a lower bound on the reserved tile. Therefore, more space has to be reserved in the cache to bypass arrays. This will decrease the space for other arrays that is left in the cache. The result is that in order to control the memory overhead a less optimal miss-rate can be obtained for platforms without bypass.

6.5.2 Cache in-place of copy candidates

The inplace optimizations as described in paragraph 4.3.4 can be applied here. If several small copy candidates are stored in the cache, it is likely that some copy candidates can be inplaced in the cache since in general the lifetime of small copy candidates is limited to one loop-nest only. However, the copy candidates are copied from bigger arrays whose lifetimes often overlap. Therefore the cache inplace cannot be combined with memory inplace and must be implemented by using multiple interleaving layouts. It depends on the driver if this inplace optimizations can reduce the memory overhead. The gain of the inplace (creating room for bigger tiles) is not always bigger than the extra memory needed to store an extra interleaving layout in memory.

6.6 Experiments and results for the Cavity Detection-driver

In this paragraph we apply the data-layout conflict miss reduction technique on the Cavity Detection algorithm using both tile-size refinement and padding. The optimizations are preceded by the MHLA optimization step that proposes copy sizes for the arrays. The decisions made during the optimization phase are discussed together with the achieved results.

6.6.1 Introduction to the experiments

The optimization has the following goals
- Data-layout transformations are applied on the set of arrays that is selected to be stored in a direct mapped cache. The focus is the reduction of data layout conflict misses.
- Two image sizes and two cache sizes are used to show the optimization possibilities.
The miss-rate of the direct-mapped cache with the best data-layout is compared to the 'optimal' miss-rate of a fully associative cache with optimal replacement policy and similar size. The difference between those simulations is used to evaluate how good the miss-rate of a direct mapped cache can approach the miss-rate of an ideal cache in terms of conflict misses.

The experiments will mainly use simulation results to optimize the code, however also a link is made to the proposed method to use the GCD to predict the miss rate.

In the considered version of the cavity detection algorithm, three signals are selected to be stored in 'layer l' of the memory hierarchy (containing a small direct mapped cache). These are the following signals: the 'in_image' array, a reuse-copy of the 'gauss_image' array and a reuse-copy of 'ce_image' array (see Figure 6.13). Other signals in the algorithm either became scalars in previous optimization-steps (and are stored in the register file) or are intended to be stored in higher memory layers because their size is relatively big compared to the selected cache-layer size. The applied optimization steps are

![Top-level array with cache](image)

Figure 6.13 Selected signals to be stored in the cache

those shown in Figure 1.2 except of the last part of the data layout optimization that is done now. The total storage in the algorithm is reduced drastically. There is no need any longer to store complete (intermediate) images as shown in Figure 3.1. The main arrays are two line-buffers 'Gauss_image' and 'Ce_image' that store both only three lines of the image. Those arrays are the interesting ones to analyze. Both arrays have equal access patterns. Therefore only one array needs to be analyzed. The 'gauss_image' array is studied. The 'in_image' array has 3 elements only. This array is stored at its own cache-locations with a tile-size of 3 elements.

6.6.2 Analyses

The proposed copy-sizes for the gauss- and ce_image array is 9 elements. The size of the in_image array is 3 elements. Our target architecture for optimization is going to be an architecture with a main-memory and a small cache of 32 bytes. This is the smallest power of 2 cache size so that the sum of the array-sizes intended to be stored in the cache does fit. Even some slack-space is left to adjust the tile-sizes of the arrays to minimize the amount of conflict-misses. We will also optimize for a cache of 16 byte. Then there is not enough space to store the complete signals, the tiles need to be reduced. The optimizations focus on those three arrays only. We have to eliminate the rest of the signals, since in our simulation environment no cache-bypass possibilities are present. So the signals stored in the main memory cannot be copied to the CPU without passing the cache.

The following information is combined:
- Information about the usage of the gauss_image array by the application
- The proposed copy-size
- Information from miss-rate simulations of the array

The gauss_image array is a 2-dimensional array of 3 x N elements. A block of 3 x 3 elements is constantly active (the central-element is compared to its neighbors). During execution the center shifts in the x-direction. If the whole line is 'scanned', the center shifts in the y-direction (see Figure 6.14). This block of 3 x 3 elements is reused, therefore the copy-size is 9.

![Group of 9 elements being reused](image)

Figure 6.14 Group of 9 elements being reused: those elements should be stored in the cache
The mapping of the 3 x 3 block to the cache is going to be studied if this array is ‘tiled’ with a specific tile-size. The storage order in memory is fixed: the 3 lines in the x-direction are consecutively stored (see Figure 6.15). The block of 9 elements is now split in 3 groups of 3 elements with a mutual ‘group distance’ in memory of N (=44) elements. The aim is to make those 9 elements map to ‘free’ (different) cache-locations. A good tile-size must be found around the initial size of 9. The size is going to be changed to obtain a good mapping and also array-padding is taken into account. It will be shown that we can play with both the tile-size and padding together to minimize the amount of conflict misses. If the tile-size has to increase too much compared to the initial size (there is not enough space in the cache), array padding can help. In order to decide if a tile-size is good or not, the locations in the cache of the first elements of the groups of 3 are going to be studied. If the ‘distance’ in cache between those first elements is 3 or more (modulo tile-size), the 9 elements will map to different cache-positions and the conflict-miss-rate will be low.

Figure 6.15 Storage of 9 elements in the memory belonging to the same copy

The (relative) memory-positions of those first elements are: 0, N and 2*N (0, 44 and 88). When tiling the gauss_image array with a tile-size of 9, the relative positions in the cache of those elements are:

- 0 MOD 9 = 0,
- 44 MOD 9 = 8,
- 88 MOD 9 = 7.

The distances between those elements in the cache is: 7, 1 and 1. Two of the distances are smaller than 3. Therefore some of the 9 different elements map to same cache-locations. This results in a not-optimal cache-usage because of conflicts. For different tile-sizes this analysis is done. The results are summarized in table 6.1.

Table 6.1 Placing of elements in the cache for different tile-sizes

<table>
<thead>
<tr>
<th>tile-size</th>
<th>'group distance' in memory</th>
<th>1st relative cache location</th>
<th>2nd relative cache location</th>
<th>3rd relative cache location</th>
<th>distances in cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>44</td>
<td>0 mod 9 = 0</td>
<td>44 mod 9 = 8</td>
<td>88 mod 9 = 7</td>
<td>7, 1, 1</td>
</tr>
<tr>
<td>10</td>
<td>44</td>
<td>0 mod 10 = 0</td>
<td>44 mod 10 = 4</td>
<td>88 mod 10 = 8</td>
<td>4, 4, 2</td>
</tr>
<tr>
<td>11</td>
<td>44</td>
<td>0 mod 11 = 0</td>
<td>44 mod 11 = 0</td>
<td>88 mod 11 = 0</td>
<td>0, 0, 11</td>
</tr>
<tr>
<td>12</td>
<td>44</td>
<td>0 mod 12 = 0</td>
<td>44 mod 12 = 8</td>
<td>88 mod 12 = 4</td>
<td>4, 4, 4</td>
</tr>
</tbody>
</table>

Only in the case with a tile-size equal to 12 all distances are 3 or more. In this case no conflict misses are expected. This will be showed by simulation. An extra degree of freedom next to the tile-size is the ‘group distance’ in memory. This we can change by array padding. Therefore the dimension-size of the gauss_image array is extended. For some values of padding, the analysis is shown in table 6.2.

Table 6.2 Placing of elements in the cache for different tile-size and padding combinations

<table>
<thead>
<tr>
<th>tile-size</th>
<th>'group distance' in memory</th>
<th>1st relative cache location</th>
<th>2nd relative cache location</th>
<th>3rd relative cache location</th>
<th>distances in cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>45 (padding = 1)</td>
<td>0 mod 9 = 0</td>
<td>45 mod 9 = 0</td>
<td>90 mod 9 = 0</td>
<td>0, 0, 9</td>
</tr>
<tr>
<td>10</td>
<td>45</td>
<td>0 mod 10 = 0</td>
<td>45 mod 10 = 5</td>
<td>90 mod 10 = 0</td>
<td>0, 5, 5</td>
</tr>
<tr>
<td>11</td>
<td>45</td>
<td>0 mod 11 = 0</td>
<td>45 mod 11 = 1</td>
<td>90 mod 11 = 2</td>
<td>1, 1, 9</td>
</tr>
<tr>
<td>12</td>
<td>45</td>
<td>0 mod 12 = 0</td>
<td>45 mod 12 = 9</td>
<td>90 mod 12 = 6</td>
<td>6, 3, 3</td>
</tr>
<tr>
<td>9</td>
<td>46 (padding = 2)</td>
<td>0 mod 9 = 0</td>
<td>46 mod 9 = 1</td>
<td>92 mod 9 = 2</td>
<td>1, 1, 7</td>
</tr>
<tr>
<td>10</td>
<td>46</td>
<td>0 mod 10 = 0</td>
<td>46 mod 10 = 6</td>
<td>92 mod 10 = 2</td>
<td>2, 4, 4</td>
</tr>
<tr>
<td>11</td>
<td>46</td>
<td>0 mod 11 = 0</td>
<td>46 mod 11 = 2</td>
<td>92 mod 11 = 4</td>
<td>2, 2, 9</td>
</tr>
<tr>
<td>12</td>
<td>46</td>
<td>0 mod 12 = 0</td>
<td>46 mod 12 = 10</td>
<td>92 mod 12 = 8</td>
<td>8, 2, 2</td>
</tr>
<tr>
<td>9</td>
<td>47 (padding = 3)</td>
<td>0 mod 9 = 0</td>
<td>47 mod 9 = 2</td>
<td>94 mod 9 = 4</td>
<td>2, 2, 5</td>
</tr>
</tbody>
</table>
There are situations where we can calculate beforehand where the distances will be big enough. This is the case if the tile-size and the ‘group distance’ in memory is a multiple of the group length. If \( \text{tile\_size} / \text{group\_length} \) is relative-prime to \( \text{group\_distance} / \text{group\_length} \) and the tile-size is big enough to hold all elements, it is ensured that all groups map to different locations. E.g. for the case with \( \text{tile\_size} = 12 \) and the \( \text{group\_distance} = 45 \) we have to check if \( 12 \) and \( 45 \) are relative prime: \( \text{gcd}(12, 45) = 3 \). This is indeed the case. When deciding a tile-size and a padding-distance, the different costs for both should be taken into account. A bigger tile-size possibly reduces the space for other arrays in the cache. This means that the reduction of conflict-misses by the increase of this tile can force other tiles to become smaller. Resulting in more misses for other arrays. Padding enlarges the array. The cost is extra spilling of memory-locations.

Simulations are done for different tile-sizes and different padding-distances with the code where only the accesses to the gauss_image array are present (see Figure 6.16). The theoretical conflict-miss-rate minimum is indicated by the simulation with a fully associative cache with optimal replacement policy (compulsory and minimal capacity-misses are measured, see graph labeled with ‘comp+cap’). The conflict-miss simulations are made with a direct-mapped cache (compulsory, minimal capacity, associativity-conflict and data-layout-conflict-misses are measured, see graphs labeled with ‘comp+cap+assoc+dl’). The cache line-size is 1 element in both cases. This simulation shows that in the situations that are highlighted in the previous tables, the conflict-miss-rate is indeed equal to zero.

In bold face the situations are indicated where the distances are big enough and conflict misses can be avoided.

\[
\begin{align*}
10 &\quad 47 & 0 \mod 10 &= 0 & 47 \mod 10 &= 7 & 94 \mod 10 &= 4 & 4, 3, 3 \\
11 &\quad 47 & 0 \mod 11 &= 0 & 47 \mod 11 &= 3 & 94 \mod 11 &= 6 & 3, 3, 5 \\
12 &\quad 47 & 0 \mod 12 &= 0 & 47 \mod 12 &= 11 & 94 \mod 12 &= 10 & 10, 1, 1 \\
9 &\quad 48 & 0 \mod 9 &= 0 & 48 \mod 9 &= 3 & 96 \mod 9 &= 6 & 3, 3, 3 \\
10 &\quad 48 & 0 \mod 10 &= 0 & 48 \mod 10 &= 8 & 96 \mod 10 &= 6 & 6, 2, 2 \\
11 &\quad 48 & 0 \mod 11 &= 0 & 48 \mod 11 &= 4 & 96 \mod 11 &= 8 & 4, 4, 3 \\
12 &\quad 48 & 0 \mod 12 &= 0 & 48 \mod 12 &= 0 & 96 \mod 12 &= 0 & 0, 0, 12
\end{align*}
\]
6.6.3 Optimization

After these analyses shown in the previous section the three signals are taken into account together again. The cache is 32 lines big. Three elements are occupied by the in_image array. 29 locations are left for the gauss_image and ce_image array. There is enough space to select a bigger tile-size than the original 9 elements. So the first tile-size is selected where without padding all the conflict-misses are eliminated. This is the case for a tile-size of 12 elements. The cavity-code with the array-accesses is transformed in order to tile the arrays. This code is used to measure the over-all miss-rates. The not-transformed code is used as reference to measure the decrease of data-layout misses. The results are shown in Figure 6.18 (2nd bar). As can be seen, a significant amount of misses is eliminated by the tiling-approach. Since the amount of associativity-misses is very small, we can conclude that the data-layout removed practically all conflict misses (the amount of misses of the code with data-layout simulated with a direct mapped cache, reaches the amount of misses simulated with a fully-associative cache and optimal replacement policy).

Since the amount of accesses in both simulations before and after performing the data-layout transformations is equal, we can conclude that there is no register spilling that influences the simulation-results.

Bigger image-size

The cavity-detection normally works on bigger images. For a different image-size, the group-distance in memory will be different, so a new analysis of the mapping is needed. A common image size is 176 times 144 elements. The group-distance is 176. Analyses of this situation results for the gauss_image array again in an optimal tile-size of 12 elements since this is again the first tile-size for which the group-distances in cache are 3 or more. The same simulations are done as for the smaller image. The different misses are shown in Figure 6.18 (4th bar). Note that in principle for a different image size, another combination of tile-sizes will be necessary. But apparently the mapping for a group-distance of 44 elements is the same as the mapping for a group-distance of 176 elements.

![Figure 6.17 Miss-rates gauss_image array (padding is 0, 1, 2, 3 and 4) for image size 176x144](image-url)
Table 6.3 Placing of elements in the cache for one tile-size (big image)

<table>
<thead>
<tr>
<th>tile-size</th>
<th>'group distance' in memory</th>
<th>1st relative cache location</th>
<th>2nd relative cache location</th>
<th>3rd relative cache location</th>
<th>distances in cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>176</td>
<td>0 mod 12 = 0</td>
<td>176 mod 12 = 8</td>
<td>352 mod 12 = 4</td>
<td>4, 4, 4</td>
</tr>
</tbody>
</table>

Smaller cache

For a cache of 16 elements, the intended copy-sizes do not fit in the cache anymore. The size of tiles will have to be reduced. The simulations per array are used to select the right points. The in_image array is left untouched. So there are 16 – 3 = 13 locations available for the gauss_image and the ce_image array. For the image-size of 44x36 elements, the choice is made to use a tile-size of 7 for the gauss_image array and a tile-size of 6 for the ce_image array without applying array-padding. Both points for the tile-sizes 6 and 7 are relatively close to the optimal curve. For the image-size of 176x144 elements, the choice is also made to use a tile-size of 7 for the gauss_image array and a tile-size of 6 for the ce_image array. However to have a good miss-rate with a tile-size of 7 elements, the gauss_image array is padded with 1 element (see Figure 6.17). As we can see in Figure 6.18 (bar 1 and 3), the data-layout cannot remove all conflict misses anymore in the case of a cache of 16 elements.

Figure 6.18 Miss-types for 2 cache-sizes and 2 image-sizes

In the figure with the miss-rates (Figure 6.17) we observe that the data-reuse copy size of 9 is too pessimistic (with a smaller copy-size, the same amount of capacity misses is present). This is known: not all 9 element are alive during the full usage of the copy, because each time there is an update of elements within the copy. The optimal cache shows that with a tile of 7 elements, still a minimal miss-rate can be obtained. But we also see that a data-layout is more difficult to optimize if not all 9 elements are stored. In order to obtain the optimal miss-rate for a tile-size of 7 elements, a clever replacement is needed. This is not directly 'implementable' by the data-layout (for the tile-sizes 7 and 8 no minimal miss-rate is obtained by the direct-mapped cache). We see a trade-off: extra misses can be solved with a higher associativity or by reserving some more space for the tile.
intra-misses gauss_image cavity 176*144, tile-size and padding

Figure 6.19 Full padding exploration for tile-sizes 5 to 12

intra-conflict-misses gauss_image optimal padding per tile (cavity 176*144)

Figure 6.20 Minimal miss-rate per tile-size for specific padding (compared to optimal)
In order to know per tile-size what is the minimal miss-rate we can obtain, an exploration is made over different padding-sizes (see Figure 6.19). Per tile-size 15 bars are shown. The first bar indicates the total amount of accesses (to indicate that the relative change in miss-rates as a result of padding is very big). The second bar shows the optimal miss-rate. The remaining 13 bars show the miss-rate for different padding-sizes between 0 and 12. There is no need to explore the range of tile-sizes further than a tile-size of 12 because for this size all conflict misses are eliminated without any padding. A padding bigger than the tile-size is not needed either, because the miss-rates are going to repeat (e.g. miss-rate for TS = 6 and padding = 1 is equal to miss-rate with padding 1 + 6). This can be understood because if the padding-distance equals the tile-size, the first group of elements in memory is shifted ‘tile-size’ positions, the second group is shifted two times ‘tile-size’ positions. The mapping to the cache (modulo tile-size) is completely equal to the initial situation. And if the mapping is equal, the miss-rate is equal too. In Figure 6.20 the minimal miss-rates per tile-size are collected. Here we clearly see that for tile-sizes smaller than 9 we are not able to reach the optimal miss-rate by a data-layout made with tiling and padding.

All the padding and tile-size combinations that result in an optimal miss-rate for a direct-mapped cache can be drawn. From those points the interesting points in terms of padding and tile-size can be selected (pareto-points). This pareto-curve is shown in Figure 6.21. We see that there are only two interesting padding and tile-size combinations that are optimal for the miss rate.

Figure 6.21 Pareto-curve padding/tile-size for gauss_image array (cavity 176*144)

6.6.4 Conclusions

This cavity detection algorithm is used to optimize the data-layout for conflict misses in the cache. The transformations are basically steered by simulations. The mapping situations are simple and a full exploration of the search-space could be made. This is useful to explain the principles and to show what the impact of the transformations can be on the miss rate.

We showed that a significant amount of miss rate reduction can be gained. We also showed in which situations we are able to predict the miss rate using our mathematical property (not all points can be estimated by the model). From the experiment with the smaller cache size we learned that the tiling approach cannot optimize very well in situations were not the complete copy sizes fit in the cache. In those cases a more clever replacement of elements is needed to reach the optimal miss-rate.
6.7 Experiments and results for the QSDPCM-driver

In this paragraph we apply the data-layout conflict miss reduction technique on the QSDPCM driver using both tile-size refinement and padding. The optimizations are preceded by the MHLA optimization step that proposes copy sizes for the arrays. The decisions made during the optimization phase are discussed together with the achieved results.

6.7.1 Introduction to the experiments

The optimization has the following goals:

- Data-layout transformations are applied in order to optimize the QSDPCM-application for a direct-mapped cache, focusing on the reduction of data layout conflict misses.
- The analysis of the memory hierarchy layer assignment (MHLA) step are taken into account: a target cache-size is given together with initial tile-sizes for the different arrays.
- The miss-rate of the direct-mapped cache with the best data-layout is compared to the 'optimal' miss-rate of a fully associative cache with optimal replacement policy and similar size. The difference between those simulations is used to evaluate how good the miss-rate of a direct-mapped cache can approach the miss-rate of an ideal cache in terms of conflict misses.
- The overhead created by the applied data-layout will be described.
- Encountered difficulties and used strategies related to the optimization will be commented.

The target-architecture for optimization is a two-layer memory hierarchy (a main memory and a cache whose sizes are selected by MHLA). In the memory-hierarchy layer assignment step an optimal cache-layer-size is selected based on the estimated amount of accesses to the different layers and the energy-consumption per memory access. The optimal size for this cache layer is 1024 elements. Figure 6.22 shows for a range of cache sizes the total energy spent in the memory hierarchy. The energy bars are sub-divided. At the bottom the energy due to cache hits is shown. We see that this energy goes up for bigger cache sizes. Not only because there are more hits, but basically because the energy per access increases. The energy due to capacity misses is the energy needed to access the main memory in order to make the copies from main memory to the cache. The energy due to conflict misses is the extra energy needed because of the usage of a direct mapped cache without optimization for conflict misses.

![Figure 6.22 Energy per cache size with a breakdown per miss type](image)

The sizes of these bars is quite random because in each situation the mapping of elements can be very different. For the selection of the minimum cache size this energy is not taken into account. It is assumed it can be drastically reduced.

The selected copy-candidates to be stored in the 1K-cache are indicated in Figure 6.23. The selected sizes are passed to the data-layout phase as first estimation of the tile-size. In the fine-tune phase, the
tile-sizes are only slightly changed to optimize for intra-array conflict-misses. Also the technique of array-padding is used to remove data-layout conflict misses. There is no need to optimize those signals whose complete top-level array is selected to be stored in the cache. In this case each element has its own location in the cache, elements of this array will never compete for the same location. There are four arrays with a selected copy-candidate smaller than the top-level array. This potentially results in conflict misses that should be removed. Those four arrays are: prev_frame, frame, prev_sub2_frame and prev_sub4_frame. The optimization of those arrays will be discussed one by one.

The selected copy-candidate-size of an array belongs to one reference (static read or write) to this array in the code. It can be directly associated with a level in the loop-structure. On this level the access-patterns of the array in the memory is going to be studied in order to predict what is going to be the locations of the accessed elements in the cache. One array can have multiple references. The result of the selected copy size on all different references to the array is taken into account.

The different references to the arrays in the application are numbered in order to distinguish them. From the QSDPCM-application the main-function is considered. In the first instance, references to the arrays are isolated and non-relevant code is eliminated in order to have a clear view on what is happening. References to both the prev_frame and prev_sub2_frame contain a data-dependent component. This so-called 'motion-vector' depends on the outcome of the more coarse-grain motion-estimation. This means that the access of a complete block to the top-level array has a variable offset. In the first experiment this motion-vector is set to 0 to be able to make analyses that do not depend on the image being processed. This has no direct influence on the reuse of data-elements in the selected copy-size. The optimization-results will first be measured with the 'adapted' driver (references to arrays only), later the complete driver will be tested as well.

6.7.2 Optimization of the 'frame' array

This array is an array with 176x144 elements. It has 4 references with the numbers: 2, 6, 9 and 17. The references 2, 6, 17 access a block of 16x16 elements once for each combination of x and y. The x and y loops we find back in the optimization for each array, since they are the enveloping loops for the complete motion estimation. They basically point to 11 times 9 different locations in the considered images. The indicated pattern in the figure accessing particular regions is shifted as a whole for each of the 11x9 combinations of the indices x and y. Reference Nr. 9 accesses for each combination of x and y 9 times a block of 16x16 elements. Those 4 references all access the same collection of 256 elements...
(see Figure 6.24), they only read those 256 elements in different orders. For each combination of x and y the 256 elements are read \(1 + 1 + 9 + 1 = 12\) times. There are \(11 \times 9 = 99\) x-y combinations (shift of 16 elements in x and y). An initial tile-size of 256 elements is proposed by MHLA. The amount of accesses to the frame array is: \(99 \times 256 \times 12 = 304\)K. In the optimal situation (no conflict misses), \(11\) x \(9 \times 256 = 25\)K misses are expected.
Find optimal mapping:
M = 176, L = 16, TS = 256
TS' = 256 (is already multiple of L)
M' = 176 (is already multiple of L)
gcd (TS'/L, M'/L) = gcd (256/16, 176/16) = gcd (16, 11) = 1 => situation already optimal (no padding, no tile-size increase). For simulation results see Figure 6.25.

Next analysable points (not needed, because the first point is already a pareto-point):
TS' = 256 + 16 = 272
gcd (TS'/L, M'/L) = gcd (272/16, 176/16) = gcd (17, 11) = 1 => OK

Optimal point for smaller tile-size:
TS' = 256 - 16 = 240
gcd (TS'/L, M'/L) = gcd (240/16, 176/16) = gcd (15, 11) = 1 => OK but # capacity misses increased.

Note: There are many cases with good solutions because 176/16 = 11 is prime (all other numbers except 11 are co-prime to this one). Concluding the analysis of the frame array: the initial tile-size of 256 without extra padding is the best combination to select.

6.7.3 Optimization of the 'prev_frame' array

Figure 6.26 Access-pattern prev_frame array

Analyses of the situation
This array has one reference with much reuse (reference Nr. 10, selected copy-candidate-size = 18x16) and three references without reuse (Nr. 1, 5 and 18). Figure 6.26 shows the regions in the 2-dimensional array (176x144 elements) that are accessed by those four references for each iteration of the outer x and y loops. The main focus is to avoid conflict-misses for reference Nr. 10 by constructing a good data-layout and avoiding that the other three references do interfere with the elements being accessed by reference Nr. 10. For each combination of x and y there are consecutively 16x16 elements accessed by reference 1, 16x16 elements accessed by reference 5 (used to construct sub-sampled versions of the frame), 9x16x16 elements accessed by reference 10 (used for motion estimation in this region) and 16x16 elements accessed by reference 18. As can be seen from the image, there is overlap between elements being accessed by reference 1 and 5 and between 10 and 18. Between those pairs of references there will be inter-reference reuse of elements.

Figure 6.27 Shape to be stored in cache in order to exploit reuse completely
for(y=0;y<9;y++)
for(x=0;x<11;x++)
for(vy=0;vy<3;vy++)
for(vx=0;vx<3;vx++)
for(m=0;m<4;m++)
for(n=0;n<4;n++)
for(i=0;i<2;i++)
for(j=0;j<2;j++)
for(k=0;k<2;k++)
for(l=0;l<2;l++)
if(! out of frame-access) /* acc_nr = 10 */
dummy+=prev_frame[(y*16 + vy - SAl + i*2 + m*4 + k) * M +
  x*16 + vx - SAl + j*2 + n*4 + l)];
}
}

Figure 6.28 Reference Nr. 10 with its enclosing loop bodies

As described before, the data-dependent motion vector is put to 0. Originally the regions accessed by
references 10 and 18 could be shifted within a region of 60x60 elements (dashed line) for each (x, y)
itration. This could influence inter-reference reuse between references 1, 5 and 10, 18. But now there
is hardly overlap. Reference 1 and 5 do have a partly overlap (4 out of 16 elements shifted in each
direction). Exploiting the full reuse between those references implies that a set of elements needs to
be stored in the cache as indicated in Figure 6.27. The tiling-approach will not be able to store such a
shape perfectly in the cache and moreover the reserved copy-size of 16x18 elements will not be able to
store the complete shape. The complete reuse could be exploited if a copy is introduced equal to a
whole line-buffer (dashed box). In that case the tiling approach can do a good job again because a
regular rectangular shape can be stored.

Assuming that only the overlapping part of the blocks results in reuse (reference Nr. 5 has 56% overlap
with reference 1), we can predict the amount of misses. Reference 1 causes 11x9x256 = 25K misses.
Reference 5 reads only 44% new elements and reuses 56% (if the elements are not flushed because of
conflict misses). In total those two references will cause 36K misses. Reference Nr. 10 reads 9 times a
block of 16x16 elements within the region of the 18x18 block (beginning in left bottom corner, ending
in right top corner). This is repeated for each x and y combination. The amount of expected misses is
11x9x18x18 - 32K. To give an indication how a reference looks like, Figure 6.28 shows a piece of c-
code containing reference Nr. 10 together with its enclosing loop-bodies.

Reference Nr. 18 reads the elements in the center of the block of 18x18 elements (the same region as
accessed by reference Nr 10). However, the copy-size is only 18x16 elements and the last block that is
accessed by reference Nr 10 is in the right upper-corner, so not all elements needed by reference Nr 18
can still be present in the cache. One extra line of 16 elements will have to be re-read (1/16 of 256 -
6% extra misses). For reference 10 and 18 together 25K + 0.6*25K = 33K misses are expected. For all
4 references together roughly 33K + 36K = 69K misses are expected (not taking into account
boundary-conditions and assuming that the data-layout is able to map the blocks of 16x16 and 16x18
without conflicts to the cache).

The optimization
Now it is known how the prev_frame array is accessed. A detailed mapping can be chosen to minimize
the conflict misses. This array is stored in row-major order i.e. the elements in the x-direction from 0 to
M-1 are stored consecutively. The intention is to map blocks of 16x16 and 18x16 to the cache. The
initial tile-size is 18x16. The initial M-dimension is 176 elements. Therefore the mapping of groups of
L (= 18) elements (repeated at distances of M (= 176) elements) should be considered. Since M is no
multiple of 18, the gcd-calculation cannot be performed directly. M needs to be adapted.

Steps to find an optimal mapping
- Determine the required distance L (=18) between first elements of the groups in the cache
- Pad the dimension M (=176) to first multiple of 18 (M' =180) => minimal pad = 4
- Change the tile-size to multiple of 18 (stays 288 = TS') => minimal TS increase = 0
- Calculate $\text{gcd}(TS'/L, M'/L)$
- Change $TS'$ and $M'$ with steps of 18 to make $\text{gcd}$ equal to 1
- $\text{Gcd}(288/18, 180/18) = \text{gcd}(16, 10) = 2$ (not optimal)
- $\text{Gcd}(288/18, (180+18)/18) = \text{gcd}(16, 11) = 1$ good ($TS=288$, pad = 18+4)
- $\text{Gcd}((288+18)/18, 180/18) = \text{gcd}(17, 10) = 1$ good ($TS=306$, pad = 4)
- More padding is not needed because the optimal point is found for a minimal tile-size increase
- More tile-size increase is not needed because the optimal point is found for a minimal padding.

The 'worst-case' minimal pad that is ever needed to make $M$ a multiple of $L$ is $(L - 1)$ (in this case the minimal padding is only 4). The 'worst-case' minimal tile-size increase to make $TS$ a multiple of $L$ is $L - 1$ (in this case no extra padding is needed). It is not easy to guarantee how long it takes before the first combination is found where $TS'/L$ and $M'/L$ are co-prime (by increasing either $TS'$ or $M'$ with $L$). But we see that $TS/L$ and $M/L$ are in the range between 0...20. In this region there are many prime numbers. This makes the change on finding two numbers being co-prime relatively big. If desired, $L$ can be 'oversized' (make $L = 19$ in this case) and study if better combinations of tile-size increase and padding are possible. For this array there is no better combination possible.

In order to show the influence of the tiling and padding-approach on the miss-rate, simulations are made for different tile-sizes (around 288) and different padding possibilities. The results are shown in Figure 6.29. The miss-rates for a direct-mapped cache (compulsory + minimal capacity + associativity + data-layout misses) are compared to the miss rate of a fully associative cache (compulsory + minimal capacity misses). An optimal combination of padding and tile-size should reduce the difference.
between both types of simulations as much as possible. The two interesting points (minimal pad of 4 in combination with tile-size of 306 and pad of 22 in combination with minimal increased tile-size of 288) do indeed show optimal points (indicated by circles). There is a small amount of misses that is not removed. The ideal cache is able to use the cache-space slightly better than the direct-mapped cache. This effect will be discussed later.

A remarkable result is that the optimal points found by choosing the right combination of padding and tiling are better than any point that is found when no padding is applied. However, in order to understand better what is going on, some (sub)optimal points of the non-padding simulation will be considered in more detail. Two interesting tile-sizes seem to be 288 and 299. A manual analysis is made of the location where different elements are placed in the cache. The cache locations of all the first elements of the groups of 18 elements are listed. The distances between those first elements are indicating how much overlap there is between elements of the 16 different groups. The more overlap there is, the more conflicts there are.

Tile-size = 288 and M = 176:
Relative locations of the first elements of groups in memory: k * 176 for k E[0...16]
0, 176, 352, 528, 704, 880, 1056, 1232, 1408 ...
Relative locations of the first elements in the cache: (k*176) mod 288
0, 176, 64, 240, 128, 16, 192, 80, 256 ....
The locations in cache ordered:
0, 16, 32, 48, 64, 80, ..., 256, 272, 0
There is a constant distance of 16 between the different elements. If groups of 18 elements are stored, there is an overlap of 2 elements. In this case it was also possible to predict this by using the gcd: both M and TS are multiples of 16. By calculating: gcd (TS/16, M/16) = gcd (288/16, 176/16) = gcd (18, 11) = 1 it is ensured that the first points of a group are indeed separated by distances of 16. Apparently this still results in a relatively good miss-rate.

Tile-size = 299 and M=176:
Relative locations of the first elements of groups in memory: k * 176 for k E[0...16]
0, 176, 352, 528, 704, 880, 1056, 1232, 1408 ...
Relative locations of the first elements in the cache: (k*176) mod 299
0, 176, 53, 229, 106, 282 etc
The locations in cache ordered:
0, 19, 36, 53, 72, 89, 106, ..., 265, 282,
Both distances of 17 and 19 are observed. The distances of 17 result in an overlap of 1 element between groups, the distances of 19 result in no overlap. This can not be calculated using the gcd because 299 and 176 is no multiple of 17 neither of 19.

Concluding the analysis and optimization of the prev_frame array: two interesting points are found. The trade-off can be made using more memory (big padding) or more cache (big tile-size).
- Minimal pad of 4 in combination with tile-size of 306
- Pad of 22 in combination with minimal increased tile-size of 288
Which point should be selected depends on the over-all picture of all arrays. In a compact way the optimization of the other arrays will be described.

6.7.4 Optimization of ‘prev_sub2_frame’ array

This array has 88 * 72 elements. Reference 8a writes to the array. For each x, y combination a block of 8*8 elements. In total: 64 * 99 = 6336 writes. Reference 8 read the array and has a lot of reuse. It reads a block of 8*8 elements shifted through block of 12*12 elements (in total 25 positions). Their results in 99*25*64 = 158K reads (see Figure 6.31). Like for the prev_frame the data-dependent motion vector is put at 0. Originally reference 8 can be shifted within region of 28x28 elements for each x, y combination. The selected copy-size is a line buffer of 12*8 elements. The information used to avoid conflict misses is: lines of 12 (=L) elements in dimension of 88 (=M) elements.

L = 12 => pad M to a multiple of 12 => M' = 96 (padding = 8)
TS = 12 * 8 = 96 = TS'
gcd (TS'/L, M'/L) = gcd (96/12, 96/12) = 8 => combination (M'=96, TS'=96) is very bad.
Enlarge TS
\[ TS' = 96 + 12 = 108 \]
\[ \gcd \left( \frac{TS'}{L}, \frac{M'}{L} \right) = \gcd \left( \frac{108}{12}, \frac{96}{12} \right) = \gcd (9, 8) = 1 \Rightarrow \text{combination (M'=96, TS'=108) is good.} \]

Enlarge padding
\[ M' = 96 + 12 = 108 \ (\text{padding} = 20) \]
\[ \gcd \left( \frac{TS'}{L}, \frac{M'}{L} \right) = \gcd \left( \frac{96}{12}, \frac{108}{12} \right) = \gcd (8, 9) = 1 \Rightarrow \text{combination (M'=108, TS'=96) is good.} \]

The simulations in Figure 6.31 show that the combinations described above are indeed good/bad. There are more points visible that show good results. They are not all predictable by computing the gcd. For example the situation with \( TS=85 \) and \( \text{padding} = 8 \). Analyses of the cache-mapping show that in this case there is 7 times a distance of 11 elements between the begin-elements of each group (1 element overlap) and one time a distance of 8 elements (4 elements overlap). Apparently this is better than the situation for \( TS=84 \) and \( \text{padding} = 8 \). In this case there is 7 times a distance of 12 elements (no...
overlap), but one complete line of the block cannot be stored. One other predictable and interesting point is the situation where $L$ is changed to 13. Change padding and tile-size: $M$ multiple of 13 => pad $M$ to multiple of 13 => $M' = 91$ (padding = 3). Change $TS$ to multiple of 13: $TS' = 104$. Compute: gcd ($TS'/L, M'/L$) = gcd (104/13, 91/13) = gcd (8, 7) = 1 => combination ($M'=91, TS'=104$) is good.

Concluding the analysis of the prev_frame array.

Three interesting analyzable points are found:
- Pad of 8 in combination with tile-size of 108
- Pad of 20 without change in tile-size (96)
- Pad of 3 in combination with tile-size of 104

6.7.5 Optimization of ‘prev_sub4_frame’ array

Since the optimization of this array is very similar to the previous ones, only a summary is given.

Array-size: 44x36 elements. For each $x, y$ combination a block of 12x12 elements is accessed (shift of 4 elements each time). Within the block of 12x12 elements, blocks of 4x4 elements are read at each of the 81 possible positions. The copy-size is a block buffer of 12x12 elements. Conflict-miss optimization: ensure that with $L=12, N=44, TS=144$ elements don’t overlap in cache.

$L = 12 => pad M to multiple of 12 => M' = 48 (padding = 4)$

$TS = 12 * 12 = 144 => TS'$

$gcd (TS'/L, M'/L) = gcd (144/12, 48/12) = gcd (12, 4) = 4 => combination (M'=48, TS'=144) not optimal.$

Enlarge $TS$

$TS' = 144 + 12 = 156$

$gcd (TS'/L, M'/L) = gcd (156/12, 48/12) = gcd (13, 4) = 1 => combination (M'=48, TS'=156)$ is good.

Enlarge padding

$M' = 48 + 12 = 60 (padding = 16)$

$gcd (TS'/L, M'/L) = gcd (144/12, 60/12) = gcd (12, 5) = 1 => combination (M'=60, TS'=144)$ is good.

Concluding the analysis of the prev_frame array. Two interesting analyzable points:
- Pad of 4 in combination with tile-size of 156
- Pad of 16 in combination with tile-size of 48

6.7.6 Global optimization results for the different arrays together

The choice which tile-size/padding combination is going to be selected depends on several issues. The sum of tiles is constrained by the cache-size of 1K. The introduction of extra padding should not drastically increase the main-memory usage. And the miss-rate should be minimized.

The selection of tile-sizes given by MHLA exactly fills the cache of 1024 elements. This means that there is no slack-space available. No extra space needs to be created if padding can solve the conflict misses. To analyse the cost of extra padding, the memory usage is considered. The tiling-approach and the belonging interleaving of arrays in memory often results in overhead of main-memory (unused memory locations). Padding can spread the unused locations differently and does not always result in extra overhead. In order to know this effect, the memory used by each array is calculated. The interleaving pattern in memory breaks the arrays in pieces with a size equal to the tile-size. Those pieces are repeated in main-memory each cache-size. Per array we calculate the amount of pieces (with and without padding). The array with the biggest amount of pieces determines the main-memory-size. As long as other arrays do not exceed this maximum memory-size if extra padding is introduced padding is ‘for free’. The influence of padding on the memory-usage is determined.

The array with the most repeated pieces in main-memory is the ‘frame’ array. With a tile-size of 256 elements, there are 176x144 / 256 = 99 pieces that have to be stored. This array did not need any change in order to be optimized for conflict misses. The prev_frame array needs: 176x144 / 288 = 88 repeated pieces for the initial tile-size. Conflict misses could be eliminated for this tile-size by introducing a padding of 22. The amount of repeated pieces becomes then: (176+22)x144 / 288 = 99.
The same number as the 'frame' array. No extra memory is spilled, only the free locations are spread in a better way. For the 'prev_sub2_frame' array: (88+20) x 72 / 96 = 81 repetitions (instead of 66 before padding). For the 'prev_sub4_frame' array: (44+16) x 36 / 144 = 15 (instead of 11 before padding). The other arrays of the QSDPCM also being stored in the memory all need less than 99 repetitions.

The total memory usage after detailed data-layout (99 times cache-size) is: 99 x 1024 = 100K. Initially the memory usage was the sum of the declared sizes from all the different arrays: 25344 + 25344 + 2 x 6336 + 2 x 1584 + 64 + 5 x 16 + 4 x 4 = 66K. This means that the QSDPCM-driver can be optimized for the direct-mapped cache of 1K if 1.5 times more memory is spent.

Table 6.1 shows the results if all the array-references of the QSDPCM application are simulated at the same time. The different situations show miss-rates for the application where: 1. The data in main-memory is not organized at all (the linker placed the different arrays consecutively in memory). 2. The data in main-memory is ordered as proposed by MHLA, not optimized for conflict misses. 3. The data in main memory is optimized for conflict misses too. The miss-rate increase is compared to the ideal miss-rate of a fully associative cache with optimal replacement policy (independent to the data-layout). The most optimal data-layout comes with a miss rate that is still is 15% higher than the optimal miss rate. One of the reasons is the fact that no in-place possibilities in the cache are explored. Therefore the optimal controller is able to use locations in the cache for one array if the lifetime of another array is not overlapping (more experiments will be done with a situation where in-place is done). Another reason is that also the individual arrays have a difference between the simulations with the two types of caches.

This difference will be discussed in more detail. The results show that it is interesting to do a detailed analyse of the different arrays in order to minimize the data-layout conflict misses. Directly using the tile-sizes proposed by MHLA does not result in optimal situations. But a detailed optimization is not possible if the MHLA does not provide the initial tile-sizes. The combination of both makes a good optimization possible.

**Table 6.4 Global results: comparison optimized / non-optimized data-layout for direct mapped cache of 1K**

<table>
<thead>
<tr>
<th>Applied optimizations</th>
<th># accesses</th>
<th>compulsory + min capacity misses</th>
<th>compulsory + min capacity + associativity + data-layout misses</th>
<th>% miss-increase (fix-opt capacity misses to dm cache)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>1.3 x 10^6</td>
<td>94383</td>
<td>204164</td>
<td>116%</td>
</tr>
<tr>
<td>MHLA</td>
<td>1.3 x 10^6</td>
<td>94410</td>
<td>134833</td>
<td>43%</td>
</tr>
<tr>
<td>MHLA + detailed data-layout analyses</td>
<td>1.3 x 10^6</td>
<td>94410</td>
<td>108861</td>
<td>15%</td>
</tr>
</tbody>
</table>

**Why are not all conflict-misses removed?**

The aim of a good data-layout is to achieve with a direct-mapped cache the miss-rate of a fully associative cache with optimal replacement policy. Table 6.5 shows simulation results for the arrays of the QSDPCM-application. In the last column of this table, the percentage extra misses is shown that is present if the optimal cache is replaced by a direct-mapped cache. Here we want to explain what is the reason for this increase (for one array even a miss-rate increase of almost 30%).

**Table 6.5 Global results: comparison optimized / non-optimized data-layout for individual signals**

<table>
<thead>
<tr>
<th>optimized arrays</th>
<th># accesses</th>
<th>tile size</th>
<th>compulsory + min capacity misses</th>
<th>compulsory + min capacity + associativity + data-layout misses</th>
<th>% miss-increase (fix-opt capacity misses to dm cache)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prev_frame</td>
<td>302534</td>
<td>288</td>
<td>63164</td>
<td>67275</td>
<td>6.5%</td>
</tr>
<tr>
<td>frame</td>
<td>304330</td>
<td>256</td>
<td>25346</td>
<td>25546</td>
<td>0.8%</td>
</tr>
<tr>
<td>prev_sub2_frame</td>
<td>159976</td>
<td>96</td>
<td>9997</td>
<td>12900</td>
<td>29%</td>
</tr>
<tr>
<td>sub2_frame</td>
<td>164740</td>
<td>64</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>prev_sub4_frame</td>
<td>115892</td>
<td>144</td>
<td>2580</td>
<td>2964</td>
<td>14%</td>
</tr>
<tr>
<td>sub4_frame</td>
<td>129892</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>mean2</td>
<td>25348</td>
<td>64</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>m4, n4, t4, QC_t2, QC_m2, m2, n2, t2, QC_t4</td>
<td>117616</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>
We'll focus on the prev_sub2_frame that shows the biggest difference between both simulations. The part of the code accessing this array is fully isolated and is changed to do the experiments to determine the source of the extra misses. The initial situation for this array: a miss-rate-increase of 29% is observed (10092=>13057 misses).

1. Now only reference 8 is considered and 8a is removed (possible inter-reference-effects between 8a and 8 are eliminated) => a resulting increase of 17% is observed (10187=>11869 misses).
2. Now the access-pattern is changed to eliminate the reuse between y-iterations (y-direction steps enlarged from 8 to 12, so there is no overlap in the y-direction anymore) => remains 17% increase (10650=>12521 misses).
3. Conditions are removed (on boundaries a full block is read, extra accesses, but the accesses are more regular) => still increase of 14% observed (11304=>12974 misses).
4. The code is changed (and padding adapted) to remove overlap in x-direction between different x-iterations => only increase of 0.3% remaining (14364=>14414 misses).

The transformation of step 1 has a big impact. Our first explanation therefore was: the optimal cache profits from the fact that reference 8a brings elements to the cache that are needed by reference 8. But this explanation is weak because there is no direct overlap between the regions accessed by reference 8a and 8 (see Figure 6.30). There is a big time-distance between the moments that both references access the same elements. The elements accessed in-between can never be stored in the cache. The second explanation is better: the optimal cache can 'bypass' elements written by reference 8a (e.g. store them always in the same line). In this way the reuse between different x-iterations of reference 8 (see Figure 6.32) is not destroyed. The direct mapped cache cannot avoid that reference 8a replaces all elements that are brought to the cache by reference 8. The amount of misses in both cases can roughly be estimated. Without inter-x reuse: 144x99 misses with inter-x reuse: 144x99 - 4x4x99 misses. The difference is about 12%. This is roughly the difference we observe between 29% and 17%.

The transformation of step 2 doesn't influence the difference in miss-rate. This is not strange since reuse in the y-direction is not local in time at all. Without boundary-conditions (step 3), only complete blocks are accessed. The data-layout is constructed to deal with complete blocks. If parts of blocks are accessed, less elements need to be stored and the remaining cache-locations can be used to exploit some extra local reuse. Only the optimal-cache can profit from this irregular behaviour. The last part of the difference is removed by transformation 4. Now there is no reuse anymore between different x and y iterations.

The last two illustrations in Figure 6.32 show that the reuse between x-iterations can be better exploited than the direct-mapped cache can do. An optimal cache can use every location that contains an element whose lifetime is exceeded to store one of the elements that can be reused between x-iterations. We tried to increase the performance of the direct mapped cache in this situation by analysing the reuse of different elements within the accessed region (see Figure 6.33). Since the copy size is restricted (12x8 elements), not all elements can be stored. A simple experiment is done. We tried to store those elements that have the highest reuse. A split is made into 2 regions: Region A and B (basic group splitting). Region A contains all elements being most frequently accessed and are stored in the cache (tile-size = copy-size). Elements from region B are bypassed. With a tile-size of 96, all misses in region A can be solved. There is no cache space left for elements in region B. Those should be bypassed (each access is counted as a miss). Region B contains 33% of the total amount of elements, but has only 15%
(=23760) of the total amount of accesses (=99x64x25=158400). But, if all those accesses are counted as misses, the miss-rate is several times higher (110%) than the optimal one. Moreover, a conflict becomes visible: if we are in another area of the image (one row higher in the y-dimension), the roles of regions A and B completely swap. The elements that were in the often-accessed region first, are hardly accessed now. The assignment of tile-size to regions is fixed and cannot be changed depending on the locations that are currently accessed in the image.

Concluding
The last 15-20% difference between the optimal and the dm-cache seems not easy to remove. In theory this could be eliminated if the granularity of splitting is back to scalar-level with a control-mechanism that knows the future. But the cost and the complexity for this is very high.

6.7.7 Simulating the complete QSDPCM driver (data-dependent behavior and full functionality included)

In order to perform a reasonable exploration for the QSDPCM driver, the first problem to solve is the fact that MHLA currently doesn’t distinguish between different data-types. In this case it simply works with a cache of 1K elements, not taking into account the fact that the different arrays are a mixture of integers (4 bytes) and chars (1 byte). A work around is to change all arrays to integers and simulating with a cache-size of 1024 integers (=4K bytes). The second problem is related to cache-bypassing. MHLA assumes that arrays can bypass the cache if it doesn’t want to store those arrays in the cache. The target-platform of the simulator doesn’t support bypassing. In order to simulate the complete driver, there are two arrays that need to be bypassed (‘rec2’ and ‘prev_rec2’). Since this is not possible, those arrays will have to be assigned to the cache too. Therefore a cache-region of 132 elements is ‘reserved’ for bypass. A new run of the MHLA tool with a cache size of 1024 – 132 = 892 is made. Now three extra arrays (QC_t2, QC_m2, QC_t4) are not assigned to the cache layer anymore. 128 cache elements will be used to bypass both rec-arrays (a quite big cache part has to be assigned, otherwise a lot of extra memory is spilled). The remaining 4 elements are used to bypass the other (small) QC-arrays. The new run of the tool also chooses another copy-candidate-size for the prev_sub4_frame array (48 instead of 144). New optimal combinations of tile-size / padding are listed below. From the possible tile and padding situations, a data-layout is constructed.

<table>
<thead>
<tr>
<th>array-name</th>
<th>tile-size / (padding) combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>prev_frame</td>
<td>288 / 22</td>
</tr>
<tr>
<td>frame</td>
<td>256 / 0</td>
</tr>
<tr>
<td>prev_sub2_frame</td>
<td>96 / 20</td>
</tr>
<tr>
<td>sub2_frame</td>
<td>64 / 0</td>
</tr>
<tr>
<td>prev_sub4_frame</td>
<td>48 / 16</td>
</tr>
<tr>
<td>sub4_frame</td>
<td>16 / 0</td>
</tr>
<tr>
<td>mean2</td>
<td>64 / 0</td>
</tr>
</tbody>
</table>
The memory overhead is still determined by the ‘frame’ array, because for this array the proportion (declared size / tile size) is dominant. 99 repeated cache blocks are needed in the main memory. For the ‘prev_frame’ array, (with the same declared size as the ‘frame’ array) the proposed padding of 22 can be applied freely, because the expanded ‘prev_frame’ array within its tile of 288 results in not more than 99 repeated cache blocks in main memory. The tile size chosen of both bypassed ‘rec’ arrays is such that exactly the 99 repeated cache blocks are used in memory (so for those arrays the minimal tile is chosen without extra memory overhead). For all the other arrays, the amount of repeated cache blocks is smaller than 99 (the tile size and padding combinations shown in the first column of table 6.6 are chosen).

Several simulations are done in order to measure the impact of the data-layout optimizations. The main results are given by the simulations with the codes before and after detailed data-layout using a direct mapped cache. This is compared to the simulation with a fully associative cache with optimal replacement policy. Some extra simulations show the impact of other aspects: one simulation measures the impact of using the tile sizes by MHLA directly without detailed padding optimization. For another simulation, an attempt is made to show what can be gained with padding only. And extra simulations are done in order to understand the source of the remaining associativity conflict misses. The results are shown in Figure 6.34.

The difference in miss rate before and after data layout is significant. A reduction of 56% is obtained. The memory increased with a factor of 1.5 in order to construct this layout. The tiling approach using directly the tile sizes proposed by MHLA also shows a reduced miss rate: the elimination of (inter array) conflict misses has already a positive impact. However, this result is not very strong. The intra array simulations showed that the amount of intra array conflict misses can be very sensitive to the choice of padding / tile size. Directly applying the tiles from MHLA can therefore result in both good

<table>
<thead>
<tr>
<th>Array</th>
<th>Declared Size</th>
<th>Tile Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>m4</td>
<td>16 / 0</td>
<td></td>
</tr>
<tr>
<td>n4</td>
<td>16 / 0</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>16 / 0</td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td>4 / 0</td>
<td></td>
</tr>
<tr>
<td>n2</td>
<td>4 / 0</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>4 / 0</td>
<td></td>
</tr>
<tr>
<td>QC_t2</td>
<td>1 / 0</td>
<td></td>
</tr>
<tr>
<td>QC_m2</td>
<td>1 / 0</td>
<td></td>
</tr>
<tr>
<td>QC_t4</td>
<td>1 / 0</td>
<td></td>
</tr>
<tr>
<td>prev_rec2</td>
<td>64 / 0</td>
<td></td>
</tr>
<tr>
<td>rec2</td>
<td>64 / 0</td>
<td></td>
</tr>
</tbody>
</table>

Data layout:
- Memory overhead: factor 1.5
- Miss rate reduction: 56%

Figure 6.34 Data-layout optimization results full QSDPCM application
The miss rate indicated with the label 'only padding' is to show what can be gained with only padding techniques for this driver (without the tiling approach). First of all it should be mentioned that this number only gives an indication. The applied padding technique is based on work presented by Panda [11]. But since not all details of this work are taken into account, the results cannot be defended very strongly.

The applied ideas are: 1. each array is mapped to the cache in such a way that the selected working set (equal to the tile size proposed by MHLA) does not result in intra array conflicts (intra array padding); 2. the inter array conflicts are taken into account by adjusting the relative placing (offset) per array in order to prevent conflicts between working sets of different arrays (inter array padding). However, this is not trivial. Working sets of different arrays are spread in different patterns through the cache and also move differently through the cache during the execution of the application. Therefore only conflicts between clusters of arrays with comparable behaviour could be taken into account. The results are poor. Maybe this driver is not suited for padding only, because many different arrays are alive at the same time. There is really a lot of data being used within a relatively small cache. If all those arrays are using the complete cache, it is hard to avoid conflict misses between arrays. With our tiling approach that definitely removes all inter array conflict misses we have a strong tool to create order in the chaos of different arrays with different access patterns.

Extra experiments focussed on the gap between the miss rate of the direct mapped cache with optimal data layout and the miss rate of a fully associative cache with optimal replacement policy (see Figure 6.35). The question to be answered is: what is the reason for this gap and what can be done in order to reduce it.

The first part of this gap can be explained by the extra cache-space reserved for bypassing. A fully associative cache can bypass arrays by assigning them to one cache line only. The rest of the 'bypass space' can be used to give room to other arrays. The impact of this is measured by using a cache size of $1024 - 128 = 896$ elements when simulating the optimal case. The initial gap has a size of 35K misses. The bypass is responsible for 6K misses. If the target platform supports to decide per array if it can bypass the cache, this overhead can be eliminated.

A second contribution to this gap is already extensively described: for each array independently the optimal cache can do a slightly better job than the direct mapped cache (exploiting some extra reuse). Part of this gap could be solved if we could decide per array reference if it should be bypassed or not. In this way it can be avoided that references without reuse destroy the cache-usage of references with reuse. A special bypass-read-instruction is needed to achieve this. Depending on the complexity of the reuse of the array, the discrepancy is smaller or bigger per array. The sum of the contributions to the

![Figure 6.35 Data-layout related conflict misses, not solved by current layout](image-url)
gap for the different arrays is about 11K misses in this situation. We didn’t succeed in reducing this amount by e.g. basic group structuring.

A third contribution is related to optimizations in which more than one array is involved. The optimal cache is able to analyze lifetimes of arrays in a very detailed way. For some copy candidates that are present in the cache, the lifetime is very short. This candidate can be in-placed with other candidates. Therefore, extra cache space is available to store more data and extra reuse can be exploited. The effect of this is measured in the following way: the gap between direct mapped and optimal cache is measured per array and afterwards the gap is measured for groups of arrays (in this situation, the size of the optimal cache is set to the sum of the tile sizes of the arrays in the group). The four important arrays (being optimized for conflict misses) showed the gap of 11K misses if the individual gaps are summed. However, the combined simulations for those arrays show a gap of 21K misses. The main contributors are the ‘prev_sub2_frame’ and the ‘prev_sub4_frame’ array with tile-sizes of 96 and 48 elements respectively. The lifetimes of those particular sizes are not overlapping and the increase of tile size can introduces quite some extra reuse.

The solution looks simple: create extra space in the cache by really doing the cache in-place. But this is not so trivial. The in-place possibilities are not so big that they create enough space that a smaller cache of 512 bytes cache can be introduced (with the same combination of tile-sizes). This means that the space created in the cache is used to increase other tiles. However, selecting a bigger copy candidate for e.g. ‘prev_sub2_frame’ and the ‘prev_sub4_frame’ extends the lifetimes of those arrays in such a way that in-place is not possible anymore. A new combination of tile sizes with cache in-place possibilities taken into account does not bring significant changes in the layout. The experiments we did with in-place resulted in miss rates very close to the one with optimal data-layout without in-place (around 1% higher) and also in the memory overhead was no difference. However, we expect that in other situations (other drivers) in-place will be able to reduce memory overhead either to reduce the miss rate.

When finding out why for one array not all conflict misses can be avoided, the conclusion was that the situation becomes to complex. The optimal cache can decide per element where to store and what to replace. In the situation with more array considered at the same time, the same is true. The situation is even more complex (different arrays with different access patterns). Again the conclusion is: there is an amount of misses that can in theory be removed by a better data layout (if the situation is considered at scalar level), but this will come with high cost in complexity and addressing.
Now $21K + 6K = 27K$ of the gap is explained. The part that is the result of bypass can be eliminated by choosing a platform that actually supports bypass, but the remaining part is not trivial to eliminate. There is a part of the gap that is not explained. Possibly it is related to a not fully optimal assignment of tile sizes to the arrays (should be solved by improving the reuse models in the MHLA step). It is an accumulation of small effects that are currently not completely explainable. But this remaining gap of $35K - 27K = 8K$ is not big anymore in relation to the optimal miss-rate (115K). However, the total not removable gap is still 30% of the optimal miss rate. But in total we removed 195K misses and there are only 27K misses left to reduce for future work.

In order to have a reference how well the performance is when an optimal data layout is applied, the miss rates of the direct mapped cache are compared to a 2-way and a 4-way associative cache with least recently used replacement policy (see Figure 6.36). Those caches have a more ‘intelligent’ controller and should be able to solve (part of) the conflict misses themselves. The direct mapped cache with data layout easily outperforms the 2-way associative cache without data layout and has nearly the same miss rate as a 4-way associative cache without data layout. The data-layout code is also simulated on the 2- and 4 way cache. On the 2-way cache, the data layout also improves the miss rate, however on the 4-way cache the miss-rate is not improved if our data layout is applied. Note that the same data-layout is used as is used for the simulations with a direct mapped cache. The layout could be adapted to 2 and 4 way caches (using a finer interleaving pattern, see figure 5.2) for refined simulation results.
7. Conclusions and Scope for Future Work

In this section we conclude the thesis with final remarks about the main contributions. Finally we list the most important opportunities for future research in the context of cache optimizations.

7.1 Conclusions

In the beginning of the thesis, no prediction could be made about the final stage of the work. Some previous work was available, but this could not always be re-used. Some principles like the tiling approach we have taken over, but parts about the prediction of the miss-rates had to be reconsidered.

In this thesis we have tried to clarify however the possibilities and limitations of the tiling approach by both looking at the ability of optimizing in different situations, and also looking at situations where we could not solve all conflicts we would like to. We have introduced extra data-layout optimization by a combination of tiling and array padding.

We have discovered that it is not trivial to de-couple optimizations for minimal capacity misses and for data-layout conflict misses for direct mapped caches. But especially for the data-reuse dominated drivers we could indicate situations with clear conflict miss problems. We were even able to predict and eliminate those misses effectively. For all the drivers we have passed through our process of optimization we are able to reduce the cache miss rate by taking into account the cache-parameters while designing a memory-data-layout. The principal factor of overhead (the memory usage) has been kept within reasonable margins.

Using our data-layout optimizations we have been able to solve many cache misses that are normally present when a direct mapped cache is used. Therefore it is not needed to introduce 2-or 4-way associative caches since we can solve these misses in a power efficient way using a direct mapped cache.

7.2 Possibilities for Future Work

- Refinement of miss-prediction model:
The optimizations we have presented in this thesis are mainly driven by experiments. Based on the observations we tried to understand the behaviour of the cache from a compiler point of view. This understanding is used to develop optimization-approaches for the reduction of cache misses. Although our focus is the reduction of conflict misses, the transformations in the data-layout also incorporated minimal capacity misses. We tried to develop techniques to predict the amount of misses for both data-reuse dominated and not dominated applications. It turned out that the prediction of real conflict misses is not so trivial. For a specific group of access-patterns we were able to predict situations where the miss-rate had to be low. However, this is not a real model yet. A possible mathematical approach is presented, but a better formalization is needed here. The focus should be to come to a model that can predict the miss rate for more than only 'some optimal points'.

- Full integration of the inter-array inplace technique:
We proposed to integrate the inter-array inplace technique with the cache-miss optimization, but more knowledge is needed here. Both techniques affect the data-layout and the inter-inplace can potentially help reducing the memory overhead caused by the tiling approach. However, the inplace possibilities depend on the driver being studied. For example in the QSDPCM not much inplace possibilities were present, since many of the arrays are continuously alive. We believe that in other situations, where many arrays are used locally in time, the inter-inplace of arrays is needed to come to good optimization results. Extra experiments should clarify the relation between this inter array inplace and the cache miss-optimizations. Especially the possibility of cache-inplace
becomes important to avoid that the tiling approach reserves locations in the cache for arrays that are only used in a part of the application. Those cache locations should be reused by other arrays.

- Dealing with the limitation of one tile-size:
The weighed approach that is used in the proposed miss model can possibly be integrated with MHLA to solve the problem of multiple tile-sizes. When designing a data-layout, only one size for an array can be chosen. If MHLA selects multiple sizes for one array (for different references of the array), still only one decision for the tile size can be implemented in the data layout. The presented miss model (and the combination of the models for different references in different loop-nest) can probably be used to decide on one final tile-size. The model can also potentially help MHLA to select array sizes if no data-reuse copies are present and it has to assign parts of arrays to the cache.

- Extension to block-sizes bigger than one data-element:
Until now we assumed the cache block size (line-size) contains only one data-element. Therefore the cache is only used to explore temporal locality. With bigger block sizes also the spatial locality becomes important. Pure data-layout optimizations will not be able to optimize much in relation to spatial locality. But for example the decision on the storage order of array dimensions can do so. The decision on the optimal block size is also to be made. Here is a trade-off: small block sizes will cause many cache misses because there are many blocks to fetch. Big block sizes will also cause many misses because not all elements in big blocks will be needed together. In between there is an optimal block-size that should be found. Once the block size is known and the optimizations for spatial locality are done, the optimizations for conflict misses as presented in this theses can by applied. Assuming that we would like to store a combination of complete blocks in the cache, the optimization-techniques will not change only the ‘granularity’ is different. We don’t consider individual array elements anymore, but always blocks of elements together. In order to integrate this block extension smoothly some extra thinking is needed.

- Extended cache control:
The presented optimizations are mainly based on the usage of a direct-mapped cache. The motivation for this is the simple controller with little power consumption. However, we saw that some extra freedom in the cache control could reduce the amount of conflict misses. One of the most important ones is the ability of cache-bypassing. This bypassing can be considered on several levels: bypass decisions per array (it is decided to store a array in the cache yes or no), bypass decisions per reference (it is decided per reference in the source code if the accessed elements should be stored in the cache). It should be investigated if the introduction of this extra flexibility has a positive influence on the over-all power cost.

- Extensions concerning tooling:
The most important tool to support the experiments until now was the code transformation. Time-consuming address transformations could be made quickly. The next part that is interesting to automate will be the fine-tune phase of the tile-sizes and the decision of the padding distances, based on the copy sizes selected by MHLA and information about the access patterns (array dimensions, copy dimensions). This information is known within the MHLA tool and will have to be passed. This new tool should calculate the interesting tile-size and padding combinations using the GCD-property (or an extended version of it). It will produce different pareto points for each array. A second step is needed to combine those points from the different arrays taking into account memory overhead and the cache-size constraint. In this way a global data-layout is constructed and offset and tile-size parameters are passed to the transformation tool. If also inter-array inplace optimizations should be taken into account, the situation becomes more complex. Before this can be automated, the methodology should be more refined first.
Bibliography


[8] Kulkarni, C.


[12] Patterson, D. and J. Hennessy

*MIPS IV Instruction Set*, revision 3.1. MIPS Technologies, INC., Mountain View, CA, January 1995.

[14] Strobach, P.