

Department of Computer Science
The University of Texas at Austin

Institute for Informatic
The Technical University of Munich

Diploma Thesis

Software Methods for Avoiding Cache Conflicts

Frank Kuehndel

August 25, 1998

Supervisor (Austin): Prof. Ph.D. Calvin Lin
Supervisor (Munich): Univ.-Prof. Dr. Dr.h.c. Juergen Eickel

I assure that I wrote this diploma thesis myself and that I used only the stated sources and tools.

Ich versichere, dass ich diese Diplomarbeit selbstaendig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, August 25, 1998

.....

Acknowledgement

I want to thank

- Prof. Calvin Lin for many hours of discussion, for the many hints which really improved this text and for correcting numerous spelling mistakes.
- all those people who made it possible for me to write my thesis in America. If I would write here all their names it would fill the entire page but I want to thank especially my parents and Dr. Linda Conrad from the International Office of the Technical University of Munich and Becky R. Conn from the International Office of the University of Texas at Austin.

Abstract

Caches improve the speed of programs by reducing the number of accesses to the slowly main memory. Unfortunately, most programs which work with arrays suffer from cache conflicts. Cache conflicts slow down the programs and counteract the advantages of the cache. The speed of affected programs can be improved by avoiding cache conflicts. This thesis concentrates on the question *how to avoid cache conflicts with software methods*. I describe

- the state of the art software methods
- how to pad for tiling
- a new padding algorithm (Odd-Padding)
- a new technique to avoid cache conflicts (Tetris)
- experiments which compare these techniques

Furthermore, I prove the odd-padding algorithm correct.

Contents

1	Introduction	2
2	The State of the Art	6
2.1	A historically interesting paper	6
2.2	Finding the optimal tile size	6
2.3	Copying the data into a continuous buffer	6
2.4	Changing the way the data is stored in memory: Padding	6
2.5	Changing the way the data is stored in memory: Tetris	9
2.6	Changing the way the data is stored in memory for stencil operations	9
2.7	Other ideas	10
2.8	An analysis of real programs	10
3	The Padding for Tiling Guide	12
3.1	How do you start: things you should know	12
3.2	How do you pad a single two-dimensional array?	13
3.3	How can you access that array?	16
3.4	How do you pad for tiles with odd base coordinates?	18
3.5	How do you pad for tiles with odd sizes?	19
3.6	How do you pad for several arrays?	20
3.7	How can you access several arrays?	23
3.8	How do you handle multi-dimensional arrays?	23
3.9	How do you pad for hierarchical caches?	25
3.10	How do you pad for stencil operations?	26
3.11	How do you choose a padding algorithm?	27
4	The Odd-Padding Proofs	32
4.1	The core theorem	32
4.2	The Odd-Padding-algorithm theorem	34
4.3	The Odd-Padding-formula theorem	36
4.4	The multi-array-access-algorithm theorem	38
4.5	The multi-array-access-formula theorem	40
5	The Tetris Idea	45
5.1	Tetris basics	45
5.2	Tetris access rules	46
5.3	Tetris mapping rules	46
5.4	Tetris and hierarchical caches	47
5.5	Tetris and multiple loops	48
5.6	Tetris Copy-Buffer	50
5.7	Tetris matrix multiplication example	51
5.8	Tetris — towards an algorithm	54
6	The Experiments	56
6.1	Discussion of the graphs	56
6.2	Summary	62
7	Conclusion	63

1 Introduction

One of my supervisors told me I should write here:

- caches are good
- cache conflicts are bad
- this thesis tells you how you can avoid cache conflicts with software methods

Calvin Lin

He has probably not thought of the possibility that I could cite him. Nevertheless, these few lines get really to the point, but let me explain them in more detail.

Data stored in the cache can be accessed much faster than data stored in memory. Therefore it is possible to speed up programs which deal with large arrays by *tiling*, also known as *blocking*. The idea of tiling is to work only on small blocks or tiles which do fit into the cache even if the whole array itself does not. The loops in a program are changed to work on these small tiles instead of the whole array. When the tile fits into the cache, the data once read from memory can be reused without accesses to the slower main memory.

An example for tiling found in Lam et al. [3] is matrix multiplication. Figure 1 shows two versions of that algorithm, one (a) without tiling and one (b) with tiling. The tiled version can reuse a fraction of a line of the Z array at the k loop and a whole block of Y at the i loop no matter how big the matrix is. For the non-tiled version, in contrast, the cache must hold at least a line of Z to be able to reuse it and a whole array to reuse Y .

Tiling is reasonably easy to implement and, therefore, often realized. Unfortunately it is not simply a matter of choosing the tile small enough to fit into the cache. The problem is the way the cache maps memory addresses to cache lines. One may think the cache replaces the content of that cache line that was unused for the longest time. Wrong! Multi-way caches do this only for sets with a very small number of cache lines. Direct mapped caches do not do it at all. Instead caches choose the cache line by applying the modulo function. For memory address adr and cache size C_S the number of the cache line used is $(adr \bmod C_S)$.

For tiling this means that whether a tile can be completely stored in the cache or not is left to chance. See the figure on the right for what may happen. Therefore, tiling must be accompanied by a method which makes sure that the whole tile is stored in the cache. This accompanying method avoids cache conflicts. Please, have a look at box 3 on page 5. It explains some words and phrases which are used in this text.

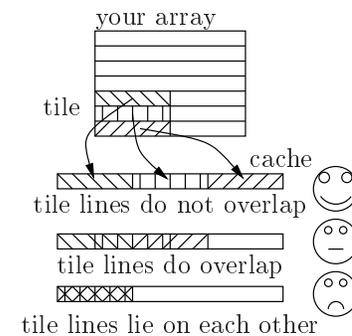
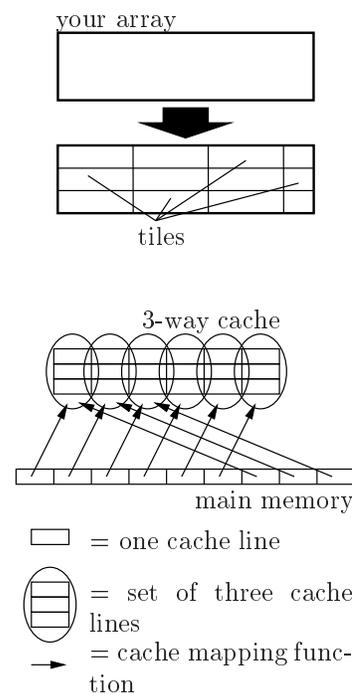
This paper describes methods for avoiding cache conflicts in the context of tiling. *The aim of this thesis is to teach you how to avoid cache conflicts.* The rest of this text is organized as follows:

Section 2 discusses the papers that I believe are most important in this area.

Section 3 describes padding in more detail — one of the most powerful methods for avoiding cache conflicts.

This part of the text is written in a “programmer’s guide” style to keep it most readable. A new algorithm (Odd-Padding) to calculate the amount of pad needed is presented, as well.

Section 4 proves the Odd-Padding algorithm correct.



Box 1: What is new?

Which contributions do I make to this area of Computer Science? The following list tries to give a very detailed answer.

- I describe in box 6 the *Brute-Force-Padding* algorithm. I have not seen this algorithm in another paper but it is likely that it is known because it is so simple.
- Section 3 is a *Programmer's Guide for Padding*. While other researchers do discuss padding — namely Panda, Nakamura, Dutt and Nicolau [5] and Rivera and Tseng [7] and others — they do not go into such a level of detail and do not put all the pieces together. In particular, the discussion of the access rules (sections 3.3 and 3.7), the discussion of odd base addresses and odd tile sizes (sections 3.4 and 3.5) and the discussion of padding for hierarchical caches (section 3.9) appear here for the first time. Moreover, Rivera and Tseng [7] can not guarantee reuse when the arrays become larger than the cache, whereas I discuss only that case in section 3.10.
- The *Odd-Padding algorithm* (see box 8) is new. This includes the proofs of the algorithm (see section 4) as well as its average and worst case behavior (see box 7).
- Everything about *Tetris* (see section 5) is completely new.

Box 2: Variable names

adr = memory address	C_S = cache size	T_L, T_H = tile length, height
A_{adr} = base address of A	CL_S = cache line size	T_x, T_y = tile coordinates
B_{adr} = base address of B	A_L = array length	UT_L = user tile length
UB_{adr} = user base adr. of B	UA_L = user array length	$B\dots$ = second cache ...

The unit of the values is usually the *cache line length*, that is, the values in the formulas are multiples of CL_S unless explicitly mentioned otherwise. All figures, formulas and programs in this paper assume row-major order memory layout of multidimensional arrays — the way C stores arrays in memory.

Section 5 describes Tetris — a new method for avoiding cache conflicts. Tetris can handle situations where padding fails.

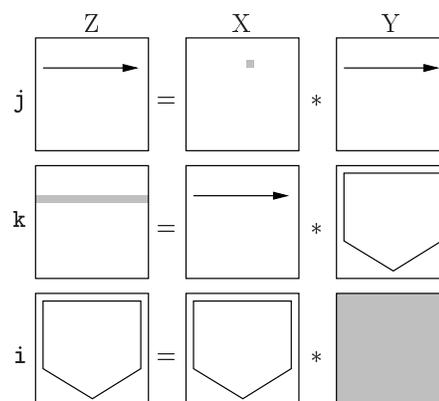
Section 6 shows some experiments which compare the different methods.

Box 1 tells you in detail which contributions I make in this thesis. Box 2 explains some variables which I use throughout the text.

```

(a) for i := 1 to N do
    for k := 1 to N do
        for j := 1 to N do
            Z[i,j] += X[i,k] * Y[k,j]

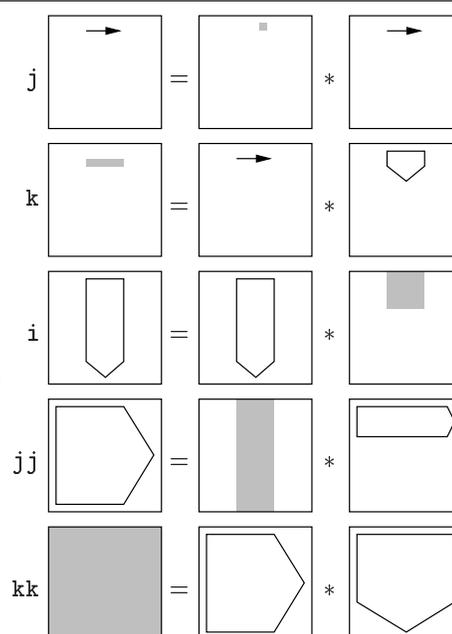
```



```

(b) for kk := 1 to N by B do
    for jj := 1 to N by B do
        for i := 1 to N do
            for k := kk to min(kk+B-1,N) do
                for j := jj to min(jj+B-1,N) do
                    Z[i,j] += X[i,k] * Y[k,j]

```



 = access to a line for the first time
 = access to an area for the first time
 = reuse of a line
 = reuse of an area

Figure 1: The original (a) and the tiled version (b) of matrix multiplication from [3]. The original must read $2N^3 + N^2$ words from memory when the arrays are so big that not even a line can be stored in the cache (worst case). The tiled program accesses only a small tile of size B in the inner loops. If B is small enough the cache can hold this data and the program need only $2N^3/B + N^2$ direct memory accesses.

Box 3: Glossary

array layout see *memory layout*

cache A small but very fast memory buffer used to store data which has been read from or should be written to the main memory. The hope is that the cache speeds up programs because accesses to it are much faster than accesses to the main memory. See section 1.

cache conflicts see *cache interference*

cache interference When accessing some data, other useful data is thrown out of the cache. The name *self-interference* means that data loaded into the cache throws useful data from the same variable out of the cache; in contrast, *cross-interference* occur when useful data from another variable is kicked out.

cache line When the cache reads or writes data from or to the main memory it always reads or writes a whole block — even if only one element of that block is accessed by the processor. These blocks are called *cache lines* and usually between 4 and 64 bytes long. Each cache use one certain size.

cache misses Accessed data is not in the cache and must be fetched from the main memory. *Capacity misses* or *compulsory misses* are caused intentionally, for example when moving from one tile to the next one. *Conflict miss* means that data is not in the cache which should be in the cache. The source of *Conflict misses* is *cache interference*.

cache thrashing The worst case *cache interference* is called *cache thrashing*. *Cache thrashing* happens when the processor consecutively access different memory locations which are mapped to the same cache line in the cache. This causes the cache to non-stop load a *cache line* from one location and throw it out immediately to replace it by a *cache line* from another location. Loops affected from *cache thrashing* are often a order of magnitude slower than unaffected loops.

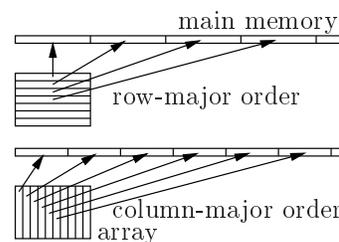
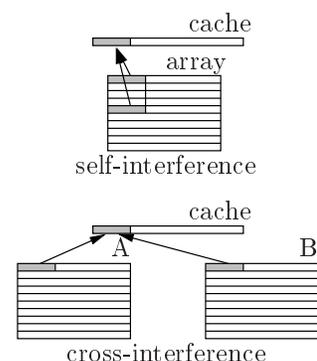
data layout see *memory layout*

memory layout The way data — usually arrays — are stored in memory. Arrays commonly stored either in *row-major order* (C) or *column-major order* (FORTRAN). See figure at the right. The *Tetris memory layout* is described in section 5.1.

ping pong see *cache thrashing*

reuse Data which is accessed is in the cache. A cache line is four times *reused* if it is accessed four times while the cache line is in the cache (after it had once been loaded).

stencil operation A certain way to access arrays. See figure 6 for an example.



2 The State of the Art

Here I discuss papers from other researchers which I believe are the major contributions to this area. Almost all papers describe an idea to avoid cache interferences. I will try to point out the strong and the weak sides of these suggestions.

2.1 A historically interesting paper

One of the first ideas to avoid cache interferences selects the tile size in such a way that there are no self-interferences. Lam, Rothberg and Wolf present an iterative algorithm in their paper [3]. Given the length of an array their algorithm tries to find the largest square tile that does not cause self-interferences. The advantage of this method is that it is not necessary to change the way the array is stored in memory (*memory layout*).

The disadvantage is an often very small tile so that a large part of the cache remains unused. In the worst case when the array length is a multiple of the cache size, as described in box 4, the tile size becomes one element i. e. the programmer must accept that very bad performance. Therefore, it is not surprising that Lam, Rothberg and Wolf finally came to the conclusion that the best results were obtained by copying the data which should be reused into a continuous buffer.

2.2 Finding the optimal tile size

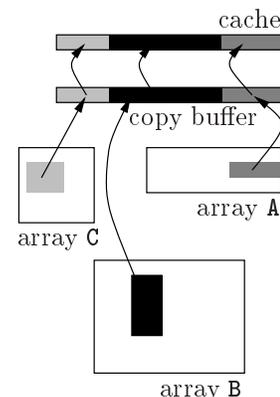
The method of Lam et al. [3] often fails to utilize large parts of the cache because it searches only for square tiles. Coleman and McKinley present an iterative algorithm [2] which searches for the biggest rectangular tile. Figures 2, 3 and 4 show how the length of an array influences the tile size. Rectangular tiles can use most or all of the cache without changing the way the data is stored in memory.

A program employing this algorithm must accept any given tile shape, which may be very small and long, or very flat. Moreover, as with the algorithm from Lam et al., Coleman and McKinley's method fails to handle the worst case where the array length is a multiple of the cache size.

2.3 Copying the data into a continuous buffer

Temam, Granston and Jalby [8] suggest a way to copy the data as proposed earlier by Lam et al. [3]. The idea is to copy those data which are often accessed (reused) into a buffer. Since the buffer is continuous and not bigger than the cache, there is no cache interference possible when accessing it. Moreover, this buffer can store any data even from different variables. This way, virtually all cache interference can be avoided.

Note, however, that the copy operation itself may cause cache conflicts (interferences). Furthermore, data which is read in a loop must be copied before the loop. Data that is written or changed must be copied back after the loop. All of this slows down the program. Therefore, Temam, Granston and Jalby's algorithm analyzes the cost of copying very carefully and decides whether it is worth doing it or not in each given situation.



2.4 Changing the way the data is stored in memory: Padding

Instead of simply accepting the way the data is stored in memory *padding* changes the data layout by adding unused dummy elements. There are two kinds of padding:

intra-variable padding makes the lines of an array longer by adding a certain amount of dummy elements to the end of each line. Done correctly, intra-variable padding avoids self-interference and can guarantee

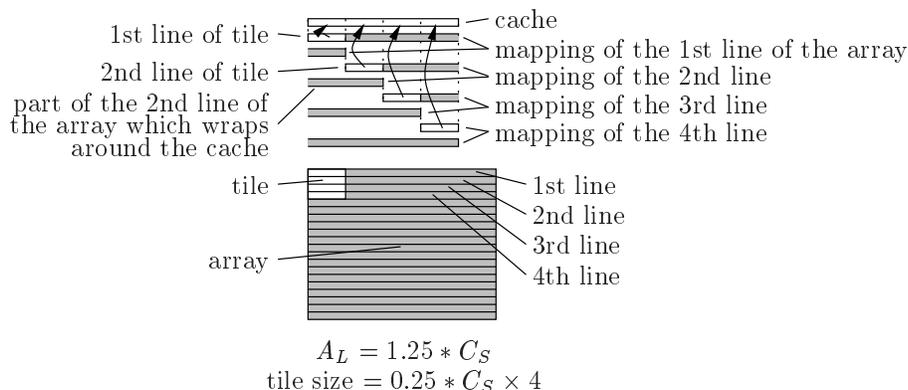


Figure 2: An array with row-major memory layout — which is the way C stores arrays — is mapped into the cache. A line of the array which reaches the end of the cache wraps around and continues from the beginning of the cache. To distinguish between the end of one line and the beginning of the next one, there is a small vertical gap between the individual lines in the figure. A program which accesses only the white tile part of the array uses the whole cache without causing any cache interferences. Note: the maximal length of a tile T_L is $(A_L \bmod C_S)$ but a shorter tile length is often preferable. For example if $(A_L \bmod C_S)$ is greater than $C_S/2$ the tile with maximal T_L would consist of only one single line.

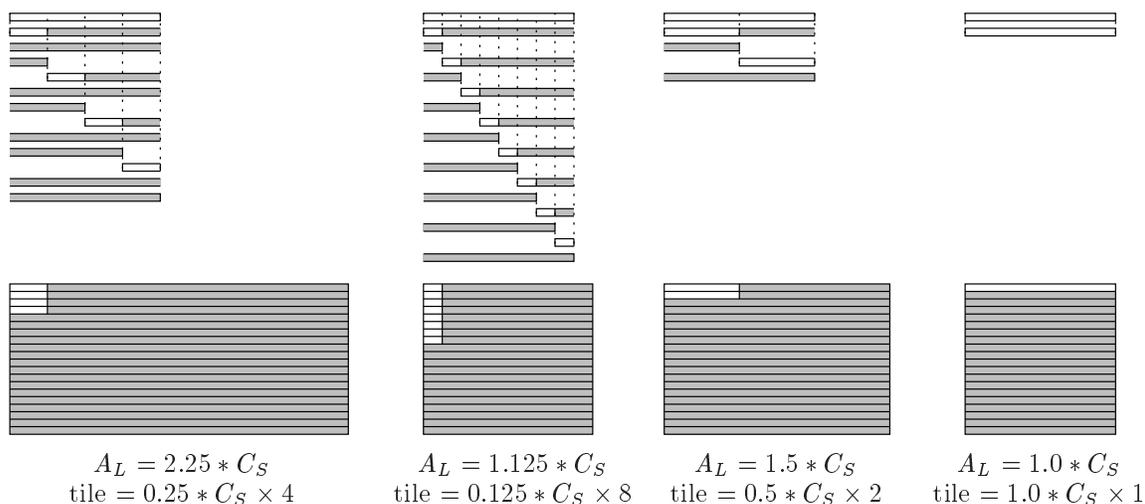
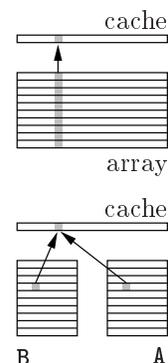


Figure 3: The length of the array influences the length of a tile. Note how the array length — the number written below the arrays — results in different tile sizes. Compare this picture also to figure 2. Different methods to avoid self-interferences have been suggested. Methods which do not change the data layout like those from Lam et al. [3] and Coleman and McKinley [2] may return very small and degenerate tiles. Methods which do change the length of the array like padding from Panda et al. [5] can make any intended tile — no matter what length and height — fit into the cache, provided the cache is large enough to hold a tile of that size.

Box 4: The worst case data layout

How much a program actually suffers from the layout of its data depends on many factors. Among these factors are the way the program accesses the data, the size of the arrays involved, the size of the tiles if it uses tiling, the start address of one array in relation to other arrays and hardware features like stream buffers. But experience has shown that two situations heavily slow down almost all programs. These situations are:

- The length of a line of an array is an exact multiple of the cache size — assuming row-major order — that is, $A_L = i * C_S$ where $i \in \mathbb{N}$. This causes heavy self-interference because all elements of a column of that array are mapped to the very same cache line — or small set of cache lines if the cache is a multi-way cache.
- The distance between the base address of two similar arrays is an exact multiple of the cache size i. e. $|A_{adr} - B_{adr}| = i * C_S$ where $i \in \mathbb{N}$. This can cause heavy cross-interference because elements of arrays **A** and **B** with the same coordinates are mapped to the same cache line — or small set of cache lines if the cache is a multi-way cache.



These situations can be considered to be the worst case because they can slow down programs by an order of magnitude compared with programs which suffer only from a “usual” amount of cache interferences. Therefore this bad cases should, wherever possible, be avoided. Methods to avoid them are copying [8] and padding [6]. Here padding uses only a very small pad; just enough to avoid the worst case without a guarantee for reuse of specific data.

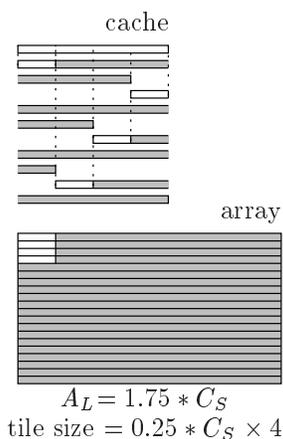


Figure 4: Here is an example where the chosen tile length is different from the maximal possible tile length. The maximal possible length T_L would be $(A_L \bmod C_S) = 0.75 * C_S$. The first line of this tile would use 75% of the cache, so that there won't be enough space in the cache to store a complete second line. The result would be a degenerated tile which consists of only one line and uses only three-fourth of the cache. The chosen tile length is $T_L = 0.25 * C_S$ and the tile height T_H is four lines. Therefore this shorter tile uses the whole cache.

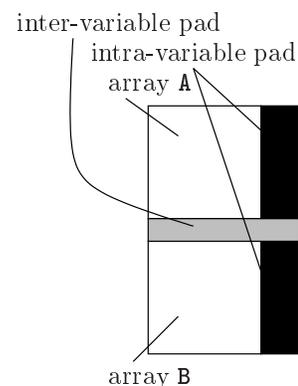
For convenience, the arrays in my figures start at the beginning of the cache. This is, of course, unlikely to happen in practise. The start or base address of the array may fall into any position of the cache. A tile line may even be cut off at the end of the cache and wrap around to continue at the beginning of it, so that a part of the tile line is at the end of the cache and another part at the beginning of it. However, the array base address is irrelevant for self-interference as long as it starts at a cache line border.

reusability of data. That is: data which can be reused will not be accidentally thrown out of the cache.

inter-variable padding changes the base or start address of a variable by adding dummy elements before that variable. Inter-variable padding changes the distance between several variables or arrays in memory and, therefore, can avoid or reduce cross-interference if done in the right way.

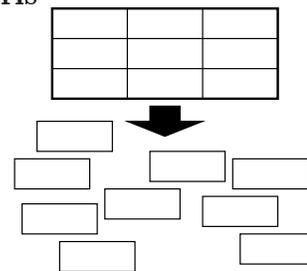
Despite its obvious disadvantage — wasted memory — padding is one of the most powerful techniques to avoid cache interference. Rivera and Tseng [6] present a padding technique which only avoids the worst cases, as described in box 4. Their method needs only very little “wasted memory” for padding because they do not intend to guarantee the reuseability of any data i. e. they do not care whether or not a whole tile can be stored in the cache without causing cache interference.

A much more powerful method has been presented by Panda, Nakamura, Dutt and Nicolau [5]. Their algorithm searches iteratively for the smallest intra-variable pad, so that a given tile can be accessed without self-interference. They also describe (very shortly) how to extend their method to access several arrays in the same loop by using inter-variable padding. These arrays must have the same size and must be accessed in a fairly similar way. This is a clear shortcoming of padding. Furthermore, this method uses much more memory than Rivera and Tseng’s technique. Section 3 discusses padding in more detail.



2.5 Changing the way the data is stored in memory: Tetris

Observe that an array which is small enough to fit completely into the cache does not cause self-interference. Therefore, the main idea is to split a big array into many small pieces. Tetris distributes the small pieces in memory in such a way that several arrays can be accessed in the same loop without causing cross-interference. This is done by dividing the memory in cache size chunks. The small pieces are then distributed into these chunks so that each chunk has a piece from each array, and the pieces always occupy the same part of the chunks. See figure 5 for an example.

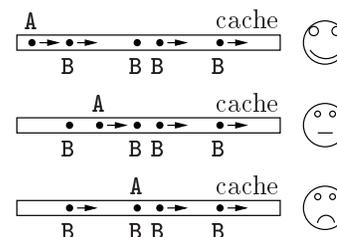


The advantage is that Tetris allows accessing different arrays with different sizes, different tile sizes and different access patterns. The drawback is the relative difficulty in implementing Tetris. I recommend using padding before Tetris. Another shortcoming is its wasted memory, similar to that of padding. Tetris is described in section 5.

2.6 Changing the way the data is stored in memory for stencil operations

Strictly speaking, the idea described here is padding again, with an emphasis on changing the base address of the involved arrays (inter-variable padding). The difference is the way the data is accessed and reused. With tiling, a tile is selected and all data read once from memory are stored in the cache until the next tile is processed. In a stencil operation a certain access pattern (the stencil) sweeps over the whole array as shown in figure 6. The intention is to load an element of the array into the cache when the stencil first hits it and to hold it in the cache until the stencil has been moved entirely over that element.

Rivera and Tseng [7] — in an extension of their earlier work [6] — present a method which arranges several arrays in such a way that there will be as few conflicts as possible. These arrays must be accessed by stencil operations. The basic idea is to change the base addresses of the arrays in such a way that the accesses to one array are mapped into the cache without falling between accesses of some other array. Their method does not use too much dummy memory because they do intra-variable padding only for the worst case. Moreover, they fail to guarantee reuse when the length of the arrays



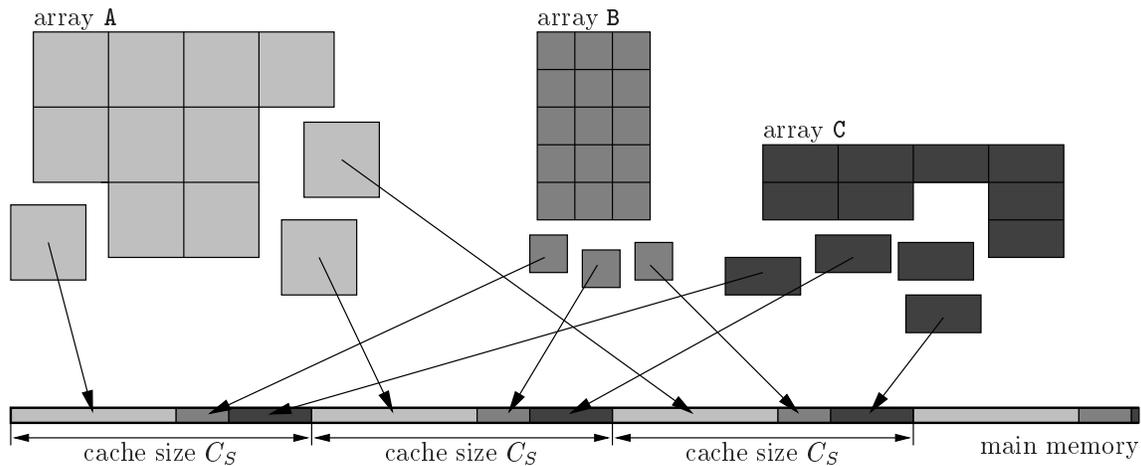


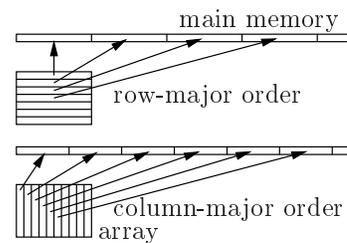
Figure 5: The idea of Tetris is to split the arrays into small pieces and to uniformly distribute the pieces into memory chunks which are exactly as large as the cache.

gets large. Stencil operations can be combined with tiling and padding. Hence I describe them in more detail in section 3.

2.7 Other ideas

There are many other ideas which change loops or the way the data is stored in memory (data layout). They usually neither guarantee that data once read is stored in the cache for future access (reuse) nor that there will be fewer cache conflicts. Instead, these ideas are based more on the believe that, in the general case, they will improve performance somewhat.

To represent them all I picked the paper by Cierniak and Li [1]. Their idea is to bring data that is accessed in one iteration of the inner loop as close together as possible. They change both the way the data is stored in memory and the loop iterations to achieve this goal. To store an array they choose between row-major order or column-major order. Their method can handle several loops and arrays. Moreover, they show an example where changing the loops and the data layout together does give better results than either of these alone. However, they do not consider whether this leads to reuse or not.



2.8 An analysis of real programs

McKinley and Temam [4] analysed the cache efficiency of the Perfect Benchmarks. Here are some of their results:

- The prevalent kind of reuse in the entire program is accessing the same element of a cache line several times. In contrast the commonly held assumption says: “The reuse of other elements of a cache line is dominant.”
- Loops have a more balanced reuse than whole programs. They usually access the same element again, as well as other elements of a once loaded cache line.
- When a loop causes the processor to wait for a memory access, the cause is usually a cache interference. This is in opposition to the common assumption that a loop spends most of the time waiting for capacity misses.

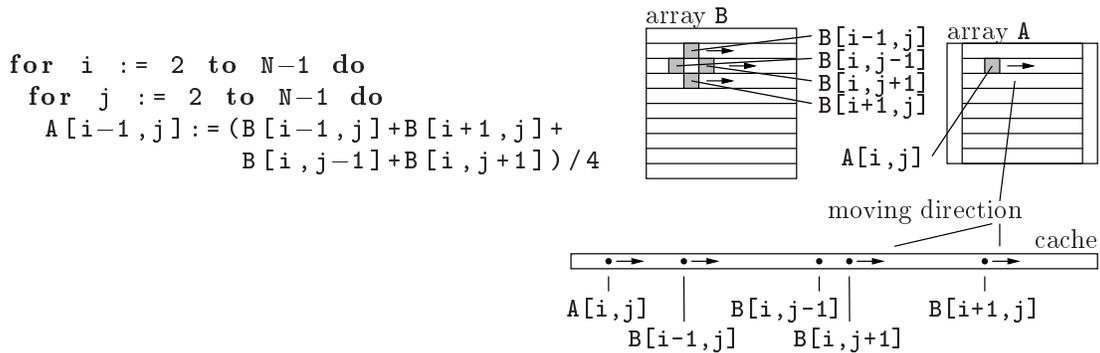


Figure 6: This is a stencil operation as discussed by Rivera and Tseng in [7]. The access to array **B** is the stencil which is moved over the whole array. The trick is to choose the base addresses of the arrays in such a way that the **A**-access does not fall between the **B**-accesses. This way, the data once loaded from memory by $B[i+1, j]$ can be reused three times without reading it from memory again.

Note that the array **A** must have the same length as **B** although the frame of **A** remains unused. If this would not be the case then the relative position of the **A**-access in the cache would change (minus two elements) when moving to the next line of the arrays.

- Waiting on capacity misses, occurs mainly when the processor proceeds from one loop to the next.
- The largest number of successful accesses to the cache — that is: the cache holds already the data — happens within loops.
- The cache fails most often when the execution moves from one loop to the next.
- In many cases only one word is accessed from a cache line and the rest remains unused. This causes bad cache utilization.
- Most accesses to memory have a very regular pattern.

Moreover, they found that the tile size chosen by tile size selection algorithms, e.g. Lam et al. [3], is too small to make tiling effective.

3 The Padding for Tiling Guide

Tiling or blocking is a well known technique to increase the speed of programs which work with arrays. Tiled algorithms make better use of the cache and thereby decrease the time a program needs to access its data. But tiling alone is not enough. It must be accompanied by a method which ensures that the data actually stay in the cache.

Here, I explain how you pad your arrays so that your tiled loops do not cause cache interference. I describe what can be done with padding and where padding fails. *This section is written as a programmer's guide.* I assume you have a program which works on arrays and you want to speed up this program by making better use of your cache. I further assume you have already tiled your loops or you are going to do so. This is not a guide for tiling! You need to know how to tile a program (see figure 1 for a tiling example).

3.1 How do you start: things you should know

Why do it? to make most efficient use of your time

What to do? know some tricks and tips

This is a list of general hints. These hints help you to speed up and to debug your program. They are not directly related to padding but you should consider them before you do padding.

- Analyze your program before you start to optimize it. The critical point here is to really use an analysis tool. These tools tell you how much time your program spends in each statement. If you do not use such a tool you will find yourself spending much time to optimize code which does not account for much execution time.
- It is always a good idea to keep the elements of an array accessed in a loop as close together as possible. This way it is more likely that data is still or already in the cache when you access it.
- If you can combine several loops to a single one and the loops access the same arrays, do so. This makes it more likely that data is already in the cache when accessed. This is often as effective as padding and tiling.
- When you can replace an array by a smaller one or, even, by a simple variable, do so. You reduce the number of memory accesses and avoid spoiling the cache. This is often more effective than padding and tiling.
- Be sceptical about optimizations done by your compiler. Loop optimization and tiling are believed to be very well understood and, consequently, implemented in most compilers. For some reason or another compilers tend to “optimize” those loops you have already tiled and optimized by hand. This often degenerates performance.

Let your compiler generate an assembler listing and print it on paper. No, I do not want you to understand details! Just draw an arrow from each jump or branch instruction to its destination label. This way, you should fast find the loops produced by your compiler.

- Know your hardware. When you start to optimize for your cache, you made a decision to optimize for your hardware. So know the game pieces you are playing with. It is not enough just to know how large your cache is.
- If your hardware can read and write without going through the cache, use that facility for data which is not accessed again. If you do so, you avoid spoiling the cache with data which can not be reused.
- When you optimize for a cache, optimize for the smallest top-level cache, first. It is the fastest and gives you the best speed.

Box 5: The process of padding

Padding avoids cache conflicts by making your array longer and change it's base address. First, there are two things you need to know (see also box 8 on page 20):

- UA_L : the user array length — the minimum length of your array.
- UT_L : the user tile length — the minimum length of a tile.

Then padding tells you:

- A_L : the real array length — that is UA_L plus the required intra-variable pad.
- T_L and T_H : the real tile length and height — T_L is UT_L plus zero or more unused elements.
- A_{adr} and B_{adr} : the base addresses of you arrays.

Figures 2, 3 and 4 may create the impression that your chosen array length UA_L implies a tile size T_L . The algorithms from Lam, Rothberg and Wolf [3] and Coleman and McKinley [2] actually go this way but the resulting tile size is often very degenerated.

Padding works exactly the other way around: once you have chosen your user tile length UT_L it tells you which array length A_L you should use (where $A_L \geq UA_L$). That is: padding chooses you array length so that you get your intended tile size.

Depending on which padding algorithm you use, the real tile length T_L is subjected to some restrictions. Therefore, you may need to change your intended tile length UT_L to T_L so that T_L meets the restrictions. In the text, for didactic reasons I first tell you how to calculate the array length A_L given T_L in section 3.2. Later, I describe how you find T_L given UT_L (sections 3.4 and 3.5). You need to change the base address of your arrays only if you work with several arrays. This is described in section 3.6.

- You do not need to pad arrays which are so small that they fit completely into the cache.
- Think in cache lines. Your cache works with whole cache lines not with single array elements.
- Note that reads *and* writes slow down your program. Most people think only about read accesses when optimizing but write accesses can cause as much trouble as read accesses.
- It is exceedingly hard to debug cache behavior because you cannot see into your cache. If you do not use an analyze program you will most likely not even recognize when your program's speed is hurt by bad cache usage. A trick is to convert the addresses of the memory accesses into cache line numbers ($adr \bmod C_S$) and to print them.

In the rest of this section I assume you have a direct mapped cache. If you have a multi-way cache, divide its size by the number of ways and use that value as cache size C_S ; e. g. a 3-way cache with 1536 cache lines gives $C_S = 512$. If you have several caches use the size of the smallest one. If not stated differently I assume all lengths are in cache lines — that is: the size of the cache and the length of an array or tile is given in cache lines. If your computer uses cache lines with different lengths, use the longest one. Heights are given in lines and never in cache lines.

I have no experience with virtual mapped caches. Therefore, the methods presented in this paper may or may not work for virtual mapped caches. You need to take care of what ever is necessary to make padding work with that kind of caches; consider especially the virtual page length and the table look aside buffer.

3.2 How do you pad a single two-dimensional array?

Why do it? to avoid self-interference

What to do? align your array base address at a cache line border and extend the line length of the array

There are two things you need to do:

1. make your array start at a cache line border
2. extend the length of all lines of your array

In sections 3.4 and 3.5 you learn why you need to align the array at a cache line border. It is not always necessary to start an array at a cache line border (see box 6 for a counter example). On the other hand, aligning an array at a cache line border costs you only a few byte and it does not do any harm if you do it unnecessarily. Therefore, it is wise to start all arrays at a cache line border.

If a cache line holds CL_S elements of your array and the base address for your array is adr (in array elements) then you can calculate the next higher address which is aligned at a cache line border, using this function:

$$A_{adr}(adr, CL_S) = adr + (CL_S - (adr \bmod CL_S)) \bmod CL_S$$

Note: the outer modulo-function is not surplus (consider the case where $(adr \bmod CL_S)$ is 0). You need to allocate $C_S - 1$ elements more for your array, to have enough space for the alignment.

Now, you must extend the length of the lines of your array to avoid self-interference (see figure 7). I assume row-major memory layout, if your array is stored in column-major order then you need to extend the height of the columns. This kind of padding is called intra-variable padding. At the moment, there are two algorithms to select a pad length. One is described in Panda et al. [5], and the other is my invention: Odd-Padding. Here I describe the Odd-Padding algorithm.

You need to fulfill the following requirements¹:

- The size of your cache² is a power of two i. e. $C_S \in \{1, 2, 4, 8, 16, 32, 64, \dots\}$.
- Your cache line size is a power of two i. e. $CL_S \in \{1, 2, 4, 8, 16, 32, 64, \dots\}$.
- You have chosen a tile length T_L which is also a power of two and is given in multiples of cache lines.
- You know that your array must be at least UA_L cache lines long.

Then the height of your tile — in lines — is:

$$T_H = \frac{C_S}{T_L}$$

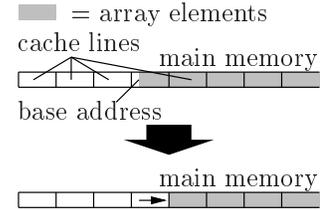
You do not need to use all lines. T_H is just the upper maximum. The length of your array must be an odd multiple of the tile length i. e. $A_L = iT_L$ where $i \in \{1, 3, 5, 7, 9, 11, 13, \dots\}$. That A_L must be an odd multiple of T_L is the core of the Odd-Padding algorithm and based on a property of the modulo-function (see section 4.1 for a proof). You can use this formula to calculate such a length for your array:

$$A_L(UA_L, T_L) = UA_L + (2T_L - ((UA_L + T_L) \bmod (2T_L))) \bmod (2T_L)$$

The new array length A_L may be up to $2T_L CL_S - 1$ elements longer than the initial array length UA_L . (Recall: T_L is the tile size in cache lines, CL_S is the cache line size in elements. Therefore, $T_L CL_S$ is the tile size in elements.) Confused? How about an example?

¹If the first two requirements do not meet your situation, you need to look at the theorem of section 4.2 to figure out how to handle your case. Requirement three is weakened in sections 3.4 and 3.5.

²I assume a direct mapped cache. See the end of section 3.1 for how to calculate C_S if you have a multi-way cache.



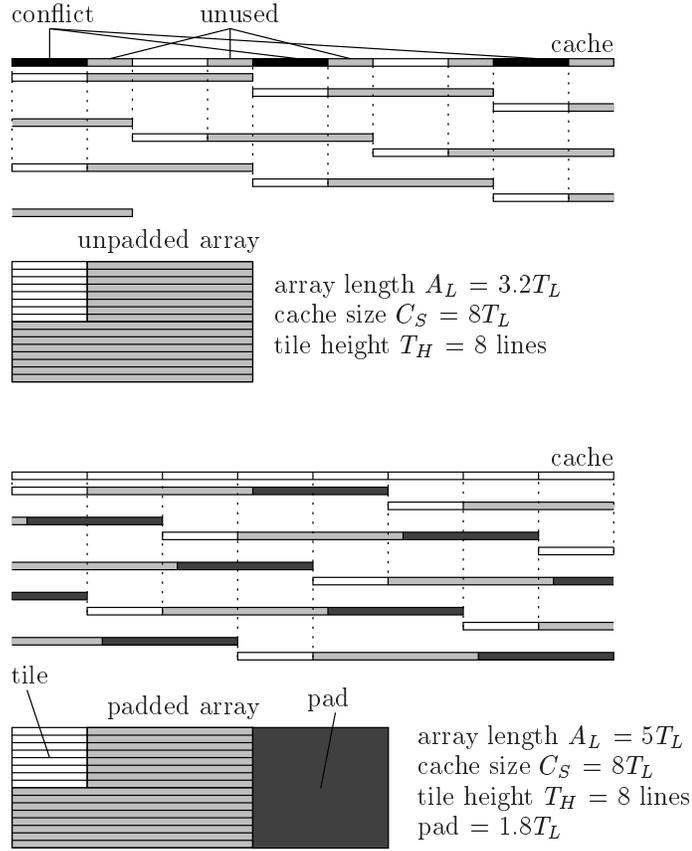


Figure 7: Here is an example of the effect of intra-variable padding. In the top figure, the array is not padded. Only two of its eight tile lines are mapped into the cache without causing cache interference. A large part of the cache remains unused.

In the bottom figure, the same array has been padded to the next odd multiple of the tile size. All tile lines are mapped into the cache without any interference and the whole cache is used.

Example 1: Padding for a two dimensional array

Assume the following situation:

- Your cache line size CL_S is 8 elements.
- Your cache size C_S is 128 cache lines (that is 1024 elements).
- You tile length T_L is 4 cache lines.
- You want to use an array with a length of $UA_L = 32$ cache lines and a height of 256 lines.

The padded length of your array must be an odd multiple of the tile length:

$$A_L(32, 4) = 32 + (8 - ((32 + 4) \bmod 8)) \bmod 8 = 36$$

You need to create an array with 256 lines, each line has 36 cache lines, that is 288 elements per line. Note that $A_L/T_L = 36/4 = 9$ is odd as required.

It remains to allocate that array and to align its base address at a cache line border. The array has $256A_LCL_S = 73728$ elements but you need to add $CL_S - 1 = 7$ elements for the aligning purpose, so that you allocate 73735 elements.

```
adr = alloc( 73735 )
```

(Note: I assume here that the `alloc`-function returns a physical address. If the virtual page length is a multiple of the cache size then the virtual address can be used, too.) Assume the operating system returns you address $adr = 12345$. You need to use the start of the next cache line as the base address of your array:

$$A_{adr}(12345, 8) = 12345 + (8 - (12345 \bmod 8)) \bmod 8 = 12352$$

$A_{adr} = 12352$ is the base address for your array. The size of your array is 256 lines with 288 elements. The tile length is $T_L CL_S = 4 * 8 = 32$ elements and your tile height is $C_S / T_L = 128 / 4 = 32$ lines.

This array, for example, could be the **Y** array of the tiled matrix multiplication from figure 1 if the tile size **B** is 32. Note: the cross-interference with arrays **X** and **Z** would not be avoided, only the self-interference of array **Y** would be removed.

For padding you must change these parts of your program: the allocation of your array as described above, all accesses to that array to take into account the new array length and the new array base address, and the loops — where possible — so that they work on your array in tiles which are $T_L CL_S$ elements long and T_H lines high.

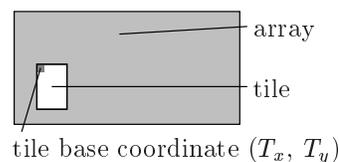
From the text of this section you may have correctly concluded that you can save memory by choosing the tile length T_L small. Note however that your hardware may benefit from long tiles, especially if you have stream buffers. My experience shows that short tile length degenerate program speed on computers with stream buffers.

3.3 How can you access that array?

Why do it? to get the optimum out of padding and tiling

What to do? know the rules about accessing your array

Let me first define what the tile base coordinate is. It is simple: the tile base coordinate (T_x, T_y) is the coordinate of the leftmost and topmost array element in a tile.

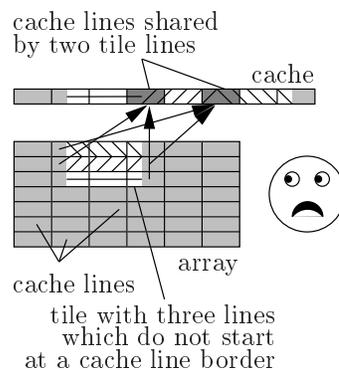


The basics of tiling are easy: once you access an element of a tile, the cache line which holds that element is loaded into the cache. That cache line stays in the cache until the tile is moved to some other place of the array. Therefore, you should access a once loaded cache line as often as you can before you move the tile to another position. This also implies the converse: if you access cache lines just once, tiling and padding will not improve the speed of your program.

An example for good tiling is the access to the **Y** array in the tiled version of the matrix multiplication algorithm from figure 1. In the **i** loop, all elements of the same tile of the **Y** array are accessed over and over again. Once for each different value of **i**. The elements of a tile are loaded from memory when first accessed and afterwards are always read directly from the much faster cache.

Unfortunately, things are not always so easy. Here are some rules which you need to pay attention to:

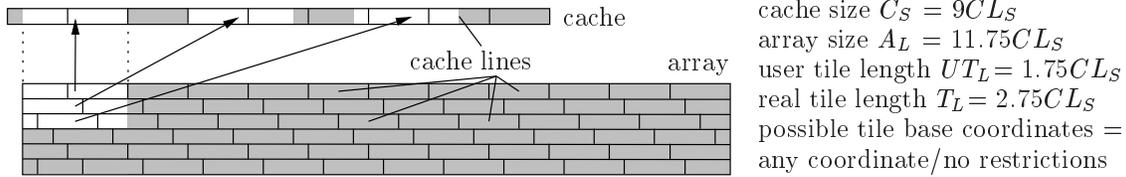
- All your memory accesses must go to that tile and must stay in the boundary of it. When you access another memory location, a cache line belonging to the tile is kicked out of the cache and must be reloaded from the slowly main memory when accessed later, no matter whether you access the same array which you have tiled, another array or even a non-array variable. In section 3.6 I tell you how to access several arrays without causing interferences.



Box 6: Brute-Force-Padding

Brute-Force-Padding is another method for intra-variable padding. The disadvantage of Brute-Force-Padding is that you need to use a pad up to the length of your cache minus two times the tile length. This is too much pad for Brute-Force-Padding being useful in practise. But it still may be used for theoretical considerations or in very rare programming situations.

The advantage of Brute-Force-Padding is that it can deal with odd situations. Cache size, array length and tile size must not even be multiples of a cache line size. Brute-Force-Padding chooses the array length to be either $A_L = iC_S + T_L$ where $i \in \mathbb{N}$ or $A_L = iC_S - T_L$ where $i \in \mathbb{N} \setminus \{0\}$. Consider this all-odd-example:



The length of this array is a multiple of the cache size plus the length of the tile i.e. $A_L = C_S + T_L = 9CL_S + 2.75CL_S$. The effect is that tile line 1 is mapped precisely behind tile line 0. Tile line 2 behind tile line 1 and so on...

Furthermore, the real tile length of the example array is the user tile length plus a cache line size i.e. $T_L = UT_L + CL_S = 1.75CL_S + 1CL_S$. Therefore, there will always be a cache line border between the end of one line of the tile and the next one. This allows the tile base x-coordinate to start at any arbitrary position and it permits the array to start at any arbitrary position in a cache line.

Of course, you can create such a tile length and array length with padding. Here is how it works: (Let UA_L be the array length you want and UT_L the desired tile length, all values are given in array elements)

Consider these questions:

1. Does your array start at a cache line border?
2. Is your desired array length UA_L a multiple of the cache line size CL_S ?
3. Is your desired tile length UT_L a multiple of the cache line size?
4. Do you want to move the tile base x-coordinate only to multiples of the cache line size?

If you can answer all of the above questions with “yes” then your tile size is $T_L = UT_L$. If you answer some questions with “no”, or if you are unsure, your must use tile size $T_L = UT_L + CL_S$. The maximal height of the tile is $T_H = \lfloor C_S/T_L \rfloor$. The length of your array must be:

$$A_L = \begin{cases} \text{if } t < UA_L \bmod C_S \text{ then } UA_L - (UA_L \bmod C_S) + t + C_S \\ \text{else } UA_L - (UA_L \bmod C_S) + t \end{cases}$$

where $t = \begin{cases} \text{if } T_L < UA_L \bmod C_S \leq C_S - T_L \text{ then } C_S - T_L \\ \text{else } T_L \end{cases}$

Note: theorem (42) is an instance of this formula and is proven in section 4.5.

The draw back of Brute-Force-Padding is your need to add up to the length of your cache minus two times the tile length and minus one element to every array line $A_L - UA_L \leq C_S - 2T_L - 1$.

Box 7: How expensive is padding?

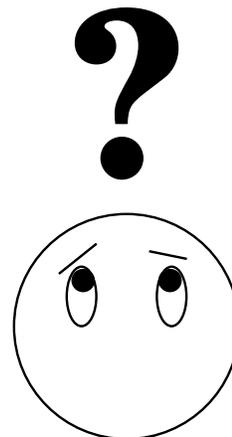
For padding, you pay mainly with storage space. Here is how much memory you lose in the worst case (all variables are given in elements):

cache line alignment	all methods	$CL_S - 1$
inter-variable padding	all methods	$C_S - 1$
intra-variable padding	Brute-Force-Padding	$(C_S - 2T_L) - 1$
	Odd-Padding	$2T_L - 1$
	panda et al.'s DAT	$(C_S - 2T_L) - 1$

Panda et al. [5] do not write how much pad their DAT algorithm uses in the worst case. From experiments I know that it uses much more pad in the worst case than the Odd-Padding algorithm (see also section 3.11). Since the Brute-Force-Padding algorithm would always find a solution for the parameters used in the DAT algorithm, the DAT algorithm will at least find the same solution. Therefore, I use the worst case of the Brute-Force-Padding algorithm as the worst case of the DAT algorithm.

Another important question is: how much faster will your program be? Unfortunately it is exceedingly hard to answer that question. It mainly depends on how much your program suffers from cache interference before you do padding, how your hardware works and how good your implementation of padding is — debugging cache problems is very difficult.

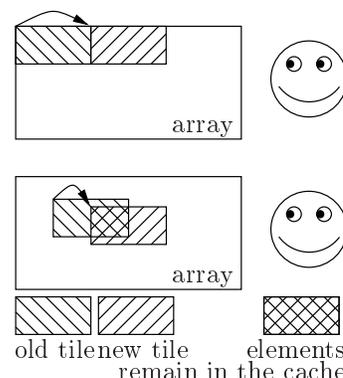
It would be helpful to know how much other programs win in average but padding is relatively new and I do not know any reliable data, yet. My best guess is that a padded loop will make *at least* 5% in most cases where padding makes sense and over 50% in very rare cases.



- The positions where you can move the tile base coordinate to are restricted. To be precise: you can move the tile y-coordinate T_y to any line you want — as long as it stays in the array, of course. If you move T_y to a far away corner of the universe, these nasty space-time-anomalies will occur and you and your computer may be swallowed by black hole. The problem is the T_x coordinate of your tile. It can only be moved to a cache line border. Since you have aligned the array base address at a cache line border — now you see why you need to do so — T_x must be a multiple of CL_S . If the base x-coordinate of a tile does not fall on a cache line border, the start of one tile line and the end of another one will fall into the same cache line and give rise to cache-interference. In sections 3.4 and 3.5 I describe how to get rid of this restriction.

Note: the frequent case where the tile is moved horizontally in such a way that the new tile starts where the old one ends is allowed because the tile length is a multiple of the cache line size.

When you move the tile base coordinate to a new position, so that the old tile overlaps partly the new one, the elements in the overlapping region will stay in the cache and do not need to be reloaded.



3.4 How do you pad for tiles with odd base coordinates?

Why do it? to use tiles which are not aligned at cache line borders

What to do? enlarge the tile by adding unused elements

What if you need to move the tile base coordinate to different positions which are not aligned at a cache line border? Assume you wish to use a tile length UT_L which is a multiple of the cache line size CL_S and not necessarily a power-of-two. The case where your intended tile length is not even a multiple of CL_S is handled in section 3.5.

You need to make the real tile length T_L some elements longer than UT_L before you calculate the array length, using the method given in section 3.2. That is: you pad the tile length with dummy elements which you do not really use. You must make T_L at least one cache line longer than your intended tile length. That is, if UT_L and T_L are given in multiples of CL_S :

$$T_L = UT_L + 1$$

You may add more than one multiple of CL_S . This is especially helpful when you use the Odd-Padding algorithm and you need to make T_L a power of two. Note, however, that the Odd-Padding algorithm may not necessarily utilize the cache best when you do so (see the counter example on the right). You calculate the array length A_L with this new, longer T_L using the formula of section 3.2.

It is important to understand that you are not allowed to use — that is to access — these additional elements. These unused elements make sure that there is a cache line border between any tile line. Moreover, if you extend UT_L by at least one cache line size, you do not need to align your array at a cache line border because your tile base coordinates may then start at any arbitrary position.

3.5 How do you pad for tiles with odd sizes?

Why do it? to use tiles which are not a multiple of the cache line size

What to do? enlarge the tile by adding unused elements

My reason for including this section is to prevent people from making a simple mistake. When you have a user tile length UT_L which is not a multiple of the cache line size CL_S you *cannot* always choose the next multiple of the cache line size as your tile length T_L . Here, you can learn why you cannot do it and how you can find a good tile length.

In general what you need to do is: extend your intended tile length UT_L to the next multiple of CL_S and add another CL_S as pad. For example let UT_L be 22 elements and $CL_S = 8$ elements. Extending UT_L to the next multiple of CL_S gives 24 and further adding one CL_S on it results in a real tile length of $T_L = 4$ cache lines or 32 elements. As formula this is (UT_L and CL_S are given in elements and T_L is given in cache lines):

$$T_L(UT_L, CL_S) = (UT_L + (CL_S - (UT_L \bmod CL_S)) \bmod CL_S) / CL_S + 1$$

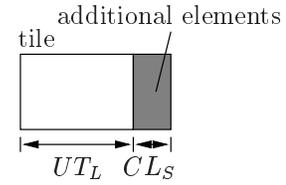
As in section 3.4 you are not allowed to access these additional elements and you can add more multiples of CL_S to T_L if you want to. You can move the tile to any arbitrary base coordinate and, hence, do not need to align the array at a cache line border.

There is a special case where you do not need to add the additional cache line size. This case occurs when

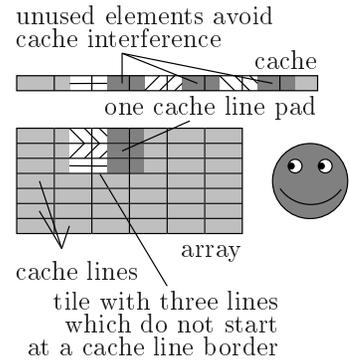
$$\gcd(UT_L \bmod CL_S, CL_S) = (UT_L \bmod CL_S)$$

and you do not move the tile base coordinate to any other position than:

$$(i(UT_L \bmod CL_S), j) \text{ where } i, j \in \mathbb{N}$$



Odd-padding does not make best use of the cache when UT_L is not a power of two. Assume:
 $UT_L = 5, CL_S = 32$
 odd-padding:
 $T_L = 8, T_H = 4$
 brute force padding (see box 6):
 $T_L = 5, T_H = 6$
 Brute force padding can utilize two tile lines more in this case.



Box 8: Padding — a quick overview

UT_L, UA_L you know your array length and your tile length
 ↓
 T_L, T_H from your intended tile length you calculate the necessary real tile length T_L and the tile height (sections 3.4 and 3.5)
 ↓
 A_L knowing T_L and UA_L you calculate the padded array length (section 3.2)
 ↓
 A_{adr}, B_{adr} you allocate your arrays and align the first at a cache line border (section 3.2) (if necessary) and then calculate the base address of all further arrays (if any) relative to the address of the first one (section 3.6)

Note that your array must also start at such a position. If this case meets your situation, you can use this formula to calculate your tile size:

$$T_L(UT_L, CL_S) = (UT_L + (CL_S - (UT_L \bmod CL_S)) \bmod CL_S) / CL_S$$

Let me first explain this formulae and then why a smaller tile size is enough.

Example 2: When can you use smaller tiles?

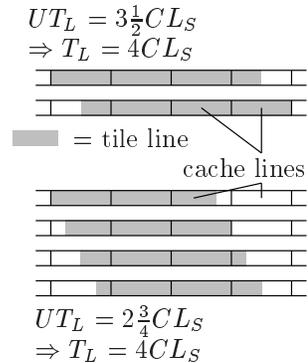
Write UT_L as fraction — the way children do it — if there is a one on top of the fraction, after shortening, then you can use the later formula for T_L . Here are some examples:

There must be a one

UA_L	$3.5CL_S$	$8.25CL_S$	$5.125CL_S$	$3.75CL_S$	$5.875CL_S$	$2.375CL_S$
as fraction	$3\frac{1}{2}CL_S$	$8\frac{1}{4}CL_S$	$5\frac{1}{8}CL_S$	$3\frac{3}{4}CL_S$	$5\frac{7}{8}CL_S$	$2\frac{3}{8}CL_S$
possible base x-coordinate	$\frac{1}{2}iCL_S$	$\frac{1}{4}iCL_S$	$\frac{1}{8}iCL_S$			

Now, let me explain why you can use a shorter tile in some cases. A tile line with odd length uses one or two cache lines only partially, the leftmost or the rightmost or both of them. Tiles with lines which use always only one cache line partially can use shorter real tile length. Tiles which use sometimes both the leftmost and the rightmost cache line partially need a cache line more (see the example at the right). The possible x-coordinates play also a role. If the $UT_L = 3.5CL_S$ tile could be moved in $0.25CL_S$ steps, it would also sometimes use both cache lines at its border partially and, therefore, would need a real tile size of $(T_L = 5)^3$.

Note that I did not prove any of my claims of sections 3.4 and 3.5.



3.6 How do you pad for several arrays?

Why do it? to avoid cross-interference when accessing several arrays in the same loop

³For usage with the Odd-Padding algorithm you would need to use $T_L = 8$ because T_L must be a power of two.

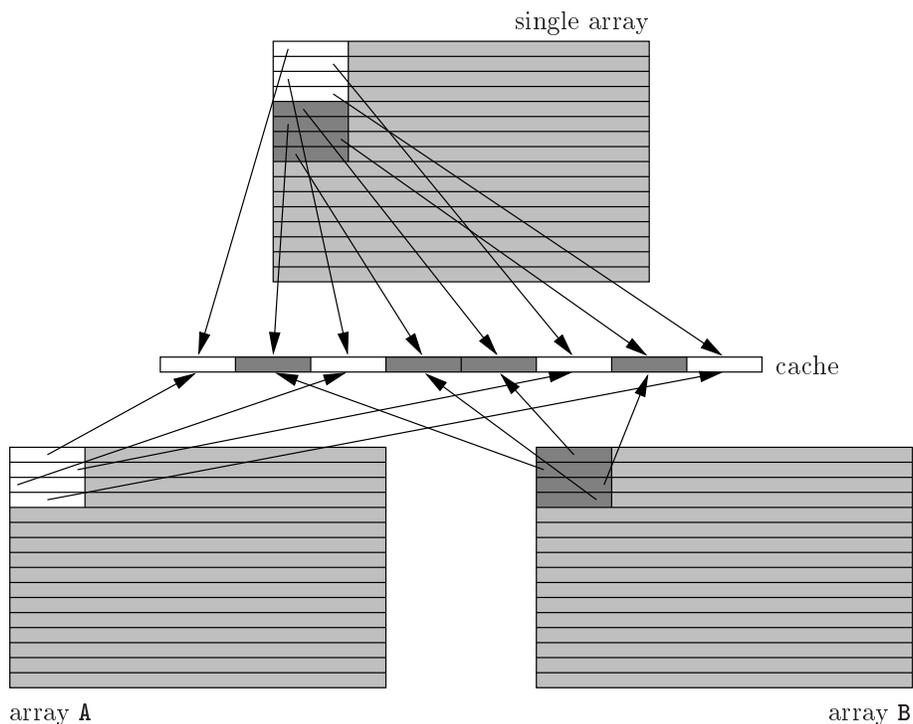


Figure 8: Here, I explain the relationship between padding for one array and padding for several arrays. All arrays in this figure are five times as long as a tile line. The cache is eight tile lines long. The padded array in figure 7 has the same relation between array length and cache size.

The tile of a single array (top) has eight lines. When using two arrays (bottom), these eight tile lines are distributed between these two arrays, so that each array has exactly four lines. Array A uses the top four tile lines of the single array and array B uses the bottom four lines. Distributing the tile lines this way avoids cross-interference between the two arrays. The trick is to find the right base address for array B.

What to do? pad the base address of the arrays (inter-variable padding)

The basic idea here is to distribute the available tile lines among several arrays by arranging their base address appropriately. This is done by inter-variable padding — through addition of unused elements at the begin of the array. There is a strong relation to the one-array case (see figure 8).

Assume you want to access n arrays. Let these arrays have numbers 0 (the first one) to $n - 1$ (the last one). Each array must have the same length A_L and use the same tile length T_L and tile height T_H . Here is how you do it:

1. First, find the appropriate tile length T_L by referring to sections 3.4 and 3.5. If you want to use the algorithm of Panda et al. [5], a tile length which is a multiple of the cache line size CL_S will do it. If you want to use the Odd-Padding algorithm from section 3.2 you need a tile length which is a power of two.
2. Since you use n arrays, the height of a tile is:

$$T_H = \left\lceil \frac{C_S}{nT_L} \right\rceil$$

All arrays must use the same T_L and T_H .

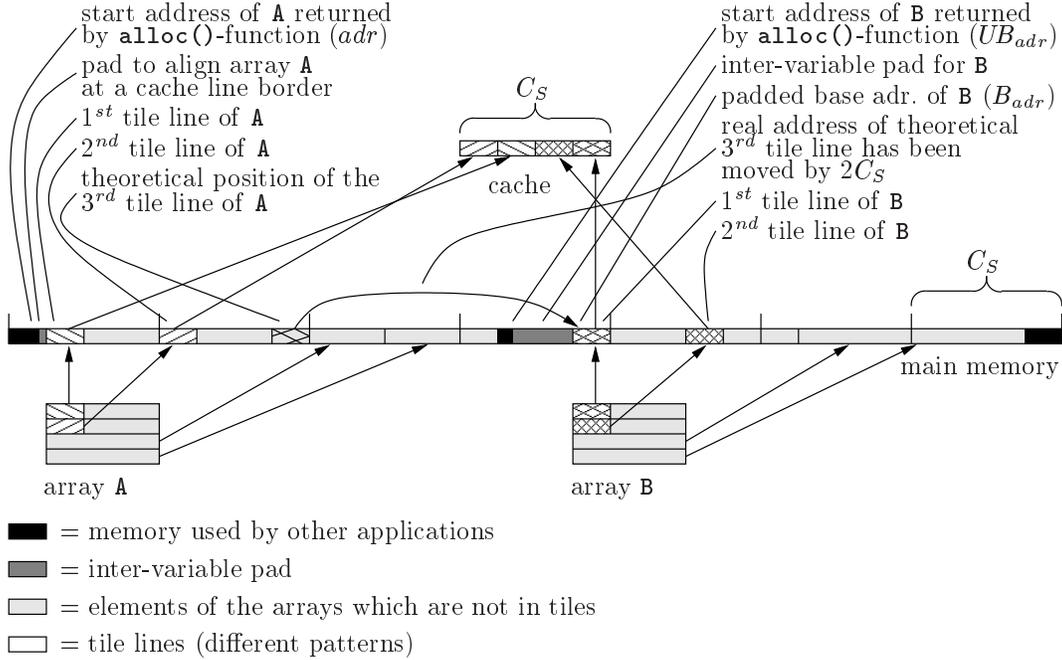


Figure 9: Inter-variable padding works by moving the base addresses of the arrays around. The arrays are three times as long as a tile and the cache is four times as long as a tile line. The long vertical lines in the main memory show where a cache mapping starts/ends. Note that the distance between the start of a cache mapping and a tile line is exactly the distance which determines where the tile line is mapped into the cache. For example the second tile line of array A starts exactly at a cache mapping begin in main memory. Consequently, it is mapped at the beginning of the cache.

The dark gray parts are inter-variable padding. For simplicity intra-variable padding is not shown in this figure. The base address of array A has been padded to align the array at a cache line border. The base address of array B has been padded to the start of this theoretical 3rd tile line. Note that array B is automatically aligned at a cache line border because this 3rd tile line starts always at whatever array A is aligned to.

3. With the new tile length, calculate the array length A_L , using the Odd-Padding algorithm from section 3.2 or the method from panda et al. [5]. It is important that all arrays have the same length.
4. Allocate the first array (A) and align its base address A_{adr} at a cache line border:

$$A_{adr}(adr, CL_S) = adr + (CL_S - (adr \bmod CL_S)) \bmod CL_S$$

adr is the address — in array elements — which the `alloc()`-function of the operating system returns, when you allocate the array. You need to allocate $C_S - 1$ elements more for your array, because the alignment leaves so much elements unused. See section 3.2 for an example. There is an exception to this rule: if you have chosen T_L appropriately and section 3.4 or 3.5 waive the requirement to align the array at a cache line border then you do not need to do it.

5. Now, you allocate the remaining arrays 1 to $n - 1$. You calculate the base address using this formula:

$$B_{adr}(A_{adr}, UB_{adr}, o) = \begin{cases} \text{if } (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S \\ \text{then } UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S + C_S \\ \text{else } UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S \end{cases}$$

where

$$o = (T_H A_L v) \bmod C_S$$

You must allocate $C_S C_L S - 1$ elements more per array because so much inter-variable pad may be needed. UB_{adr} is the address the `alloc()`-function returns to you. v is the number of the array you are allocating (1 to $n - 1$). A_{adr} is the padded address of the first array.

Clearly, the trick is to find the right pad for the arrays 1 to $n - 1$. How does it work? The above B_{adr} function will result in something like $A_{adr} + iC_S + o$. iC_S does not matter for the cache mapping and can be ignored. Assume the case where n is 4 and the tile height T_H is 8. The resulting addresses would be:

$$\begin{aligned} A_{adr} + (0 * 8A_L) \bmod C_S & \quad \text{for array 0} \\ A_{adr} + (1 * 8A_L) \bmod C_S & \quad \text{for array 1} \\ A_{adr} + (2 * 8A_L) \bmod C_S & \quad \text{for array 2} \\ A_{adr} + (3 * 8A_L) \bmod C_S & \quad \text{for array 3} \end{aligned}$$

That is: array 0 uses tile lines 0–7 of a single array. Array 1 starts exactly where the 8th tile line of a single array would go and, therefore, uses tile lines 8–15 of that imaginary single array. Array 2 starts exactly where the 16th tile line of a single array would begin and uses tile lines 16–23. Array 3 uses tile lines 24–31 of that theoretical single array (enjoy also figure 9).

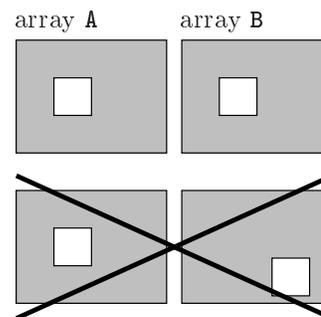
3.7 How can you access several arrays?

Why do it? to avoid cross-interference and get the maximum out of padding

What to do? know the rules

The rules are the same as those stated in section 3.3. To keep this paper small and to avoid boring the reader I do not repeat them here. There is one additional new rule when handling several arrays. This rule goes like this: For all arrays accessed at the same time, i. e. in the same loop, you must use the same base coordinate for all tiles. See figure 10 for what happens when you violate this rule. This rule is a severe restriction of the usefulness of padding. If this restriction is a problem, you may want to try one of these methods to avoid cache interference: copying (Temam et al. [8]) or Tetris.

The tile base coordinate is a concept which I introduced to explain how you must handle your arrays. It is nothing which you need to tell your hardware or operating system about. You simply access the arrays in the described way. Your hardware will then operate correctly (... er ... hopefully).



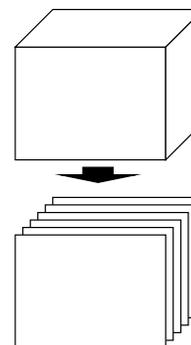
3.8 How do you handle multi-dimensional arrays?

Why do it? to avoid cache interference when accessing arrays with more than two dimensions

What to do? chop the array into two-dimensional planes

If you need to work on arrays with more than two dimensions, simply dissect them into a number of two dimensional planes and pad and access them as described in the previous sections. Sometimes you will find yourself in a situation where you need to access several — but not all — planes at once. I will show you on an example how you can handle this:

Assume you have a 3D-array with 6 planes but you access only three of them at the same time. First you access planes 0, 1 and 2. Next 1, 2, 3



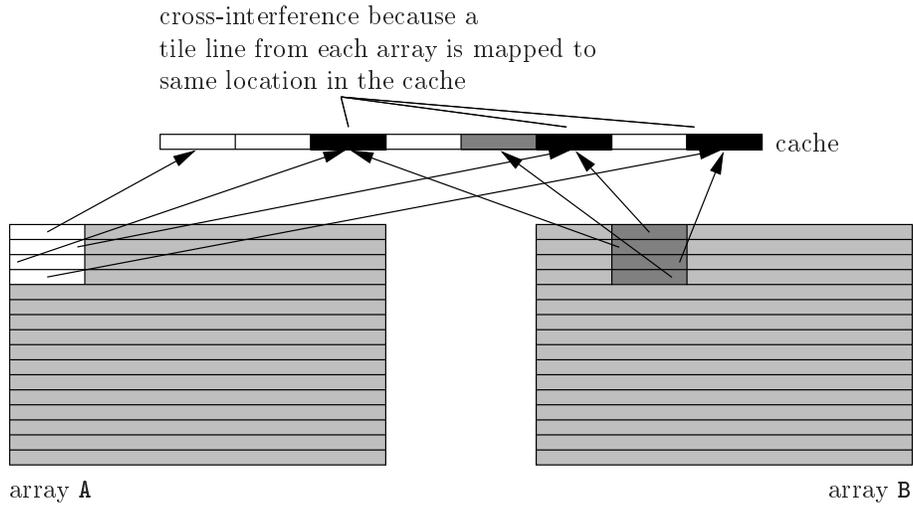


Figure 10: The accesses to the arrays in this example do not use the same tile base coordinates for both arrays and, therefore, cause cache interference. Compare this with figure 8.

Box 9: The Odd-Padding algorithm — a quick overview

- your tile length and your cache size is a power of two
- your array and your tile start always at a cache line border

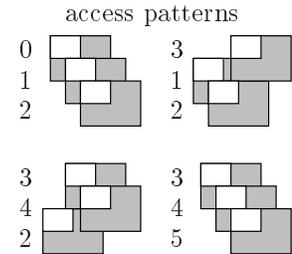
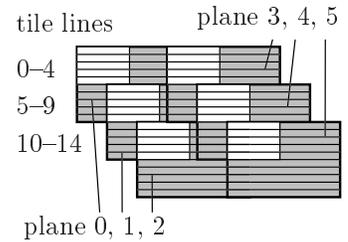
↓

- your tile height is the cache size divided by the tile length
- your array length must be an odd multiple of your tile length

then 2, 3, 4 finally 3, 4, 5. How do you pad this? Assume your cache can hold 16 tile lines, then for three arrays each tile has a height of 5 lines and one line remains unused. Further I assume you have padded the array length appropriately.

You pad the first three planes (0, 1, 2) as if you had only these three arrays using the method described in section 3.6. You pad plane 3 using the B_{adr} -function with the same o you used for array 0. Then you pad plane 4 with the o you used for array 1 and finally you pad array 5 with the o of plane 2. The result is this:

plane #	padded using o of array	tile lines reserved
0	0	0–4
1	1	5–9
2	2	10–14
3	0	0–4
4	1	5–9
5	2	10–14



This way, there will be no interference with any access combination mentioned above. For example when you access planes 2, 3, 4 then tile lines 10–14, 0–4 and 5–9 are used respectively. There are no overlapping tile lines.

3.9 How do you pad for hierarchical caches?

Why do it? to make best use of all caches you have

What to do? sorry, man, there is (almost) no way ...

First of all, let me clarify: in this section I discuss only padding for hierarchical tiling for *caches*. Hierarchical tiling can be used for other stuff but then it uses totally different mapping functions and, therefore, the methods described in this text do not apply.

Let me assume you have two caches: a small and fast one and one which is bigger and slower but still faster than your memory. The idea behind hierarchical tiling is: to use not only the small cache but also the big one to speed up your program. You need to have or create an access pattern like this: first you access one tile, then you change the tile base coordinates and access some other tiles, later you will come back and re-access the first tile. Hierarchical tiling tries to hold the data of the first tile in the big cache, so that it need not to be read from memory when the tile is accessed the second time.

For hierarchical tiling to make sense your big cache must be

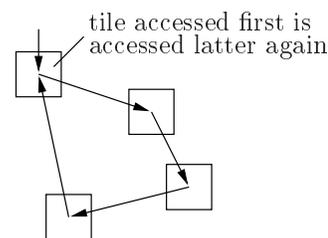
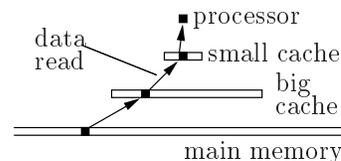
- much faster than the memory
- much bigger than the first cache (McKinley and Temam [4] found that the cache size must be made much bigger to gain a significant win.)

But this is a combination you usually do not find in real computers. The “big” cache is either small and fast or large and slow.

Nevertheless, I know possibilities to combine padding with hierarchical tiling but there is only one which makes sense to me and it is a very restricted case. I explain you that case here but my advise is to avoid hierarchical tiling as long as you have no reason to believe that it will be a great advantage in your case. Most likely you will find yourself spending much time and gaining only a small win. If you still believe hierarchical tiling is an advantage for you, consider reading about Tetris because Tetris has more powerful means to utilize several caches.

Here is how you pad for two caches: the smaller/faster cache has size C_S , and tile size T_H , T_L with tile base coordinates T_x , T_y . The bigger cache has size BC_S and tile size BT_H , BT_L and tile base coordinates BT_x , BT_y .

- The size of the big cache, BC_S , must be a power of two.
- The tile in the big cache must have the same length as the tile in the small cache, i.e. $BT_L = T_L$. (I told you: it is a restricted case!)
- The parameters of the small cache must meet all requirements of the Odd-Padding algorithm, that is:
 - The size of your small cache is a power of two i.e. $C_S \in \{1, 2, 4, 8, 16, 32, 64, \dots\}$.
 - Your cache line size is a power of two i.e. $CL_S \in \{1, 2, 4, 8, 16, 32, 64, \dots\}$.
 - You have chosen a tile length T_L which is also a power of two and is given in multiples of cache lines.
 - You know that your array must be at least UA_L cache lines long.
- You pad the length and the start address of your array using the parameters of the small cache. *You must use the Odd-Padding algorithm*, as described in section 3.2 or section 3.6 if you deal with several arrays.



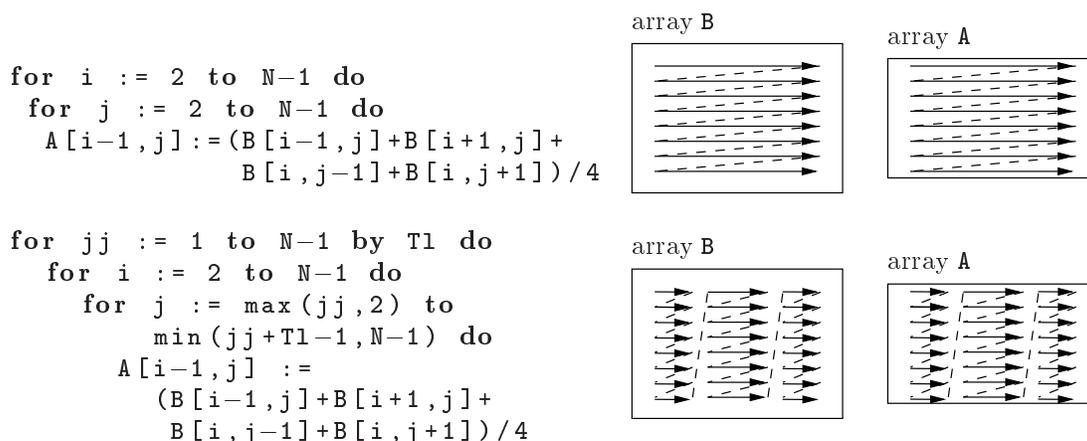
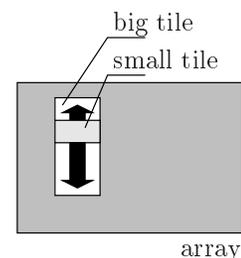


Figure 11: The top figure shows how the untilted stencil operation processes the array. The bottom figure shows the tiled one, using tile size T_1 . Note that the arrays are three times T_L long but the tiled loop leaves out one element of the left and right column. Letting a tile start one element from the left is not legal because a tile must start at a cache line border.

That is all you need to do. The small and the big tile have the same length but not the same height:

$$T_H = \frac{C_S}{T_L} \qquad BT_H = \frac{BC_S}{BT_L} = \frac{BC_S}{T_L} = \frac{T_H(BC_S)}{C_S}$$

You can access each tile using the rules of section 3.3 or section 3.7 if you use several arrays (in that case you need to divide T_H and BT_H by n). But you need to follow this additional rule: The small tile must always be inside the big tile, especially $T_x = BT_x$. When you access a cache line for the first time, the cache line is loaded into the big and the small cache. It will remain in the small cache until you change the base coordinates of the small tile. It will remain in the big cache until you change the base coordinates for the big tile, no matter how often you change the base coordinates of the small tile.



3.10 How do you pad for stencil operations?

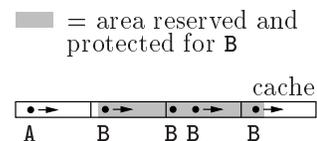
Why do it? to avoid cache-interference when doing stencil operations

What to do? apply the methods of the former sections

I do not want to discuss stencil operations in general. See the paper of Rivera and Tseng [7] for a more general discussion. Instead I concentrate upon the relation between stencil operations and padding. Furthermore, I discuss only the example shown in figure 6 on page 11. Please, have a look on that figure.

If the arrays are small enough, you do not need to use tiling. For the example from figure 6, if the cache holds a little bit more than two complete lines of the arrays then tiling would be unnecessary. I assume that the arrays are greater than that, so that you must tile the loop as shown in figure 11.

How to choose T_L ? Assume you have a cache with 128 cache lines. I advise you to split your cache in four tile lines, each 32 cache lines long, as shown in the figure at the right. Then the access to **A** and each access to a line of **B** get an own tile line. That is a bit generous because the accesses to **B** require only to protect the gray area from any other use but the Odd-Padding algorithm can only pad the arrays for a power of two number of



tile lines and that is four here. You could use the remaining white space for access to other arrays if you have such possibilities.

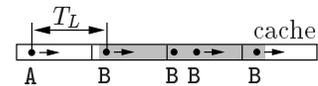
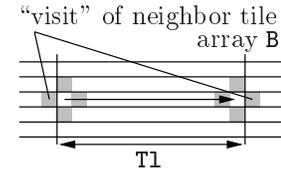
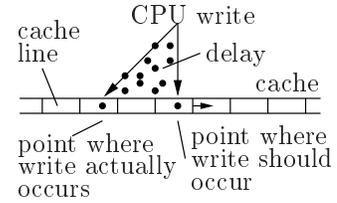
Note: the access to **A** is a write. In contrast to read-accesses writes may be delayed by the write back buffer of your CPU. That is: they may happen later than they appear in the program. Therefore, you need to reserve two to eight cache lines behind a write access to avoid conflicts with the next cache operation.

Tile your loop in such a way that it will use only 31 cache lines, that is: leave one cache line unaccessed. In figure 11 T_L should be $(T_L - 1)CL_S$. The reason is that the accesses to array elements $B[i, j-1]$ and $B[i, j+1]$ will have a look into the neighbor tile. If you leave one cache line unused, this cache line is used for that glimpse and no useful data is thrown out of the cache.

You need to pad the whole thing as described in section 3.6. To be more precise: you first pad the length of both arrays. They will need to have the same length but not the same height. You do not need the first and last “frame” line of array **A**. The situation is exactly as shown in figure 6.

The array **A** is the one with number 0 and is aligned at a cache line border as described in section 3.2. Then array **B** is allocated (with $C_S CL_S - 1$ elements more for padding) and its base address is calculated using the B_{adr} -function from section 3.6. Here is an important difference to that section: your arrays use different tile heights therefore you use $o = T_L$ for this B_{adr} calculation (see the picture at the right). Technical speaking you can choose any o between 1 and $2T_L - 2$ or so. This would just move the gray area with the accesses to **B** around in the cache. With such o 's you could also place other array accesses appropriately in the cache.

The distance between the **B**-accesses is defined by the array length and — since you have padded the array length — by tile length T_L . Note: I do not prove anything from this section.



3.11 How do you choose a padding algorithm?

For intra-variable padding you can choose between several algorithms. This is interesting because it is the kind of padding where you most likely spent most memory for. Choosing the right algorithm for intra-variable padding can save you much memory. (For inter-variable padding and cache line alignment there is only one possible method, so you do not have a choice.) You do not want to use the following techniques for padding:

Lam, Rothberg and Wolf’s method [3] often utilizes only a small part of the cache. Moreover, it is not a padding technique because it tells you your tile size after you choose your array length. But most of all: you can not choose your tile size, it tells you what size you must use.

Coleman and McKinley’s algorithm [2] is also not a padding technique. As Lam, Rothberg and Wolf’s method [3] it tells you a tile size after you choose your array length. This tile size is rectangular and uses most of the cache but again: you can not choose your tile size or shape. You must accept whatever tile size this algorithm returns. It may be a very degenerate tile.

Rivera and Tseng’s method [6] does only worst case padding. That is: they only avoid cases which are described in box 4. This technique does not guarantee you that there will be no cache interference. Consequently, you do not know which size your tile may have.

Rivera and Tseng’s method [7] does only worst case padding and inter-variable padding. It is specially designed to handle stencil operations.

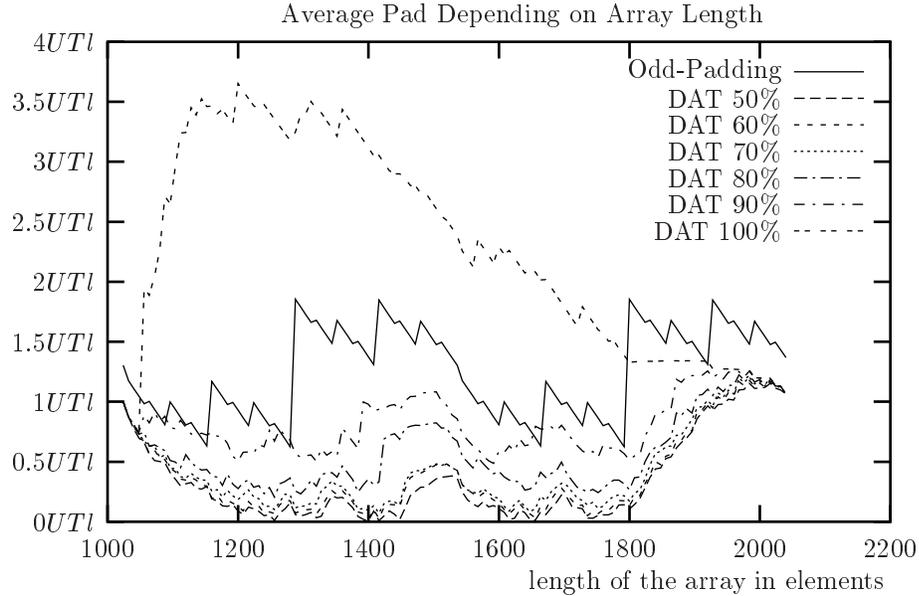


Figure 12: This plot shows the average pad required for different user array length UA_L . I assume a cache with 1024 elements and a cache line length of 8 elements. Other values of C_S and CL_S will give very similar curves. For each user array length between 1024 and 2048 — I measure one point all 8 elements — I calculate the average pad length. That is: I calculate the pad for all tile sizes between 8 elements and 256 elements in steps of 8 elements, dividing each single pad by the tile length and use the average as the value for that point. The meaning of the different percentage numbers for Panda et al.’s DAT algorithm are described in box 12.

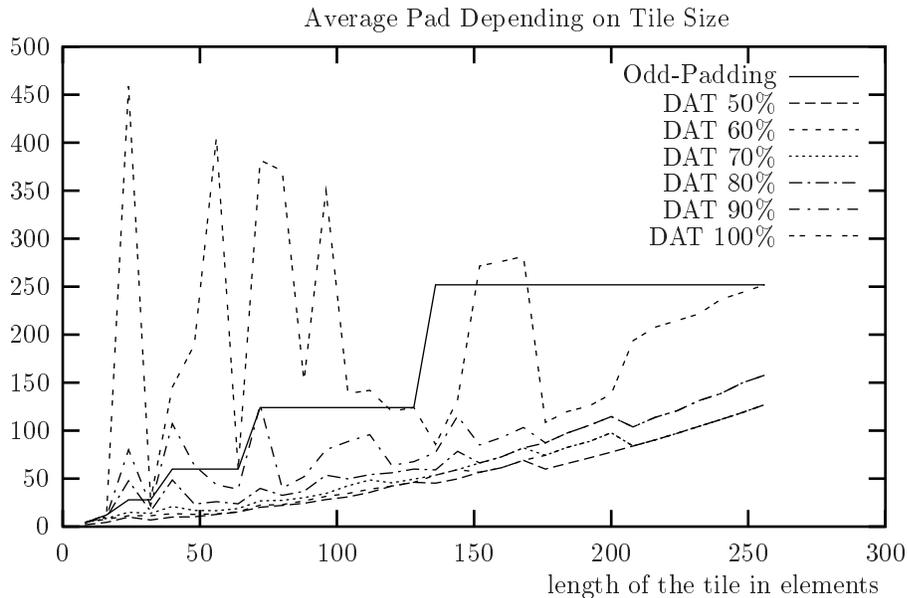


Figure 13: This plot shows the average pad for different tile length. I assume a cache with 1024 elements and a cache line length of 8 elements. Other values of C_S and CL_S will give very similar curves. For each user tile length UT_L between 8 and 256 I calculate the average pad length. For each given tile length the average pad is produced by calculating the required pads for all array lengths from 1024 to 2048 in steps of 8 elements. The meaning of the different percentage numbers for Panda et al.’s DAT algorithm are described in box 12.

Box 10: Which one is better: Odd-Padding or DAT?

Which algorithm is better: Odd-Padding or DAT? Panda et al.'s DAT [5] algorithm is better than the Odd-Padding algorithm if you ignore the worst case behavior of DAT. Odd-Padding has been designed to handle power-of-two tile lengths when you want to utilize the whole cache. For this case the Odd-Padding algorithm finds the shortest possible pad. But the DAT algorithm finds the same pad in that case. In all other cases the Odd-Padding algorithm will perform poorly:

The tile length is not a power of two. In this case you must extent the tile length by adding unused elements to it as described in sections 3.4 and 3.5. For example for a user tile length of 17 cache lines you would need to add 15 unused cache lines to get the real tile length of 32 cache lines. About 50% of your cache would remain unused. In average over all possible multiple of a cache line long tiles, the Odd-Padding algorithm utilizes about 75% of the cache. Note: the Odd-Padding algorithm utilizes 100% of the cache for all power-of-two tile length.

You do not want to use the whole cache. If you do not use the whole cache, most likely, there is a shorter pad than the one the Odd-Padding algorithm calculates. Panda et al.'s DAT algorithm will find this shorter pad and, therefore, is better.

Furthermore, both algorithms avoid the same amount of cache interference, so that no one makes your program faster than the other. Both algorithms are applicable in the same situations, so that no algorithm can handle a case which the other can not handle. Note however, that I introduced a method to handle hierarchical tiling which requires the Odd-Padding algorithm (see section 3.9). The real problem of Panda et al.'s DAT algorithm is that it uses much more pad than the Odd-Padding algorithm in the worst case.

Box 11: Why should you use the Odd-Padding algorithm?

Box 10 explains that the DAT algorithm from Panda et al. [5] is in general the better padding algorithm. So why should you bother with the Odd-Padding algorithm? Here are some points where the Odd-Padding algorithm has an advantage over the DAT algorithm:

- The Odd-Padding algorithm is a simple formula (see box 9): make your tile length a power-of-two, align your tile always at a cache line border and make your array length an odd multiple of the tile length. Simple to remember, simple to apply. You do not need to look up and hack in a whole procedure when you just want to try something out or when wasting some memory does not matter.
- In the worst case the Odd-Padding algorithm uses a pad which is shorter than two times the power-of-two tile size. Panda et al.'s algorithm uses much more pad in the worst case. Sometimes you may want to use that fact to make sure that you do not need too much memory for the pad.
- If your tile length is a power-of-two and you want to use the whole cache, there is no better solution than the one calculated by the Odd-Padding algorithm.
- I proved the Odd-Padding algorithm correct, so you can rely on it.

The most significant advantage of the Odd-Padding algorithm are the facts that it is a simple formula and has an acceptable worst case pad.

Box 12: What does DAT 90% mean?

For the algorithm from Panda et al. (DAT) I can choose how many rows my tile should have. For DAT with 100% I use all rows which fit into the cache. For DAT with 90% I used 90% of the rows which would fit into the cache. For DAT with 50% I use half of the rows which would fit into the cache. I round the number of rows towards zero if necessary, i. e. 70% of 11 rows gives 7 rows. For the Odd-Padding algorithm, I can not choose the utilization of the cache but I always need to extend the tile length towards the next larger power-of-two number. This results in an average cache utilization of about 75% for the Odd-Padding algorithm.

Cierniak and Li's method [1] does not even intend to avoid any cache interference.

Copying by Temam, Granston and Jalby [8] is a very powerful method but not a method for padding. I compare it to padding in section 6.

Tetris (see section 5) is also a very powerful method but not a method for padding, too. I compare it to padding in section 6.

The Brute-Force-Padding algorithm (see box 6) wastes too much memory to be useful in practise. It may be used for theoretical issues or in very strange situations.

It remains Panda, Nakamura, Dutt and Nicolau's DAT algorithm [5] and the Odd-Padding algorithm (see box 9). For the rest of this section I want to compare these two algorithms. Box 10 discusses which one is the better one and box 11 tells you when you may want to use the Odd-Padding algorithm. I also compare DAT with the Odd-Padding algorithm in two graphs:

figure 12 shows how much pad these algorithms use depending on the array length.

figure 13 shows how much pad these algorithms use depending on the tile length.

Please, have a look at these figures. Here, I discuss **figure 12** in more detail:

? What is the essence of figure 12?

The plot shows that the DAT algorithm needs less pad than the Odd-Padding algorithm if you utilize 90% or less of the cache. The DAT 100% case shows clearly that the worst case of the DAT algorithm is much worse than the one of the Odd-Padding algorithm. Note however, that although the difference between DAT's 100% line and the Odd-Padding line is large, the Odd-Padding uses only about 75% of the cache in average (see box 10) and comes not even close to the 100% cache utilization.

? Why has the Odd-Padding curve such a strange look?

The strange look of the Odd-Padding line results from the fact that the algorithm can only handle power-of-two tile lengths. For some array sizes the Odd-Padding algorithm needs only a very small pad in average over all the different tile sizes. For some other array lengths the Odd-Padding algorithm needs very large pads in average. This produces the large jumps in the line of the Odd-Padding algorithm.

? Why is the average height of the Odd-Padding curve not one UT_L ?

The Odd-Padding algorithm has an average pad length of one power-of-two tile length T_L . The average height of the curve in the graph is somewhat higher than one user tile length. The reason is that the user tile length UT_L is not always a power-of-two and I must divide the pad by UT_L and not by T_L . Since UT_L is often smaller than T_L the resulting curve is a bit higher than one UT_L .

Now, let me discuss **figure 13** in more detail:

? Why has the Odd-Padding curve such steps in figure 13?

In average the Odd-Padding algorithm uses a pad which is as long as one tile. This and the fact that the tile length must be a power-of-two causes the steps in the line of the Odd-Padding algorithm. For example: for all user tile sizes between 65 and 128 elements the Odd-Padding algorithm uses a real tile size of 128 elements which gives an average pad of 128 elements over all array lengths. This is the 128 elements high step which you see between tile sizes 65 and 128 in figure 13.

? Why is the DAT curve sometimes higher than the Odd-Padding curve?

The DAT algorithm requires the most pad for 100% cache utilization and small tile length. However, it actually utilizes the cache with up to 100% where as the Odd-Padding algorithm fails to do so if the user tile length is not a power-of-two. For example: for a tile length of 24 elements — the tallest peak of DAT's 100% curve — the Odd-Padding algorithm uses a real tile length of 32 elements and, therefore, utilizes the cache only about 75%.

? Is there a relation between the DAT 100% curve and the Odd-Padding curve?

Note that the 100% curve of DAT meets the curve of the Odd-Padding algorithm at all power-of-two tile length like 8, 16, 32, 64, 128 and so on. The pad calculated by the Odd-Padding algorithm is the shortest possible for power-of-two tile length if you want to use the whole cache, therefore, both curves must meet at this spots.

? Why has DAT's 100% curve such tall peaks?

The high peaks of DAT's 100% curve are a result of the following: If you want to use the whole cache, there is sometimes no other solution than the one described in box 6 (Brute-Force-Padding). But the Brute-Force-Padding algorithm has a worst case of about the cache size minus two times the tile length. For the tile lengths where this large worst case pad is often required DAT's 100% curve has a tall peak. Since longer tile lengths reduce the worst case padding length of the Brute-Force-Padding algorithm, the peaks of the DAT 100% curve get smaller when the tile length increase.

? What is the essence of figure 13?

The Odd-Padding algorithm performs acceptable for small tile lengths but looses for larger ones. With increased tile length both algorithm need more pad.

To sum it up: Panda et al.'s DAT algorithm produces a shorter pad in average — at least when you use 90% or less of your cache — and is, therefore, better than the Odd-Padding algorithm. Nevertheless, the Odd-Padding algorithm may find some applications because it is easy to remember and to apply and has a better worst case behavior than the DAT algorithm.

4 The Odd-Padding Proofs

Proofs are very important but I know that most people will not bother with them. Therefore, the innocent reader may skip this section.

In this part of the text I prove five theorems:

The core theorem is used in the proofs of the Odd-Padding-algorithm theorem and the multi-array-access-algorithm theorem. It is a property of the modulo function and shows that no line of a tile is mapped on another one if a tile line has length one.

The Odd-Padding-algorithm theorem proves that there is no self-interference when you use the Odd-Padding-algorithm for just one array. I also show the impossibility to find pads which are shorter than the ones used by the Odd-Padding-algorithm if you want to utilize the whole cache.

The Odd-Padding-formula theorem shows that the Odd-Padding-formula comes up with the correct array length for the Odd-Padding-algorithm.

The multi-array-access-algorithm theorem proves cross- and self-interference freeness, when you use the Odd-Padding-algorithm with several arrays. This is actually a more complex version of the Odd-Padding-algorithm theorem.

The multi-array-access-formula theorem ensures that this formula returns the right base address for the padding of several arrays with the Odd-Padding-algorithm.

In the proofs I will use some fact which I hope are common known or at least believable without proof. I call them axioms and put them in box 13 on page 34.

4.1 The core theorem

I want to prove

$$h, u \in \mathbb{N} \setminus \{0\} \quad (1)$$

$$\wedge p, q \in \mathbb{N} \wedge p, q < u \quad (2)$$

$$\wedge \gcd(h, u) = 1 \quad (3)$$

$$\wedge (ph) \bmod u = (qh) \bmod u \quad (4)$$

\Rightarrow

$$p = q$$

Assume that all variables are given in multiples of the tile length T_L and not in cache line size CL_S . Then h is the length of the array and u the size of the cache. Since the basis unit is the length of a tile, two tiles in the cache either fall exactly on each other or not. There is no possibility for one tile line to partial overlap the other. The theorem (4) states that tile line p is only mapped on q if p is q .

Note that I am a liar. In section 3.2 I tell you h — the length of the array in tile sizes — must be odd and u — the cache size — must be a power of two. If u is a power of two, all its prime factors are 2, whereas h is odd and does not contain any 2 as prime factor. Hence, the $\gcd(h, u)$ is 1. Therefore, the beast I prove here is stronger than necessary.

Proof 1: The core theorem

$$ph \bmod u = qh \bmod u \quad (1.1)$$

= {apply the axiom for mod (6)}

$$ph - \left\lfloor \frac{ph}{u} \right\rfloor u = qh - \left\lfloor \frac{qh}{u} \right\rfloor u \quad (1.2)$$

$$\begin{aligned}
&= \left\{ + \left\lfloor \frac{ph}{u} \right\rfloor u, -qh \right\} \\
&(p - q)h = \left(\left\lfloor \frac{ph}{u} \right\rfloor - \left\lfloor \frac{qh}{u} \right\rfloor \right) u \tag{1.3} \\
&= \{\text{consider the cases } p = q \text{ and } p \neq q\}
\end{aligned}$$

Case 1: $p = q$

$$(p - q)h = \left(\left\lfloor \frac{ph}{u} \right\rfloor - \left\lfloor \frac{qh}{u} \right\rfloor \right) u \tag{1.4}$$

= {rewrite $p = q$ }

$$(q - q)h = \left(\left\lfloor \frac{qh}{u} \right\rfloor - \left\lfloor \frac{qh}{u} \right\rfloor \right) u \tag{1.5}$$

= {both sides collapse to 0}

true (1.6)

= {rewrite this with the assumption from this case which is also **true**}

$$p = q \tag{1.7}$$

Case 2: $p \neq q$

$$(p - q)h = \left(\left\lfloor \frac{ph}{u} \right\rfloor - \left\lfloor \frac{qh}{u} \right\rfloor \right) u \tag{1.8}$$

= {substitute v for $\left(\left\lfloor \frac{ph}{u} \right\rfloor - \left\lfloor \frac{qh}{u} \right\rfloor \right)$ }

$$(p - q)h = vu \tag{1.9}$$

= $\left\{ \begin{array}{l} \text{To be an equality both sides must have the same prime factors. From } \gcd(h, u) = 1 \text{ (3) follows} \\ \text{that } h \text{ and } u \text{ do not share any prime factors. Therefore, } v \text{ must deliver all prime factors of } h, \\ \text{that is } h \text{ divides } v. \text{ Substitute } v = v'h \text{ where } v' = v/h. \text{ Note: } (p - q) \neq 0 \wedge h > 0 \text{ from (1) } \Rightarrow \\ (p - q)h \neq 0 \Rightarrow vu \neq 0 \text{ because } (p - q)h = vu. vu \neq 0 \Rightarrow v \neq 0 \Rightarrow v' \neq 0. \end{array} \right\}$

$$(p - q)h = v'hu \tag{1.10}$$

= {/h This step is legal because $h > 0$ (1)}

$$(p - q) = v'u \tag{1.11}$$

\Rightarrow { $\Rightarrow \geq$ }

$$(p - q) \geq v'u \tag{1.12}$$

\Rightarrow { $p \geq (p - q)$ because $0 \leq q$ from (2)}

$$p \geq v'u \tag{1.13}$$

= { $p < u$ from (2) and $v' \neq 0$ }

false (1.14)

= {from the assumption of this case: $(p = q) = \mathbf{false}$ }

$$p = q \tag{1.15}$$

Box 13: The axioms used in the proofs

Here are the axioms which I use without proving them. I hope these axioms are known or at least believed without me proving them. Most of them can be established by using plain arithmetic and structural induction over the definition of the modulo function (5) and floor function.

$$a \in \mathbb{N} \wedge b \in \mathbb{N} \setminus \{0\}$$

$$\wedge a \bmod b = \mathbf{if} \ a < b \ \mathbf{then} \ a \ \mathbf{else} \ (a - b) \bmod b \quad (5)$$

$$a \in \mathbb{N} \wedge b \in \mathbb{N} \setminus \{0\}$$

$$\wedge a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b \quad (6)$$

$$a, i \in \mathbb{N} \wedge b \in \mathbb{N} \setminus \{0\}$$

$$\wedge (a + ib) \bmod b = a \bmod b \quad (7)$$

$$a, c \in \mathbb{N} \wedge b \in \mathbb{N} \setminus \{0\}$$

$$\wedge ((a \bmod b) + c) \bmod b = (a + c) \bmod b \quad (8)$$

$$a, b \in \mathbb{N} \wedge c, d \in \mathbb{N} \setminus \{0\} \wedge a < c$$

$$\wedge (a + bc) \bmod (dc) = a + (b \bmod d)c \quad (9)$$

$$a, b, d \in \mathbb{N} \wedge c \in \mathbb{N} \setminus \{0\}$$

$$\wedge (a + d) \bmod c = (b + d) \bmod c \Rightarrow a \bmod c = b \bmod c \quad (10)$$

$$a, b, e, f \in \mathbb{N} \wedge c \in \mathbb{N} \setminus \{0\} \wedge a, b < c$$

$$\wedge a + ec = b + fc \Rightarrow a = b \wedge e = f \quad (11)$$

$$b_1, b_2 \in \mathbb{B} \wedge a, c \in \mathbb{N}$$

$$\wedge \mathbf{if} \ b_1 \ \mathbf{then} \ a \ \mathbf{else} \ (\mathbf{if} \ b_2 \ \mathbf{then} \ a \ \mathbf{else} \ c) = \mathbf{if} \ b_1 \vee b_2 \ \mathbf{then} \ a \ \mathbf{else} \ c \quad (12)$$

Furthermore, I reason in the proof of the core theorem about prime factors and some times move expressions into **if** branches.

\Rightarrow {from (1.7) and (1.15)}

$$p = q \quad (1.16)$$

□

A thanks to Prof. J. Misra here. He told us in his Spring 1998 class to replace proofs by contradiction through proofs by induction. In the first place, this was a proof by contradiction: given the facts, assume one tile line falls on another one ... When I tried to change the proof into an induction I found this much nicer one.

4.2 The Odd-Padding-algorithm theorem

I want to prove that the Odd-Padding-algorithm does avoid self-interference. That is, no cache line in the tile will be mapped onto another one, regardless of the base address of the array adr and regardless of the base coordinates of the tile T_x and T_y .

$$adr, T_x, T_y, x_p, y_p, x_q, y_q \in \mathbb{N} \quad (13)$$

$$\wedge h, T_H, T_L \in \mathbb{N} \setminus \{0\} \quad (14)$$

$$\wedge x_p, x_q < T_L \wedge y_p, y_q < T_H \quad (15)$$

$$\wedge A_L = hT_L \quad (16)$$

$$\wedge C_S = T_H T_L \quad (17)$$

$$\wedge \gcd(h, T_H) = 1 \quad (18)$$

$$\wedge m(T_{adr}(x_p, y_p)) = m(T_{adr}(x_q, y_q)) \quad (19)$$

\Rightarrow

$$x_p = x_q \wedge y_p = y_q$$

where m is the cache mapping function and T_{adr} is the address of the cache line with coordinates x, y in memory:

$$m(adr) = adr \bmod C_S \quad (20)$$

$$T_{adr}(x, y) = adr + T_y A_L + T_x + y A_L + x \quad (21)$$

The Odd-Padding-algorithm requires that C_S and T_L are a power-of-two and A_L is an odd multiple of T_L (see box 9). Mathematically, this implies that the Odd-Padding-algorithm chooses the array length A_L in such a way that its greatest common divisor with the cache size C_S is T_L i.e. $\gcd(C_S, A_L) = T_L \Rightarrow \gcd(T_H T_L, h T_L) = T_L \Rightarrow \gcd(T_H, h) = 1$. Line (17) ensures that T_L divides C_S and line (16) ensures that T_L divides A_L . This theorem says: “any two cache lines of the tile with the coordinates x_p, y_p and x_q, y_q fall only on each other if they are the same.”

Proof 2: The Odd-Padding theorem

$$m(T_{adr}(x_p, y_p)) = m(T_{adr}(x_q, y_q)) \quad (2.1)$$

$$= \{\text{unfold definition of } T_{adr} \text{ (21)}\}$$

$$m(adr + T_y A_L + T_x + y_p A_L + x_p) = m(adr + T_y A_L + T_x + y_q A_L + x_q) \quad (2.2)$$

$$= \{\text{unfold definition of } m \text{ (20)}\}$$

$$(adr + T_y A_L + T_x + y_p A_L + x_p) \bmod C_S = (adr + T_y A_L + T_x + y_q A_L + x_q) \bmod C_S \quad (2.3)$$

$$\Rightarrow \{\text{use axiom (10) to remove } adr, T_y A_L, T_x\}$$

$$(x_p + y_p A_L) \bmod C_S = (x_q + y_q A_L) \bmod C_S \quad (2.4)$$

$$= \{\text{rewrite } C_S = T_H T_L \text{ from (17) and } A_L = h T_L \text{ from (16)}\}$$

$$(x_p + y_p h T_L) \bmod (T_H T_L) = (x_q + y_q h T_L) \bmod (T_H T_L) \quad (2.5)$$

$$= \{\text{axiom (9) applies because } x_p, x_q < T_L \text{ (15)}\}$$

$$x_p + ((y_p h) \bmod T_H) T_L = x_q + ((y_q h) \bmod T_H) T_L \quad (2.6)$$

$$\Rightarrow \{\text{axiom (11) applies because } x_p, x_q < T_L \text{ (15)}\}$$

$$x_p = x_q \wedge (y_p h) \bmod T_H = (y_q h) \bmod T_H \quad (2.7)$$

$$\Rightarrow \{\text{apply the core theorem (4) from section 4.1 because } y_p, y_q < T_H \text{ (15) and } \gcd(h, T_H) = 1 \text{ (18)}\}$$

$$x_p = x_q \wedge y_p = y_q \quad (2.8)$$

□

Is the Odd-Padding-algorithm the only solution which utilizes the whole cache or are there other possibilities? Solutions where the cache size C_S is not a multiple of the tile length T_L cannot use the whole cache, so at

least a few cache lines will be unused. It remains to check for solutions where T_L divides C_S . The Odd-Padding-algorithm chooses $\gcd(h, T_H) = 1$ but what if h and T_H share a common factor? Let this factor be n with $n > 1$ so that $\gcd(hn, T_H n) = n$ and the assumptions (15), (16), (17) rewrite to:

$$\begin{aligned} y_p, y_q &< T_H n \\ A_L &= hnT_L \\ C_S &= T_H nT_L \end{aligned}$$

I choose $x_p = x_q = x$ and $y_p = y$, $y_q = (y + mT_H)$ where $1 \leq m < n$ so that $y_p \neq y_q$. Let me rewrite proof step (2.5) with these variables:

$$\begin{aligned} &(x + yhnT_L) \bmod (T_H nT_L) = (x + (y + mT_H)hnT_L) \bmod (T_H nT_L) \\ = &\{\text{use axiom (9) because } x < T_L n \text{ (15)}\} \\ &x + ((yh) \bmod T_H)T_L n = x + ((yh + mT_H h) \bmod T_H)T_L n \\ = &\{\text{remove } mT_H h \text{ by applying axiom (7)}\} \\ &x + ((yh) \bmod T_H)T_L n = x + ((yh) \bmod T_H)T_L n \end{aligned}$$

This equality is always true despite the fact that I have chosen $y_p \neq y_q$. To express this finding in words: if h and T_H share a common factor then some tile lines are mapped on each other and cause cache interferences. This in turn implies that the Odd-Padding-algorithm uses the only possible solutions which utilize the whole cache. Note: other algorithms e. g. Panda, Nakamura, Dutt and Nicolau [5] may find the same amount of pad as the Odd-Padding algorithm.

4.3 The Odd-Padding-formula theorem

I want to prove that the function that calculates the array length A_L

$$A_L(UA_L, T_L) = UA_L + (2T_L - ((UA_L + T_L) \bmod (2T_L))) \bmod (2T_L) \quad (22)$$

returns the next bigger or equal array length which is an odd multiple of T_L . Let the initial or user array length UA_L be

$$UA_L = 2iT_L + jT_L + k \quad \text{where} \quad (23)$$

$$k = UA_L \bmod T_L \quad (24)$$

$$j = \frac{(UA_L - k) \bmod (2T_L)}{T_L} \quad (25)$$

$$i = \frac{UA_L - jT_L - k}{2T_L} \quad (26)$$

Any address UA_L can be expressed using k, j, i . The above formulas say how to calculate k, j, i from a given UA_L . Note that this implies

$$0 \leq k < T_L \quad (27)$$

$$0 \leq j < 2 \quad (28)$$

I prove the theorem below. This is not the most beautiful form but proving $UA_L \leq A_L(UA_L, T_L) < UA_L + 2T_L$ and $\gcd(A_L(UA_L, T_L), T_L) = T_L \wedge A_L(UA_L, T_L)/T_L = 2N + 1$ will be two long proofs with many case splits. I

believe these facts can be seen from the theorem below and stating the longer proofs will not be an advantage for the reader.

$$T_L \in \mathbb{N} \setminus \{0\} \quad (29)$$

$$\wedge \quad UA_L, i, j, k \in \mathbb{N} \quad (30)$$

$$\wedge \quad A_L(UA_L, T_L) = \mathbf{if} \ j = 0 \vee k = 0 \ \mathbf{then} \ (2i + 1)T_L \quad (31)$$

$$\qquad \qquad \qquad \mathbf{else} \ (2(i + 1) + 1)T_L$$

... and here goes the proof:

Proof 3: The Odd-Padding-formula theorem

$$UA_L + (2T_L - ((UA_L + T_L) \bmod 2T_L)) \bmod 2T_L \quad (3.1)$$

$$= \{\text{replace } UA_L \text{ with } 2iT_L + jT_L + k \text{ (23)}\}$$

$$2iT_L + jT_L + k + (2T_L - ((2iT_L + jT_L + k + T_L) \bmod 2T_L)) \bmod 2T_L \quad (3.2)$$

$$= \{\text{simplify}\}$$

$$(2i + j)T_L + k + (2T_L - ((k + (2i + j + 1)T_L) \bmod 2T_L)) \bmod 2T_L \quad (3.3)$$

$$= \{\text{apply (9) because } k < T_L \text{ from (27)}\}$$

$$(2i + j)T_L + k + (2T_L - (k + ((2i + j + 1) \bmod 2)T_L)) \bmod 2T_L \quad (3.4)$$

$$= \{\text{apply (7)}\}$$

$$(2i + j)T_L + k + (2T_L - k - ((j + 1) \bmod 2)T_L) \bmod 2T_L \quad (3.5)$$

$$= \{\text{apply definition of mod (5)}\}$$

$$(2i + j)T_L + k + (2T_L - k - (\mathbf{if} \ j + 1 < 2 \ \mathbf{then} \ j + 1 \ \mathbf{else} \ (j + 1 - 2) \bmod 2)T_L) \bmod 2T_L \quad (3.6)$$

$$= \{\text{move } T_L \text{ into the } \mathbf{if}\}$$

$$(2i + j)T_L + k + (2T_L - k - \mathbf{if} \ j < 1 \ \mathbf{then} \ (j + 1)T_L \ \mathbf{else} \ ((j - 1) \bmod 2)T_L) \bmod 2T_L \quad (3.7)$$

$$= \{\text{note } j \in \{0, 1\} \text{ (28)}\}$$

$$(2i + j)T_L + k + (2T_L - k - \mathbf{if} \ j = 0 \ \mathbf{then} \ T_L \ \mathbf{else} \ (0 \bmod 2)T_L) \bmod 2T_L \quad (3.8)$$

$$= \{\text{move } 2T_L - k \text{ and mod } 2T_L \text{ into the } \mathbf{if}\}$$

$$(2i + j)T_L + k + \mathbf{if} \ j = 0 \ \mathbf{then} \ ((2 - 1)T_L - k) \bmod 2T_L \ \mathbf{else} \ (2T_L - k) \bmod 2T_L \quad (3.9)$$

$$= \{\text{apply (5) because } T_L - k < 2T_L \text{ from (27)}\}$$

$$(2i + j)T_L + k + \mathbf{if} \ j = 0 \ \mathbf{then} \ T_L - k \ \mathbf{else} \ (2T_L - k) \bmod 2T_L \quad (3.10)$$

$$= \{\text{apply definition of mod (5)}\}$$

$$(2i + j)T_L + k + \mathbf{if} \ j = 0 \ \mathbf{then} \ T_L - k \quad (3.11)$$

$$\qquad \qquad \qquad \mathbf{else} \ \mathbf{if} \ 2T_L - k < 2T_L \ \mathbf{then} \ 2T_L - k \ \mathbf{else} \ (2T_L - k - 2T_L) \bmod 2T_L$$

$$= \{\text{rewrite } \mathbf{if} \ \text{with } -k < 0 \Rightarrow k > 0\}$$

$$(2i + j)T_L + k + \mathbf{if} \ j = 0 \ \mathbf{then} \ T_L - k \ \mathbf{else} \ \mathbf{if} \ k > 0 \ \mathbf{then} \ 2T_L - k \ \mathbf{else} \ -k \bmod 2T_L \quad (3.12)$$

$$= \{\text{exchange } \mathbf{if}\text{-cases } k > 0 \text{ to } k = 0 \text{ (recall that } 0 \leq k < T_L)\}$$

$$\begin{aligned}
& (2i + j)T_L + k + \mathbf{if } j = 0 \mathbf{ then } T_L - k \mathbf{ else if } k = 0 \mathbf{ then } 0 \bmod 2T_L \mathbf{ else } 2T_L - k & (3.13) \\
= & \{ \text{move } (2i + j)T_L + k \text{ into the } \mathbf{ifs} \} \\
& \mathbf{if } j = 0 \mathbf{ then } (2i + j)T_L + k + T_L - k & (3.14) \\
& \quad \mathbf{else if } k = 0 \mathbf{ then } (2i + j)T_L + k \mathbf{ else } (2i + j)T_L + k + 2T_L - k \\
= & \{ \text{rewrite } j \in \{0, 1\} \text{ (28) and } k \text{ in the } \mathbf{then} \text{ and } \mathbf{else} \text{ branches} \} \\
& \mathbf{if } j = 0 \mathbf{ then } (2i + 1)T_L \mathbf{ else if } k = 0 \mathbf{ then } (2i + 1)T_L \mathbf{ else } (2(i + 1) + 1)T_L & (3.15) \\
= & \{ \text{collapse } \mathbf{ifs} \text{ to one } \mathbf{if} \text{ using axiom (12)} \} \\
& \mathbf{if } j = 0 \vee k = 0 \mathbf{ then } (2i + 1)T_L \mathbf{ else } (2(i + 1) + 1)T_L & (3.16)
\end{aligned}$$

□

What is the relation between this theorem and the Odd-Padding-algorithm theorem (19)? If you want to use the Odd-Padding-algorithm you need to find an array length which fulfills the assumptions of that theorem. For a cache size $C_S = 2^i, i \in \mathbb{N}$ the formula (22) delivers such an array length. Assume your intended array length is UA_L and you want to use a tile with length T_L and height $T_H = C_S/T_L$ (all length in cache lines with the exception of T_H , of course). Since T_L must divide C_S evenly, you must select a power of two, that is $T_L = 2^j, j \in \mathbb{N}, j \leq i$. The assumptions (16), (17), (18) of the Odd-Padding-algorithm are:

$$\begin{aligned}
A_L &= hT_L \\
C_S &= T_H T_L \\
\gcd(h, T_H) &= 1
\end{aligned}$$

From $C_S = T_H T_L$ and your choices it follows that $T_H = 2^{i-j}$. Then you use the Odd-Padding-formula to calculate A_L . The result is $A_L = (2n + 1)T_L, n \in \mathbb{N}$ — as stated by theorem (31) which I just proved. Therefore, $h = (2n + 1)$, that is, h is odd and does not contain any 2 as prime factor. T_H consists solely of 2s as prime factors. Hence, the requirement $\gcd(h, T_H) = 1$ of the Odd-Padding-algorithm theorem is fulfilled. To sum it up: the Odd-Padding-formula will deliver the right array length if your cache size is a power of two.

Proving the correctness of the formula that aligns array **A** with initial base address adr at a cache line border CL_S would be somewhat similar to proof 3 above.

$$A_{adr}(adr, CL_S) = adr + (CL_S - (adr \bmod CL_S)) \bmod CL_S \quad (32)$$

Moreover, this formula is simpler and somewhat similar to the one proved above. Therefore, I omit that proof.

4.4 The multi-array-access-algorithm theorem

In section 4.2 I proved that the Odd-Padding-algorithm avoids cache interference when used for one array. In this section I prove that there will be no interference even when you use it to access several arrays at once.

This proof and this theorem are very similar to the ones from section 4.2. Actually, if you set the number of arrays, n , to one, then this theorem becomes the one of section 4.2. Nevertheless, I want to give both theorems because the theorem of section 4.2 is confusing enough and the theorem here has even more variables. Therefore, I hope it is helpful for the reader to be able to compare the two theorems and see where they differ.

Here is the theorem I want to prove:

$$adr, r, T_x, T_y, i_p, v_p, x_p, y_p, i_q, v_q, x_q, y_q \in \mathbb{N} \quad (33)$$

$$\wedge h, n, T'_H, T_L \in \mathbb{N} \setminus \{0\} \quad (34)$$

$$\wedge v_p, v_q < n \wedge x_p, x_q < T_L \wedge y_p, y_q < T'_H \quad (35)$$

$$\wedge T_H = nT'_H + r \quad (36)$$

$$\wedge A_L = hT_L \quad (37)$$

$$\wedge C_S = T_H T_L \quad (38)$$

$$\wedge \gcd(h, T_H) = 1 \quad (39)$$

$$\wedge m(T_{adr}(i_p, v_p, x_p, y_p)) = m(T_{adr}(i_q, v_q, x_q, y_q)) \quad (40)$$

\Rightarrow

$$v_p = v_q \wedge x_p = x_q \wedge y_p = y_q$$

Let me explain the new variables. n is the number of arrays accessed at the same time. Since there are T_H tile lines available in the cache, the maximum tile height per array T'_H is reduced to $T'_H = \lfloor T_H/n \rfloor$. The floor function may cause some tile lines to remain unused. r is the number of those unused lines. Note that the reduced height of the tile is expressed in $y_p, y_q < T'_H$ (35).

Here, the address calculation is more complex because this theorem deals with several arrays. Let these arrays be numbered from 0 to $n - 1$ and let v be the number of an array. The reference address — that is the address of array 0 — is adr . The other arrays are stored with a relative distance to that array. The multi-array-access-formula which I prove in section 4.5 will calculate such addresses. The distance is a multiple, i , of the cache size — which is irrelevant, of course — and $((T'_H A_L v) \bmod C_S)$ where v is the number of the array. This gives the new address function T_{adr} :

$$T_{adr}(i, v, x, y) = adr + iC_S + (T'_H A_L v) \bmod C_S + T_y A_L + T_x + y A_L + x \quad (41)$$

The theorem (40) above says: “two array accesses are only mapped to the same line in the cache if they belong to the same array and have the same coordinates in the tile.”

Proof 4: The multi-array-access-algorithm theorem

$$m(T_{adr}(i_p, v_p, x_p, y_p)) = m(T_{adr}(i_q, v_q, x_q, y_q)) \quad (4.1)$$

$$= \{\text{unfold definition of } T_{adr} \text{ (41)}\}$$

$$\begin{aligned} & m(adr + i_p C_S + (T'_H A_L v_p) \bmod C_S + T_y A_L + T_x + y_p A_L + x_p) = \\ & m(adr + i_q C_S + (T'_H A_L v_q) \bmod C_S + T_y A_L + T_x + y_q A_L + x_q) \end{aligned} \quad (4.2)$$

$$= \{\text{unfold definition of } m \text{ (20)}\}$$

$$\begin{aligned} & (adr + i_p C_S + (T'_H A_L v_p) \bmod C_S + T_y A_L + T_x + y_p A_L + x_p) \bmod C_S = \\ & (adr + i_q C_S + (T'_H A_L v_q) \bmod C_S + T_y A_L + T_x + y_q A_L + x_q) \bmod C_S \end{aligned} \quad (4.3)$$

$$= \{\text{apply axiom (7) to get rid of } i_p C_S \text{ and } i_q C_S\}$$

$$\begin{aligned} & (adr + (T'_H A_L v_p) \bmod C_S + T_y A_L + T_x + y_p A_L + x_p) \bmod C_S = \\ & (adr + (T'_H A_L v_q) \bmod C_S + T_y A_L + T_x + y_q A_L + x_q) \bmod C_S \end{aligned} \quad (4.4)$$

$$= \{\text{apply axiom (8) to remove the inner mod } C_S\}$$

$$\begin{aligned} & (adr + T'_H A_L v_p + T_y A_L + T_x + y_p A_L + x_p) \bmod C_S = \\ & (adr + T'_H A_L v_q + T_y A_L + T_x + y_q A_L + x_q) \bmod C_S \end{aligned} \quad (4.5)$$

$$\Rightarrow \{\text{use axiom (10) to remove } adr, T_y A_L, T_x\}$$

$$(T'_H A_L v_p + y_p A_L + x_p) \bmod C_S = (T'_H A_L v_q + y_q A_L + x_q) \bmod C_S \quad (4.6)$$

= {simplify}

$$(x_p + (T'_H v_p + y_p) A_L) \bmod C_S = (x_q + (T'_H v_q + y_q) A_L) \bmod C_S \quad (4.7)$$

= {rewrite $C_S = T_H T_L$ from (38) and $A_L = h T_L$ from (37)}

$$(x_p + (T'_H v_p + y_p) h T_L) \bmod T_H T_L = (x_q + (T'_H v_q + y_q) h T_L) \bmod T_H T_L \quad (4.8)$$

= {use axiom (9) because $x_p, x_q < T_L$ (35)}

$$x_p + ((T'_H v_p + y_p) h \bmod T_H) T_L = x_q + ((T'_H v_q + y_q) h \bmod T_H) T_L \quad (4.9)$$

\Rightarrow {use axiom (11) because $x_p, x_q < T_L$ (35)}

$$x_p = x_q \wedge (T'_H v_p + y_p) h \bmod T_H = (T'_H v_q + y_q) h \bmod T_H \quad (4.10)$$

$$\Rightarrow \left\{ \begin{array}{l} \text{Observe that } T'_H v_p + y_p \text{ and } T'_H v_q + y_q \text{ are smaller than } T_H: \\ \\ T'_H v_p + y_p < T_H \\ = \{T_H = n T'_H + r \text{ from (36)}\} \\ T'_H v_p + y_p < n T'_H + r \\ \Rightarrow \{\text{replace } v_p \text{ by } n - 1 \text{ because } v_p < n \text{ (35)}\} \\ \\ (n - 1) T'_H + y_p < n T'_H + r \\ = \{\text{subtract } (n - 1) T'_H \text{ on both sides because } n > 0 \text{ and } T h' > 0 \text{ (34)}\} \\ y_p < T'_H + r \\ \Rightarrow \{\text{Since } y_p < T'_H \text{ from (35) and } r \geq 0 \text{ from (33)}\} \\ \\ \mathbf{true} \\ \\ \text{The same argument is true for } T'_H v_q + y_q < T_H. \text{ Therefore, I can apply the core theorem (4) from} \\ \text{section 4.1 because } T'_H v_p + y_p, T'_H v_q + y_q < T_H \text{ and } \gcd(h, T_H) = 1 \text{ (39)} \end{array} \right.$$

$$x_p = x_q \wedge (T'_H v_p + y_p) = (T'_H v_q + y_q) \quad (4.11)$$

\Rightarrow {use axiom (11) because $y_p, y_q < T'_H$ (35)}

$$v_p = v_q \wedge x_p = x_q \wedge y_p = y_q \quad (4.12)$$

□

This theorem also holds for just one array. Then n would be 1. v_p and v_q must be 0 because of assumption (35). The conclusion would reduce to $x_p = x_q \wedge y_p = y_q$. Therefore, this theorem subsumes the Odd-Padding-algorithm theorem from section 4.2.

4.5 The multi-array-access-formula theorem

The multi-array-access-function B_{adr} aligns the base address of array \mathbf{B} relative to the base address of the array \mathbf{A} . The difference between the two base addresses modulo C_S is o . To express it more concretely: UB_{adr} is the initial or first possible base address of array \mathbf{B} . A_{adr} is the base address of array \mathbf{A} which is — for

the usage in the multi-array-access-theorem — the first of the n arrays. The relative distance o should be $((T'_H A_L v) \bmod C_S)$ where v is the number of the **B** array, A_L the array length after intra-variable padding and $T'_H = \lfloor T_H/n \rfloor$ the tile height.

The function B_{adr} calculates the next possible start address for the array **B** so that its relative position to array **A** is o modulo C_S .

$$\begin{aligned} B_{adr}(A_{adr}, UB_{adr}, o) = & \mathbf{if} (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S & (42) \\ & \mathbf{then} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S + C_S \\ & \mathbf{else} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S \end{aligned}$$

I want to prove this theorem:

$$A_{adr}, UB_{adr}, o \in \mathbb{N} \quad (43)$$

$$\wedge C_S \in \mathbb{N} \setminus \{0\} \quad (44)$$

$$\wedge o < C_S \quad (45)$$

$$\wedge m(B_{adr}(A_{adr}, UB_{adr}, o) - A_{adr}) = o \quad (46)$$

This theorem says: “if you calculate the base address of array **B** using the B_{adr} function then the distance of the mappings of **A** and **B** in the cache will be o .”

Proof 5: The multi-array-access-formula theorem

$$m(B_{adr}(A_{adr}, UB_{adr}, o) - A_{adr}) \quad (5.1)$$

= {unfold definition of B_{adr} -function (42)}

$$m \left(\left(\begin{array}{l} \mathbf{if} (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S \\ \quad \mathbf{then} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S + C_S \\ \quad \mathbf{else} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S \end{array} \right) - A_{adr} \right) \quad (5.2)$$

= {move $-A_{adr}$ into the **if**}

$$m \left(\begin{array}{l} \mathbf{if} (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S \\ \quad \mathbf{then} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S + C_S - A_{adr} \\ \quad \mathbf{else} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S - A_{adr} \end{array} \right) \quad (5.3)$$

= {unfold the cache mapping function m (20)}

$$\left(\begin{array}{l} \mathbf{if} (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S \\ \quad \mathbf{then} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S + C_S - A_{adr} \\ \quad \mathbf{else} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S - A_{adr} \end{array} \right) \bmod C_S \quad (5.4)$$

= {move $\bmod C_S$ into the **if**}

$$\begin{aligned} & \mathbf{if} (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S \\ & \quad \mathbf{then} (UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S + C_S - A_{adr}) \bmod C_S \\ & \quad \mathbf{else} (UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S - A_{adr}) \bmod C_S \end{aligned} \quad (5.5)$$

= {apply axiom (7) to remove $+C_S$ }

$$\begin{aligned} & \mathbf{if} (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S \\ & \quad \mathbf{then} (UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S - A_{adr}) \bmod C_S \\ & \quad \mathbf{else} (UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S - A_{adr}) \bmod C_S \end{aligned} \quad (5.6)$$

= {well, collapse both cases}

$$\begin{aligned}
& (UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S - A_{adr}) \bmod C_S & (5.7) \\
= & \{\text{Remove the inner modulo functions using axiom (8)}\} \\
& (UB_{adr} - UB_{adr} + (o + A_{adr}) - A_{adr}) \bmod C_S & (5.8) \\
= & \{\text{without words } \dots \} \\
& o \bmod C_S & (5.9) \\
= & \{\text{apply the definition of the modulo function (5) with } o < C_S \text{ from (45)}\} \\
& o & (5.10)
\end{aligned}$$

□

What is the relation between this theorem and the multi-array-access-algorithm? You use this function $B_{adr}(adr, UB_{adr}, (T'_H A_L v) \bmod C_S)$ to calculate the padded base address of the v^{th} array given the base address adr of the 0^{th} array. The theorem proven above then results in:

$$\begin{aligned}
& m(B_{adr}(adr, UB_{adr}, (T'_H A_L v) \bmod C_S) - adr) = (T'_H A_L v) \bmod C_S \\
= & \{\text{unfold the cache mapping function } m \text{ (20)}\} \\
& (B_{adr}(adr, UB_{adr}, (T'_H A_L v) \bmod C_S) - adr) \bmod C_S = (T'_H A_L v) \bmod C_S \\
= & \{\text{mod } C_S \text{ removes some } i \text{ times } C_S \text{ where } i \in \mathbb{N}\} \\
& B_{adr}(adr, UB_{adr}, (T'_H A_L v) \bmod C_S) - adr - iC_S = (T'_H A_L v) \bmod C_S \\
= & \{+adr + iC_S\} \\
& B_{adr}(adr, UB_{adr}, (T'_H A_L v) \bmod C_S) = adr + iC_S + (T'_H A_L v) \bmod C_S
\end{aligned}$$

The right side of that expression is exactly the part of the array base address I assume in the definition of the T_{adr} function (41). To put it in simple words: if you use the formula $B_{adr}(adr, UB_{adr}, (T'_H A_L v) \bmod C_S)$ to calculate the base address of your arrays then your arrays will have the correct base address for the multi-array-access-algorithm.

Additionally, I need to prove $UB_{adr} \leq B_{adr}(A_{adr}, UB_{adr}, o) < UB_{adr} + C_S$. I split this into two proofs and start with $UB_{adr} \leq B_{adr}(A_{adr}, UB_{adr}, o)$.

Proof 6:

$$\begin{aligned}
& UB_{adr} \leq B_{adr}(A_{adr}, UB_{adr}, o) & (6.1) \\
= & \{\text{unfold the definition of the } B_{adr} \text{ function (42)}\} \\
& UB_{adr} \leq \mathbf{if} (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S & \\
& \quad \mathbf{then} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S + C_S & (6.2) \\
& \quad \mathbf{else} UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S
\end{aligned}$$

$$= \left\{ \begin{array}{l} \text{substitute} \\ \\ a = (o + A_{adr}) \bmod C_S \\ b = UB_{adr} \bmod C_S \\ \\ \text{The modulo function returns values which are smaller than its second argument. Therefore, } a < C_S \\ \text{and } b < C_S. \end{array} \right\}$$

$$\begin{aligned}
& UB_{adr} \leq \mathbf{if} \ a < b \ \mathbf{then} \ UB_{adr} - b + a + C_S \ \mathbf{else} \ UB_{adr} - b + a \\
= & \ \{\text{consider the cases individually}\}
\end{aligned} \tag{6.3}$$

Case 1: $a < b$

$$\begin{aligned}
& UB_{adr} \leq \mathbf{if} \ a < b \ \mathbf{then} \ UB_{adr} - b + a + C_S \ \mathbf{else} \ UB_{adr} - b + a \\
= & \ \{\text{remove the if with the assumption of this case}\}
\end{aligned} \tag{6.4}$$

$$\begin{aligned}
& UB_{adr} \leq UB_{adr} - b + a + C_S \\
= & \ \{-UB_{adr} + b\}
\end{aligned} \tag{6.5}$$

$$\begin{aligned}
& b \leq a + C_S \\
= & \ \{\text{Since } b < C_S \text{ and } a \geq 0\}
\end{aligned} \tag{6.6}$$

$$\mathbf{true} \tag{6.7}$$

Case 2: $a \geq b$

$$\begin{aligned}
& UB_{adr} \leq \mathbf{if} \ a < b \ \mathbf{then} \ UB_{adr} - b + a + C_S \ \mathbf{else} \ UB_{adr} - b + a \\
= & \ \{\text{remove the if with the assumption of this case}\}
\end{aligned} \tag{6.8}$$

$$\begin{aligned}
& UB_{adr} \leq UB_{adr} - b + a \\
= & \ \{-UB_{adr} + b\}
\end{aligned} \tag{6.9}$$

$$\begin{aligned}
& b \leq a \\
= & \ \{\text{with the assumption of this case } a \geq b \text{ follows}\}
\end{aligned} \tag{6.10}$$

$$\mathbf{true} \tag{6.11}$$

$$= \ \{\text{from (6.7) and (6.11)}\}$$

$$\mathbf{true} \tag{6.12}$$

□

This theorem ensures that the function B_{adr} will return a padded base address for \mathbf{B} which is greater or equal the initial address UB_{adr} .

Now I prove the other part $B_{adr}(A_{adr}, UB_{adr}, o) < UB_{adr} + C_S$.

Proof 7:

$$B_{adr}(A_{adr}, UB_{adr}, o) < UB_{adr} + C_S \tag{7.1}$$

$$= \ \{\text{unfold the definition of the } B_{adr} \text{ function (42)}\}$$

$$\left(\begin{array}{l} \mathbf{if} \ (o + A_{adr}) \bmod C_S < UB_{adr} \bmod C_S \\ \quad \mathbf{then} \ UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S + C_S \\ \quad \mathbf{else} \ UB_{adr} - UB_{adr} \bmod C_S + (o + A_{adr}) \bmod C_S \end{array} \right) < UB_{adr} + C_S \tag{7.2}$$

$$= \left\{ \begin{array}{l} \text{substitute} \\ \\ a = (o + A_{adr}) \bmod C_S \\ b = UB_{adr} \bmod C_S \\ \\ \text{The modulo function returns values which are smaller than its second argument. Therefore, } a < C_S \\ \text{and } b < C_S. \end{array} \right\}$$

$$\begin{aligned}
& (\mathbf{if } a < b \mathbf{ then } UB_{adr} - b + a + C_S \mathbf{ else } UB_{adr} - b + a) < UB_{adr} + C_S & (7.3) \\
= & \{\text{consider the cases individually}\}
\end{aligned}$$

Case 1: $a < b$

$$\begin{aligned}
& (\mathbf{if } a < b \mathbf{ then } UB_{adr} - b + a + C_S \mathbf{ else } UB_{adr} - b + a) < UB_{adr} + C_S & (7.4) \\
= & \{\text{remove the } \mathbf{if} \text{ with the assumption of this case}\}
\end{aligned}$$

$$\begin{aligned}
& UB_{adr} - b + a + C_S < UB_{adr} + C_S & (7.5) \\
= & \{-UB_{adr} - C_S + b\}
\end{aligned}$$

$$\begin{aligned}
& a < b & (7.6) \\
= & \{\text{this is the assumption of this case}\}
\end{aligned}$$

$$\mathbf{true} \tag{7.7}$$

Case 2: $a \geq b$

$$\begin{aligned}
& (\mathbf{if } a < b \mathbf{ then } UB_{adr} - b + a + C_S \mathbf{ else } UB_{adr} - b + a) < UB_{adr} + C_S & (7.8) \\
= & \{\text{remove the } \mathbf{if} \text{ with the assumption of this case}\}
\end{aligned}$$

$$\begin{aligned}
& UB_{adr} - b + a < UB_{adr} + C_S & (7.9) \\
= & \{-UB_{adr} + b\}
\end{aligned}$$

$$\begin{aligned}
& a < C_S + b & (7.10) \\
= & \{\text{This is true because } a < C_S\}
\end{aligned}$$

$$\mathbf{true} \tag{7.11}$$

$$\begin{aligned}
= & \{\text{from (7.7) and (7.11)}\} \\
& \mathbf{true} & (7.12)
\end{aligned}$$

□

This theorem ensures that the pad added to UB_{adr} is really smaller than C_S . That is: the function (42) returns the next possible base address for array \mathbf{B} .

5 The Tetris Idea

Tetris is a method which avoids cache interference by changing the way the arrays are stored in memory. Unlike padding Tetris requires a much heavier transformation of the memory layout. Tetris does avoid as much cache interference as padding but it is more generally applicable and can handle situations where padding fails (see box 14 on page 49). In fact, I invented Tetris due to the restrictions of padding.

At the moment Tetris is just an idea. It does not come with an algorithm. If Tetris proves itself to be useful, it would be necessary to develop an algorithm in the future. Nonetheless, Tetris can be implemented by hand. This section gives away the fundamental ideas behind Tetris. Whereas Tetris changes much more heavily the way arrays are stored in memory than padding, I believe it is much more intuitive. It is more obvious which data can be accessed without giving rise to cache interference.

Tetris splits large arrays into smaller ones and distributes the small ones in the memory in such a way that several arrays can be accessed without causing cache conflicts.

5.1 Tetris basics

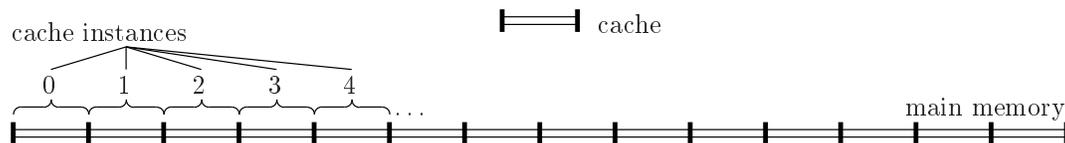
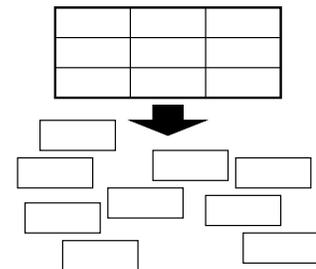
Tetris avoids both types of cache conflicts: self- and cross-interference. It uses two totally different methods to get rid of each of them:

self-interference by splitting large arrays into smaller ones

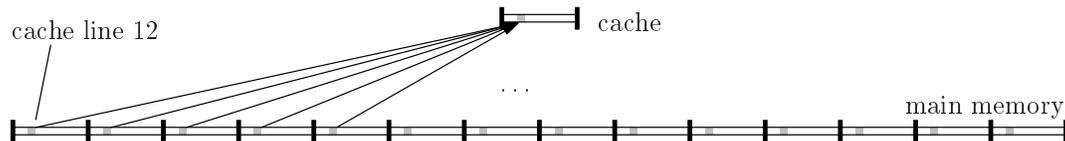
cross-interference by storing the small arrays in a very special way

An array which is bigger than the cache may cause self-interference but an array smaller than the cache cannot cause such a thing. Therefore, Tetris requires you to split your big arrays into small blocks or tiles.

This takes care of self-interference but what if you need to access several arrays? How does Tetris avoid cross-interference then? Imagine how your cache sees your memory: it sees the memory as a lot of chunks of its own size. Well, we all tend to project our own failure onto others, don't we?



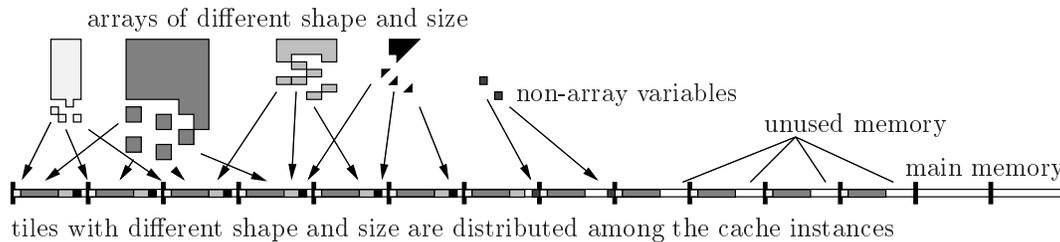
I will call each such a chunk a *cache instance* and I will enumerate them starting with 0. Imagine a cache line 12 cache lines from the beginning of cache instance 0. This cache line will be mapped onto the cache line 12 of your cache. Moreover, cache line 12 of cache instance 1 will also be mapped onto cache line 12 of your cache and so will all the other cache lines 12 of the other cache instances.



To avoid cross-interference Tetris uses the cache instances. It distributes the tiles of all arrays accessed in one loop among the cache instances. This must be done in such a way that

- All tiles of an array use the same cache lines in all cache instances which hold a tile of that array.
- Each instance contains no more than one tile from an array. Cache instances are not required to hold tiles from each array.

- Each tile of an array has the same size. A tile which may consist of several lines is always a multiple of a cache line long but some elements may remain unused.
- All cache instances start at a cache line border.
- The distance between any two cache instances is either zero or a multiple of the cache size.
- All cache instances are exactly as big as the cache. Note: not all space in a cache instance must be used. Some cache lines may just remain unused.

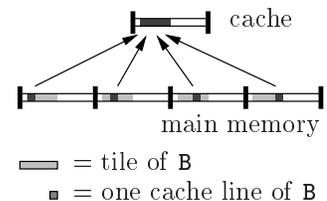


See also figure 5. You may also reserve a cache line for normal (non-array) variables which are accessed in the loop. McKinley and Temam [4] found that such accesses happen frequently in loops. The Tetris memory layout will avoid all cross-interference.

5.2 Tetris access rules

How can you access those arrays? The answer is: on a cache line base! The first thing to observe is: you can access all tiles of one cache instance at the same time. But you can go further: since arrays do not interfere even across instances, you can read a tile from array A from, let's say, instance 3 and a tile from array B from instance 5 in the same loop. But you can go even further: as long as you do not access the same cache line you can read elements of the same array from different instances. For example, assume array B uses cache lines 4 to 12. Then you could read cache line 4 from instance 0, cache line 5 from instance 1, cache line 6 from instance 2 and so on ... just to give one possible example.

Note: you are not forced to use such difficult access patterns. You just can access one whole tile at once and when you finished working on it go to the next one. But at least you have the possibility to use more complex access patterns if you need to.

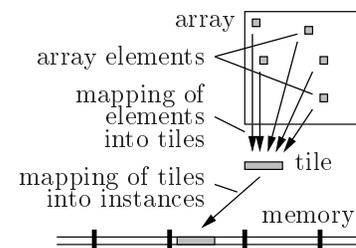


5.3 Tetris mapping rules

There are two mapping functions involved in Tetris (in practise you will probably only deal with one which is the union of these two):

mapping of the tiles in the cache instances This function answers the question: "In which cache instance and at which position inside that instance can I find tile xyz ". Note: all tiles of an array use the same position in all cache instances which store a tile of that array.

mapping of array elements into tiles This function answers the question: "In which tile and in which position inside the tile do I find the array element with coordinates x, y, z, \dots ".



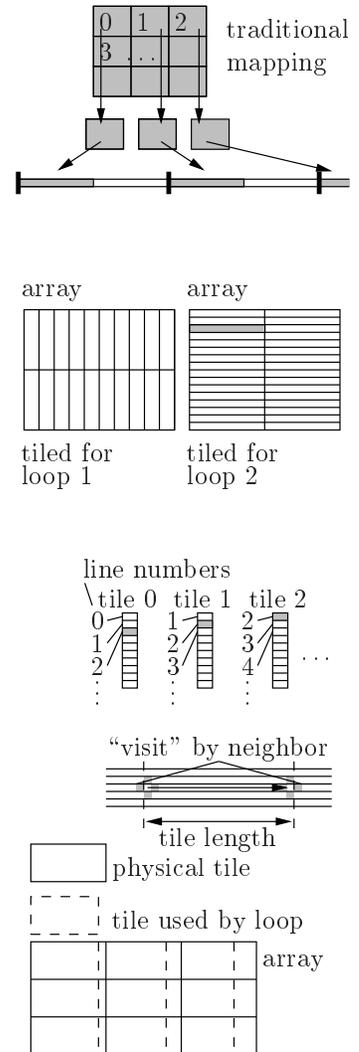
How do you choose these functions? The answer is: there are no restrictions. You are on your own. Most likely you want to have a simple mapping function like the “traditional” padding mapping: the leftmost, topmost block is mapped into tile 0 and that is mapped row-major into instance 0.

Sometimes, you will come across situations where different — perhaps more complex — mapping functions will be an advantage. For multi-dimensional arrays for example you can chop them into planes as you did for padding but it is not longer necessary. You can choose any mapping even multi-dimensional tiles are possible — that is: the elements of a tile come from a volume and not only from a plane or line. If you have a stream-buffer, you may want to arrange the elements of a tile in such a way that the loops access one cache line after the other without skipping some lines.

Another reason for using more complex mapping functions are different access patterns of several loops. Imagine you have two loops, the first requires narrow and very tall tiles, the second needs wide but short tiles. For this example, let the second loop require tiles which are only one line tall. How can you handle this? Make your tiles narrow (one or two cache lines long) and as high as the first loop requires but rotate the lines by one for each tile (see figure at the right and pay special attention to the line numbers). When the second loop accesses line 2, for example, it can use the line 2 from as many tiles as a single tile has lines. The reason is that line 2 in tile 1 occupies the space of line 1 in tile 0 and line 2 in tile 2 occupies the space of line 0 in tile 0 and so on ... Therefore, when loaded into the cache, the cache lines which hold data of line 2 occupy another space in the cache for each tile (as shown in the figure of section 5.2).

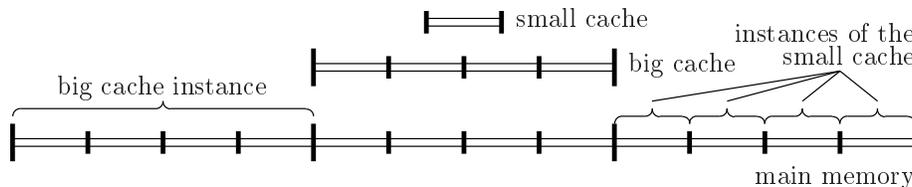
A note to stencil operations (see figure 6): The problem is that these operations have a look at the neighboring tiles. As with padding the solution is not to change the memory layout but to let the tiles, which the loops use, be a cache line smaller than the physical tile. The unused cache line is then used for the glimpse into the neighbor tiles without causing loss of useful data.

To sum it up: You are completely free in choosing how to map your array elements into tiles and which cache instance to choose for which tile but one mapping may be better in your situation than another one.



5.4 Tetris and hierarchical caches

How can you use several caches? I want to explain this with an example. Assume you have two caches and the big cache is four times larger than the small one. That is four instances of the small cache fit into one instance of the big cache.



Consequently, tile line 12 of four consecutive small cache instances will be mapped onto four different places in the big cache but into the same place of the small cache.

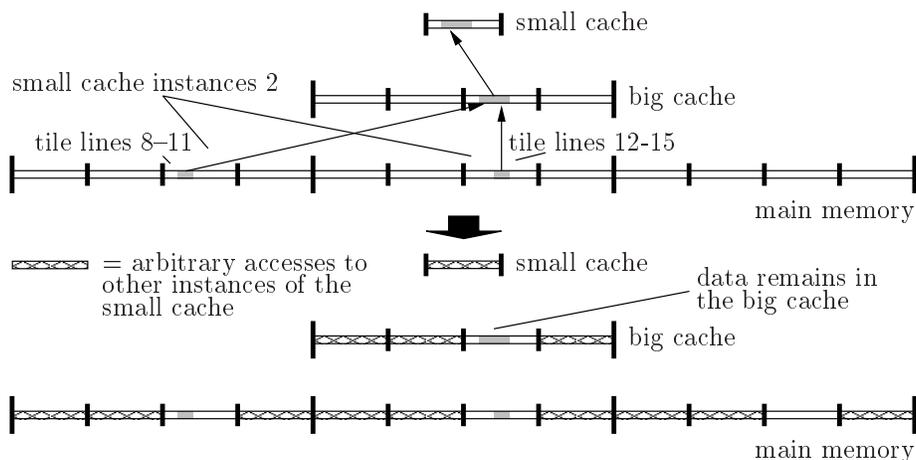
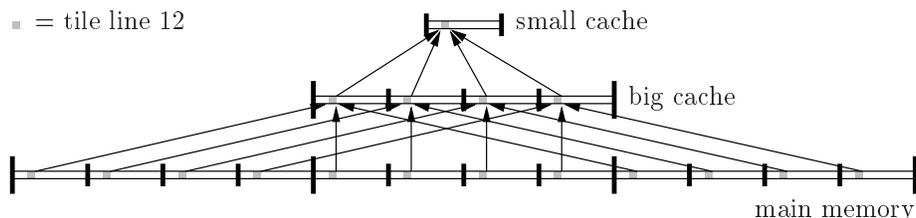


Figure 14: In the top figure cache lines 8 – 11 and 12 – 15 of different instances 2 of the small cache are accessed. In the bottom figure other instances of the small cache are accessed but not cache lines 8 – 15 of small cache instance 2. Despite the fact that these cache lines are kicked out of the small cache, they remain in the big cache. If the program later accesses the very same memory locations of these cache lines again, they will be in the big cache and do not need to be read from the main memory.



When you read or write data from the i^{th} small cache instance within a big cache instance it will be stored in the big cache until you read or write data from/to another i^{th} small cache instance, even when you access small cache instances which are not the i^{th} in between. Let me make an example (see figure 14): assume array **B** occupies cache lines 8 to 15 in the small cache instances. Now, you access cache lines 8 to 11 in the small cache instance 2 within the big cache instance 0 and cache lines 12 to 15 in the small cache instance 2 within the big cache instance 1. Then you access all cache lines in all other small cache instances with the exception of all instances 2. Finally, you re-access small cache instance 2 within the big cache instance 0 and 1 to read cache lines 8 to 11 and 12 to 15 respectively. These cache lines will still be in the big cache but not in the small cache.

5.5 Tetris and multiple loops

What do you need to do if your program has several loops which access arrays? Basically, there are two problems:

1. The loops access different arrays but also share some arrays. You will probably not have a problem if the loops do not share at least one array.
2. The loops access the same array but have different access requirements. That is: they have different access patterns or need different tile sizes or shapes.

In section 5.3 I discuss an example which handles problem 2. The trick is to find a smart mapping and to reorder the loops appropriately. If this does not help, you need to have a look at section 5.6.

Box 14: Tetris versus Padding

Both Tetris and padding can completely avoid cache interference. The real difference between them is that padding can only handle very restricted situations whereas Tetris can handle most situations.

	Tetris	padding
intuitive	✓	○
algorithm available / easy to implement	○	✓
changes array layout only a bit	○	✓
tiles with odd sizes and odd base address	✓	✓
multi-dimensional arrays	✓	✓
several arrays in the same loop	✓	?
different array sizes	✓	○
different tile sizes	✓	○
different access patterns	✓	○
several loops	✓	?
hierarchical caches	✓	?
arrays with non-rectangular shape	✓	○
non-array variables	✓	○
wastes memory	✓	✓

The question marks say: “it is possible but very restricted.” Let me briefly discuss the points:

intuitive That Tetris is more intuitive is my subjective opinion.

algorithm available / easy to implement Section 3 presents several padding algorithms.

changes array layout only a bit Padding just makes the lines of the arrays longer and moves the base addresses around. Tetris requires a total different array layout (see section 5.1).

tiles with odd sizes and odd base address For padding see sections 3.4 and 3.5. For Tetris see section 5.3.

several arrays in the same loop Padding can only handle different arrays if they have the same sizes and are accessed very similarly (see sections 3.6 and 3.7).

different array sizes Tetris can work with different array sizes (see sections 5.1 and 5.3). Padding does not accept different array sizes (see section 3.6).

different tile sizes Tetris can work with different tile sizes (see sections 5.1 and 5.3). Padding does not accept different tile sizes (see section 3.6).

different access patterns Tetris accepts different access patterns (see section 5.2). Padding requires very similar access patterns for all arrays (see section 3.7).

several loops Tetris can handle multiple loops (see section 5.5). Padding can handle several loops only if the same arrays are accessed and the same tile size is used.

hierarchical caches For Tetris see section 5.4. Hierarchical padding works only if the big tile is as long as the small tile (see section 3.9).

arrays with non-rectangular shape Tetris can handle non-rectangular arrays when you can find an appropriate mapping function (see section 5.3). For padding arrays must be rectangular.

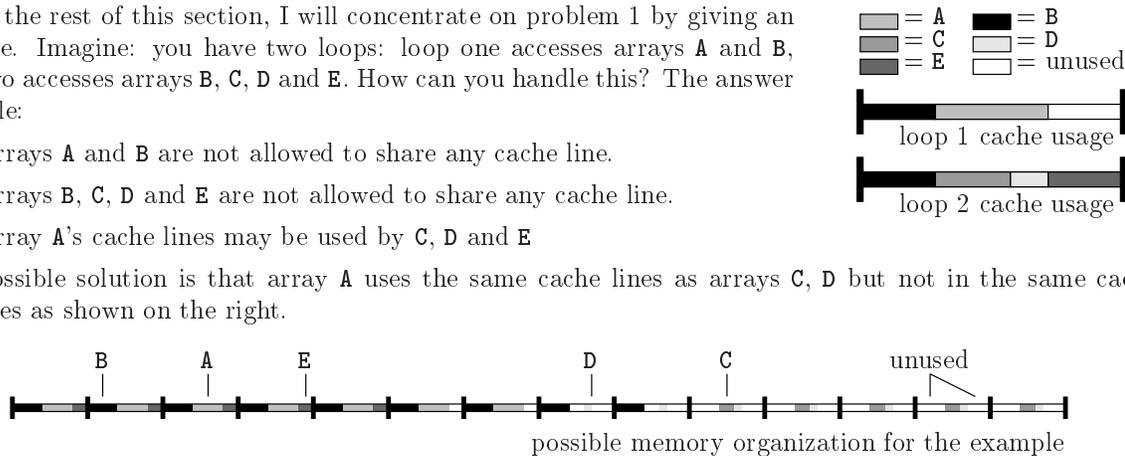
non-array variables For Tetris see section 5.1. Padding can only pad array variables.

wastes memory Which technique wastes more memory depends on your program and your padding algorithm and your implementation of Tetris.

For the rest of this section, I will concentrate on problem 1 by giving an example. Imagine: you have two loops: loop one accesses arrays **A** and **B**, loop two accesses arrays **B**, **C**, **D** and **E**. How can you handle this? The answer is simple:

- Arrays **A** and **B** are not allowed to share any cache line.
- Arrays **B**, **C**, **D** and **E** are not allowed to share any cache line.
- Array **A**'s cache lines may be used by **C**, **D** and **E**

One possible solution is that array **A** uses the same cache lines as arrays **C**, **D** but not in the same cache instances as shown on the right.



Usually, there will be several solutions for problem 1. You may want to choose the one that fulfills all requirements (tile size etc.), makes best use of your cache and does not waste too much memory.

5.6 Tetris Copy-Buffer

Tetris gives you much possibilities to cope with most cache problems but there are still situations where Tetris fails. Such situations arise when you need to deal with too many access patterns, too many loops or/and too many arrays so that you can not find a Tetris memory layout which takes care of every thing. This is the worst case of Tetris. I can think of two problems:

1. You need to change the way the array is stored in memory.
2. A loop accesses several arrays and at least one array uses the same cache lines as one or more other arrays. That is, the arrays would overlap in the cache. (I assume you have stored all arrays in the “Tetris-way”, that is: they are chopped in small pieces and are distributed into cache instances.)

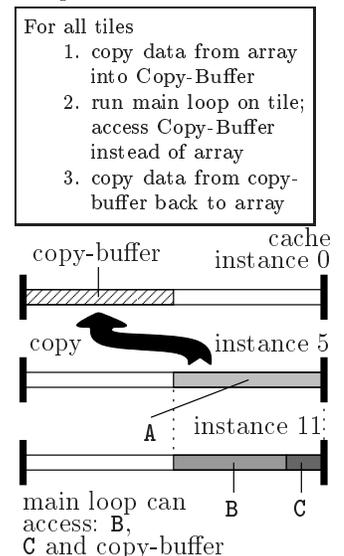
If this happens you are forced to copy data to avoid cache conflicts. Tetris comes with an own copying technique — I call it *Copy-Buffer*⁴. That is, even in the worst case Tetris does not perform so bad. Let me first describe copying in general and afterwards the Tetris Copy-Buffer idea.

Basically, copying works like this: You have a loop which does the “real” work on the tile. I will call this loop the main loop. If the main loop reads data from **A**, you copy this data into a buffer before your enter the main loop. Then the main loop works on the tile but instead of accessing array **A** directly it accesses its copy in the buffer. When the main loop finished working on the tile, you copy the data from the buffer back into the original array **A** if the main loop has changed this data. See box at the right.

Now, I describe the Tetris Copy-Buffer technique. The basic idea is to use the Tetris memory layout to

- avoid all cache interference
- keep the cache lines of the Copy-Buffer all the time in the cache
- avoid to copy many arrays

This makes Tetris Copy-Buffer much faster than usual copying as described by Temam et al. [8]. Usual copying either copies only some of the arrays accessed and must accept cache interference during the execution of the main



⁴Note: if you copy to permanently change the way the array is stored in memory then Tetris Copy-Buffer is not applicable but read on: you can still learn how to avoid cache interference while copying.

loop or it copies all arrays but suffers heavily from interference while copying. Usual copying must reload at least some of the cache lines of the Copy-Buffer because they get kicked out due to cache interference.

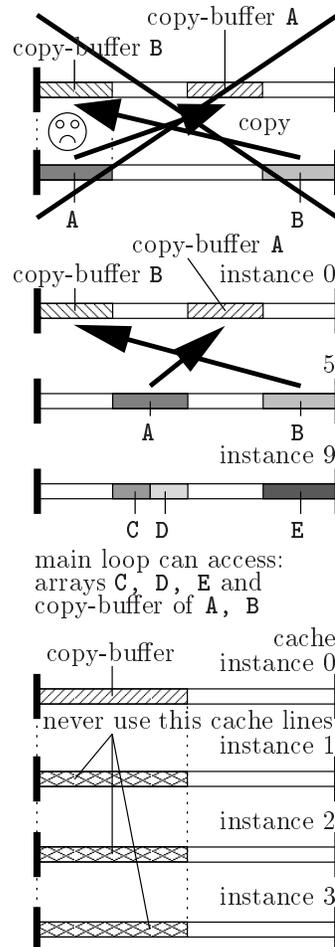
The trick on Tetris Copy-Buffer is the way the location of the buffer is chosen:

- The buffer is never moved to another location.
- The cache lines of the buffer do not cause any interference. That is, no other array whether copied or not uses the same cache lines in another cache instance.

Here are the detailed rules for Tetris Copy-Buffer:

- There must be enough space in the cache to hold the Copy-Buffer and all other arrays and variables you want to access in the main loop. If this is not the case, e. g. the Copy-Buffer requires 50% of the cache and another array requires 80% of the cache, consider the possibility that you have chosen tile sizes, that are too big.
- The Copy-Buffer must occupy tile lines which are not used by any array or variable which is accessed in the main loop or by a copy operation. That is, the main loop can access the Copy-Buffer without causing cache interference and no copy operation will throw out the cache lines of a Copy-Buffer.
- The Copy-Buffer for an array is always in the same cache instance. Even when you move from one tile to the next, you must not move the buffer. If you move the buffer, the cache lines of the old buffer location will be thrown out of the cache and the lines of the new buffer will be loaded. This causes unnecessary delays.

You may want to know how much better the Tetris Copy-Buffer is compared to usual copying. The answer is: it depends entirely on how much cache interference are caused by usual copying. The point here is that any argument about cache interference in general involves so many parameters that it would not be scientific.



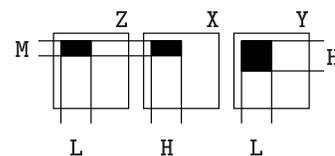
5.7 Tetris matrix multiplication example

The previous sections describe the Tetris idea from an abstract point of view. In this section I give a concrete example. I will show an implementation of the matrix multiplication algorithm using the Tetris data layout. I further describe how I developed this program. I present the Tetris matrix multiplication example in C code because it involves some difficult pointer calculations which are not easily represented in an abstract syntax.

I decided to use the very same tiled matrix multiplication algorithm⁵ which is shown in figure 1. For simplicity, I restricted all arrays to be square and have the same size. These are the steps which I took to develop the program:

1. I started with analyzing the algorithm. Especially, the parameters which determine the tile size are important. I named the length of the tile for the Y array L and its height H. The tile of the X array must have H as its length, that is: it must be as long as the Y-tile is heigh. The length

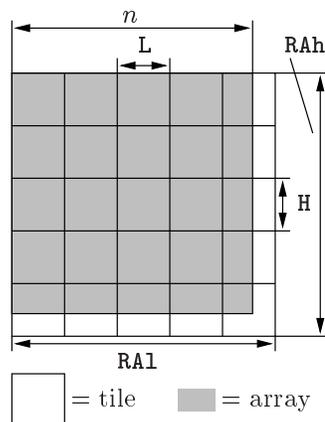
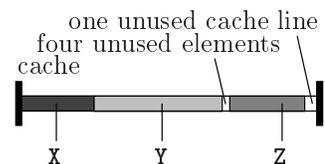
C_S	= 128 cache lines
CL_S	= 8 words
L, H	= 22 words
M	= 12 words
Y-tile	uses 61 cache lines
X, Y-tiles	use 33 cache lines each



⁵Mathematicians know faster ways to multiply matrices. I use this simple algorithm because I focus only on cache issues.

of the tile of the **Z** array must be **L**, that is: it must be as long as the **Y**-tile. I chose **M** as the height of both the **X** and the **Z**-tile. I figured out it would be the best to choose a large and square tile for **Y**. The tiles of the **X** and **Z** arrays could be smaller.

2. I decided how the arrays should be distributed into cache instances. I assigned half of the cache to each of the other arrays and a quarter of the cache to the **X** and **Z** arrays. Since accesses to **X** do not occur in the inner loop and are relatively seldom it is unnecessary to reserve a part of the cache for it. Using a quarter of the cache will actually slow down the Tetris program because the **Y**-tile is smaller than necessary but I thought it is instructive to show that Tetris can handle this. A further speed increase would be possible by reserving only one line of length **L** for the **Z**-tile but that would result in too much unused memory. I would need a whole cache instance for each line of length **L** to store the **Z** array.
3. Given this layout I calculated the tile size in elements. I use a Cray T3E computer with DEC Alpha 21164 processors. The top level cache has 1024 words and the largest cache line size is 8 words. This computer has stream buffers so that I wanted to read as many consecutive cache lines as possible to make best use of them. I chose **L**, **H** = 22. This gives the next smaller square which fits into half of the cache. The remaining cache lines are best used by choosing **M** = 12.



4. Then, I created the mapping functions for each array (see section 5.3). A mapping function calculates the address of an element from its **x**, **y** coordinates. The functions must, of course, honor the Tetris memory layout, that is the fact that the arrays are distributed into several cache instances. The part of these functions which is multiplied by $CS * CLS$ calculates the address of the cache instance and the right part the address of the element within the cache instance. Since the algorithm access the elements in row-major order, I stored the elements in row-major order within the cache instances, leaving no space between the tile lines. This way I can utilize the stream buffers best. **XADR**, **YADR** and **ZADR** are the mapping functions for the arrays **X**, **Y** and **Z** respectively:

```
#define XADR(y,x) (X + ( (y / M)*RAh + (x / H) )*(CS*CLS) + (y % M)*H + (x % H))
#define YADR(y,x) (Y + ( (y / H)*RA1 + (x / L) )*(CS*CLS) + (y % H)*L + (x % L))
#define ZADR(y,x) (Z + ( (y / M)*RA1 + (x / L) )*(CS*CLS) + (y % M)*L + (x % L))
```

X, **Y** and **Z** are the base addresses of the arrays. $CS * CLS$ is the size of the cache in elements and, therefore, the size of the cache instances. **RA1** is the length of the array in whole multiples of **L**. For example, let **L** be 22 then for a **Y** array with length 100, **RA1** would be 5 since $5 * 22$ is the next greater than 100 multiple of 22. **RAh** is the height of the array in multiples of **H**.

5. Next, I wrote the initialization of the program as shown in figure 15. I tested this part very carefully to ensure that the data was stored in the right way. The initialization of the arrays is not show. I simply used the above mapping functions to calculate the address of each element.
6. Afterwards, I wrote the first version of the matrix multiplication and tested its correctness. I still used the mapping functions to calculate the addresses of the elements in the inner loop.
7. Finally, I optimized the matrix multiplication by removing the expensive address calculation from the inner loop. This resulted in the code shown in figure 16. It is possible to remove the address calculation completely from the matrix multiplication by replacing it through some simple additions and a bunch of pointer variables but it will not result in significant improved speed.

```

#define CLS 8 /* length of a cache line in elements */
#define CS 128 /* size of the cache in cache lines */
#define L 22 /* length of Y and Z-tile in elements */
#define H 22 /* height of Y-tile and length of X-tile */
#define M 12 /* height of X and Z-tile */
#define WORD (sizeof(double)) /* length of a word in bytes */

#define MYMIN(n,m) ((n) < (m) ? (n) : (m)) /* minimum of n and m */
#define CEIL(v,s) ((v) + ( (s) - ((v) % (s)) ) % (s)) /* round v to next multiple of s */

double *Mtx; /* pointer to the allocated memory */
int RAL; /* real array length (in multiples of L) */
int RAh; /* real array height (in multiples of H) */
double *X, *Y, *Z; /* array base addresses */
int jj, kk, i, k, j; /* index variables */
double *YY, *YYY, *XX, *ZZ, *ZZZ; /* auxiliary pointers */
double r; /* auxiliary variable to hold element of X */

/* calculate real array length and height */
RAL = CEIL(n,L) / L;
RAh = CEIL(n,H) / H;

/* how many cache instances do I need? */
i = RAL * RAh; /* number of cache instances used by Y */
j = RAh * CEIL(n,M) / M; /* number of cache instances used by X and Z */
if(j > i) i = j; /* which one uses more cache instances? */

/* allocate memory */
Mtx = malloc( i*( CS * CLS + CLS -1 ) * WORD );
assert( Mtx );

/* calculate base addresses i. e. offsets into the first cache instance */
X = (double *) CEIL((long) Mtx ,CLS);
Y = X + CEIL((long) (H*M), CLS);
Z = Y + CEIL((long) (L*H), CLS);

if( (Z + CEIL((long) (L*M), CLS)) - X > (long) CS * CLS)
{
    printf("The tiles exceed the size of your cache!\n");
    exit(EXIT_FAILURE);
}

... initialize matrices (not shown) ...

```

Figure 15: The initialization and address calculation part of the Tetris matrix multiplication.

```

/* matrix multiplication */
for( kk = 0; kk < n; kk += H )
{
  for( jj = 0; jj < n; jj += L )
  {
    for( i = 0; i < n; i++ )
    {
      XX = X + ( (i / M)*RAh + (kk / H) )*(CS*CLS) + (i % M)*H;
      YY = Y + ( (kk / H)*RAL + (jj / L) )*(CS*CLS);
      ZZ = Z + ( (i / M)*RAL + (jj / L) )*(CS*CLS) + (i % M)*L;
      for( k = kk; k < MYMIN( kk + H, n ); k++ )
      {
        YYY = YY;
        r = *(XX++);
        ZZZ = ZZ;
        for( j = jj; j < MYMIN( jj + L, n ); j++ )
        {
          *(ZZZ++) += r * *(YYY++);
        }
        YY += L;
      }
    }
  }
}

```

Figure 16: The main loops of the Tetris matrix multiplication.

Besides several mistakes with the pointer calculation, I had to fight two main problems: the Cray compiler “optimized” my loops when I run it in high optimization levels and I had to align my code by hand. That is: I had to insert some no-op instructions before the matrix multiplication loops to gain maximum performance.

5.8 Tetris — towards an algorithm

In this section I discuss some of the issues which may arise if someone wants to find a Tetris algorithm. Most of the readers may want to skip this section. This is not a complete list of points which must be considered for a Tetris algorithm; it is more a collection of some thoughts about it.

Tetris was born to be used in compilers. When I saw the restrictions of padding, it was clear to me that padding is too weak to be used by compilers, so I thought of something more powerful and came up with Tetris. Since Tetris changes the data layout heavily, it is certainly easier to use with a programming language in which the array layout is unknown to the programmer.

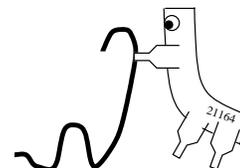
To get the best results when applying Tetris, all possibilities — like mapping functions, loop changings, tile sizes and so on — must be considered. Most likely, this requires human creativity and can not be done by a computer. Nevertheless, it should be possible to find an algorithm which can handle most cases with very good results by applying some heuristics. Here are some points which such an algorithm may need to take into consideration:

- Which variables are accessed in the inner loops?
- Which variables are shared by several loops?
- Which restrictions/dependencies exist between loops and tile sizes?

Box 15: The myth of the transparent cache

For more than a decade scientists all over the world have been studying how to make best use of the cache — without much success. The hardware people promised a transparent cache. So far they have not kept their promise. Allow me to make some philosophic comments about the hardware issue.

From the point of view of a hardware designer, Tetris is just a software simulation of a fully addressable fast memory buffer. Tetris abuses the cache to play the roll of this buffer. Tetris is a comparatively difficult replacement for such a buffer. Hardware designers seem to dream of improving the speed of programs with hardware which does not need any software support, like caches. Should Tetris ever become wide spread used — what I do not know — this dream would have come to its worst end. Tetris represents a huge software effort to circumvent a fully automated hardware feature — the cache.



Is there no better solution? First, we need some or several kinds of fast and small, close to the processor memory. But is a cache the only thing we can think of? I believe, the interference problem with arrays — the very one against padding and Tetris fight — is an intrinsic cache problem, caused by the mapping function. I do not say that having a fully addressable fast memory buffer is the ultimate solution to the interference problem but I say that this problem will not be solved with a cache. I believe that too many people concentrate on caches instead of searching for other solutions.

- How should the algorithm select a tile size?
- How to tile the loops? Note: it may sometimes be necessary to tile loops differently than for usual tiling because Tetris permits — with some restrictions — to access tiles which are distributed among several cache instances. See the stencil operation and the two-loops example discussed in section 5.3.
- How should the algorithm find a mapping function? That is, how should the algorithm map tiles to cache instances and how to map the elements of the array into the tiles? Section 5.3 has an example which shows that it is necessary to be a bit creative here; especially if the algorithm deals with several distinct loops.
- How should the algorithm avoid wasting too much memory?
- When should the algorithm copy data?
- How should the algorithm handle multi-dimensional arrays?
- Which non-array variables should be stored using Tetris?
- When does hierarchical tiling make sense? That is: when does it improve the program speed — my hierarchical tiled matrix multiplication slows down the program — or when does hierarchical tiling result in other advantages?
- How should the algorithm support other hardware features like virtual mapped caches or stream buffers?

The best approach towards an algorithm would probably be to first gain experience by applying Tetris to a number of programs by hand. Afterwards, an algorithm could be developed which ignores most of the above questions, for example by choosing the same tile size for all tiles, by not considering hierarchical tiling and by ignoring the Copy-Buffer technique. Later the algorithm could be extended to handle more cases and to be more efficient. Well, so much for theory ...

6 The Experiments

Basically, there are three methods which are very strong in avoiding cache conflicts: padding, copying and Tetris. Unfortunately, it is hard to argue which one is better. Padding can not handle all situations but it is easier to implement as copying and Tetris. In general, I can not say how much a certain technique improves a program. It depends on the program and the hardware.

In this section I try all methods on a certain program (matrix multiplication) and on a certain hardware (Cray T3E) to compare them. Box 16 describes the programs and algorithms in detail. Box 17 describes the hardware and how I made the experiments. Box 18 explains why I have chosen matrix multiplication. Box 19 tries to answer the question which method is the fastest. Figures 17 and 18 show the results of the experiments.

6.1 Discussion of the graphs

In this section I try to explain why the curves of figures 17 and 18 look the way they look. Note that I can not open the cache and see what is going on there. Therefore, the explanations given here are my best guess and not a general truce.

Is there always no speed difference between copying, padding and Tetris?

You do not see a big difference between the execution times of the copying, padding, Tetris-Copy-Buffer and Tetris programs. The reason is that the matrix multiplication can reuse data a lot and causes relatively few loads from main memory after the program has been tiled. If I had used the stencil operation from figure 6 — which reuses a once loaded element just three times and accesses the memory much heavier — the difference between the methods would be much more significant.

Are the tiny differences between copying, padding and Tetris meaningful?

Yes. The differences are not just a measurement error. These tiny differences in the execution time are actually cross-interference or overhead from copying. For example: Tetris is always faster than Tetris-Copy-Buffer which has some overhead from copying.

Are the optimized programs unaffected by Stream Buffer issues?

No. When you compare the enlargements of the two graphs, you will see that all fast programs are one to four nano-seconds slower without Stream Buffers than with them. Moreover, I programmed the Tetris version Stream Buffer friendly, therefore, it loses more than padding and copying when the stream buffers are switched off.

Why is Tetris slower than padding and copying without Stream Buffers?

The padded program suffers from cross-interference but there is not much interference in the matrix multiplication. The copying program suffers from the copying overhead but that is small because there is much reuse in the matrix multiplication algorithm. Nevertheless, both programs should be slower than Tetris and Tetris-Copy-Buffer even without Stream Buffer support. The point here is that the Tetris program use a smaller tile size which slows it down.

Should not the padded program be faster than the copying program?

Theoretically, the answer should be yes. I have no explanation for the fact that both programs run at the same speed.

Why are the copying, padding and Tetris curves not completely even?

When the user array length is not an exact multiple of a tile size then some tiles are only partly used. The loop overhead is higher for partly used tiles than for full tiles. This causes the jittering. The up and down becomes smaller for larger array sizes because larger arrays have much more full tiles than partly used tiles.

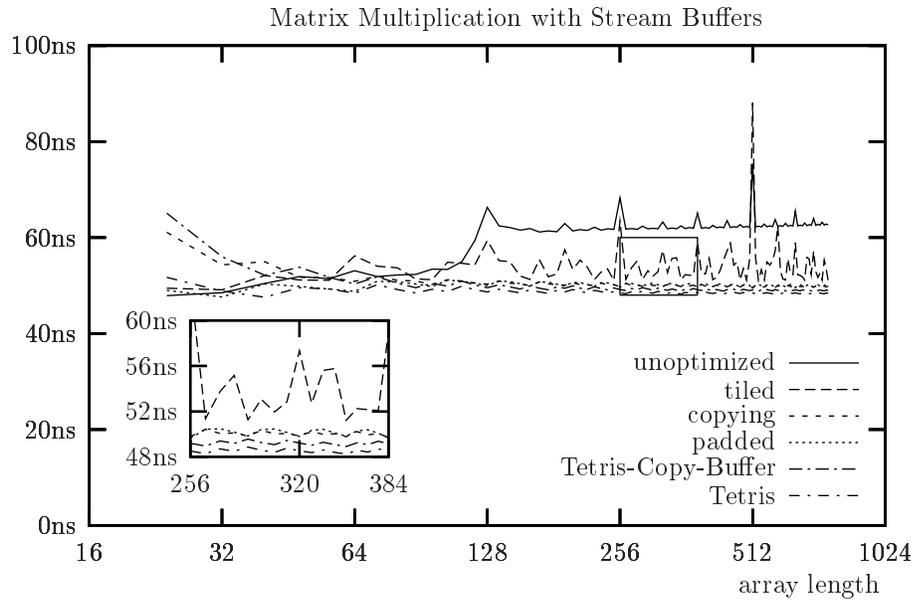


Figure 17: This plot shows how much time the different programs spent for the matrix multiplication depending on the array length. The time is divided by the cubic array length. For your convenience I enlarged the small box in the graph.

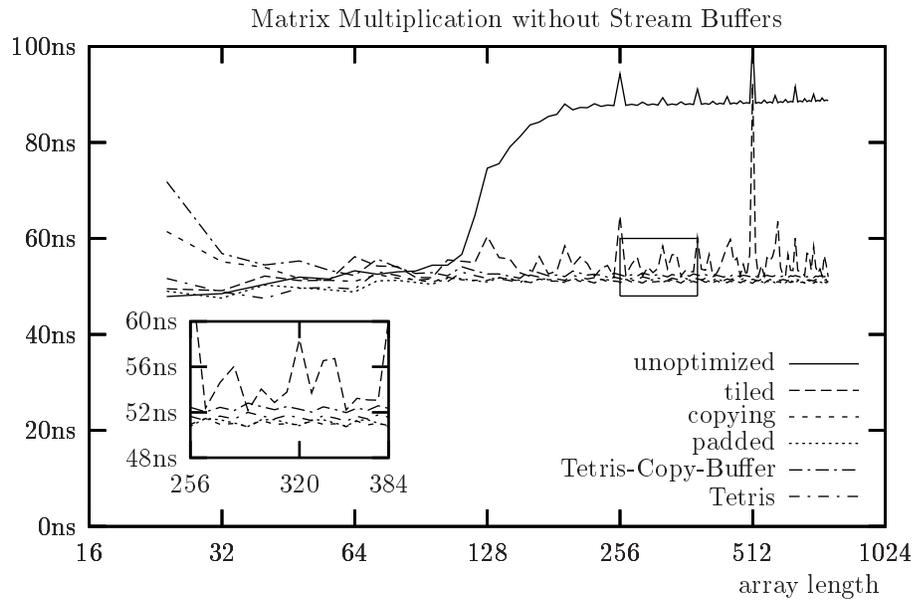


Figure 18: Since you may not have Stream Buffers I made the experiment from figure 17 again but switched off the Stream Buffers. Again, I enlarged a part of the graph.

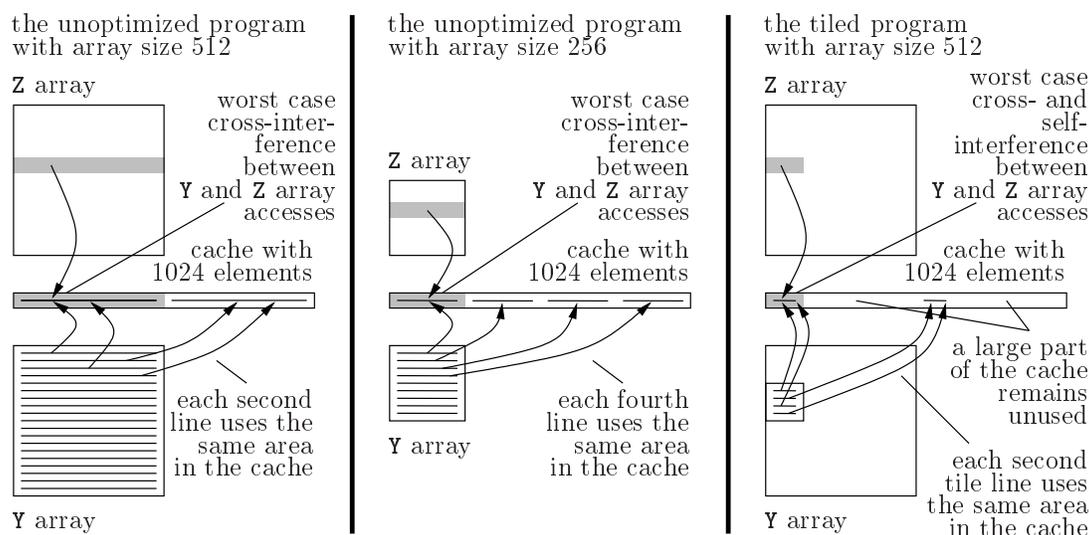


Figure 19: Have a look at figure 1: in the k -loop, one line of the Z -array is reused while each line of the Y -array is loaded. At array length 512, exactly two lines fit into the 1024 element cache. The unoptimized program uses the same cache lines for the array lines 0, 2, 4, 6, 8, ... of the Y -array. This is not a problem but, by chance, the single line of the Z -array uses also the very same cache lines as this lines of the Y -array (left figure). Actually, element i of the Y -array line uses the same cache line as the element i of the Z -array line. This causes cache thrashing — the cache lines of the Y and Z -arrays throw each other out of the cache. This slows down the program and causes the peak in the unoptimized curve at array length 512.

The middle figure shows the situation for array length 256. Four lines of the array fit into the cache but only one causes cache thrashing. Therefore, the cache thrashing slows down the program only by half of the amount of time as it has slow down the program for array length 512. So the peak in the curve for array length 256 is only half as high as for 512. For array length 128 the peak is only a quarter as high as for 512 because eight lines fit into the cache and only one of them causes this worst case cross-interference.

For the tiled program the peak in it's curve at array length 512 is even higher. The tiled program does not use the Stream Buffers as effective as the unoptimized program does, instead it hopes the cache is able to hold all elements of the tile. The tiled program suffers from the same cross-interference as the unoptimized program but in addition it makes bad use of the cache (right figure).

Since every second tile line uses the same cache lines and a tile is only 32 elements long, only a tiny part of the cache gets used. When the tiled program tries to reuse the first tile line, the line has long been thrown out of the cache. The effect is not so dramatic for array length 256 and 128 due to the secondary cache. This twelve times larger 3-way cache with it's random replacement strategy can then hold enough cache lines of the tile to make reuse possible.

Box 16: The sample programs

All programs which I use in the plots implement the matrix multiplication algorithm shown in figure 1 on page 4. I choose all arrays to have the same square sizes. For example, when the plot says array length 128 then all arrays have a user array length of 128 elements and a height of 128 lines. All programs are written in C. I use the optimization `-O1` of the Cray C-compiler for all programs. Furthermore, I tried to make all programs — with the exception of the Tetris program — as fast as possible by choosing the best tile sizes, access patterns and by making the programs stream buffer friendly if possible.

unoptimized The unoptimized program is the very code shown in figure 1 (a). This program does not implement any optimizations like padding or copying.

tiled The tiled program is the code shown in figure 1 (b). This program does not implement any other optimizations like padding or copying — just simple tiling. The tile size — **B** in figure 1 — is 32. The arrays have exactly the size $A_L \times A_L$.

copying The copying program implements the code shown in figure 1 (b) but it copies the tile of the **Y**-array into a continuous buffer before entering the **i**-loop. The program uses a 32×32 element tile for the **Y**-array so that the whole cache is used. Lam et al. [3] found that copying the **Z**-array as well produces more overhead as benefit.

padded The padded program uses the code shown in figure 1 (b). The length of array **Y** is padded to avoid self-interference. I use the Odd-Padding algorithm but note that the choice of the padding algorithm does not influence the number of the remaining cache conflicts and, therefore, does not influence the speed of this program. Arrays **X** and **Z** can not cause self-interference because it is only one line accessed in the inner loop and there is no reuse of that line in the **i** and **j**-loops. To avoid worst case cross-interference between arrays **Y** and **Z** the base address of array **Y** is padded so that there is always a relative distance of at least 16 elements between these arrays. The tile size — **B** in figure 1 — is 32. Experiments have shown that a smaller tile size slows down the program.

Tetris The Tetris program code is shown in figures 15 and 16 on page 54. I use the same parameters as described in section 5.7. Especially the **Y**-tile size is 22×22 and the **X** and **Z**-tile size is 22×12 . This choice is not optimal but I wanted to show that Tetris can handle three arrays with different tile sizes and access patterns. Section 5.7 explains how to improve this program.

Tetris-Copy-Buffer The Copy-Buffer version of Tetris is basically the same program as shown in figures 15 and 16. It also uses the same parameters as the Tetris program. I deliberately changed the array layout so that the **Y**-array uses the same cache lines as the **X** and **Z**-arrays — in different cache instances, of course. Therefore, I need to use the Copy-Buffer technique (see section 5.6) to copy the **Y**-tile into a buffer before the **i**-loop. The copying program and the Tetris-Copy-Buffer program can be compared because they copy the same amount of data during the whole execution of the programs despite the fact that the tile sizes are different. The different tile sizes do not affect the overhead caused by copying, instead they somewhat speed up the main work loop of the copying program.

I had problems with the code alignment by all programs but some programs seem to be more sensible to it than others. Bad code alignment sometimes doubled the execution time.

Box 17: The experimental setup

The measurements for figures 17 and 18 are made on a Cray T3E, using the programs described in box 16. The T3E has DEC Alpha 21164 processors. The graphs show the execution time of the matrix multiplication divided by $(UA_L)^3$. That is: I normalized the graphs using the function $(UA_L)^3$. The time shown is the time needed by one processor for executing the matrix multiplication loops. The initialization of the programs and matrices is not measured. For array lengths between 24 and 768 elements, I took one sample for each multiple of 8 array length. Note that the x-axis which shows the user array length UA_L in elements has a logarithmic scale.

The DEC Alpha 21164 processor has two levels of caches. The first cache (data cache) is a direct mapped 1024 element cache with a cache line size of 4 elements. The second cache is a 3-way 12288 element cache with a cache line length of 8 elements. Which cache line out of a three-cache-line-set gets replaced is chosen randomly. One element is 8 bytes long which is the size of `double`.

The Cray T3E has six data Stream Buffers which work as replacement for the third cache level. These stream buffers automatically detect accesses to consecutive memory locations and start prefetching those cache lines. For figure 18 I switched the Stream Buffers off by setting the environment variable `SCACHE_D_STREAMS` to 0.

Box 18: Why do I use matrix multiplication?

Here are some points which explain why matrix multiplication is a good choice:

- Matrix multiplication is simple and wide known.
- It is a famous example for cache experiments.
- It has much reuse which is important to show cache effects.
- It accesses several arrays so that I can show that Tetris can handle them all.
- It has different access patterns so that padding can not avoid all cache interference. That is: padding should not be as fast as Tetris.
- I can implement matrix multiplication using different tile sizes which proofs that Tetris can handle this, too.

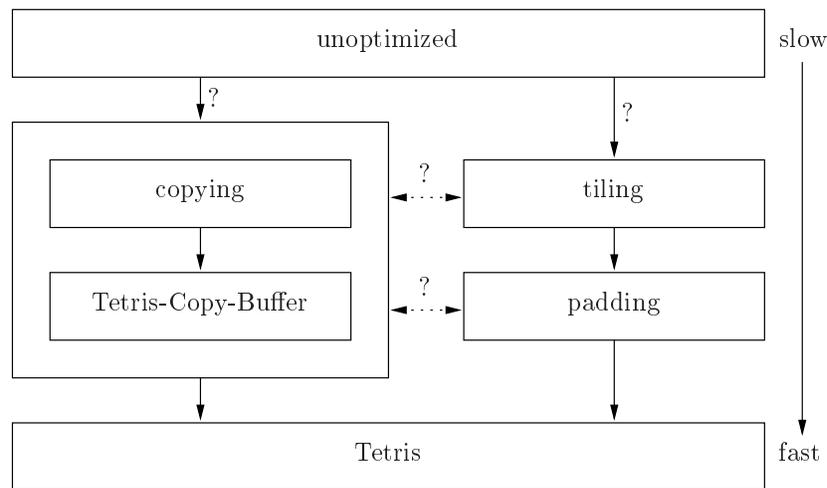
On the other hand, there are some reasons why matrix multiplication is a bad choice:

- It has too much reuse and too few direct memory accesses. This is why you do not see a significant difference between most programs.
- It has too few cross interference. Therefore, you can not see a difference between Tetris and padding.

In the end I need to choose an example algorithm for the experiments and no matter which one I choose it will have its advantages and disadvantages.

Box 19: Which method is faster?

There are several methods which avoid cache conflicts. Which one is the best for your program? How much faster your program will be when you implement one of these techniques depends on many parameters — like your hardware, the memory access patterns of your program, the cache interference caused by your program and the cache interference avoided by the method. This are too many parameters, so that I can not tell you how much you will win when you use one of these methods. I can not even promise you that your program will not be slower than before. But from experience and theoretical considerations I can, at least, tell you some relations among these methods.



A \longrightarrow B B is as fast as A or faster

A $\xrightarrow{?}$ B B is most likely faster as A but in some cases slower

A $\longleftrightarrow^{?}$ B it is unknown which one is faster

Most likely a tiled program is faster as the unoptimized original program but I saw some (badly) tiled programs which were slower than the original; mainly because the original version could make better use of the Stream Buffers.

A padded program is always faster than the tiled program because it uses the very same loops but avoids at least some cache conflicts. I never saw a padded program which was slower than the original and I do not expect to do so but theoretically it would be possible. For example: when the loop overhead caused by tiling slows down the padded program more than it wins by avoiding cache conflicts.

I know that Tetris-Copy-Buffer is faster than copying because it avoids cache conflicts while it copies. Whether these both methods are faster than your original program or not depends on how much cache conflicts they can avoid, how often once copied data is reused and how high the overhead caused by copying is. For the same reasons these methods can be faster or slower than tiling. In situations where padding can avoid all cache conflicts a padded program is sure faster than copying and Tetris-Copy-Buffer. If padding can not avoid all cache conflicts, then it depends on the relation between the time spent for the remaining cache conflicts of padding to the time spent for the copying overhead.

Tetris is faster than Tetris-Copy-Buffer and copying because it has not to pay the overhead of copying — here, I assume that Tetris can avoid all cache conflicts. Tetris is as fast as padding when padding can avoid all cache conflicts because Tetris can avoid the same conflicts as padding. In a situation where Tetris can avoid more cache conflicts as padding, Tetris will be faster.

? Why does the unoptimized curve rise so much around 110?

The secondary cache of the Alpha processor can store 12288 elements, that is: it can store one complete array (**Y**) of size 110. Even the unoptimized program can hold that array in the cache as long as the array length is smaller than 110.

? Why are most programs a bit faster for very small arrays?

Because of the first cache which can hold 1024 elements and is faster than the second one. Most programs use a tile size of 32×32 which exactly fits into the first cache. The Tetris programs use a tile size of 22×22 and must already deal with four tiles for array size 24. Therefore, their curves are higher at these points.

? What causes the peaks in the curve of the tiled program?

The tiled program suffers from self- and cross-interference. Depending on how long the arrays are and which base address they have, the amount of cache conflicts raises or drops. Some array lengths are odd multiples of the tile length 32. Then the array length fulfills the requirements of the Odd-Padding algorithm and there are no self-interference but there will still be cross-interference.

? What causes the peaks in the curve of the unoptimized program?

The peaks are caused by worst case cross-interference (cache thrashing) as described in box 4. Figure 19 explains the details.

? What causes the tall peak at array size 512 in the curve of the tiled program?

Worst case cross- and self-interference in the cache as described in box 4. Please, see figure 19 for details, again.

6.2 Summary

It is hard to predict the effect of a cache conflict avoiding method on the speed of a given program. For matrix multiplication copying, padding and Tetris run almost at the same speed. For other programs the out come may be totally different. The smooth curves show that these methods really get rid of most cache interference. Furthermore, Tetris proved that it can handle different tile sizes and access patterns. Even the worst case for Tetris — when it is necessary to employ the Tetris-Copy-Buffer technique — is comparatively fast.

7 Conclusion

Tiling can improve the speed of programs but it must be accompanied by a technique which avoids cache conflicts. Three of these techniques are reasonable strong: copying, padding and Tetris.

technique	advantage	disadvantage
padding	easy to implement	can not handle all situations
copying	strong and flexible	causes much overhead
Tetris	strong and flexible	no algorithm available/hard to implement

Copying is not much used because people fear the overhead but the experiments with matrix multiplication show that it is quite good in some cases. In the future Padding and Tetris have to prove their usefulness in practise.

A few years ago, If I would have talked about the state of the art of padding to a programmer, I might very well have had to face this question:

“You mean that the thing is supernatural?”

Sherlock Holmes in Sir Arthur Conan Doyle’s
The hound of the Baskervilles

A glimpse at the “work of others” section in this paper shows clearly that this is not longer true. Padding can be considered to be well understood but it is new and not wide spread by now.

There are several algorithms for intra-variable padding. Panda et al.’s [5] method and the two introduced in this paper: Odd-Padding and Brute-Force-Padding change the length of the array involved and have different requirements for the tile length. Whereas Panda et al.’s [5] method searches for an appropriate pad, the Odd-Padding and the Brute-Force-Padding algorithm are based on mathematical properties of the modulo-function. I not only showed the worst-case memory consumption of the Brute-Force-Padding algorithm and the Odd-Padding algorithm but even proved that the Odd-Padding algorithm is correct.

padding algorithm	advantage	disadvantage
Brute-Force-Padding	simple, flexible	uses too much pad
Panda et al.’s DAT	shortest average pad, flexible	bad worst case behavior
Odd-Padding	simple, shortest worst case pad	not so flexible, longer average pad

Padding is powerful enough to handle a lot of common cases like: tiles with odd sizes and base coordinates, several or multi-dimensional arrays, stencil operations. But there are serious weaknesses which restrict the usefulness of padding: for accessing several arrays at once, all arrays and tiles must have the same size. Furthermore, the way you can access the arrays is restricted, hierarchical tiling works only in a special case, and I carefully never talked about how to pad arrays which are accessed in different loops with different access requirements. Padding fails in such prominent cases like matrix multiplication, where it can avoid all self-interference but not all cross-interference because the arrays involved have different access patterns. In such cases copying (Temam et al. [8]) or Tetris must be employed.

Tetris is a powerful method to avoid cache conflicts but it does not come with an algorithm. At the moment Tetris can only be implemented by hand and some intuition is necessary to get best results. Should Tetris prove to be useful in future then it would be necessary to develop an algorithm.

References

- [1] Michal Cierniak and Wei Li. Unifying data and control transformations for distributed shared-memory machines. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30 no. 6 of *SIGPLAN Notices*, pages 205–217, La Jolla, CA, USA, June 1995. ACM.
- [2] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30 no. 6 of *SIGPLAN Notices*, pages 279–290, La Jolla, CA, USA, June 1995. ACM.
- [3] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 26 no. 4 of *SIGPLAN Notices*, pages 63–74, Palo Alto, California, USA, April 1991. IEEE and ACM.
- [4] Kathryn McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 31 no. 9 of *SIGPLAN Notices*, pages 94–104, Cambridge, MA, USA, September 1996. ACM.
- [5] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and Alexandru Nicolau. A data alignment technique for improving cache performance. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 587–592, Austin, TX, USA, October 1997. IEEE, IEEE Computer Society Press.
- [6] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 1998. ACM.
- [7] Gabriel Rivera and Chau-Wen Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 ACM International Conference on Supercomputing (ICS'98)*, Melbourne, Australia, July 1998. ACM.
- [8] Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings SUPERCOMPUTING '93*, pages 410–419, Portland, OR, USA, November 1993. IEEE and ACM SIGARCH, IEEE Computer Society Press.