Periodic scheduling for cache-miss minimisation

Periodic Scheduling for Cache-Miss Minimisation

Periodic Scheduling for Cache-Miss Minimisation

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op
maandag 27 augustus 2001 om 16.00 uur

door

Ramon Antoine Wiro Clout

geboren te Oosterhout

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. E.H.L. Aarts
en
prof.dr. M. Rem

Copromotor: dr.ir. W.F.J. Verhaegh

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Clout, Ramon Antoine Wiro.

Periodic Scheduling for Cache-Miss Minimisation /
Ramon Antoine Wiro Clout. -
Eindhoven: Eindhoven University of Technology
Thesis Eindhoven. - With index, ref. - With summary in Dutch
ISBN: 90-74445-49-7
Subject headings: scheduling, combinatorial optimisation, cache memory

Preface

This thesis is the result of many years of working in the field of cache optimisation. The research started as a graduation project for my computing science studies at Eindhoven University of Technology. Martin Rem and Frans Sijstermans, who supervised this project, encouraged me to continue the work as a Ph.D. student. The following years, that have not always been easy, have shown me the difficulties, but more importantly the beauty and satisfaction of doing research.

Many people have contributed to the results presented in this thesis. I wish to thank them.

First, I would like to thank my daily supervisor Wim Verhaegh. His warm and unique way of supporting this research has given me the strength to continue the work. Our constructive and pleasant discussions generated many of the ideas on which this thesis is based. It often proved that one hour of discussion generated several months of work.

I am very grateful to Emile Aarts for his continuous support. Besides providing an excellent working environment, he has taught me the importance of unambiguous and clear presentation of the work. I would like to thank Martin Rem for introducing me to this fascinating world, and for stressing that many results, however insignificant they may appear to me, are worth being reported upon.

The research has been carried out at the Philips Research Laboratories (Nat.Lab.) in Eindhoven. I have enjoyed working in this stimulating environment and owe thanks to many colleagues, especially to the members of the 'Algoritmen-club', for their open and constructive discussions. I am grateful to the management of the Nat.Lab. for giving me the opportunity to carry out the research described in this thesis.

Finally, I thank my family and friends for their support. My parents and sister I thank for giving me a safe home to return to. Ronald and Hans, you know what writing a thesis is all about. I would like to thank you for your support and friendship over the years. Arjan, Martijn, and Roy, the past years would not have been the same without our weekly 'kookclub' and our regular visits to the theatre and the swimming pool. I value these happenings highly.

Eindhoven, June 2001 Ramon Clout

Contents

1

Introduction

This thesis concerns the problem of periodic scheduling for cache-miss minimisation. Periodic scheduling refers to the problem of determining an order for a given set of operations that have to be executed repeatedly. Our interest in periodic scheduling originates from the field of digital video signal processing. More precisely, it originates from the problem of compiling a description of a video algorithm into code that can be executed by a general purpose processor, while optimising the use of the processor cache.

This chapter contains background information on the field of video signal processing in Section 1.1, and on processors in Section 1.2. In Section 1.3 we give an informal statement of the scheduling problem we consider. Next, Section 1.4 contains an overview of related work. The chapter ends with an outline of the thesis in Section 1.5.

1.1 Video signal processing

Signal processing concerns the transformation of input streams into output streams. Applications of signal processing can be found in TV, radio, radar, medical diagnosis, and telecommunications. In these areas a shift takes place from analog systems to digital ones. This shift is enabled by advances in integrated circuit technology. Digital signal processing can be used for picture and sound enhancement, and for

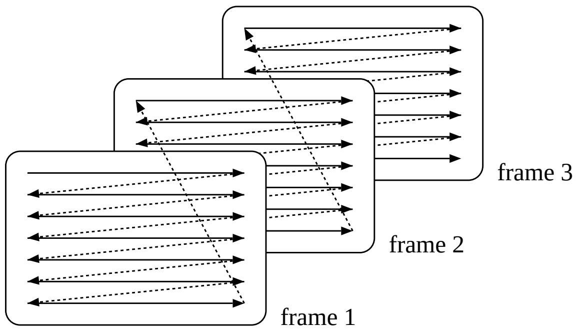


Figure 1.1. The scanning process for non-interlaced PAL.

new features that support applications such as video conferencing.

A digital video signal is obtained by sampling a moving picture that can be regarded to be continuous in time and space. By digitising an image in time, space, and amplitude, the image is represented in such a way that it can be handled by digital video signal processing algorithms. The spatial and temporal digitisation is referred to as *sampling*, the amplitude digitisation is referred to as *quantisation*.

Frames are still pictures that are the result of temporal sampling. Each frame consists of a number of *lines*, and each line consists of a number of *pixels*. The exact number of frames per second, lines per frame, and pixels per line depends on the video standard that is used. In the PAL standard, for example, we have 25 frames per second, 625 lines per frame, and 864 pixels per line, which gives a total of 13.5 million pixels per second. Out of these 625 lines only 576 are visible, and out of the 864 pixels only 720 are visible.

Spatial sampling is performed in order to map the two-dimensional spatial information into one-dimensional temporal information. For a non-interlaced version of the PAL standard the scanning process is shown in Figure 1.1. For each frame pixels are scanned from left to right, and lines are scanned from top to bottom. PAL video has 25 frames per second, which results in one frame every 40 ms. The time between successive lines is $1/625 \cdot 40$ ms $= 64$ μs, and the time between successive pixels $1/864 \cdot 64$ μs ≈ 74 ns. In Figure 1.2 this scanning process is depicted as a three-dimensional periodic arrival of visible pixels. The time between the last pixel of a line and the first pixel of the next line is called the *line blanking*, and is $(864 - 720) \cdot 74$ ns ≈ 9 μs for our example. This time can be used by a scanner to move to the beginning of the next line. Similarly, the *frame blanking* is the time between the last visible pixel of a frame, and the first pixel of the next frame.

After quantisation the pixels can be represented as integers, representing light intensity and colour. A video signal can thus be represented as a stream of integers.

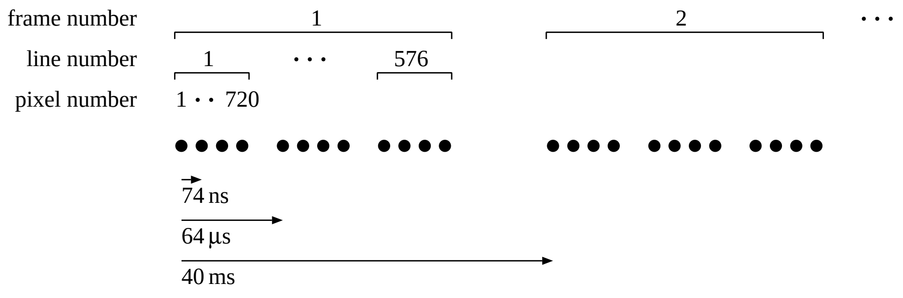


Figure 1.2. Three-dimensional periodic arrival of visible pixels.

An algorithm that transforms such an input stream into an output stream is called a digital video signal processing algorithm, or *video algorithm*, for short.

The high sampling frequency, together with the number of operations a video algorithm requests per pixel results in a high computational demand. For example, a video algorithm that performs 100 operations per pixel on a stream of 13.5 million pixels per second results in a demand of over 1 billion operations per second. Until recently, only *application specific integrated circuits* (ASICs), which are tailored to performing dedicated tasks, and *programmable video signal processors*, which are heavily tailored towards video signal processing, were able to implement such video algorithms. Advances in processor technology have enabled the possibility of executing such algorithms on general purpose processors.

Application specific processors are inflexible in the sense that they are designed for one application only. The flexibility of video processors, and even more for general purpose processors enables the possibility of fast prototyping of new algorithms. The price for this flexibility is paid by a worse resource usage, which results in larger circuits and higher power consumption.

As examples of these approaches that have been developed at the Philips Research Laboratories we mention the Phideo design methodology [Van Meerbergen et al., 1995] which supports high-level synthesis of video algorithms into application specific circuits, such as the I.McIC MPEG2 video encoder [Van der Werf et al., 1997], and the VSP programming environment [Vissers et al., 1995] which supports code generation of video algorithms for programmable video signal processors. An example of a general purpose processor tailored towards video signal processing is the TriMedia processor [Slavenburg, Rathnam & Dijkstra, 1996]. This thesis focuses on methods for mapping video algorithms onto general purpose processors. In the next section we give characteristics of such processors, after which this mapping problem is stated informally.

1.2 Processors

As indicated above, speed of modern microprocessors has increased considerably in the past 20 years. Besides integrated circuit technology, the current performance of processors is largely due to exploitation of instruction level parallelism. This can be done in the following two ways. In the first place, pipelining is applied to let executions of operations overlap in time. Second, multi-issue processors allow simultaneous start of multiple operations per clock cycle. As operations can only be started if all input data are present and if sufficient resources are available, operations must be well ordered for an effective use of the processor. This scheduling process can be handled by a compiler, but can also be performed by the processor.

Unfortunately, the latency of the memory system, which is the time needed to retrieve data from memory, has not kept up with the speed of the processors. Hennessy & Patterson [1996] show that memory performance has increased with approximately 7% annually over the past 20 years. Processor performance on the other hand has increased with 35% per year until 1986 and with 55% per year in the past 14 years. This leads to a performance gap between processors and memory; processors are able to request data from memory faster than the memory is able to service these requests.

A way to reduce the gap between memory and processor performance is to insert a cache between the processor and the memory. Caches are fast but small pieces of memory located close to the processor that act as a buffer for data that are used frequently. In this way, the data requested by the processor can be delivered much faster. However, as the cache is small it cannot contain all the data that are needed during the execution of a program. If the processor requests data that are not available in the cache, we say that a *cache miss* occurs and the requested data must be fetched from memory, which takes a long time. Upon a cache miss, a cache must furthermore decide which data must be removed from the cache to facilitate storage of the new data. This process is called *cache replacement*. Cache replacement policy, as well as cache size are parameters that must be determined during the design of a processor.

General purpose processors are flexible in the sense that they can execute a great variety of programs efficiently. To this end, a cache should be targeted towards the set of programs that are likely to be executed on that processor. Much research has been done on the problem of designing a cache. Basically, there are two ways of approaching the design problem [Przybylski, 1990]. In order to make the common case fast, a set of representative applications is chosen for which the memory system is analysed using trace-driven simulations. A trace is a sequence of memory references that contains sufficient information for the computation of the cache hit rate, i.e., the fraction of the memory references that result in a hit in the cache. Another way of attacking the cache design problem is by analytical modelling of a cache. Analytical models give insight in the phenomena that determine the performance of a cache.

Processors use cache memory for both instruction and data references. Processors can have separate caches for instruction references and data references. Such caches are called *data caches* and *instruction caches*, respectively. *Unified caches* can contain both instructions and data. In this thesis we focus on data references and data caches only.

Complementary to the cache design problem is the problem of optimally using a cache for a given application, which is the problem we study in this thesis. Given a processor with a cache and a description of an algorithm in a programming language, it is the task of a compiler to translate the algorithm into object code for the processor in such a way that the scarce resources on the processor, in our case the cache, are used optimally.

We divide the field of cache optimisation into two categories: local optimisations and global optimisations. Local optimisations deal with small parts of a program, typically one or two loop nests, whereas global optimisation considers a whole program. Local optimisations have the disadvantage that locally optimal solutions for each part of the program lead to sub-optimal solutions for the whole program. Next to these categories, we make a distinction in the level of detail in which we study the scheduling problem. A high level of detail is achieved by optimisation that considers every individual access to memory. Optimisation with a low level of detail considers arrays or sections of arrays rather than individual array elements. Again, optimising with a low level of detail may lead only to sub-optimal solutions. Therefore, this thesis focuses on global optimisation with a high level of detail.

1.3 Informal problem statement

For the scheduling problem subject to this thesis we assume that a cache and a video algorithm are given. Parameters that define a cache are, amongst others, cache size and replacement policy. As many video algorithms can be described by a set of operations that must be executed repeatedly, a video algorithm is represented as a series of nested loops. In this way, an execution of an operation of a video algorithm is uniquely identified by the corresponding values of its loop iterators. As a consequence, these operations also produce and consume data in a repetitive way, which is described using multidimensional arrays.

The order of execution of operations can now be defined by periods for every loop iterator, which denote the time between two consecutive iterations of the corresponding loop. Each operation is given a start time, indicating the moment in time at which the first execution takes place. In this thesis periods and start times are used only to give a partial order on the executions of the operations. During scheduling we must assign values to the periods and the start times of all operations. In addition to this *time assignment* we must also assign a memory position to every element of the multidimensional arrays, which is reflected in a so-called *address assignment*.

We consider two kinds of constraints. First, we have *precedence constraints*, which are due to data dependencies. Data dependencies express the relationship between production and consumption of data, and precedence constraints specify that consumption of data may only take place after the corresponding production. Second, we have *address constraints* that express that data may not be overwritten as long as they are not consumed. Hence, a memory location can only be reused after the previous data item has been consumed for the last time.

The objective for scheduling is minimisation of the number of processor cycles that are required for the execution of the schedule, i.e., a time assignment and an address assignment. As a cache miss results in a large number of processor cycles, we determine total execution time of a schedule by counting the total number of cache misses.

Informally, we state the scheduling problem as follows. Given a video algorithm and cache parameters, find a schedule, consisting of a time assignment and an address assignment, that satisfies the precedence constraints and address constraints, and that minimises the number of cache misses.

1.4 Related work

The discrete nature of the scheduling problem we consider allows for a formulation in terms of a combinatorial optimisation problem. For the theory of combinatorial optimisation we refer to Papadimitriou & Steiglitz [1982] and Schrijver [1986]. The computational complexity of many of these problems is studied by Garey & Johnson [1979] and Papadimitriou [1995].

In the area of off-line non-preemptive periodic scheduling, work has been done by Korst [1992] and De Kock [1999], who consider the mapping of video signal processing algorithms onto programmable video signal processors, and by Verhaegh [1995], who considers the problem of scheduling multidimensional periodic operations in high-level synthesis. Contrary to these approaches, where operations are scheduled on several processors, we restrict the scheduling problem to a single processor. For a general introduction to scheduling we refer to Pinedo [1995].

An early and thorough survey on the influence of various cache parameters is given by Smith [1982]. The specific topic of using trace-driven analysis is handled by Hill & Smith [1984]. Przybylski, Horowitz & Hennessy [1988], Przybylski

[1990], and Smith & Goodman [1983] analyse caches using analytical models of typical workloads. These approaches, however, do not consider periodic operations.

Bacon, Graham & Sharp [1994] and Wolfe [1996] give overviews of compile time optimisations. They both give an overview of loop transformation techniques for cache optimisation. A general introduction on compilers is given by Aho, Sethi & Ullman [1986].

A well known compiler optimisation algorithm for caches, *blocking* or *tiling*, breaks computations for large arrays into several computations on sub-arrays. Techniques for finding tile sizes have been reported by Lam, Rothberg & Wolf [1991], Wolf & Lam [1991], Kennedy & McKinley [1992], and Coleman & McKinley [1995].

Other algorithms that rearrange loops are often included in source-to-source compilers. Kennedy & McKinley [1993] and Singhai & McKinley [1997] use *loop fusion* and *loop distribution* to enhance cache performance. McKinley, Carr & Tseng [1996] propose a compound optimisation algorithm that incorporates several loop reordering techniques. These optimisation approaches are all local optimisation techniques. Our global optimisation algorithm generalises these reordering techniques in terms of periods and start times in Chapter 7.

Another class of algorithms aims at optimising cache performance by taking the placement of data in memory into account. Recent work has been done by Panda & Dutt [1999], Strout, Carter, Ferrante & Simon [1998], Rivera & Tseng [1998], and Calder, Krintz, John & Austin [1998]. Topham & González [1999] propose randomness in the indexing function of the cache in order to eliminate cache conflicts. Combinations of loop transformations and data placement are given by Kandemir, Ramanujam & Choudhary [1999] and Manjikian & Abdelrahman [1997].

Gannon, Jalby & Gallivan [1988] present program transformation techniques for caches and local memories. Philbin, Edler, Anshus, Douglas & Li [1996] use fine-grained thread scheduling for cache optimisation. Schutte & Van Kempen [1997] propose cache optimisation algorithms for a class of image processing algorithms. The problem of optimising video algorithms for combined instruction cache performance and data cache performance is studied by Clout [1994].

Besides reordering memory accesses, the latency of fetching data from memory can be shortened by prefetching data. A hardware-controlled prefetching method is already mentioned by Smith [1982], who suggests considering the next line for prefetching upon a cache reference. Jouppi [1990] and Palacharla [1994] propose the addition of stream buffers to the cache that prefetch streams of subsequent cache blocks. Chen & Baer [1992] include *reference prediction tables* that store predictions for regular data accesses, and which generate prefetching requests.

Zucker, Flynn & Lee [1995] combine the use of a prediction table and stream buffers for multimedia applications, such as an MPEG2 decoder. Software controlled prefetching as proposed by Callahan, Kennedy & Porterfield [1991] and Mowry, Lam & Gupta [1992] assumes that the instruction set of a processor includes a *prefetch instruction*, which can be inserted by either a programmer or a compiler. A combination of software and hardware control for prefetching is given by Chen [1997] and Struik, Van der Wolf & Pimentel [1998]. The former approach synchronises the prefetching on the instruction addresses, the latter on the data addresses. In Chapter 6 of this thesis we come back to prefetching issues.

1.5 Thesis outline

In this thesis we study the mapping of video algorithms onto processors with caches, where the objective is to minimise the execution time of a video algorithm on such a processor by minimising the number of cache misses. We aim at finding an appropriate model for video algorithms that allows for a global optimisation approach. Contrary to local optimisations, that only look at a limited number of loop nestings, our global approach treats all loops at once. Furthermore, we aim at analysing the complexity of our scheduling problem and at finding a solution approach to this problem. The thesis is organised as follows. In Chapter 2 we informally present the most important concepts that we use in this thesis. In Chapter 3 we give a formal model for video algorithms, a formal execution model for these algorithms (schedule), a model for caches, and a formal problem definition of the multidimensional periodic cache scheduling problem. Chapter 4 handles the formal complexity of this scheduling problem and some related sub-problems. As not all schedules allow for an efficient implementation on a single processor, we introduce in Chapter 5 some constraints on schedules that make an efficient implementation possible. In Chapter 6 we propose a method for efficiently estimating the number of cache misses for a given schedule and a given cache. This method is used in Chapter 7, where we give ingredients for a local search approach that is aimed at finding good schedules. We summarise the main results of this thesis in Chapter 8. Notes on notation used in this thesis can be found in the symbol index.

2

Conceptual Model

In this chapter we present the concepts *video algorithms*, *caches*, and *cache performance*. These concepts are described in an informal way. A formal presentation is given in Chapter 3. Section 2.1 gives characteristics of video algorithms. In Section 2.2 we present the relevant cache terminology. In Section 2.3 we break down a measure for processor performance into several parts, which are discussed one by one. Section 2.4 discusses several ways of measuring the performance of a cache.

2.1 Video algorithms

We aim at exploiting the repetitive nature of video algorithms. This nature exists both in the operations that must be performed for every pixel and in the storage of the pixels in memory. For every pixel on every video line of every video frame a similar action usually has to be performed. The memory is also accessed in a repetitive way, meaning that pixel information for the second pixel on the third video line of the first frame can usually be accessed in a similar way as the second pixel on the third video line of the seventh frame.

Hence, for video algorithms we use a formalism that can handle this repetition well. In this chapter we only give a flavour of this model by means of an example; in Chapter 3 it is formalised.

```
for $i := 0\ldots 2 \rightarrow$
   for $j := 0\ldots 1 \rightarrow$
      $Z[j][i+1] := \mathrm{sum}(Y[i][j], Z[j][i])$
for $k := 0\ldots 1 \rightarrow$
   $Z[k][0] := \mathrm{cpy}(Y[0][k])$
for $i := 0\ldots 3 \rightarrow$
   for $j := 0\ldots 1 \rightarrow$
      $Y[i][j] := \mathrm{avg}(X[3i+j], X[3i+j+1])$
```

Figure 2.1. Example of a part of a video algorithm with nested loops and multi-dimensional arrays.

As video signals are periodic, their execution must also be periodic. Therefore we specify them as nested loops. In Figure 2.1, we give a set of operations that perform some calculations on an input array X. Here, we have three operations: operation 'sum' with two input arguments $Y[i][j]$, $Z[j][i]$ and one output argument $Z[j][i+1]$, operation 'cpy', and operation 'avg'. The arguments of the operations are elements of multidimensional arrays, where the *index expressions*, for example $3i+j+1$, are affine expressions in the *loop iterators* i and j.

This program looks much like a program in an *imperative* programming language like Pascal or C, but there are marked differences. In Figure 2.1 there is no direct relationship between the syntactic order of loops and a *possible* order of execution of the operations. In general there are many valid orders, that are restricted only by data dependencies. These dependencies are implicit in the program and exist between each production and consumption of the same array element. In Figure 2.2 we have depicted all data dependencies for the example. Every data dependency induces a *precedence constraint* on the possible order of execution of operations, that is, every array element must be written before it can be read. A possible order of execution of the operations that meets all precedence constraints is given in Figure 2.3. This order is not complete as executions that appear above each other may be executed in arbitrary order. A compact representation of such partial orders is given in Chapter 3.

Another difference with imperative programming languages is the memory allocation. A memory allocation assigns a memory address to each element of each array. This allocation, which is implicit for many programming languages, has not been specified in the program above. For arrays in the programming language C, the mapping of array elements to memory locations is described by an address

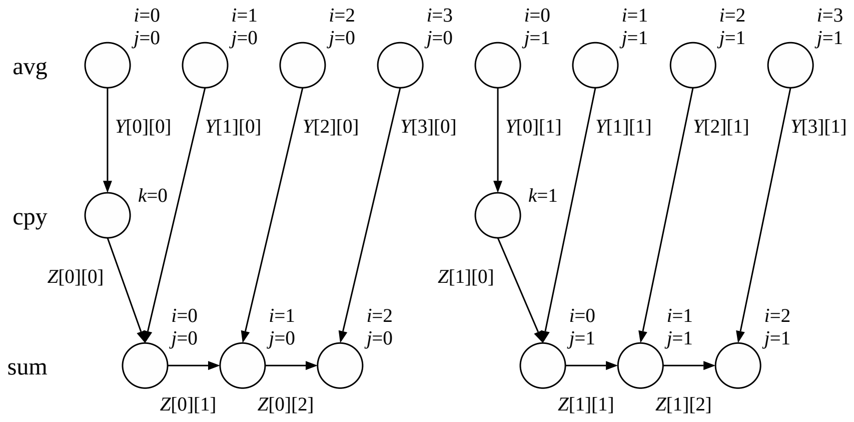


Figure 2.2. Data dependencies for the example of Figure 2.1. Every circle represents an execution of an operation and is annotated with the values for the iterators. The data dependencies are denoted by arrows and annotated with the corresponding array elements.

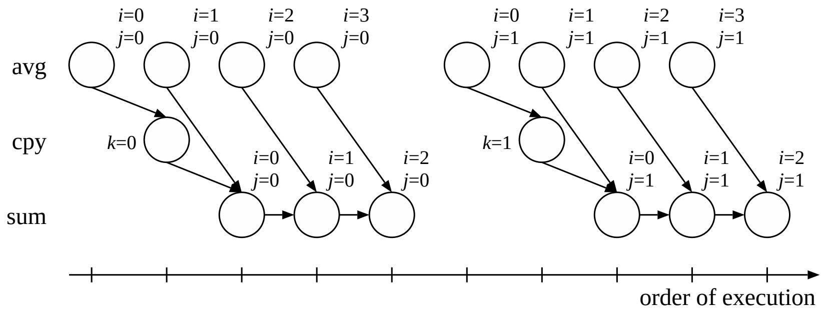


Figure 2.3. A possible order of executions of operations where all precedence constraints are met. Executions that appear to the right of another execution must be performed later.

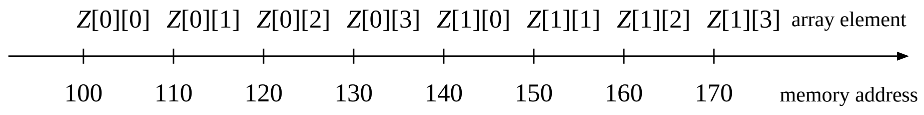


Figure 2.4. A possible memory assignment for a 2×4 array Z with C-like storage. Array element $Z[i][j]$ is placed at memory location $100 + 10(4i + j)$.

function a, which is given by

$$a(X[n_{N-1}]\dots[n_1][n_0]) = o(X) + s(X) \sum_{k=0}^{N-1} n_k \prod_{l=0}^{k-1} S_l,$$

for an $S_{N-1} \times \dots \times S_1 \times S_0$ array X, and $s(X)$ being the size of one array element. The offset $o(X)$ determines the location of array element $X[0]\dots[0][0]$ in memory. This position is usually chosen at compile time for global variables, or at run time for local variables or dynamically created variables. For example, a 2×4 array Z with array offset $o(Z) = 100$ and element size $s(Z) = 10$, is arranged in memory as depicted in Figure 2.4.

The above assignment method is called *row-major* storage, as the larger coefficients appear for the indices n_k with larger k. The opposite method, *column-major* storage, is used in the programming language Fortran.

In this thesis, we use a generalisation of such memory assignments, given by

$$a(X[n_{N-1}]\dots[n_1][n_0]) = o(X) + \sum_{k=0}^{N-1} n_k c_k(X),$$

where the *address offset* $o(X)$ and *address coefficients* $c_k(X)$ remain to be chosen. Here we must take care that array elements that are alive at the same time are not mapped onto the same memory address. The *lifetime* of an array element is defined as the interval between its production and its last consumption. Hence, we are looking for a memory assignment for all arrays in such a way that array elements with overlapping lifetimes are mapped onto different memory addresses. The order of the executions of operation 'sum' that we chose in Figure 2.3 allows that array elements $Z[k][0]$, $Z[k][1]$, and $Z[k][2]$ are all mapped to the same memory address. Hence we can use a memory assignment $a(Z[i][j]) = o(Z) + s(Z) \cdot (i + 0j) = o(Z) + s(Z)i$. This memory assignment is possible as the lifetimes of these array elements do not overlap for the given order.

In this thesis we aim at methods for finding an execution order for all operations and a memory assignment for each array in such a way that the processor cache is used optimally. To this end, we first give an introduction to processor caches, and accompanying performance criteria.

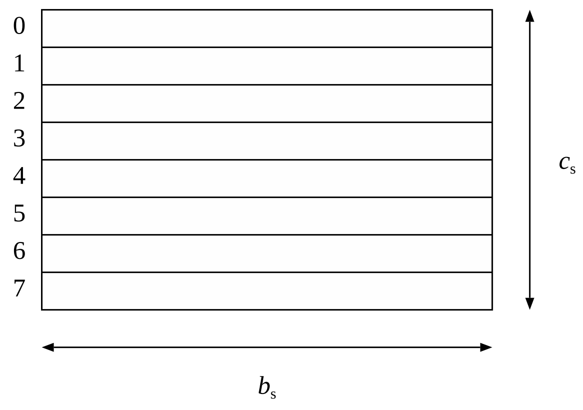


Figure 2.5. A cache is a memory consisting of blocks with block size b_s. The size of a cache, denoted by c_s, is expressed in the number of blocks.

2.2 Caches

In this section we give an overview of caches. This overview is not intended to be complete; we limit ourselves to the relevant terminology. We closely follow the description of caches given by Hennessy & Patterson [1996].

A cache is a fast but small piece of memory between processor and main memory that acts as a buffer for data that are used frequently. We say that a *cache miss* occurs if the processor requests a data item that is not available in the cache. The opposite, a request for a data item that is present in the cache, is called a *cache hit*. A *block* is the granularity for storing data in a cache, i.e., the amount of data that the cache requests from memory upon a cache miss. A cache hit or cache miss will consequently occur for an entire cache block. Figure 2.5 shows a cache with c_s cache blocks, every block having a size b_s. The unit of block size is the size of the smallest element that can be addressed in memory, usually a byte.

A cache is designed in such a way that it takes advantage of *locality of reference*. References to the same memory address tend not to be uniformly distributed over time, but instead, two references to the same memory address are likely to be executed close to each other in time. This kind of locality is called *temporal locality*. Also, if data items are close together in memory, they tend to be referenced close to each other in time. This is called *spatial locality*.

In order to exploit spatial locality, blocks usually consist of successive data items in memory. Hence, if a data item is requested from memory, a block of data is fetched, consisting of data items that are spatially close. When a processor requests data at memory address $a \in \mathbb{Z}$, the position within the corresponding block is called

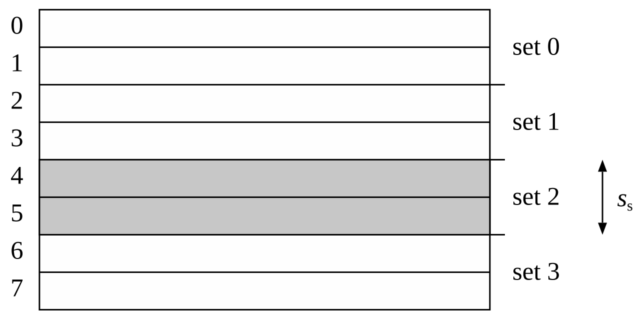


Figure 2.6. The positions within a cache where a block can be placed is restricted in set-associative caches. The size of each set, denoted by s_s, is also called the *associativity* of the cache. The number of sets, in this case four, is denoted by n_s.

the *block offset*, and given by $a \bmod b_s$, where b_s is the size of a single block. The *block address* equals $a_b(a) = a \operatorname{div} b_s$.

Now, there are four basic issues to be addressed in a cache design:

- Placement of a block in the cache.
- Finding a block in the cache.
- Replacement of a block upon a cache miss.
- Policies for writes to the cache.

Placement of a block in the cache. Often, there are restrictions on the position within a cache where a block can be placed. In Figure 2.6 the cache has been divided into four sets of equal size. Each memory block can only be placed in a predetermined set. It is common practice that a set is selected using low order bits of the block address; the set where address a is placed is given by its *index* $s(a) = a_b(a) \bmod n_s$, where n_s is the number of sets in the cache. The number of blocks in each set is also called the associativity of the cache and is given by $s_s = \frac{c_s}{n_s}$. A cache with set size $s_s = 1$ is called a *direct mapped* cache, as the mapping of each memory block to a position within the cache is unique. A cache with only one set, with set size $s_s = c_s$, is called a *fully set-associative* cache. Here, every memory block can appear at every position in the cache. In general, caches with associativity n are called *n-way set-associative*. Caches with a higher degree of associativity usually result in fewer cache misses, but are more difficult to build in hardware [Hill, 1988].

Finding a block in the cache. In order to find out whether data requested by a processor is present in the cache, each cache block is tagged with its block address. As is depicted in Figure 2.7 we do not need to store the whole block address, as

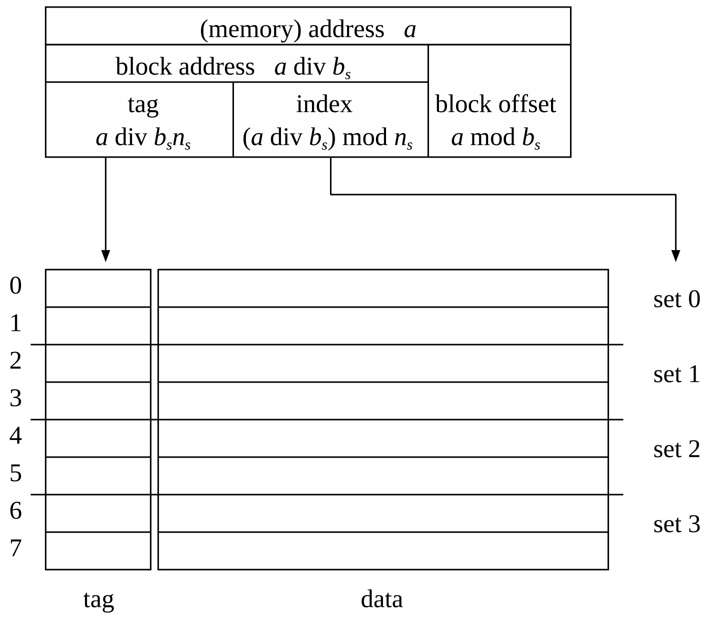


Figure 2.7. Organisation of a 2-way set-associative cache.

a block can only appear in one set, and hence the index, i.e., the part of the block address that determines the set, does not have to be stored in the tag.

Replacement of a block upon a cache miss. Upon a cache miss, the cache must decide which block is removed from the cache. As a block can only appear in one set, a choice must be made between the blocks within this set. Several replacement algorithms have been presented in literature, which can be divided into two categories [Smith, 1982], being usage-based and non-usage-based algorithms. Usage-based replacement algorithms keep track of the use of blocks and select a victim for replacement based on this information. Non-usage-based algorithms choose a victim for replacement irrespective of the previously accessed blocks.

We give four examples of replacement algorithms, two usage-based (MIN and LRU), and two non-usage-based (Random and FIFO).

MIN. The *MIN*-algorithm by Belady [1966] replaces the block that is used again the farthest into the future.

Least Recently Used (LRU). The principle of locality tells that items that were accessed recently tend to be accessed again soon. Based on this principle, the most unlikely block to be accessed soon is the one that was accessed least recently.

First In First Out (FIFO). Blocks are replaced cyclically. The cache block that was least recently replaced, is the victim.

Random. Within the set a victim for replacement is chosen randomly.

The MIN-algorithm has been proved optimal with respect to the number of cache misses, but requires knowledge about the future and is therefore hard to implement. The hardware cost for LRU replacement increases with associativity and is often only approximated. Both FIFO and Random are easy to implement in hardware. As usage-based replacement usually performs better than non-usage-based replacement, and as LRU is widely used in modern processors, we focus on caches with LRU replacement.

Policies for writes to the cache. Upon a write it must be decided where the data item is modified. With *write through* data are changed in memory. Only if data already reside in cache, the contents of the cache are changed. As a consequence the data that are stored in the cache and the corresponding data in memory are always identical. This also means that upon a read miss, the block can simply be thrown out of the cache. *Write back*, on the other hand changes data in the cache only. Write back has the advantage that it generates fewer accesses to memory. In case of write back caches, blocks that have been changed in the cache, must be written back to memory, when they are replaced. An optimisation that is often used is the addition of a *dirty* bit to each cache block which registers whether data in the cache have been updated. Only if the dirty bit for a block has been set, the data must be written back to memory upon replacement of that block.

Furthermore, upon a write it must be decided whether to fetch data in case of a cache miss. The strategy where a block is fetched upon a write miss, is called *write allocate*. This strategy is common for write back caches, as subsequent writes to the same block are handled by the cache. The *no-write allocate* strategy changes the block only in memory. As all writes change memory in a write through cache, this is a common strategy for these caches.

Write through caches have the additional advantage that the content of the blocks that are present in the cache are equal to the contents of the corresponding blocks in memory. For multiprocessors that share memory it is important that all caches have the same copy of a memory block. This so-called *cache consistency* is implemented easier with write through caches.

In general write back caches handle writes faster than write through caches. Furthermore, if more writes to the same block result in hits, they will require only one write back.

As is explained in the next sections, we are mainly interested in finding an execution order and memory assignment that minimises the number of cache misses for a given program and a given cache. The cache parameters that are needed for measuring the number of cache misses are the cache size c_S, the associativity s_S, the block size b_s, and the replacement algorithm. In this thesis we focus on LRU caches with write back and write allocate policy.

2.3 Processor performance

System performance optimisation involves optimal use of scarce resources, for example processors and memory. In our case we want to optimise the performance of a processor, where we use the term processor for a CPU and cache together. Hence it is worth taking a look at a measure of processor performance, being the time needed for the execution of a program, which is defined by Hennessy & Patterson [1996] as follows.

$$\text{processor time} = \frac{\text{seconds}}{\text{program}} = \underbrace{\frac{\text{instructions}}{\text{program}}}_{1} \times \underbrace{\frac{\text{clock cycles}}{\text{instruction}}}_{2} \times \underbrace{\frac{\text{seconds}}{\text{clock cycle}}}_{3}$$

The number of instructions per program (1), also called the instruction path length or instruction count, is a measure for the amount of work that the processor has to perform. In the first place, the programmer is responsible for minimising this parameter. Optimising compilers can improve this, for example by removing redundant computations. This parameter is further influenced by the choice of instruction set. A processor with a small instruction set, for example RISC (Reduced Instruction Set Computer), will generally need more instructions than a processor that can execute complex instructions, such as CISC (Complex Instruction Set Computer). The last parameter, the number of seconds per clock cycle, is largely determined by the hardware technology.

Both parameter (1) and (3) fall outside the scope of this thesis. We zoom in on the average number of clock cycles per instruction (2), usually denoted by its abbreviation CPI, which can be split up [Przybylski, 1990] as follows.

$$\frac{\text{clock cycles}}{\text{instruction}} = \underbrace{\frac{\text{CPU cycles}}{\text{instruction}}}_{2a} + \underbrace{\frac{\text{references}}{\text{instruction}}}_{2b} \times \underbrace{\frac{\text{clock cycles}}{\text{reference}}}_{2c}$$

The average number of clock cycles per instruction consists of the average number of clock cycles that the CPU needs to execute an instruction and the average number of clock cycles that the cache needs for all memory references for the instruction. An important influence on the mean number of CPU clock cycles per instruction (CPI) is the architecture of the CPU. Modern CPUs allow the execution of multiple instructions in parallel. This level of instruction level parallelism (ILP) influences the CPI. As not all instructions require the same amount of clock cycles, the complexity of the instructions is another factor that affects the CPI.

	(1)	(2a)	(2b)	**(2c)**	(3)
instruction set	⋆	⋆	⋆		
compiler	⋆	⋆	⋆	★	
CPU-architecture		⋆			⋆
cache organisation				⋆	⋆

Table 2.1. Influences on the various performance parameters discussed in Section 2.3, indicated with stars.

Parameters (2b) and (2c) constitute the time spent by the cache. The first factor, the average number of references per instruction, depends on the used compiler. For example, by storing data with short lifetimes in registers, the number of memory references can be decreased. The second factor, the average number of clock cycles per reference, depends on the organisation of the cache, and on the sequence of references that is fed to the cache. This sequence basically is an ordered list of memory addresses that are referenced. As explained in Section 2.1, both the order of the list and the addresses can be influenced by the compiler.

Table 2.1 lists all relevant parameters. This scheme lists influences on the various performance parameters. This table has been derived from Przybylski [1990], where we added the influence of a compiler on performance parameter (2c). Only the most important influences have been discussed in the text. In the next section we look more closely at performance parameter (2c), the average number of clock cycles per reference, or stated differently, the time spent by the processor cache for memory references. The focus in this thesis is on the influence of a compiler on this parameter.

2.4 Cache performance

For a cache, the mean number of clock cycles per reference depends on the *miss rate*, i.e., the fraction of the memory references that cause a miss in the cache, as follows.

$$\text{average time per reference} = \text{hit time} + \text{miss rate} \times \text{miss penalty},$$

where the *hit time* is the time required to fetch data from the cache, and the *miss penalty* is the additional time the cache needs to fetch this data from memory. In this thesis we assume that hit time and miss penalty are constants. In general neither of the two is constant. For example, processors that can issue multiple requests to the cache at the same time may suffer from conflicts in the cache, resulting in a variable hit time and variable miss penalty. Assuming that the cache hit time and the cache miss penalty are constants, minimisation of the average time per reference for a video algorithm is equivalent to minimisation of the cache miss rate.

A method that is often used to measure the miss rate of a cache for an execution of a program is to perform a cache simulation. For a cache simulation one typically runs the program and registers which subsequent memory accesses take place. This information is sufficient to compute the number of cache misses. A disadvantage of cache simulation is the excessive execution times of such simulations. If we want to analyse a program with twice as many memory accesses, the cache simulation will take twice as long. As video signal processing algorithms usually generate many memory accesses, this will result in long simulation times.

For a PAL signal, for example, we have 720 visible pixels on 576 lines per video frame and 25 video frames per second. If we assume that we need one memory reference for each visible pixel, 10,368,000 memory accesses per second take place. A cache simulation for such an amount of accesses takes several minutes. For example, simulation of 10,000,000 memory accesses takes approximately 200 seconds on a Pentium processor running at 200 MHz.

An additional disadvantage of cache simulation is the difficulty to identify bottlenecks. As a simulation only reports the number of cache misses, it is hard to find the references in the original program that are responsible for the misses.

Another way of obtaining the number of cache misses is to analyse a program at source code level. Ghosh, Martonosi & Malik [1998] propose a method that generates so-called *cache miss equations* (CMEs) for every reference in a loop nest. Every solution for these equations represents a possible cache miss. Counting the number of solutions for a system of CMEs can then be performed using methods described by Clauss [1996]. From this the references with most misses can be identified. Unfortunately, at the moment CMEs do not allow for program analysis across loop nests.

Algorithms that optimise for cache performance at compile time often use approximations of the number of cache misses rather than the exact number. McKinley, Carr & Tseng [1996] use the number of distinct cache lines that a single loop nest accesses as an indication for the number of cache misses. Wolf & Lam [1991] use the number of memory accesses per iteration of a loop nest as a metric. Both estimations can be computed efficiently but work only for a single loop nest.

In Chapter 6 we give a method for the evaluation of the number of cache misses for a set of loop nests. We aim at a fast evaluation, that gives a good estimation of the number of caches misses. A precise count of the number of cache misses cannot be expected to be possible in reasonable time as is shown in Chapter 4. Nevertheless, the repetitiveness of video algorithms enables a compact description of the access sequence, which is used in Chapter 6 to estimate the number of cache misses effectively and efficiently.

3

Formal Model

In this chapter we model video algorithms by means of multidimensional periodic operations in a *program graph*. A *schedule* that satisfies accompanying *constraints* determines a possible *execution* of a program graph. We give a formal *cache model* which we use to define the objective function of our scheduling problem, i.e., the number of cache misses. Throughout this chapter we use a model of multidimensional periodic operations based on the model presented by Verhaegh [1995].

The model in this chapter consists of three parts. In Section 3.1 we give a model of video algorithms that we want to optimise. Schedules are defined in Section 3.2, and feasibility of schedules is the topic of Section 3.3. The formal cache model is defined in Section 3.4. The problem is formulated in Section 3.5. Section 3.6 contains some special properties of program graphs and schedules.

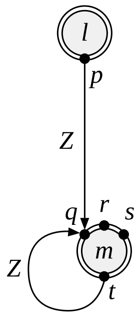
3.1 Multidimensional periodic operations

Usually, video algorithms are described using nested loops and multidimensional arrays. For an example see Figure 3.1, in which a matrix multiplication algorithm for 50×50 matrices is given by two loop nests, of which the former initialises an array Z and the latter does the actual multiplication of arrays X and Y and stores the result in array Z. In a so-called *program graph* the statements in the inner loop are represented by *operations*. For the example of Figure 3.1 the program graph

```
operation $l$:
for $i_0 := 0 \ldots 49 \rightarrow$
    for $i_1 := 0 \ldots 49 \rightarrow$
        $Z[i_0][i_1][0] := 0$

operation $m$:
for $i_0 := 0 \ldots 49 \rightarrow$
    for $i_1 := 0 \ldots 49 \rightarrow$
        for $i_2 := 0 \ldots 49 \rightarrow$
            $Z[i_0][i_1][i_2+1] := Z[i_0][i_1][i_2] + X[i_0][i_2] * Y[i_2][i_1]$
```

Figure 3.1. A program for matrix multiplication consisting of two loop nests.



$$\boldsymbol{I}(l) = \begin{bmatrix} 49 \\ 49 \end{bmatrix} \quad \boldsymbol{A}(p) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \quad \boldsymbol{b}(p) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

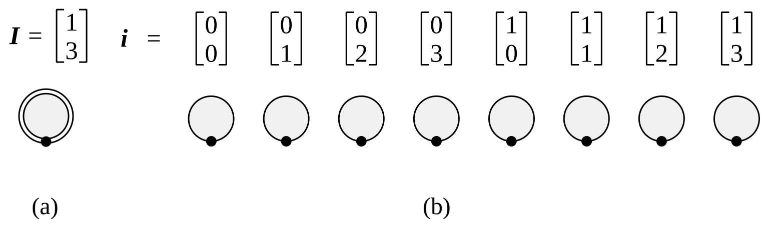
$$\boldsymbol{A}(r) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \boldsymbol{A}(s) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \boldsymbol{b}(r) = \boldsymbol{b}(s) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\boldsymbol{I}(m) = \begin{bmatrix} 49 \\ 49 \\ 49 \end{bmatrix} \quad \boldsymbol{A}(q), \boldsymbol{A}(t) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \boldsymbol{b}(q) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{b}(t) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Figure 3.2. Program graph of the matrix multiplication example of Figure 3.1. The graph consists of two operations, denoted by a double circle, and annotated with their iterator bound vectors. The black dots represent ports of the operations, which are annotated with their index matrices and their index offset vectors. The meaning of the symbols is explained in the text.

is depicted in Figure 3.2. Here, operation l corresponds to the assignment in the first loop nest, and operation m to the assignment in the second loop nest. In the program the array expressions are the arguments of the operations. The second loop nest in the example has three read arguments, $Z[i_0][i_1][i_2]$, $X[i_0][i_2]$, and $Y[i_2][i_1]$, and one write argument, $Z[i_0][i_1][i_2+1]$. In the program graph these arguments are modelled by ports and denoted by black dots in the figure. In general an operation can have multiple read ports and multiple write ports. Operations without read ports or without write ports are the input and output ports of the program graph. A program graph is formally defined as follows.

Definition 3.1 (program graph). A program graph $\mathcal{G}$ is given by a 7-tuple



$\boldsymbol{I} = \begin{bmatrix}1\\3\end{bmatrix}$
$\boldsymbol{i} = \begin{bmatrix}0\\0\end{bmatrix} \begin{bmatrix}0\\1\end{bmatrix} \begin{bmatrix}0\\2\end{bmatrix} \begin{bmatrix}0\\3\end{bmatrix} \begin{bmatrix}1\\0\end{bmatrix} \begin{bmatrix}1\\1\end{bmatrix} \begin{bmatrix}1\\2\end{bmatrix} \begin{bmatrix}1\\3\end{bmatrix}$
(a)
(b)

Figure 3.3. (a) An operation with two dimensions of repetition ($\delta = 2$) and one port. (b) The individual executions of the operation are denoted by means of single circles.

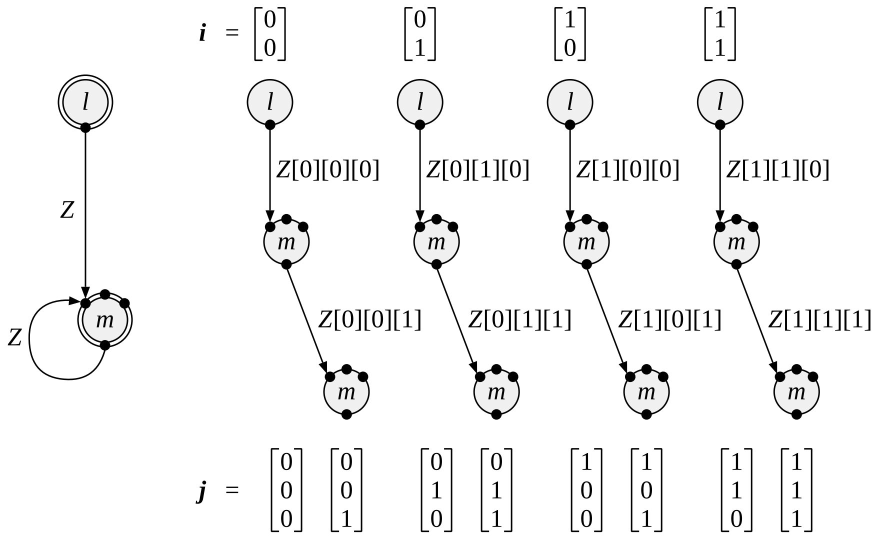
$(V,R,W,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, where

- V is a finite set of multidimensional operations,
- $R(v)$ denotes a set of *operation read ports*, for each operation $v \in V$,
- $W(v)$ denotes a set of *operation write ports*, for each $v \in V$,
- $\boldsymbol{I}(v) \in \mathbb{N}^{\delta(v)}$ denotes an *iterator bound vector*, for each $v \in V$,A
- $E \subseteq W \times R$ is a set of edges representing data dependencies, where $W = \bigcup_{v \in V} W(v)$ and $R = \bigcup_{v \in V} R(v)$,
- $\boldsymbol{A}(p) \in \mathbb{Z}^{\alpha(p) \times \delta(v)}$ denotes an *index matrix*, for each $v \in V$ and $p \in P(v) = R(v) \cup W(v)$,
- $\boldsymbol{b}(p) \in \mathbb{Z}^{\alpha(p)}$ denotes an *index offset vector*, for each $p \in P = R \cup W$.

□

Here, $\delta(v)$ denotes the number of loops by which operation v is enclosed, where the upper bounds of the loops are given by a vector $\boldsymbol{I}(v)$ with length $\delta(v)$. This means that in each loop $k = 0,\dots,\delta(v)-1$ the iterator ranges from 0 to $\boldsymbol{I}_k(v)$. For instance, operation l in the matrix multiplication example of Figure 3.1 has $\delta(l) = 2$ surrounding loops and iterator bound vector $\boldsymbol{I}(l) = \begin{bmatrix}49 & 49\end{bmatrix}^{\mathrm{T}}$. An execution of an operation v can thus be characterised by a vector $\boldsymbol{i} \in \mathbb{Z}^{\delta(v)}$, with $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(v)$. The set of all iterator vectors for an operation v is called the *iterator space*, and denoted by $\mathcal{I}(v) = \{\, \boldsymbol{i} \in \mathbb{Z}^{\delta(v)} \mid \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(v) \,\}$.

In Figure 3.3 all executions are depicted for an operation with iterator bound vector $\boldsymbol{I} = \begin{bmatrix}1 & 3\end{bmatrix}^{\mathrm{T}}$. An operation v without enclosing loops has dimension of repetition $\delta(v) = 0$. This operation has exactly one execution, which is denoted by the empty vector $[\,]$. In the same way, we talk about executions of ports. The iterator space of a port $p \in P(v)$ of an operation $v \in V$ is given by $\mathcal{I}(p) = \mathcal{I}(v)$



$\boldsymbol{i} = \begin{bmatrix}0\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\end{bmatrix}$ $\begin{bmatrix}1\\0\end{bmatrix}$ $\begin{bmatrix}1\\1\end{bmatrix}$
l
m
Z
$Z[0][0][0]$
$Z[0][1][0]$
$Z[1][0][0]$
$Z[1][1][0]$
$Z[0][0][1]$
$Z[0][1][1]$
$Z[1][0][1]$
$Z[1][1][1]$
$\boldsymbol{j} = \begin{bmatrix}0\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\1\end{bmatrix}$ $\begin{bmatrix}0\\1\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\\1\end{bmatrix}$ $\begin{bmatrix}1\\0\\0\end{bmatrix}$ $\begin{bmatrix}1\\0\\1\end{bmatrix}$ $\begin{bmatrix}1\\1\\0\end{bmatrix}$ $\begin{bmatrix}1\\1\\1\end{bmatrix}$

Figure 3.4. An example of data dependencies of the 2×2 version of the matrix multiplication example of Figure 3.1, depicted by means of edges between port executions. The edges are labelled with the array elements associated with the dependencies.

and its dimension by $\delta(p) = \delta(v)$. The set of all executions of port p is denoted by $\mathcal{E}(p) = \{\, (p,\boldsymbol{i}) \mid \boldsymbol{i} \in \mathcal{I}(p)\,\}$. We use $\mathcal{E} = \bigcup_{p\in P} \mathcal{E}(p)$ for the set of all port executions.

Data consumption and data production of operations is described by ports. Each operation has a set of read ports, along which it reads data, and a set of write ports, on which it writes data. Data consumed or produced by an execution $\boldsymbol{i}$ of port p are described by an element of a multidimensional array with $\alpha(p)$ dimensions. The array element that is accessed by execution $\boldsymbol{i}$ of port p is given by an *index vector* $\boldsymbol{n}(p,\boldsymbol{i}) \in \mathbb{Z}^{\alpha(p)}$ as a linear expression in $\boldsymbol{i}$ by

$$\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{A}(p)\boldsymbol{i} + \boldsymbol{b}(p),$$

using the index matrix $\boldsymbol{A}(p)$ and index vector $\boldsymbol{b}(p)$. For instance, for port s of our matrix multiplication example of Figure 3.1 we have

$$\begin{aligned}\boldsymbol{n}(s,\boldsymbol{i}) &= \boldsymbol{A}(s)\boldsymbol{i} + \boldsymbol{b}(s)\\ &= \begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\end{bmatrix}\begin{bmatrix}i_0\\ i_1\\ i_2\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix}\\ &= \begin{bmatrix}i_2\\ i_1\end{bmatrix}.\end{aligned}$$

In Figure 3.2 the index matrices and index offset vectors for all ports in the matrix multiplication example are given.

Data dependencies are described by edges between ports; the presence of an edge $(p,q) \in E$ means that data produced by an execution $\boldsymbol{i}$ of write port p is consumed by an execution $\boldsymbol{j}$ of read port q if they refer to the same array element, i.e., $\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})$. Figure 3.4 depicts all data dependences of the matrix multiplication example.

In order to identify which ports access same array, we introduce the notion of array clusters.

Definition 3.2 (array clusters). Two ports p and q are said to access the same array, denoted by $p \bowtie q$, if they are weakly connected in the program graph, i.e., there is a list of ports $(p_0,...,p_n)$, with $n \geq 0$, $p_0 = p$, $p_n = q$, and

$$(p_i, p_{i+1}) \in E \ \vee \ (p_{i+1}, p_i) \in E,$$

for each $i = 0,\dots,n-1$. Now, an array cluster $A \subseteq P$ is defined as a set of weakly connected ports, i.e., $p \bowtie q$ for all $p,q \in A$, such that the set cannot be extended. The set of all array clusters is denoted by $\mathcal{A}$. □

In our matrix multiplication example of Figure 3.1 ports p, q, and t are weakly connected, so they form an array cluster $\{p,q,t\}$. Ports r and s are not connected to any other port. Hence for this example we have array clusters $\mathcal{A} = \{\{p,q,t\},\{r\},\{s\}\}$. These three clusters correspond to the arrays Z, X, and Y, respectively, in Figure 3.1.

In order to make data dependencies unique, we introduce a so-called *single-assignment* property, which is formally defined as follows.

Definition 3.3 (single assignment). An array $A \in \mathcal{A}$ is said to have the single-assignment property if and only if for each $p,q \in A \cap W$, each execution $\boldsymbol{i} \in \mathcal{I}(p)$, and each execution $\boldsymbol{j} \in \mathcal{I}(q)$ we have

$$\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \ \Rightarrow \ p = q \ \wedge \ \boldsymbol{i} = \boldsymbol{j}.$$

□

The single-assignment property means that each array element may be written at most once.

3.2 Schedules

So far we have not discussed the order of execution of operations, nor the actual memory positions for array elements. These are the decision variables of our scheduling problem. A schedule is defined by a *time assignment*, which gives a partial order on the execution of operations, and an *address assignment*, which

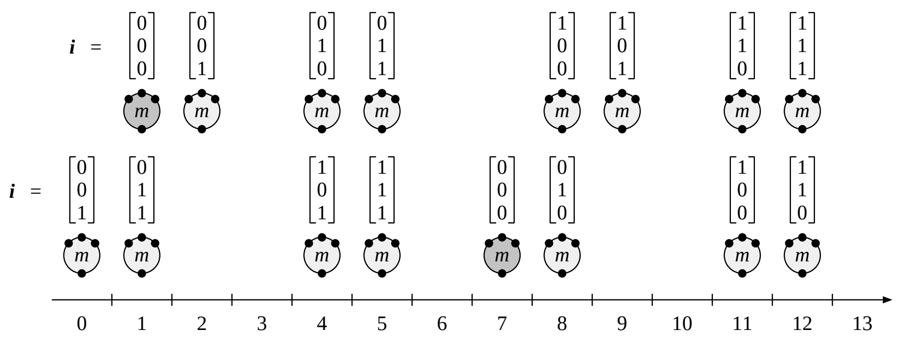


Figure 3.5. Two possible time assignments for operation m of the 2×2 variant of the matrix multiplication example of Figure 3.1. The first assignment has start time $s(m) = 1$ and period vector $\boldsymbol{p}(m) = \begin{bmatrix}7 & 3 & 1\end{bmatrix}^{\mathrm{T}}$, the second one has start time $s(m) = 7$ and period vector $\boldsymbol{p}(m) = \begin{bmatrix}4 & 1 & -7\end{bmatrix}^{\mathrm{T}}$.

gives a mapping of array elements onto actual memory locations, which are also called addresses.

First we discuss the time assignment. In our model we use time only to define a partial order on the execution of operations. A time assignment, for instance, that assigns times 4 and 9 to two executions e and f of operations, respectively, merely demands that e is executed before f. If two executions e and f are assigned the same time, these executions are not ordered by the schedule. In this case e may be executed before f or the other way around.

A time assignment is defined as follows.

Definition 3.4 (time assignment). Given is a program graph $(V, R, W, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$. Then a time assignment τ is a pair $(\boldsymbol{p}, s)$, where

- $\boldsymbol{p}(v) \in \mathbb{Z}^{\delta(v)}$ denotes a *period vector*, for each operation $v \in V$, and
- $s(v) \in \mathbb{Z}$ denotes a *start time* for each operation $v \in V$.

□

The start time and period vector of an operation $v \in V$ fix the time of execution $\boldsymbol{i}$ of operation v through the expression

$$t(v, \boldsymbol{i}) = \boldsymbol{p}^{\mathrm{T}}(v)\boldsymbol{i} + s(v).$$

The start time is the time at which execution $\boldsymbol{i} = \boldsymbol{0}$ of operation v takes place. In general periods p_j may be negative. As a result, the start time of an operation can be different from the time at which the first execution of an operation takes place.

Two possible time assignments for the matrix multiplication example of Figure 3.1 are shown in Figure 3.5. For the first one, execution $\boldsymbol{i}$ of operation m takes place at

$$t(m,\boldsymbol{i}) = \boldsymbol{p}^{\mathrm{T}}(m)\boldsymbol{i} + s(m) = \begin{bmatrix} 7 & 3 & 1 \end{bmatrix} \begin{bmatrix} i_0 \\ i_1 \\ i_2 \end{bmatrix} + 1 = 7i_0 + 3i_1 + i_2 + 1.$$

The time assignment also determines the order of execution of all ports. Each execution $\boldsymbol{i}$ of port $p \in P(v)$ of operation v takes place at the same moment as execution $\boldsymbol{i}$ of the operation itself, i.e., $t(p,\boldsymbol{i}) = t(m,\boldsymbol{i})$. Therefore, we define $\boldsymbol{p}(p) = \boldsymbol{p}(v)$ and $s(p) = s(v)$ for all ports $p \in P(v)$.

Besides a time assignment we need a way to determine the address on which each array element is stored in memory.

Definition 3.5 (address assignment). Given a program graph $(V,R,W,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, an address assignment μ is a pair $(\boldsymbol{c},o)$, where

- $\boldsymbol{c}(A) \in \mathbb{Z}^{\alpha(A)}$ denotes an *address coefficient vector,* for each $A \in \mathcal{A}$, and
- $o(A) \in \mathbb{Z}$ denotes an *address offset,* for each $A \in \mathcal{A}$.

□

The address assignment μ fixes the address of array element with index vector $\boldsymbol{n}$ of array cluster $A \in \mathcal{A}$ by

$$a(A,\boldsymbol{n}) = \boldsymbol{c}^{\mathrm{T}}(A)\boldsymbol{n} + o(A).$$

The address at which the data belonging to execution $\boldsymbol{i}$ of port $p \in A$ is then given by

$$a(p,\boldsymbol{i}) = a(A,\boldsymbol{n}(p,\boldsymbol{i})) = \boldsymbol{c}^{\mathrm{T}}(A)\boldsymbol{n}(p,\boldsymbol{i}) + o(A).$$

Definition 3.6 (schedule). Given a program graph $\mathcal{G}$, a schedule $\sigma = (\tau,\mu)$ is the combination of a time assignment τ and an address assignment μ. □

3.3 Constraints

Constraints on schedules are introduced as not all schedules are valid. Precedence constraints limit the possible time assignments by demanding that each execution of a write port that refers an array element takes place before all executions of read ports that refer the same array element.

Definition 3.7 (precedence constraints). Given are a program graph $\mathcal{G} = (V,R,W,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$ and a time assignment $\tau = (\boldsymbol{p},s)$. Then, for each execution $\boldsymbol{i}$ of a write port $p \in W$, and for each execution $\boldsymbol{j}$ of a read port $q \in R$, with $(p,q) \in E$, the precedence constraints specify that

$$\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \;\Rightarrow\; t(p,\boldsymbol{i}) < t(q,\boldsymbol{j}).$$

□

A time assignment τ is called a *feasible time assignment* with respect to a program graph $\mathcal{G}$ if all precedence constraints are met. For the time assignments in Figure 3.5 the first assignment is feasible. The second one is infeasible as none of the precedence constraints for ports q and t are satisfied.

Next, we have address constraints, demanding that the memory location of an array element may be not overwritten during its lifetime. The lifetime of an array element is the time interval from the production until the last consumption. An address conflict for write ports p, r, and read port q occurs if there are executions of p and q that access the same array element, i.e., the same index vector of the same array, and the address belonging to that array element has been overwritten by an execution of r between the executions of p and q.

Definition 3.8 (address constraints). Given are a program graph $\mathcal{G} = (V,R,W,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, a feasible time assignment $\tau = (\boldsymbol{p},s)$, and an address assignment $\mu = (\boldsymbol{c},o)$. Then, for each execution $\boldsymbol{i}$ of a write port $p \in W$, and for each execution $\boldsymbol{j}$ of a read port $q \in R$, with $(p,q) \in E$ and $\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})$, the address constraints specify that for each execution $\boldsymbol{k}$ of a write port $r \in W$

$$a(p,\boldsymbol{i}) = a(r,\boldsymbol{k}) \ \wedge \ t(p,\boldsymbol{i}) \leq t(r,\boldsymbol{k}) < t(q,\boldsymbol{j}) \ \Rightarrow \ p = r \ \wedge \ \boldsymbol{i} = \boldsymbol{k}$$

if r and q belong to the same operation, or

$$a(p,\boldsymbol{i}) = a(r,\boldsymbol{k}) \ \wedge \ t(p,\boldsymbol{i}) \leq t(r,\boldsymbol{k}) \leq t(q,\boldsymbol{j}) \ \Rightarrow \ p = r \ \wedge \ \boldsymbol{i} = \boldsymbol{k}$$

if r and q belong to different operations. □

We need stronger address constraints for the case where ports $r \in W$ and $q \in R$ belong to different operations, as at time $t(r,\boldsymbol{k}) = t(q,\boldsymbol{j})$ executions $\boldsymbol{j}$ of port q and $\boldsymbol{k}$ of port r are not ordered. For the case where ports r and q belong to the same operation, we assume that for an execution of this operation the execution of read port q occurs before the execution of write port r, resulting in weaker address constraints.

The address constraint for ports p, q, and r with $(p,q) \in E$ is denoted by $(p \to q,r)$. The address constraints for the matrix multiplication example of Figure 3.1 are $(p \to q,p)$, $(p \to q,t)$, $(t \to q,p)$, and $(t \to q,t)$.

An address assignment μ is called a *feasible address assignment* with respect to a program graph $\mathcal{G}$ and time assignment τ if all address constraints are met.

A schedule (τ,μ) is called a *feasible schedule* with respect to a program graph $\mathcal{G}$ if τ is feasible with respect to $\mathcal{G}$ and μ is feasible with respect to $\mathcal{G}$ and τ.

3.4 Objectives

The objective of our scheduling problem is to minimise the number of cache misses of a program. When computing the number of cache misses for an execution of a schedule, we need knowledge about the size of the cache, the associativity, the block size, and the replacement policy. As LRU is the most effective replacement scheme in practice, we assume LRU replacement in the remainder of this chapter. The following definition of a cache together with a schedule contains sufficient information for determining the number of cache misses.

Definition 3.9 (cache). A cache $\mathcal{C}$ is a triple (c_s, s_s, b_s), where

- $c_s \in \mathbb{N}_+$ denotes the size of the cache in blocks,
- $s_s \in \mathbb{N}_+$ with $s_s \mid c_s$, denotes the associativity, or, the size of each cache *set* in blocks,
- $b_s \in \mathbb{N}_+$ denotes the size of each *block* in bytes.

The number of sets of which a cache consists is given by $n_s = \frac{c_s}{s_s}$. An address a maps to set $s(a) = a \bmod n_s$. The block address of an address a is $a_b(a) = a \operatorname{div} b_s$. □

3.4.1 LRU cache model

We give a formal model of cache misses in a cache with LRU replacement. During the execution of a program, a CPU accesses an address $a(e)$ for each execution $e \in \mathcal{E}$ of a port.

A so-called *compulsory miss* for port execution $e \in \mathcal{E}$ occurs if it is the first to access block address $a_b(e)$. To determine this, we define $M(e)$ as the set of port executions that access block address $a_b(e) = a(e) \operatorname{div} b_s$ and that execute before e, i.e.,

$$M(e) = \{\, f \in \mathcal{E} \mid a_b(f) = a_b(e) \;\wedge\; t(f) < t(e) \,\}.$$

A compulsory miss for e occurs if no port executions access $a_b(e)$ before e, i.e.,

$$comp(e) \;\Leftrightarrow\; M(e) = \emptyset. \tag{3.1}$$

If port execution e is the first port execution that accesses block address $a_b(e)$, and there is another port execution f that accesses $a_b(e)$ at the same time $t(f) = t(e)$, then both e and f will cause a compulsory miss in this cache model. Counting these misses twice can be avoided by assigning different times to these port executions.

A so-called *expiration miss* for port execution $e \in \mathcal{E}$ in a fully set-associative cache occurs if there are too many port executions accessing block addresses different from $a_b(e)$ between e and the *most recent* port execution preceding e accessing $a_b(e)$. We denote the time at which this port execution takes place by

$$m(e) = \max_{f \in M(e)} t(f).$$

The set of block addresses accessed between $m(e)$ and $t(e)$ is given by

$$Ac(e) = \{\, a_b(f) \mid f \in \mathcal{E} \;\wedge\; m(e) \le t(f) \le t(e) \,\}.$$

A port execution e causes an expiration miss in a fully set-associative cache if and only if e does not cause a compulsory miss and the set of block addresses between the previous access to $a_b(e)$ and e itself exceeds the size of the cache, i.e., $|Ac(e)| > c_s$. As port executions that take place at the same time are not ordered, we once again assume that all port executions $f \in \mathcal{E}$ with $t(f) = t(e)$ execute before e.

The treatment of an expiration miss in a n-way set-associative cache is analogous to treatment in the fully set-associative case, but now we are not interested in the accesses to the whole cache, but only in those to the set that e accesses, i.e., we consider the set

$$S(e) = \{ a_b(f) \mid f \in \mathcal{E} \ \wedge \ m(e) \leq t(f) \leq t(e) \ \wedge \ s(a(f)) = s(a(e)) \}.$$

Port execution e causes an expiration miss if there is not enough room in the set that is accessed by e, i.e.,

$$ex(e) \ \Leftrightarrow \ \neg comp(e) \ \wedge \ |S(e)| > s_s. \tag{3.2}$$

As $s(a(e)) = s(a(f))$ for all executions e and f in the fully set-associative case, $S(e) = Ac(e)$, and hence (3.2) is a general characterisation of expiration misses, irrespective of the associativity.

Every port execution causes either a hit or a miss in the cache. As a port execution cannot cause both a compulsory miss and an expiration miss, the total number of cache misses is the sum of the number of compulsory misses and the number of expiration misses, i.e.,

$$\begin{aligned} c_{\mathcal{G},\mathcal{C}}(\sigma) &= |\{ e \in \mathcal{E} \mid comp(e) \ \vee \ ex(e) \}| \\ &= |\{ e \in \mathcal{E} \mid comp(e) \}| + |\{ e \in \mathcal{E} \mid ex(e) \}|. \end{aligned}$$

The miss rate is the fraction of port executions that causes a miss, and is given by

$$\varphi_{\mathcal{G},\mathcal{C}}(\sigma) = \frac{c_{\mathcal{G},\mathcal{C}}(\sigma)}{|\mathcal{E}|}.$$

3.4.2 FIFO cache model

We can do the same exercise for caches employing a *least recently replaced* strategy. In literature this replacement scheme is referred to as FIFO.

As the occurrence of a miss depends on data being available in the cache, we are interested in knowing when data enter the cache. If a block address is accessed for the first time during the execution of a schedule, a compulsory miss is caused. Data will then remain in the cache up to the s_s^{th} miss after the data entered the cache. As before, we define

$$M(e) = \{ f \in \mathcal{E} \mid a_b(f) = a_b(e) \ \wedge \ t(f) < t(e) \},$$

and again a compulsory miss occurs if and only if $M(e) = \emptyset$.

For expiration misses we are interested in the moment the last miss on $a_0(e)$ occurred, which is given by

$$l(e) = \max_{f \in M(e) \wedge miss(f)} t(f).$$

We then count the number of misses since the last miss, by considering the set

$$S(e) = \{\, f \in \mathcal{E} \mid miss(f) \wedge l(e) \leq t(f) < t(e) \wedge s(a(e)) = s(a(f)) \,\},$$

and we derive that a miss occurrence is given by

$$miss(e) \Leftrightarrow M(e) = \emptyset \vee |S(e)| > s_s.$$

The definition for $miss(e)$ is indeed a valid definition as $miss(f)$ is used in the definition for $f(e)$ and $S(e)$ only for $t(f) < t(e)$. The number of cache misses $c_{\mathcal{G},\mathcal{C}}(\sigma)$ and miss rate $\varphi_{\mathcal{G},\mathcal{C}}(\sigma)$ are defined in a similar way as we did for the LRU cache. In the remainder of this thesis the number of cache misses $c_{\mathcal{G},\mathcal{C}}(\sigma)$ and the miss rate $\varphi_{\mathcal{G},\mathcal{C}}(\sigma)$ refer to caches with LRU replacement, as LRU is more effective than FIFO [Chrobak & Noga, 1999].

3.5 Formal problem statement

We now can define the problem that is studied in this thesis.

Definition 3.10 (multidimensional periodic cache scheduling (MPCS)). Given a program graph $\mathcal{G}$ and a cache $\mathcal{C}$, find a feasible schedule σ for which $c_{\mathcal{G},\mathcal{C}}(\sigma)$ is minimal. □

3.6 Special properties

In this section special properties of program graphs and schedules are discussed that are often used in the remainder of this thesis.

3.6.1 Lexicographical executions

A loop nest is said to have lexicographical executions if the periods are ordered in such a way that all executions within an inner dimension take place in the period of an outer dimension. For weak lexicographical executions the first and last execution in an inner dimension of two successive executions of the outer dimension may take place at the same time. For an example see Figure 3.6, where operation u has lexicographical executions, operation v only has the weak lexicographical property, and w has no lexicographical property. In the following definition we formalise these properties.

Definition 3.11 (lexicographical execution). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$ and a period vector $\boldsymbol{p} \in \mathbb{Z}^\delta$. The iterator bound vector and period vector

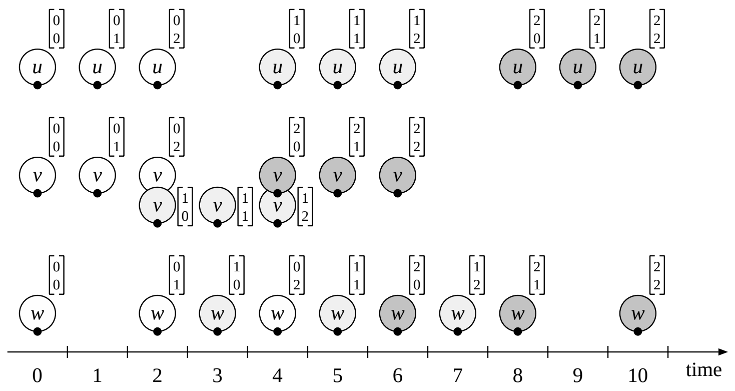


Figure 3.6. Three operations u, v, and w with iterator bound vector $\boldsymbol{I} = [2\ 2]^{\mathrm{T}}$, start time 0 and period vectors $\boldsymbol{p}(u) = [4\ 1]^{\mathrm{T}}$, $\boldsymbol{p}(v) = [2\ 1]^{\mathrm{T}}$, and $\boldsymbol{p}(w) = [3\ 2]^{\mathrm{T}}$. The executions of the outer dimension have been shaded differently.

are said to have the lexicographical execution property, denoted by $\mathrm{lex}(\boldsymbol{I},\boldsymbol{p})$, if and only if for all vectors $\boldsymbol{i},\boldsymbol{j} \in \mathbb{Z}^{\delta}$ with $\boldsymbol{0} \leq \boldsymbol{i},\boldsymbol{j} \leq \boldsymbol{I}$ holds

$$\boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{j} \ \Leftrightarrow\ \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} < \boldsymbol{p}^{\mathrm{T}}\boldsymbol{j}.$$

The vectors have the *weak* lexicographical property, denoted by $\mathrm{wlex}(\boldsymbol{I},\boldsymbol{p})$, if and only if for all vectors $\boldsymbol{i},\boldsymbol{j} \in \mathbb{Z}^{\delta}$ with $\boldsymbol{0} \leq \boldsymbol{i},\boldsymbol{j} \leq \boldsymbol{I}$ holds

$$\boldsymbol{i} \leq_{\mathrm{lex}} \boldsymbol{j} \ \Rightarrow\ \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \leq \boldsymbol{p}^{\mathrm{T}}\boldsymbol{j}.$$

□

In this definition, $\boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{j}$ holds if and only if an $m \in \{0,\ldots,\delta-1\}$ exists such that $i_m < j_m$, and $i_l = j_l$ for all $l = 0,\ldots,m-1$. The following characterisations of the execution properties can be checked in polynomial time. They are analogous to the characterisations given by Verhaegh [1995]. The lexicographical execution $\mathrm{lex}(\boldsymbol{I},\boldsymbol{p})$ holds if and only if

$$p_k > \sum_{l=k+1}^{\delta-1} p_l I_l \ \wedge\ p_k > 0,$$

for all $k = 0,\ldots,\delta-1$, and the weak lexicographical execution property $\mathrm{wlex}(\boldsymbol{I},\boldsymbol{p})$ holds if and only if

$$p_k \geq \sum_{l=k+1}^{\delta-1} p_l I_l \ \wedge\ p_k \geq 0,$$

for all $k = 0,\ldots,\delta-1$.

3.6.2 Lexicographical index orderings

The lexicographical index ordering property is similar to the lexicographical execution property. It says that two executions are ordered lexicographically if and only if their corresponding indices are ordered lexicographically.

Definition 3.12 (lexicographical index ordering). An iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$ and index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$ are said to have the lexicographical index ordering property, denoted by $\mathrm{lio}(\boldsymbol{I},\boldsymbol{A})$ if and only if for all vectors $\boldsymbol{i},\boldsymbol{j} \in \mathbb{Z}^\delta$ with $\boldsymbol{0} \leq \boldsymbol{i},\boldsymbol{j} \leq \boldsymbol{I}$ holds

$$\boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{j} \ \Leftrightarrow\ \boldsymbol{Ai} <_{\mathrm{lex}} \boldsymbol{Aj}.$$

□

As proven by Verhaegh [1995], the lexicographical index ordering can be verified in polynomial time by checking whether

$$\boldsymbol{A}_{\cdot k} >_{\mathrm{lex}} \sum_{l=k+1}^{\delta-1} A_{\cdot l} I_l \ \wedge\ \boldsymbol{A}_{\cdot k} >_{\mathrm{lex}} \boldsymbol{0},$$

for all $k = 0,\dots,\delta-1$.

3.6.3 Unique address assignment

An address assignment is called a *unique address assignment* if each array element is mapped onto a unique address.

Definition 3.13 (unique address assignment). Given a program graph $\mathcal{G}$, we say that an address assignment μ is a *unique address assignment* if for every pair of port executions $(p,\boldsymbol{i})$ and $(q,\boldsymbol{j})$ the corresponding data are stored on the same address only if they access the same array index of the same array, i.e.,

$$a(p,\boldsymbol{i}) = a(q,\boldsymbol{j}) \ \Rightarrow\ \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \ \wedge\ p \bowtie q.$$

□

Theorem 3.1. *For a program graph $\mathcal{G}$, a unique address assignment exists, and can be computed in polynomial time.*

Proof. First, we construct a unique address assignment for every array cluster $A \in \mathcal{A}$ by means of a *unique coefficient assignment*. For a given set of index vectors $N \subset \mathbb{Z}^\alpha$, a coefficient vector $\boldsymbol{c} \in \mathbb{Z}^\alpha$ is a unique coefficient assignment if for all index vectors $\boldsymbol{n},\boldsymbol{n}' \in N$, we have

$$\boldsymbol{c}^{\mathrm{T}}\boldsymbol{n} = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{n}' \ \Rightarrow\ \boldsymbol{n} = \boldsymbol{n}'.$$

If we choose $\boldsymbol{x}$ and $\boldsymbol{y} \in \mathbb{Z}^\alpha$ in such a way that $\boldsymbol{x} \leq \boldsymbol{n} \leq \boldsymbol{y}$ for all $\boldsymbol{n} \in N$, for example by taking $x_l = \min\ \{n_l \mid \boldsymbol{n} \in N\}$ and $y_l = \max\ \{n_l \mid \boldsymbol{n} \in N\}$ for all $l = 0,\dots,\alpha-1$, then the vector $\boldsymbol{c} \in \mathbb{Z}^{\alpha}$ with

$$c_k = \prod_{i=k+1}^{\alpha-1} (y_i - x_i + 1) \quad \text{for all } k = 0, \ldots, \alpha - 1,$$

is a unique coefficient assignment. Hence, for array cluster A we construct a unique coefficient assignment $\boldsymbol{c}(A)$ by choosing $N = \{\, \boldsymbol{n}(p, \boldsymbol{i}) \mid p \in A \land \boldsymbol{i} \in \mathcal{I}(p) \,\}$. We can now choose the offsets $o(A)$ in such a way that the addresses of different array clusters are disjoint. This construction of a unique address assignment takes time $\mathcal{O}(\alpha|\mathcal{A}|)$, where α is the largest array dimension of any array cluster. □

Unique address assignments have the property that they are feasible for any time assignment.

Theorem 3.2. *For a program graph $\mathcal{G}$, a unique address assignment μ is feasible for any time assignment.*

Proof. Let $(p, \boldsymbol{i})$ and $(r, \boldsymbol{k})$ be executions of write ports, and $(q, \boldsymbol{j})$ an execution of a read port of $\mathcal{G}$. Furthermore, assume that $(p, \boldsymbol{i})$ and $(r, \boldsymbol{k})$ access the same address, i.e., $a(p, \boldsymbol{i}) = a(r, \boldsymbol{k})$. As μ is a unique address assignment, we know that $\boldsymbol{n}(p, \boldsymbol{i}) = \boldsymbol{n}(r, \boldsymbol{k})$ and $p \bowtie r$. Because of the single-assignment property we find that $p = r$ and $\boldsymbol{i} = \boldsymbol{k}$, and hence the address constraints are met. As we did not use the time assignment in this proof, we know that a unique address assignment is feasible for any time assignment. □

In the construction of a unique address assignment the memory locations of two arrays do not overlap. In general this property is called a *private array assignment*.

Definition 3.14 (private array assignment). An address assignment μ is called a private array assignment if and only if for all executions $\boldsymbol{i}$ of operation p and for all executions $\boldsymbol{j}$ of q holds

$$a(p, \boldsymbol{i}) = a(q, \boldsymbol{j}) \;\Rightarrow\; p \bowtie q.$$

□

4

Complexity Analysis

This chapter deals with the complexity of the multidimensional periodic scheduling problem as formulated in Definition 3.10. In Section 4.1 the complexity of computing the cost function is studied. In Section 4.2 the complexity of checking address constraints is discussed. The scheduling problem itself is handled in Section 4.3. For the theory of NP-completeness and complexity in general we refer to Papadimitriou [1995] and to Garey & Johnson [1979].

4.1 Computing the cost of a schedule

We address the complexity of determining the number of cache misses for a given schedule and a given cache. In this section the general problem is studied as well as a special case in which lexicographical index orderings are assumed.

Definition 4.1 (cache cost computation problem (CCCP)). For a given program graph $\mathcal{G}$, a feasible schedule $\sigma = (\tau, \mu)$, and a cache $\mathcal{C}$, determine the cost of the schedule, i.e., compute $c_{\mathcal{G},\mathcal{C}}(\sigma)$. □

In complexity theory, problems are usually formulated as decision problems. We are primarily interested in the number of cache misses, but the decision variant where we decide if the number of cache misses does not exceed a given integer K is in a sense as difficult. If CCCP is solvable in polynomial time, then its decision variant is also solvable in polynomial time. If we can solve the decision variant in polynomial time, say $\mathcal{O}(f(n))$ for CCCP-instances of size n, by a certain algorithm A, then we can apply a binary search [Papadimitriou, 1995] by repeatedly applying A. Since the number of cache misses is bounded from above by the total number of port executions, $|\mathcal{E}|$, the binary search will take time $\mathcal{O}(\log(|\mathcal{E}|)\,f(n))$, which is bounded by a polynomial in the size of the problem instance.

4.1.1 Cache cost computation

For a given program graph, schedule, and cache we want to compute the number of cache misses. In this section we show that this problem cannot be solved efficiently unless P = NP. Below, we give a sketch of the proof, of which the details can be found in Theorem 4.2.

First of all we observe that if the cache is large enough to contain all data that are accessed during the execution of the schedule, then conflict misses will occur only because of limited associativity of the cache.

Hence, for fully set-associative caches of sufficient size, all misses are compulsory misses. A compulsory miss occurs for each first access to a block address. Since each execution of a port accesses exactly one block address, an upper bound on the number of distinct block addresses accessed by the execution of a schedule is the number of executions of all ports, $|\mathcal{E}|$.

Suppose that we have a fully set-associative cache of size $|\mathcal{E}|$. Now we are left with the problem of determining the number of compulsory misses, which can be computed by counting the number of distinct block addresses accessed by the schedule. Suppose that we have a set P of multidimensional periodic ports with index matrices $\boldsymbol{A}(p)$, index offset vectors $\boldsymbol{b}(p)$, iterator bound vectors $\boldsymbol{I}(p)$ for $p \in P$, then the number of block addresses accessed by the schedule is given by

$$|\{\, a_{\mathrm{b}}(p,\boldsymbol{i}) \mid p \in P \,\wedge\, \boldsymbol{i} \in \mathcal{I}(p) \,\}|\,. \tag{4.1}$$

For a program graph with two ports p and q, of which q executes only once, an address assignment that maps each index vector onto a *unique* address, and a cache with $b_{\mathrm{s}} = 1$, we can rewrite (4.1) into

$$\left|\{\, \boldsymbol{A}(p)\boldsymbol{i} \mid \boldsymbol{i} \in \mathbb{Z}^{\delta(p)} \,\wedge\, \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I}(p) \,\} \cup \{\boldsymbol{b}(q)\}\right|, \tag{4.2}$$

and for a program graph consisting of only port p into

$$\left|\{\, \boldsymbol{A}(p)\boldsymbol{i} \mid \boldsymbol{i} \in \mathbb{Z}^{\delta(p)} \,\wedge\, \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I}(p) \,\}\right|. \tag{4.3}$$

If it is possible to compute (4.2) and (4.3) in polynomial time it is also possible to determine in polynomial time whether or not an solution $\boldsymbol{i} \in \mathbb{Z}^{\delta(p)}$ exists to

$$\boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I} \,\wedge\, \boldsymbol{A}(p)\boldsymbol{i} = \boldsymbol{b}(q). \tag{4.4}$$

However, solving (4.4) is as difficult as solving *zero-one integer programming* (ZOIP) [Garey & Johnson, 1979], as we show by a reduction.

Definition 4.2 (zero-one integer programming (ZOIP)). Given are a matrix $\boldsymbol{M} \in \mathbb{Z}^{m\times n}$, a vector $\boldsymbol{d} \in \mathbb{Z}^m$, a vector $\boldsymbol{c} \in \mathbb{Z}^n$, and an integer B. Determine whether there is a vector $\boldsymbol{x} \in \{0,1\}^n$ such that $\boldsymbol{M}\,\boldsymbol{x} = \boldsymbol{d}$ and $\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \geq B$. □

ZOIP is NP-complete in the strong sense [Garey & Johnson, 1979]. The problem in (4.4) is formulated as follows.

Definition 4.3 (bounded integer solution for linear equations (SLE)). Given are a matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha\times\delta}$, a vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$, and a vector $\boldsymbol{I} \in \mathbb{N}^{\delta}$. Determine whether a vector $\boldsymbol{i} \in \mathbb{Z}^{\delta}$ exists for which $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \wedge \boldsymbol{A}\boldsymbol{i} = \boldsymbol{b}$. □

For establishing the complexity of SLE, we transform an instance of ZOIP to an instance of SLE by adding a slack variable. The inequality $\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \geq B$ is replaced by an equality $\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} - y = B$, where y is a non-negative slack variable.

Theorem 4.1. *SLE is NP-complete in the strong sense.*

Proof. When given a vector $\boldsymbol{i}$ we can check in polynomial time whether it satisfies $\boldsymbol{A}\boldsymbol{i} = \boldsymbol{b}$, so SLE is in NP. ZOIP is reduced to SLE by defining an instance $\mathcal{I}_{\mathrm{sle}}$ of SLE from an instance $\mathcal{I}_{\mathrm{zoip}}$ of ZOIP in the following way.

- $\delta = n+1$,
- $\alpha = m+1$,
- vector $\boldsymbol{I}$ with $I_0 = -B + \sum_{l=0}^{n-1} c_l^+$ and $I_k = 1$ for $k = 1,\ldots,\delta-1$,
- matrix $\boldsymbol{A} = \begin{bmatrix} -1 & \boldsymbol{c}^{\mathrm{T}} \\ 0 & \boldsymbol{M} \end{bmatrix}$, and
- vector $\boldsymbol{b} = \begin{bmatrix} B \\ \boldsymbol{d} \end{bmatrix}$,

where $x^+ = \max\{x,0\}$. Note that $\mathcal{I}_{\mathrm{zoip}}$ can be computed in polynomial time. Furthermore, the largest number in $\mathcal{I}_{\mathrm{sle}}$ is polynomially bounded in the largest number in $\mathcal{I}_{\mathrm{zoip}}$ and the size of $\mathcal{I}_{\mathrm{zoip}}$. Now we have

$$\exists_{\boldsymbol{i}\in\mathbb{Z}^{\delta}}\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \wedge \boldsymbol{A}\boldsymbol{i} = \boldsymbol{b}$$

$$\equiv$$

$$\exists_{\boldsymbol{i}'\in\mathbb{Z}^n}\ \exists_{i_0\in\mathbb{Z}}\ 0 \leq i_0 \leq I_0 \wedge \boldsymbol{0} \leq \boldsymbol{i}' \leq \boldsymbol{1} \wedge -i_0 + \boldsymbol{c}^{\mathrm{T}}\boldsymbol{i}' = B \wedge \boldsymbol{M}\boldsymbol{i}' = \boldsymbol{d}$$

$$\equiv$$

$$\exists_{\boldsymbol{i}'\in\{0,1\}^n}\ 0 \leq -B + \boldsymbol{c}^{\mathrm{T}}\boldsymbol{i}' \leq -B + \sum_{l=0}^{n-1} c_l^+ \wedge \boldsymbol{M}\boldsymbol{i}' = \boldsymbol{d}$$

$$\equiv$$

$$\exists_{\boldsymbol{i}' \in \{0,1\}^n} \ \boldsymbol{M}\boldsymbol{i}' = \boldsymbol{d} \wedge \boldsymbol{c}^{\mathrm{T}}\boldsymbol{i}' \geq B.$$

So using the relation $\boldsymbol{i}' = \boldsymbol{x}$ we can conclude that $\mathcal{I}_{\mathrm{zoip}}$ has a solution if and only if $\mathcal{I}_{\mathrm{sle}}$ has one. Hence, SLE is NP-complete in the strong sense. □

Above, we argued that for a fully set-associative cache of sufficient size, the number of cache misses is equal to the number of distinct block addresses that are accessed by the execution of a schedule. We prove this in the following lemma.

Lemma 4.1. *For a fully set-associative cache of size $|\mathcal{E}|$, the cost $c_{\mathcal{G},\mathcal{C}}(\sigma)$ of a schedule for which $\forall_{f,e \in \mathcal{E}} \ (e \neq f \Rightarrow t(e) \neq t(f))$ is given by*

$$|\{ a_{\mathrm{b}}(e) \mid e \in \mathcal{E} \}| \, .$$

Proof. First of all, because of the associativity and size of the cache, conflict misses cannot occur. Hence the cost of the schedule must consist entirely of compulsory misses. A compulsory miss occurs for a port execution e, if it is the first port execution that accesses block address $a_{\mathrm{b}}(e)$. Since it is not possible for two different port executions to execute at the same time, the first port execution to access a block address is unique. Hence the number of compulsory misses is equal to the number of distinct block addresses. □

Theorem 4.2. *CCCP cannot be solved in polynomial time unless P = NP.*

Proof. Let an instance $\mathcal{I}_{\mathrm{sle}}$ of SLE be given by a matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$, a vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$, and a vector $\boldsymbol{I} \in \mathbb{N}^{\delta}$. We construct an instance of CCCP as follows. The program graph $\mathcal{G} = (V, R, W, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ has

- two operations $V = \{u, v\}$,
- operation read ports $R(u) = \emptyset$, $R(v) = \{q\}$,
- operation write ports $W(u) = \{p\}$, $W(v) = \emptyset$,
- iterator bound vectors $\boldsymbol{I}(u) = [\,]$, $\boldsymbol{I}(v) = \boldsymbol{I}$,
- set of edges $E = \{(p, q)\}$,
- index matrices $\boldsymbol{A}(p) = [\,]$, $\boldsymbol{A}(q) = \boldsymbol{A}$, and
- index offset vectors $\boldsymbol{b}(p) = \boldsymbol{b}$, $\boldsymbol{b}(q) = \boldsymbol{0}$.

This choice of index matrices and index offset vectors ensures that the single-assignment property holds. A sufficient condition for the feasibility of a time assignment $\tau = (\boldsymbol{p}, s)$ for $\mathcal{G}$ is that the (only) execution of operation u takes place before all executions of operation v. Hence by taking

$$\begin{aligned} s(u) &= 0 \\ s(v) &= 1 \\ p_k(v) &= \prod_{i=k+1}^{\delta(v)-1} (I_i(v) + 1) \text{ for all } k = 0, \ldots, \delta(v) - 1, \end{aligned}$$

a feasible time assignment is created for which at most one execution of an operation takes place in each time slot. The (only) array cluster is denoted by $A = \{p,q\}$. For the address assignment μ we choose a unique address assignment as introduced in Theorem 3.1, which is feasible as indicated by Theorem 3.2. For a fully set-associative cache $\mathcal{C}$ of size $|\mathcal{E}|$ and block size one, the cost of schedule $\sigma = (\tau,\mu)$ for program graph $\mathcal{G}$ is then given by

$$\begin{aligned}
& c_{\mathcal{G},\mathcal{C}}(\sigma) \\
= & \quad \{ \text{ Lemma 4.1 and } b_s = 1 \text{ } \} \\
& |\{ \boldsymbol{c}^{\mathrm{T}}(A)\boldsymbol{n}(q,\boldsymbol{i}) + o(A) \mid \boldsymbol{i} \in \mathcal{I}(q) \} \cup \{ \boldsymbol{c}^{\mathrm{T}}(A)\boldsymbol{n}(p,\boldsymbol{j}) + o(A) \mid \boldsymbol{j} \in \mathcal{I}(p) \}| \\
= & \quad \{ \mu \text{ is a unique address assignment } \} \\
& |\{ \boldsymbol{n}(q,\boldsymbol{i}) \mid \boldsymbol{i} \in \mathcal{I}(q) \} \cup \{ \boldsymbol{n}(p,\boldsymbol{j}) \mid \boldsymbol{j} \in \mathcal{I}(p) \}| \\
= & \\
& |\{ \boldsymbol{A}\boldsymbol{i} \mid \boldsymbol{i} \in \mathbb{Z}^{\delta(v)} \wedge \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I} \} \cup \{\boldsymbol{b}\}|.
\end{aligned}$$

In a similar way we construct a second instance of CCCP by creating $\mathcal{G}' = (V',R',W',\boldsymbol{I}',E',\boldsymbol{A}',\boldsymbol{b}')$ having

- one operation, $V' = \{v'\}$,
- operation read ports $R'(v) = \{q'\}$,
- operation write ports $W'(v) = \emptyset$,
- iterator bound vector $\boldsymbol{I}'(v) = \boldsymbol{I}$,
- edge set $E' = \emptyset$,
- index matrix $\boldsymbol{A}'(q') = \boldsymbol{A}$, and
- index offset vector $\boldsymbol{b}'(q') = \boldsymbol{0}$.

For operation v' we use the same period vector and start time as we did in the time assignment τ for graph $\mathcal{G}$ and the same address assignment μ as we used for $\mathcal{G}$. The cost of schedule σ' for graph $\mathcal{G}'$ is then given by

$$c_{\mathcal{G}',\mathcal{C}}(\sigma') = |\{ \boldsymbol{A}\boldsymbol{i} \mid \boldsymbol{i} \in \mathbb{Z}^{\delta(v)} \wedge \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I} \}|,$$

which can be derived analogously to $c_{\mathcal{G},\mathcal{C}}(\sigma)$. Now suppose hat CCCP can be solved in polynomial time. This means that both $c_{\mathcal{G},\mathcal{C}}(\sigma)$ and $c_{\mathcal{G}',\mathcal{C}}(\sigma')$ can be computed in polynomial time. However, we can also decide whether $c_{\mathcal{G},\mathcal{C}}(\sigma)$ equals $c_{\mathcal{G}',\mathcal{C}}(\sigma')$ in polynomial time, and hence we can determine in polynomial time whether

$$\exists_{\boldsymbol{i} \in \mathbb{Z}^\delta} \, \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I} \wedge \boldsymbol{A}\boldsymbol{i} = \boldsymbol{b}$$

is feasible. Note that the size of the constructed instances of CCCP is polynomially bounded in the size of $\mathcal{I}_{\text{sle}}$. Therefore, we can solve SLE in polynomial time and as SLE is NP-complete, we can conclude that CCCP can only be solved in polynomial time if P = NP. □

4.1.2 Cache cost computation for lexicographical index orderings

Even if we assume that each port has lexicographical index orderings (see Definition 3.12), computing the number of cache misses remains difficult as is proved below.

Definition 4.4 (CCCP for lexicographical index orderings (CCCP-LL)). Given are a program graph $\mathcal{G}$, with for all ports p the lexicographical index ordering property $\mathrm{lio}(\boldsymbol{I}(p),\boldsymbol{A}(p))$, a feasible schedule σ, a cache $\mathcal{C}$, and an integer K. Does the schedule have cost at most K, i.e., $c_{\mathcal{G},\mathcal{C}}(\sigma) \leq K$? $\square$

In order to establish the complexity result for CCCP-LL we need a formulation of SLE with lexicographical properties.

Definition 4.5 (SLE-LL). Given is a matrix $\boldsymbol{A} = \begin{bmatrix}\boldsymbol{A}' & \boldsymbol{A}''\end{bmatrix}$ with $\boldsymbol{A}' \in \mathbb{Z}^{\alpha\times\delta'}$ and $\boldsymbol{A}'' \in \mathbb{Z}^{\alpha\times\delta''}$, a vector $\boldsymbol{I} = \begin{bmatrix}\boldsymbol{I}' \\ \boldsymbol{I}''\end{bmatrix}$ with $\boldsymbol{I}' \in \mathbb{N}^{\delta'}$ and $\boldsymbol{I}'' \in \mathbb{N}^{\delta''}$, and a vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. Both sub-matrices have the lexicographical index order property, i.e., both $\mathrm{lio}(\boldsymbol{I}',\boldsymbol{A}')$ and $\mathrm{lio}(\boldsymbol{I}'',\boldsymbol{A}'')$ hold. Determine whether a vector $\boldsymbol{i} \in \mathbb{Z}^{\delta'+\delta''}$ exists for which

$$\begin{gathered}\boldsymbol{A}\boldsymbol{i} = \boldsymbol{b} \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}.\end{gathered}$$

$\square$

Theorem 4.3. *SLE-LL is NP-complete in the strong sense.*

Proof. If we are given an integer vector $\boldsymbol{i}$ we can check in polynomial time whether $\boldsymbol{A}\boldsymbol{i} = \boldsymbol{b} \wedge \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}$. Hence SLE-LL is in NP. We prove NP-completeness by a reduction from SLE. Let an instance $\mathcal{I}_{\text{sle}}$ of SLE be given by a matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha\times\delta}$, a vector $\boldsymbol{I} \in \mathbb{N}^{\delta}$, and a vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. Analogously to the NP-completeness proof of PCLL by Verhaegh [1995], we construct an instance $\mathcal{I}_{\text{sle-ll}}$ of SLE-LL by choosing

$$\boldsymbol{A}_{\text{ll}} = \begin{bmatrix}\mathbf{I} & \mathbf{I} \\ \boldsymbol{A} & \mathbf{O}\end{bmatrix} \quad \boldsymbol{b}_{\text{ll}} = \begin{bmatrix}\boldsymbol{I} \\ \boldsymbol{b}\end{bmatrix} \quad \boldsymbol{I}_{\text{ll}} = \begin{bmatrix}\boldsymbol{I} \\ \boldsymbol{I}\end{bmatrix},$$

where $\mathbf{I}$ is the $\delta\times\delta$ identity matrix and $\mathbf{O}$ is the $\alpha\times\delta$ zero matrix. Now we have

$$\begin{aligned}
&\exists_{\boldsymbol{i}\in\mathbb{Z}^{2\delta}}\, \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}_{\text{ll}} \wedge \boldsymbol{A}_{\text{ll}}\boldsymbol{i} = \boldsymbol{b}_{\text{ll}} \\
\equiv\\
&\exists_{\boldsymbol{i}',\boldsymbol{i}''\in\mathbb{Z}^{\delta}}\, \boldsymbol{0} \leq \begin{bmatrix}\boldsymbol{i}' \\ \boldsymbol{i}''\end{bmatrix} \leq \begin{bmatrix}\boldsymbol{I} \\ \boldsymbol{I}\end{bmatrix} \wedge \begin{bmatrix}\mathbf{I} & \mathbf{I} \\ \boldsymbol{A} & \mathbf{O}\end{bmatrix}\begin{bmatrix}\boldsymbol{i}' \\ \boldsymbol{i}''\end{bmatrix} = \begin{bmatrix}\boldsymbol{I} \\ \boldsymbol{b}\end{bmatrix} \\
\equiv\\
&\exists_{\boldsymbol{i}',\boldsymbol{i}''\in\mathbb{Z}^{\delta}}\, \boldsymbol{0} \leq \boldsymbol{i}',\boldsymbol{i}'' \leq \boldsymbol{I} \wedge \boldsymbol{i}' + \boldsymbol{i}'' = \boldsymbol{I} \wedge \boldsymbol{A}\boldsymbol{i}' = \boldsymbol{b} \\
\equiv\\
&\exists_{\boldsymbol{i}\in\mathbb{Z}^{\delta}}\, \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \wedge \boldsymbol{A}\boldsymbol{i} = \boldsymbol{b}.
\end{aligned}$$

So, $\mathcal{I}_{\text{sle-ll}}$ has a solution if and only if $\mathcal{I}_{\text{sle}}$ has one. Note that the size of $\mathcal{I}_{\text{sle-ll}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{sle}}$. Furthermore, the largest number in $\mathcal{I}_{\text{sle-ll}}$ is polynomially bounded in the largest number in $\mathcal{I}_{\text{sle}}$ and the size of $\mathcal{I}_{\text{sle}}$. Hence, SLE-LL is NP-complete in the strong sense. □

Not only the problem of finding bounded integer solutions for linear equations resulting from two arrays with the lexicographical property is NP-complete. Also the problem with a matrix $\boldsymbol{A}$ that can be split into matrices $\boldsymbol{A}'$ and $\boldsymbol{A}''$ with $\text{lio}(\boldsymbol{I}',\boldsymbol{A}')$ and $\text{lio}(\boldsymbol{I}'',-\boldsymbol{A}'')$, is NP-complete in the strong sense, as the next theorem shows. $\text{lio}(\boldsymbol{I},-\boldsymbol{A})$ is also a lexicographical property of matrix A, as it is equivalent to

$$\boldsymbol{i} <_{\text{lex}} \boldsymbol{j} \;\Leftrightarrow\; \boldsymbol{Ai} >_{\text{lex}} \boldsymbol{Aj},$$

and hence worth to be studied.

Definition 4.6 (SLE-LL′). Given is a matrix $\boldsymbol{A} = \begin{bmatrix}\boldsymbol{A}' & \boldsymbol{A}''\end{bmatrix}$ with $\boldsymbol{A}' \in \mathbb{Z}^{\alpha\times\delta'}$ and $\boldsymbol{A}'' \in \mathbb{Z}^{\alpha\times\delta''}$, a vector $\boldsymbol{I} = \begin{bmatrix}\boldsymbol{I}' \\ \boldsymbol{I}''\end{bmatrix}$ with $\boldsymbol{I}' \in \mathbb{N}^{\delta'}$ and $\boldsymbol{I}'' \in \mathbb{N}^{\delta''}$, and vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. Furthermore, the lexicographical index ordering properties $\text{lio}(\boldsymbol{I}',\boldsymbol{A}')$ and $\text{lio}(\boldsymbol{I}'',-\boldsymbol{A}'')$ hold. Determine whether a vector $\boldsymbol{i} \in \mathbb{Z}^{\delta'+\delta''}$ exists for which

$$\boldsymbol{Ai} = \boldsymbol{b}$$
$$\boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I}.$$

□

Theorem 4.4. *SLE-LL′ is NP-complete in the strong sense.*
Proof. The proof is analogous to that of Theorem 4.3 using

$$\boldsymbol{A}_{\text{ll}} = \begin{bmatrix}\mathbf{I} & -\mathbf{I} \\ \boldsymbol{A} & \mathbf{O}\end{bmatrix} \quad \boldsymbol{b}_{\text{ll}} = \begin{bmatrix}\mathbf{0} \\ \boldsymbol{b}\end{bmatrix} \quad \boldsymbol{I}_{\text{ll}} = \begin{bmatrix}\boldsymbol{I} \\ \boldsymbol{I}\end{bmatrix}.$$

□

Now we are ready to study the complexity of CCCP-LL.

Theorem 4.5. *CCCP-LL is NP-hard in the strong sense.*
Proof. For this proof we reduce SLE-LL′ to CCCP-LL. Let an instance $\mathcal{I}_{\text{sle-ll}'}$ of SLE-LL′ be given. We construct an instance $\mathcal{I}_{\text{cccp-ll}}$ of CCCP-LL as follows. First, we introduce a program graph by choosing $\mathcal{G}$:

- two operations $V = \{u,v\}$,
- operation read ports $R(u) = \emptyset$, $R(v) = \{q\}$,
- operation write ports $W(u) = \{p\}$, $W(v) = \emptyset$,
- iterator bound vectors $\boldsymbol{I}(u) = \boldsymbol{I}'$, $\boldsymbol{I}(v) = \boldsymbol{I}''$,
- edge set $E = \{(p,q)\}$,
- index matrices $\boldsymbol{A}(p) = \boldsymbol{A}'$, $\boldsymbol{A}(q) = -\boldsymbol{A}''$, and

- index offset vectors $\boldsymbol{b}(p) = \boldsymbol{0}$, $\boldsymbol{b}(q) = \boldsymbol{b}$.

The single-assignment assumptions are met as the lexicographical index ordering property ensures that write port p accesses each index at most once. Furthermore we assume a fully set-associative cache $\mathcal{C}$ of size $|\mathcal{E}|$ and block size equal to one.

For a port with lexicographical index orderings, the number of distinct index vectors that are accessed during the execution of a schedule, is equal to the number of executions of it, which is given by

$$\prod_{l=0}^{\delta-1} (I_l + 1),$$

and which can computed in polynomial time. So, the number of index vectors accessed by port p is given by

$$K' = |\{\boldsymbol{A}'\boldsymbol{i} \mid \boldsymbol{i} \in \mathcal{I}(p)\}| = |\mathcal{I}(p)| = \prod_{l=0}^{\delta'-1} (I_l(p) + 1), \text{ and}$$

the number of index vectors accessed by port q is given by

$$K'' = |\{-\boldsymbol{A}''\boldsymbol{i} + \boldsymbol{b} \mid \boldsymbol{i} \in \mathcal{I}(q)\}| = |\mathcal{I}(q)| = \prod_{l=0}^{\delta''-1} (I_l(q) + 1).$$

Now, for the cost bound we choose $K = K' + K'' - 1$. For both operation u and v we choose a lexicographical execution, in such a way that all executions of operation u take place before any execution of operation v. In this way the precedence constraints are trivially met. For the address assignment we choose a unique address assignment, resulting in a feasible schedule.

Now we derive

$c_{\mathcal{G},\mathcal{C}}(\sigma) \leq K$

$\equiv$ $\{$ Lemma 4.1, μ is a unique address assignment $\}$

$|\{\boldsymbol{A}'\boldsymbol{i} \mid \boldsymbol{i} \in \mathcal{I}(p)\} \cup \{-\boldsymbol{A}''\boldsymbol{i} + \boldsymbol{b} \mid \boldsymbol{i} \in \mathcal{I}(q)\}| \leq K' + K'' - 1$

$\equiv$ $\{$ definition of K' and K'' $\}$

$\exists_{\boldsymbol{i}' \in \mathbb{Z}^{\delta'}} \exists_{\boldsymbol{i}'' \in \mathbb{Z}^{\delta''}} \; \boldsymbol{0} \leq \boldsymbol{i}' \leq \boldsymbol{I}' \wedge \boldsymbol{0} \leq \boldsymbol{i}'' \leq \boldsymbol{I}'' \wedge \boldsymbol{A}'\boldsymbol{i}' = -\boldsymbol{A}''\boldsymbol{i}'' + \boldsymbol{b}$

$\equiv$

$\exists_{\boldsymbol{i} \in \mathbb{Z}^{\delta}} \; \boldsymbol{0} \leq \boldsymbol{i} \leq \begin{bmatrix} \boldsymbol{I}' \\ \boldsymbol{I}'' \end{bmatrix} \wedge \begin{bmatrix} \boldsymbol{A}' & \boldsymbol{A}'' \end{bmatrix} \boldsymbol{i} = \boldsymbol{b}.$

So $\mathcal{I}_{\text{sle-ll}'}$ has a solution if and only if $\mathcal{I}_{\text{cccp-ll}}$ has one. Note that the size of $\mathcal{I}_{\text{cccp-ll}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{sle-ll}'}$. Furthermore, the largest number in $\mathcal{I}_{\text{cccp-ll}}$ is polynomially bounded in the largest number in $\mathcal{I}_{\text{sle-ll}'}$ and the size of $\mathcal{I}_{\text{sle-ll}'}$. Hence, CCCP-LL is NP-hard in the strong sense. $\square$

Membership of CCCP-LL to NP remains unknown as a compact certificate for each schedule with cost at most K must in some way contain information about which port executions produce misses (or hits). Unfortunately, we do not know a method of representing all misses by a compact certificate.

As both $\text{lio}(\boldsymbol{I},\boldsymbol{A})$ and $\text{lio}(\boldsymbol{I},-\boldsymbol{A})$ are equally interesting properties, we also introduce the problem CCCP-LL$'$ in which the number of cache misses is asked for a program graph with two ports p and q, with lexicographical index orderings $\text{lio}(\boldsymbol{I}(p),\boldsymbol{A}(p))$ and $\text{lio}(\boldsymbol{I}(q),-\boldsymbol{A}(q))$.

Theorem 4.6. *CCCP-LL$'$ is NP-complete in the strong sense.*

Proof. This result is proved analogously to Theorem 4.5 by a reduction from SLE-LL. □

4.2 Feasibility of a schedule

Verhaegh [1995] discusses the complexity of checking precedence constraints already extensively. Therefore, we restrict the discussion in this section to address constraints. We restrict ourselves to two variants of the address conflict problem, one where three ports are involved, and one with only two ports.

4.2.1 Address conflicts for three ports

An address conflict for write ports p, r and read port q occurs if data is shared between p and q, but is overwritten in memory by an execution of r that takes place between the production by p and consumption by q of the shared array element.

Definition 4.7 (address conflict for three ports (ACP3)). Given are ports p, r, and q with iterator bound vectors $\boldsymbol{I}(p) \in \mathbb{N}^{\delta(p)}$, $\boldsymbol{I}(r) \in \mathbb{N}^{\delta(r)}$, and $\boldsymbol{I}(q) \in \mathbb{N}^{\delta(q)}$, period vectors $\boldsymbol{p}(p) \in \mathbb{Z}^{\delta(p)}$, $\boldsymbol{p}(r) \in \mathbb{Z}^{\delta(r)}$, $\boldsymbol{p}(q) \in \mathbb{Z}^{\delta(q)}$, start times $s(p), s(r), s(q) \in \mathbb{Z}$, index matrices $\boldsymbol{A}(p) \in \mathbb{Z}^{\alpha(p)\times\delta(p)}$, $\boldsymbol{A}(r) \in \mathbb{Z}^{\alpha(r)\times\delta(r)}$, $\boldsymbol{A}(q) \in \mathbb{Z}^{\alpha(q)\times\delta(q)}$, and index offset vectors $\boldsymbol{b}(p) \in \mathbb{Z}^{\alpha(p)}$, $\boldsymbol{b}(r) \in \mathbb{Z}^{\alpha(r)}$, $\boldsymbol{b}(q) \in \mathbb{Z}^{\alpha(q)}$. The time assignment is feasible and the single-assignment property is met. Furthermore, two arrays A and B with $p, q \in A$ and $r \in B$, address coefficient vectors $\boldsymbol{c}(A) \in \mathbb{Z}^{\alpha(A)}$, $\boldsymbol{c}(B) \in \mathbb{Z}^{\alpha(B)}$, and address offsets $o(A), o(B) \in \mathbb{Z}$ are given. Determine whether there are vectors $\boldsymbol{i} \in \mathbb{Z}^{\delta(p)}$, $\boldsymbol{k} \in \mathbb{Z}^{\delta(r)}$, and $\boldsymbol{j} \in \mathbb{Z}^{\delta(q)}$ that satisfy

$$\begin{gathered} a(p,\boldsymbol{i}) = a(r,\boldsymbol{k}) \\ t(p,\boldsymbol{i}) \leq t(r,\boldsymbol{k}) \leq t(q,\boldsymbol{j}) \\ \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(p) \\ \boldsymbol{0} \leq \boldsymbol{k} \leq \boldsymbol{I}(r) \\ \boldsymbol{0} \leq \boldsymbol{j} \leq \boldsymbol{I}(q). \end{gathered} \tag{4.5}$$

□

We give two complexity results for ACP3. First, we prove that ACP3 is NP-complete. Next, we show that no pseudo-polynomial algorithm exists for solving ACP3, unless P = NP. The proofs show distinct difficulties in checking address constraints. The first proof shows the difficulty of checking the constraint $t(p,\boldsymbol{i}) \leq t(r,\boldsymbol{k}) \leq t(q,\boldsymbol{j})$, whereas the second proof focuses on the constraint $\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})$.

For proving NP-completeness of ACP3, we use the problem PUC [Verhaegh, 1995], which is defined as follows.

Definition 4.8 (PUC). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^\delta$, and an integer s. Determine whether there is a vector $\boldsymbol{i} \in \mathbb{Z}^\delta$ that satisfies

$$\begin{gathered} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s \\ \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{I}. \end{gathered}$$

□

PUC is NP-complete, and it can be solved in pseudo-polynomial time [Verhaegh, 1995].

Theorem 4.7. *ACP3 is NP-complete.*

Proof. When vectors $\boldsymbol{i}$, $\boldsymbol{k}$, and $\boldsymbol{j}$ are given, one can check in polynomial time whether they satisfy (4.5), so ACP3 is in NP. Next, we reduce PUC to ACP3. Let an instance $\mathcal{I}_{\text{puc}}$ of PUC be given, then we create an instance $\mathcal{I}_{\text{acp3}}$ of ACP3 by choosing

- iterator bound vectors $\boldsymbol{I}(p) = \boldsymbol{I}(q) = [\,]$, $\boldsymbol{I}(r) = \boldsymbol{I}$,
- index matrices $\boldsymbol{A}(p) = \boldsymbol{A}(q) = [\,]$, $\boldsymbol{A}(r) = \mathbf{I}$,
- index offset vectors $\boldsymbol{b}(p) = \boldsymbol{b}(q) = [\,]$, $\boldsymbol{b}(r) = \mathbf{0}$,
- period vectors $\boldsymbol{p}(p) = \boldsymbol{p}(q) = [\,]$, $\boldsymbol{p}(r) = 2\boldsymbol{p}$,
- start times $s(p) = 2s$, $s(r) = 0$, $s(q) = 2s+1$,
- address coefficient vectors $\boldsymbol{c}(A) = [\,]$, $\boldsymbol{c}(B) = \mathbf{0}$, and
- address offsets $o(A) = o(B) = 0$.

All single-assignment properties are satisfied by the choice of index matrices and index offset vectors. Both ports p and q execute only once, accessing the same array element. As the production by p takes place at time $2s$ and the consumption by q at time $2s+1$, the time assignment $\tau = (\boldsymbol{p}, s)$ is feasible. All data that are produced and consumed are stored at address 0, so an address conflict can only occur if a production of port r takes place at time $2s$ or $2s+1$. Hence, an address constraint is violated if and only if

$$\begin{aligned} & \exists_{\boldsymbol{k} \in \mathcal{I}(r)}\, a(p,[\,]) = a(r,\boldsymbol{k}) \;\wedge\; t(p,[\,]) \leq t(r,\boldsymbol{k}) \leq t(q,[\,]) \;\wedge\; \boldsymbol{n}(p,[\,]) = \boldsymbol{n}(q,[\,]) \\ \equiv \quad & \{\; a(p,[\,]) = 0 = a(r,\boldsymbol{k}) \;\} \end{aligned}$$

$$\exists_{\boldsymbol{k}\in\mathbb{Z}^\delta}\ \boldsymbol{0} \le \boldsymbol{k} \le \boldsymbol{I}(r)\ \wedge\ 2s \le 2\boldsymbol{p}^{\mathrm{T}}\boldsymbol{k} \le 2s+1$$
$$\equiv$$
$$\exists_{\boldsymbol{k}\in\mathbb{Z}^\delta}\ \boldsymbol{0} \le \boldsymbol{k} \le \boldsymbol{I}\ \wedge\ \boldsymbol{p}^{\mathrm{T}}\boldsymbol{k} = s.$$

So $\mathcal{I}_{\text{acp3}}$ has a solution if and only if $\mathcal{I}_{\text{puc}}$ has one. Furthermore, note that the size of $\mathcal{I}_{\text{acp3}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{puc}}$. Hence, ACP3 is NP-complete.
$\square$

A stronger result is possible by reducing ZOIP of Definition 4.2 to ACP3.

Theorem 4.8. *ACP3 is NP-complete in the strong sense.*

Proof. Membership of ACP3 to NP has already been established in Theorem 4.7. Let an instance $\mathcal{I}_{\text{zoip}}$ of ZOIP be given by a matrix $\boldsymbol{M} \in \mathbb{Z}^{m\times n}$, a vector $\boldsymbol{d} \in \mathbb{Z}^m$, a vector $\boldsymbol{c} \in \mathbb{Z}^n$, and an integer B. We create an instance $\mathcal{I}_{\text{acp3}}$ of ACP3 by choosing

- iterator bound vectors $\boldsymbol{I}(p) = \boldsymbol{I}(r) = [\,]$, $\boldsymbol{I}(q) = \boldsymbol{1}$,
- index matrices $\boldsymbol{A}(p) = \boldsymbol{A}(r) = [\,]$, $\boldsymbol{A}(q) = \boldsymbol{M}$,
- offset vectors $\boldsymbol{b}(p) = \boldsymbol{d}$, $\boldsymbol{b}(r) = \boldsymbol{d}+\boldsymbol{1}$, and $\boldsymbol{b}(q) = \boldsymbol{0}$,
- period vectors $\boldsymbol{p}(p) = [\,]$, $\boldsymbol{p}(r) = [\,]$, and $\boldsymbol{p}(q) = \boldsymbol{c}$,
- start times $s(p) = -1 + B^- + \sum_{i=0}^{n-1} c_i^-$, $s(r) = B$, and $s(q) = 0$,
- address coefficient vectors $\boldsymbol{c}(A) = \boldsymbol{c}(B) = \boldsymbol{0}$, and
- address offsets $o(A) = o(B) = 0$,

where $x^- = \min\{x,0\}$. The single assignment constraints for both arrays are satisfied. Again, first we show that $\tau = (\boldsymbol{p},s)$ is a feasible time assignment, which is given by the fact that for all $\boldsymbol{j} \in \mathcal{I}(q)$

$$t(p,\boldsymbol{i}) = -1 + B^- + \sum_{i=0}^{n-1} c_i^- < \sum_{i=0}^{n-1} c_i j_i = t(q,\boldsymbol{j}).$$

Using the relation $\boldsymbol{x} = \boldsymbol{j}$ we observe that $\mathcal{I}_{\text{acp3}}$ has a solution if and only if $\mathcal{I}_{\text{zoip}}$ has one, as we have

$$\exists_{\boldsymbol{i}\in\mathcal{I}(p)}\ \exists_{\boldsymbol{j}\in\mathcal{I}(q)}\ \exists_{\boldsymbol{k}\in\mathcal{I}(r)}\ t(p,\boldsymbol{i}) \le t(r,\boldsymbol{k}) \le t(q,\boldsymbol{j})\ \wedge\ a(p,\boldsymbol{i}) = a(r,\boldsymbol{k})\ \wedge\ \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})$$
$$\equiv \qquad \{\ p \text{ and } r \text{ have exactly one execution }\ \}$$
$$\exists_{\boldsymbol{j}\in\{0,1\}^n}\ -1 + B^- + \sum_{i=0}^{n-1} c_i^- \le B \le \boldsymbol{c}^{\mathrm{T}}\boldsymbol{j}\ \wedge\ a(p,[\,]) = a(r,[\,])\ \wedge\ \boldsymbol{d} = \boldsymbol{M}\boldsymbol{j}$$
$$\equiv$$
$$\exists_{\boldsymbol{j}\in\{0,1\}^n}\ \boldsymbol{c}^{\mathrm{T}}\boldsymbol{j} \ge B\ \wedge\ \boldsymbol{M}\boldsymbol{j} = \boldsymbol{d}.$$

Note that the size of $\mathcal{I}_{\text{acp3}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{zoip}}$. Furthermore, the largest number in $\mathcal{I}_{\text{acp3}}$ is polynomially bounded in the largest number in $\mathcal{I}_{\text{zoip}}$ and the size of $\mathcal{I}_{\text{zoip}}$. Hence ACP3 is NP-complete in the strong sense. □

As we did not assume A and B to be different in the proof, ACP3 remains NP-complete in the strong sense if p, q and r belong the same array cluster. Furthermore, ACP3 remains NP-complete in the strong sense for an address space limited to only one address.

4.2.2 Address conflicts for two ports

An address conflict for write port p and read port q occurs if data is shared between p and q, but is overwritten by another execution of p, taking place between the production by p and consumption of q of the shared array element.

Definition 4.9 (address conflict for two ports (ACP2)). Given are ports p and q with iterator bound vectors $\boldsymbol{I}(p) \in \mathbb{N}^{\delta(p)}$, $\boldsymbol{I}(q) \in \mathbb{N}^{\delta(q)}$, period vectors $\boldsymbol{p}(p) \in \mathbb{Z}^{\delta(p)}$, $\boldsymbol{p}(q) \in \mathbb{Z}^{\delta(q)}$, start times $s(p), s(q) \in \mathbb{Z}$, index matrices $\boldsymbol{A}(p) \in \mathbb{Z}^{\alpha(p)\times\delta(p)}$, $\boldsymbol{A}(q) \in \mathbb{Z}^{\alpha(q)\times\delta(q)}$, and index offset vectors $\boldsymbol{b}(p) \in \mathbb{Z}^{\alpha(p)}$, $\boldsymbol{b}(q) \in \mathbb{Z}^{\alpha(q)}$. The time assignment is feasible and the single-assignment property is met. Both ports are assumed to belong to the same array, say A, with address coefficient vector $\boldsymbol{c}(A) \in \mathbb{Z}^{\alpha(A)}$, and address offset $o(A) \in \mathbb{Z}$. Determine whether there are vectors $\boldsymbol{i} \in \mathbb{Z}^{\delta(p)}$, $\boldsymbol{k} \in \mathbb{Z}^{\delta(p)}$, and $\boldsymbol{j} \in \mathbb{Z}^{\delta(q)}$ that satisfy

$$\begin{gathered} a(p,\boldsymbol{i}) = a(p,\boldsymbol{k}) \\ t(p,\boldsymbol{i}) \le t(p,\boldsymbol{k}) \le t(q,\boldsymbol{j}) \\ \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \\ \boldsymbol{i} \ne \boldsymbol{k} \\ \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I}(p) \\ \boldsymbol{0} \le \boldsymbol{k} \le \boldsymbol{I}(p) \\ \boldsymbol{0} \le \boldsymbol{j} \le \boldsymbol{I}(q). \end{gathered} \tag{4.6}$$

□

Theorem 4.9. *ACP2 is NP-complete in the strong sense.*

Proof. When given vectors $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ expression (4.6) can be checked in polynomial time. Hence ACP2 is in NP. Let an instance $\mathcal{I}_{\text{zoip}}$ of ZOIP be given by a matrix $\boldsymbol{M} \in \mathbb{Z}^{m\times n}$, a vector $\boldsymbol{d} \in \mathbb{Z}^m$, a vector $\boldsymbol{l} \in \mathbb{Z}^n$, and an integer B. We create an instance $\mathcal{I}_{\text{acp2}}$ of ACP2 by choosing

- iterator bound vectors $\boldsymbol{I}(p) = \begin{bmatrix}1\end{bmatrix}$, $\boldsymbol{I}(q) = \boldsymbol{1}$,
- index matrices $\boldsymbol{A}(p) = \begin{bmatrix}\boldsymbol{0}\\1\end{bmatrix}$, $\boldsymbol{A}(q) = \begin{bmatrix}\boldsymbol{M}\\\boldsymbol{0}^{\mathrm{T}}\end{bmatrix}$,
- offset vectors $\boldsymbol{b}(p) = \begin{bmatrix}\boldsymbol{d}\\0\end{bmatrix}$, $\boldsymbol{b}(q) = \boldsymbol{0}$,

- period vectors $\boldsymbol{p}(p) = \left[1 + B^+ - \sum_{i=0}^{n-1} l_i^-\right]$, $\boldsymbol{p}(q) = \boldsymbol{l}$,
- start times $s(p) = -1 - B^+ + \sum_{i=0}^{n-1} l_i^-$, $s(q) = -B$,
- address coefficient vector $\boldsymbol{c}(A) = \boldsymbol{0}$, and
- address offset $o(A) = 0$.

Again, first we show that $\tau = (\boldsymbol{p}, s)$ is a feasible time assignment by deriving that $t(p,\boldsymbol{i}) < t(q,\boldsymbol{j})$ holds for all port executions $\boldsymbol{i} \in \mathcal{I}(p)$ of port p and $\boldsymbol{j} \in \mathcal{I}(q)$ of port q with $\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})$.

$$
\begin{aligned}
& t(p,\boldsymbol{i}) \\
= \quad & \{\ \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \ \Rightarrow\ \boldsymbol{i} = [0]\ \} \\
& -1 + B^- + \sum_{i=0}^{n-1} l_i^- \\
< \quad & \{\ 0 \le j_i \le 1\ \} \\
& -B + \sum_{i=0}^{n-1} l_i j_i \\
= \quad & \\
& t(q,\boldsymbol{j})
\end{aligned}
$$

Using the relation $\boldsymbol{x} = \boldsymbol{j}$ we observe that $\mathcal{I}_{\text{acp2}}$ has a solution if and only if $\mathcal{I}_{\text{zoip}}$ has one, as we have

$$
\begin{aligned}
& \exists_{\boldsymbol{i}\in\mathcal{I}(p)}\ \exists_{\boldsymbol{j}\in\mathcal{I}(q)}\ \exists_{\boldsymbol{k}\in\mathcal{I}(p)}\ t(p,\boldsymbol{i}) \le t(p,\boldsymbol{k}) \le t(q,\boldsymbol{j}) \ \wedge\ a(p,\boldsymbol{i}) = a(p,\boldsymbol{k}) \ \wedge \\
& \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \ \wedge\ \boldsymbol{i} \neq \boldsymbol{k} \\
\equiv \quad & \{\ p \text{ has two executions with } t(p,[0]) < t(p,[1])\ \} \\
& \exists_{\boldsymbol{j}\in\{0,1\}^n}\ 0 \le \boldsymbol{l}^{\mathrm{T}}\boldsymbol{j} - B \ \wedge\ a(p,[0]) = a(p,[1]) \ \wedge\ \begin{bmatrix}\boldsymbol{0}\\1\end{bmatrix}[0] + \begin{bmatrix}\boldsymbol{d}\\0\end{bmatrix} = \begin{bmatrix}\boldsymbol{M}\\\boldsymbol{0}^{\mathrm{T}}\end{bmatrix}\boldsymbol{j} + \boldsymbol{0} \\
\equiv \quad & \{\ \text{address coefficient vector } \boldsymbol{c}(A) = \boldsymbol{0}\ \} \\
& \exists_{\boldsymbol{j}\in\{0,1\}^n}\ \boldsymbol{l}^{\mathrm{T}}\boldsymbol{j} \ge B \ \wedge\ \boldsymbol{M}\boldsymbol{j} = \boldsymbol{d}.
\end{aligned}
$$

Note that the size of $\mathcal{I}_{\text{acp2}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{zoip}}$. Furthermore, the largest number in $\mathcal{I}_{\text{acp2}}$ is polynomially bounded in the largest number in $\mathcal{I}_{\text{zoip}}$ and the size of $\mathcal{I}_{\text{zoip}}$. Hence ACP2 is NP-complete in the strong sense. □

4.3 Multidimensional periodic cache scheduling

Besides the complexity of the multidimensional periodic cache scheduling problem, where both a time assignment and an address assignment are asked, two related problems are studied. The first one, which is called *optimal address assignment*, is the problem of finding an optimal and feasible address assignment if a feasible time assignment is already given. The second one is the problem of finding an optimal and feasible time assignment for a fixed address assignment, which is called *optimal time assignment*.

Definition 4.10 (optimal address assignment (OAA)). Given are a program graph $\mathcal{G}$, a cache $\mathcal{C}$, a feasible time assignment τ, and an integer K. Determine whether there exists a feasible address assignment μ with cost $c_{\mathcal{G},\mathcal{C}}(\sigma) \leq K$. □

Instead of deriving a complexity result for OAA, we prove the complementary problem (co-OAA) to be NP-hard. The complementary problem co-OAA decides whether *no* schedule exists with cost at most K. It is unlikely that an algorithm exists that decides OAA in polynomial time, as that would imply the existence of an algorithm that decides co-OAA in polynomial time, and hence would imply P = NP.

In the complexity proof we fix $K = 1$. In the following lemma we give necessary and sufficient conditions for schedules with cost $c_{\mathcal{G},\mathcal{C}}(\sigma) = 1$.

Lemma 4.2. *For a graph $\mathcal{G}$ with at least one port, and a fully set-associative cache of size $|\mathcal{E}|$, a schedule σ has cost $c_{\mathcal{G},\mathcal{C}}(\sigma) = 1$ if and only if*

(1) in the time slot in which the first execution of any port takes place, exactly one port executes, and

(2) $\forall_{e,f\in\mathcal{E}}\ a_\text{b}(e) = a_\text{b}(f)$.

Proof.
($\Rightarrow$) Let σ be a schedule with cost $c_{\mathcal{G},\mathcal{C}}(\sigma) = 1$. The necessity of (1) and (2) is shown by contradiction. Suppose that two executions $e \neq f \in \mathcal{E}$ exist such that both execute in the first time slot in which any port executes, then $M(e) = \emptyset$ and $M(f) = \emptyset$, which results in two compulsory misses, resulting in cost at least two. Next, if there are two executions $e, f \in \mathcal{E}$ that access different block addresses, then also at least two compulsory misses are generated. Hence both (1) and (2) are necessary conditions.
($\Leftarrow$) As the number of different block addresses that are accessed during the execution of a schedule is bounded by the number of port executions, no conflict misses can occur in a fully set-associative cache. Hence we have to prove that any schedule for which (1) and (2) hold generates exactly one compulsory miss. Suppose that $e \in \mathcal{E}$ takes place in the first time slot in which any port execution takes place. Because of (1) every execution $f \in \mathcal{E}$ takes place after e and because of (2) $a_\text{b}(f) = a_\text{b}(e)$; therefore, $e \in M(f)$ for each $f \in \mathcal{E} \setminus \{e\}$, and $M(e) = \emptyset$. Hence only port execution e produces a compulsory miss. □

Theorem 4.10. *co-OAA is NP-hard.*
Proof. Let an instance $\mathcal{I}_\text{puc}$ of PUC be given by an iterator bound vector $\boldsymbol{I} \in$

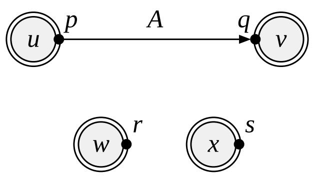


Figure 4.1. Constructed instance of co-OAA.

$\mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^\delta$, and an integer s. An instance $\mathcal{I}_{\text{co-oaa}}$ of co-OAA is constructed as follows. For the program graph $\mathcal{G} = (V, R, W, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ we choose

- four operations $V = \{u, v, w, x\}$,
- operation read ports $R(u) = \varnothing$, $R(v) = \{q\}$, $R(w) = \varnothing$, $R(x) = \varnothing$,
- operation write ports $W(u) = \{p\}$, $W(v) = \varnothing$, $W(w) = \{r\}$, $W(x) = \{s\}$,
- iterator bound vectors $\boldsymbol{I}(u) = \boldsymbol{I}(v) = \boldsymbol{I}(x) = [\,]$, $\boldsymbol{I}(w) = \boldsymbol{I}$,
- edge set $E = \{(p, q)\}$,
- index matrices $\boldsymbol{A}(p) = \boldsymbol{A}(q) = \boldsymbol{A}(s) = [\,]$, $\boldsymbol{A}(r) = \mathbf{I}$, and
- index offset vectors $\boldsymbol{b}(p) = \boldsymbol{b}(q) = \boldsymbol{b}(r) = \boldsymbol{b}(s) = \mathbf{0}$.

For the time assignment $\tau = (\boldsymbol{p}, s)$ we choose

- period vectors $\boldsymbol{p}(u) = \boldsymbol{p}(v) = \boldsymbol{p}(x) = [\,]$, $\boldsymbol{p}(w) = 2\boldsymbol{p}$, and
- start times $s(u) = 2s$, $s(v) = 2s + 1$, $s(w) = 0$, $s(x) = -1 + 2s^- + \sum_{i=0}^{\delta-1} p_i^- I_i$.

For the cache we choose a fully set-associative cache of size $|\mathcal{E}|$ and block size one. The cost bound K is chosen one. First of all we observe that indeed a feasible time assignment has been created as the only execution of port p takes place before the only execution of port q. Furthermore, note that the size of $\mathcal{I}_{\text{co-oaa}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{puc}}$. Now we prove that finding a solution to the PUC-instance is equivalent to deciding whether no feasible schedule exists with cost at most one.

($\Rightarrow$) Suppose that a vector $\boldsymbol{x} \in \mathbb{Z}^\delta$ exists such that $\mathbf{0} \leq \boldsymbol{x} \leq \boldsymbol{I} \wedge s = \boldsymbol{p}^{\mathrm{T}}\boldsymbol{x}$. We have to prove that all feasible schedules have cost at least two. From address constraint $(p \rightarrow q, r)$ it follows that $a(p, [\,]) \neq a(r, \boldsymbol{x})$ as otherwise execution $\boldsymbol{x}$ of r overwrites data produced by the execution of p, which can be shown as follows.

$(p \rightarrow q, r)$

$\equiv$

$\forall_{\boldsymbol{i} \in \mathcal{I}(p)} \forall_{\boldsymbol{j} \in \mathcal{I}(q)} \forall_{\boldsymbol{k} \in \mathcal{I}(r)}\, a(p, \boldsymbol{i}) = a(r, \boldsymbol{k}) \wedge \boldsymbol{n}(p, \boldsymbol{i}) = \boldsymbol{n}(q, \boldsymbol{j}) \wedge$
$t(p, \boldsymbol{i}) \leq t(r, \boldsymbol{k}) \leq t(q, \boldsymbol{j}) \Rightarrow p = q \wedge \boldsymbol{i} = \boldsymbol{j}$

$\equiv$ $\{$ $p \neq q$, p and q have exactly one execution $\}$

$\forall_{\boldsymbol{k} \in \mathcal{I}(r)}\ a(p,[\,]) \neq a(r,\boldsymbol{k}) \ \vee\ \boldsymbol{A}(p)[\,]+\boldsymbol{b}(p) \neq \boldsymbol{A}(q)[\,]+\boldsymbol{b}(q) \ \vee$
$\neg(2s \leq 2\boldsymbol{p}^{\mathrm{T}}\boldsymbol{k} \leq 2s+1)$

$\Rightarrow \quad \{\ \boldsymbol{x} \in \mathcal{I}(r)\ \}$

$a(p,[\,]) \neq a(r,\boldsymbol{x}) \ \vee\ 2s \neq 2\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}$

$\equiv \quad \{\ s = \boldsymbol{p}^{\mathrm{T}}\boldsymbol{x}\ \}$

$a(p,[\,]) \neq a(r,\boldsymbol{x})$

As at least two addresses are needed, and as we have a cache with block size one, at least two compulsory misses will occur and hence $c_{\mathcal{G},\mathcal{C}}(\sigma) \geq 2$.
($\Leftarrow$) Suppose that for all integer vectors $\boldsymbol{i} \in \mathbb{Z}^{\delta}$ we have $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \Rightarrow \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \neq s$. Then the existence of a feasible schedule with cost one remains to be shown. The start time of operation x makes sure that in the first time slot in which any port execution takes place, exactly one port executes. From Lemma 4.2 and block size equal to one, it follows that it is sufficient to show that any address assignment with $a(e) = a(f)$ for all $e, f \in \mathcal{E}$ is feasible. Hence we have to show that address constraints $(p \rightarrow q, p)$, $(p \rightarrow q, s)$, and $(p \rightarrow q, r)$ are met. To this end, we can derive

$(p \rightarrow q, r)$

$\equiv \quad \{\ p \neq q,\ p$ and q has exactly one execution $\}$

$\forall_{\boldsymbol{k} \in \mathcal{I}(r)}\ a(p,[\,]) \neq a(r,\boldsymbol{k}) \ \vee\ \boldsymbol{A}(p)[\,]+\boldsymbol{b}(p) \neq \boldsymbol{A}(q)[\,]+\boldsymbol{b}(q) \ \vee$
$\neg(2s \leq 2\boldsymbol{p}^{\mathrm{T}}\boldsymbol{k} \leq 2s+1)$

$\Leftarrow$

$\forall_{\boldsymbol{k} \in \mathcal{I}(r)}\ s \neq \boldsymbol{p}^{\mathrm{T}}\boldsymbol{k}.$

The two remaining constraints are proved valid in a similar way. So, the address assignment with $\boldsymbol{c}(A) = \boldsymbol{0}$ and $o(A) = 0$ for all arrays $A \in \mathcal{A}$ is feasible, and as all precedence constraints are met, a feasible schedule σ has been constructed with $c_{\mathcal{G},\mathcal{C}}(\sigma) = 1$. □

If we fix the address assignment, the question remains to find an optimal and feasible time assignment in such a way that the address assignment is feasible as well. This problem is NP-hard in the strong sense which is proved by means of a reduction from SLE-LL$'$.

Definition 4.11 (optimal time assignment (OTA)). Given are a program graph $\mathcal{G}$, a cache $\mathcal{C}$, an address assignment μ, and an integer K. Determine whether there exists a feasible time assignment τ with μ feasible with respect to τ and $\mathcal{G}$ and with cost $c_{\mathcal{G},\mathcal{C}}(\sigma) \leq K$. □

Theorem 4.11. *OTA is NP-hard in the strong sense.*

Proof. Let an instance $\mathcal{I}_{\text{sle-ll}'}$ of SLE-LL$'$ be given by matrices $\boldsymbol{A}' \in \mathbb{Z}^{\alpha\times\delta'}$, $\boldsymbol{A}'' \in \mathbb{Z}^{\alpha\times\delta''}$, vectors $\boldsymbol{I}' \in \mathbb{N}^{\delta'}$, $\boldsymbol{I}'' \in \mathbb{N}^{\delta''}$, with $\text{lio}(\boldsymbol{A}',\boldsymbol{I}')$ and $\text{lio}(-\boldsymbol{A}'',\boldsymbol{I}'')$, and a vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. An instance $\mathcal{I}_{\text{ota}}$ of OTA is now constructed by choosing

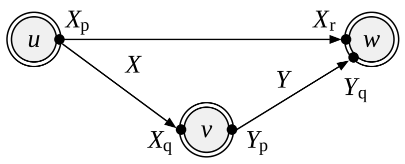


Figure 4.2. Constructed instance of co-MPCSD.

- two operations $V = \{u, v\}$,
- operation read ports $R(u) = \varnothing$, $R(v) = \{q\}$,
- operation write ports $W(u) = \{p\}$, $W(v) = \varnothing$,
- iterator vectors $\boldsymbol{I}(u) = \boldsymbol{I}'$, $\boldsymbol{I}(v) = \boldsymbol{I}''$,
- edge set $E = \{(p, q)\}$,
- index matrices $\boldsymbol{A}(p) = \boldsymbol{A}'$, $\boldsymbol{A}(q) = -\boldsymbol{A}''$, and
- index vectors $\boldsymbol{b}(p) = \boldsymbol{0}$, $\boldsymbol{b}(q) = \boldsymbol{b}$.

Furthermore, we choose

- a unique address assignment μ as given in Theorem 3.1,
- a fully set-associative cache of size $|\mathcal{E}|$ and block size one, and
- a cost bound $K = |\mathcal{E}(p)| + |\mathcal{E}(q)| - 1$.

Now we have

$$\begin{aligned}
& \exists_\tau\ \tau \text{ feasible} \wedge c_{\mathcal{G},\mathcal{C}}(\sigma) \le K \\
\Leftarrow \quad & \{ \text{ Lemma 4.1 } \} \\
& |\{ \boldsymbol{A}'\boldsymbol{i} \mid \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I}' \} \cup \{ -\boldsymbol{A}''\boldsymbol{i} + \boldsymbol{b} \mid \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I}'' \}| \le |\mathcal{E}(p)| + |\mathcal{E}(q)| - 1 \\
\equiv \quad & \{ \text{ see Theorem 4.5 } \} \\
& \exists_{\boldsymbol{i} \in \mathbb{Z}^{\delta' + \delta''}}\ \boldsymbol{0} \le \boldsymbol{i} \le \begin{bmatrix} \boldsymbol{I}' \\ \boldsymbol{I}'' \end{bmatrix} \wedge \begin{bmatrix} \boldsymbol{A}' & \boldsymbol{A}'' \end{bmatrix} \boldsymbol{i} = \boldsymbol{b}.
\end{aligned}$$

The other implication ($\Rightarrow$), cannot be concluded from Lemma 4.1 directly. A problem arises when two port executions take place at the same time t. But as was noted in Chapter 3, the cost of a schedule cannot increase by rescheduling these port executions. Hence, another feasible time assignment exists with cost at most K, that satisfies the requirement of Lemma 4.1.

We have proved that $\mathcal{I}_{\text{ota}}$ has a solution if and only if $\mathcal{I}_{\text{sle-ll}'}$ has one. Note that the size of $\mathcal{I}_{\text{ota}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{sle-ll}'}$. Furthermore, the largest number in $\mathcal{I}_{\text{ota}}$ is polynomially bounded in the largest number in $\mathcal{I}_{\text{sle-ll}'}$ and the size of $\mathcal{I}_{\text{sle-ll}'}$. Hence OTA is NP-hard in the strong sense. □

Finally, we discuss the complexity of the multidimensional periodic cache scheduling problem of Definition 3.10, whose decision variant is denoted by MPCSD, the complementary problem is denoted by co-MPCSD.

Theorem 4.12. *co-MPCSD is NP-hard in the strong sense.*

Proof. We prove NP-hardness of co-MPCSD by a reduction from SLE. Let an instance $\mathcal{I}_{\mathrm{sle}}$ of SLE be given by a matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha\times\delta}$, a vector $\boldsymbol{I} \in \mathbb{N}^{\delta}$, and a vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. We construct an instance of co-MPCSD as follows. The program graph with its read and write ports is given in Figure 4.2. We choose

- three operations $V = \{u, v, w\}$,
- operation read ports $R(u) = \emptyset$, $R(v) = \{X_{\mathrm{q}}\}$, $R(w) = \{X_{\mathrm{r}}, Y_{\mathrm{q}}\}$,
- operation write ports $W(u) = \{X_{\mathrm{p}}\}$, $W(v) = \{Y_{\mathrm{p}}\}$, $W(w) = \emptyset$,
- iterator bound vectors $\boldsymbol{I}(u) = [\,]$, $\boldsymbol{I}(v) = \boldsymbol{I}(w) = \boldsymbol{I}$,
- index matrices $\boldsymbol{A}(X_{\mathrm{p}}) = [\,]$, $\boldsymbol{A}(X_{\mathrm{q}}) = \boldsymbol{A}(X_{\mathrm{r}}) = \boldsymbol{A}$, $\boldsymbol{A}(Y_{\mathrm{p}}) = \boldsymbol{A}(Y_{\mathrm{q}}) = \mathbf{I}$,
- index offset vectors $\boldsymbol{b}(X_{\mathrm{p}}) = \boldsymbol{b}$, $\boldsymbol{b}(X_{\mathrm{q}}) = \boldsymbol{b}(X_{\mathrm{r}}) = \mathbf{0}$, $\boldsymbol{b}(Y_{\mathrm{p}}) = \boldsymbol{b}(Y_{\mathrm{q}}) = \mathbf{0}$,
- a fully set-associative cache with size $|\mathcal{E}|$ and block size one, and
- cost bound $K = 1$.

Now, we prove

$$\exists_{\boldsymbol{i}\in\mathbb{Z}^{\delta}}\ \mathbf{0} \le \boldsymbol{i} \le \boldsymbol{I} \ \wedge\ \boldsymbol{A}\boldsymbol{i} = \boldsymbol{b} \ \Leftrightarrow\ \neg\exists_{\sigma}\ \sigma \text{ feasible} \ \wedge\ c_{\mathcal{G},\mathcal{C}}(\sigma) \le 1.$$

The proof is done in two steps.

($\Rightarrow$) First, let $\boldsymbol{l} \in \mathbb{Z}^{\delta}$ such that $\mathbf{0} \le \boldsymbol{l} \le \boldsymbol{I} \ \wedge\ \boldsymbol{A}\boldsymbol{l} = \boldsymbol{b}$. Now we show that any feasible schedule has cost at least two. For a feasible schedule σ we know that both the precedence constraints and the address constraints are satisfied. From the precedence constraint for array X we deduce that $s(u) < t(v,\boldsymbol{i})$ for all executions $\boldsymbol{i} \in \mathcal{I}(v)$ with $\boldsymbol{A}\boldsymbol{i} = \boldsymbol{b}$, and in particular $s(u) < t(v,\boldsymbol{l})$. From the precedence constraint for array Y it follows that $t(v,\boldsymbol{i}) < t(w,\boldsymbol{i})$ for all $\boldsymbol{i} \in \mathcal{I}(v)$, and hence $s(u) < t(v,\boldsymbol{l}) < t(w,\boldsymbol{l})$.

From the address constraint $(X_{\mathrm{p}} \to X_{\mathrm{r}}, Y_{\mathrm{p}})$ we deduce

$$\begin{aligned}
&\forall_{\boldsymbol{i}\in\mathcal{I}(X_{\mathrm{p}})}\ \forall_{\boldsymbol{j}\in\mathcal{I}(X_{\mathrm{r}})}\ \boldsymbol{n}(X_{\mathrm{p}},\boldsymbol{i}) = \boldsymbol{n}(X_{\mathrm{q}},\boldsymbol{j}) \Rightarrow\\
&\forall_{\boldsymbol{k}\in\mathcal{I}(Y_{\mathrm{p}})}\ t(X_{\mathrm{p}},\boldsymbol{i}) \le t(Y_{\mathrm{p}},\boldsymbol{k}) \le t(X_{\mathrm{r}},\boldsymbol{j}) \ \wedge\ a(X_{\mathrm{p}},\boldsymbol{i}) = a(Y_{\mathrm{p}},\boldsymbol{k}) \Rightarrow X_{\mathrm{p}} = Y_{\mathrm{p}} \ \wedge\ \boldsymbol{i} = \boldsymbol{k}\\
\equiv\quad &\{\ X_{\mathrm{p}} \ne Y_{\mathrm{p}},\ X_{\mathrm{p}} \text{ has only one execution }\}\\
&\forall_{\boldsymbol{j}\in\mathcal{I}(X_{\mathrm{r}})}\ \boldsymbol{b} = \boldsymbol{A}\boldsymbol{j} \Rightarrow\\
&\forall_{\boldsymbol{k}\in\mathcal{I}(Y_{\mathrm{p}})}\ a(X_{\mathrm{p}},[\,]) \ne a(Y_{\mathrm{p}},\boldsymbol{k}) \ \vee\ s(u) > t(Y_{\mathrm{p}},\boldsymbol{k}) \ \vee\ t(Y_{\mathrm{p}},\boldsymbol{k}) > t(X_{\mathrm{r}},\boldsymbol{j})\\
\Rightarrow\quad &\{\ \boldsymbol{l} \in \mathcal{I}(Y_{\mathrm{p}}) = \mathcal{I}(X_{\mathrm{r}}),\ \boldsymbol{b} = \boldsymbol{A}\boldsymbol{l}\ \}\\
&a(X_{\mathrm{p}},[\,]) \ne a(Y_{\mathrm{p}},\boldsymbol{l}) \ \vee\ s(u) > t(v,\boldsymbol{l}) \ \vee\ t(v,\boldsymbol{l}) > t(w,\boldsymbol{l})\\
\equiv\quad &\{\ s(u) < t(v,\boldsymbol{l}) < t(w,\boldsymbol{l})\ \}\\
&a(X_{\mathrm{p}},[\,]) \ne a(Y_{\mathrm{p}},\boldsymbol{l}).
\end{aligned}$$

Hence, we need at least two addresses. As at most one address is mapped to each cache block, each feasible schedule will cause at least two compulsory misses, i.e., $c_{\mathcal{G},\mathcal{C}}(\sigma) \geq 2$.
($\Leftarrow$) Suppose that $\forall_{\boldsymbol{i}\in\mathbb{Z}^\delta}\ \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \Rightarrow \boldsymbol{A}\boldsymbol{i} \neq \boldsymbol{b}$. Now we show that a feasible schedule σ exists with $c_{\mathcal{G},\mathcal{C}}(\sigma) \leq 1$. To this end, we choose a time assignment with

- period vectors $\boldsymbol{p}(u) = [\,]$,
  $p_i(w) = p_i(v) = 2\prod_{j=i+1}^{\delta-1}(I_j+1)$ for all $i = 0,\ldots,\delta-1$, and
- start times $s(u) = 0$, $s(v) = 1$, and $s(w) = 2$.

As $t(u,[\,]) < t(v,\boldsymbol{i}) < t(w,\boldsymbol{i})$ for all $\mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{I}$, this time assignment is feasible. Furthermore we choose an address assignment with

- address coefficient vectors $\boldsymbol{c}(X) = \boldsymbol{c}(Y) = \mathbf{0}$, and
- address offsets $o(X) = o(Y) = 0$.

Now, all six address constraints are satisfied. We have constructed a schedule that uses only one addresses, and will produce exactly one compulsory misses, i.e., $c_{\mathcal{G},\mathcal{C}}(\sigma) = 1 \leq K$.

So, $\mathcal{I}_{\text{co-mpcsd}}$ has a solution if and only if $\mathcal{I}_{\text{sle}}$ has one. Note that the size of $\mathcal{I}_{\text{co-mpcsd}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{sle}}$. Furthermore, the largest number in $\mathcal{I}_{\text{co-mpcsd}}$ is polynomially bounded in the largest number in $\mathcal{I}_{\text{sle}}$ and the size of $\mathcal{I}_{\text{sle}}$. Therefore, co-MPCSD is NP-hard in the strong sense. □

Even if we restrict this problem to program graphs with lexicographical index orderings and a cost bound one, the problem remains hard.

Theorem 4.13. *co-MPCSD remains NP-hard in the strong sense for lexicographical index orderings and $K = 1$.*
Proof. We prove this by reducing an instance $\mathcal{I}_{\text{sle-ll}'}$ of SLE-LL$'$ into an instance $\mathcal{I}_{\text{co-mpcsd}}$ of co-MPCSD. $\mathcal{I}_{\text{sle-ll}'}$ is given by matrices $\boldsymbol{A}' \in \mathbb{Z}^{\alpha\times\delta'}$ and $\boldsymbol{A}'' \in \mathbb{Z}^{\alpha\times\delta''}$, vectors $\boldsymbol{I}' \in \mathbb{N}^{\delta'}$ and $\boldsymbol{I}'' \in \mathbb{N}^{\delta''}$, and vector $\boldsymbol{b} \in \mathbb{Z}^\alpha$. For both matrices the lexicographical index order property, $\text{lio}(\boldsymbol{I}',\boldsymbol{A}')$ and $\text{lio}(\boldsymbol{I}'',-\boldsymbol{A}'')$, hold. Again the operations and ports of the program graph of $\mathcal{I}_{\text{co-mpcsd}}$ are given by Figure 4.2. We choose

- three operations $V = \{u,v,w\}$,
- operation read ports $R(u) = \varnothing$, $R(v) = \{X_\text{q}\}$, $R(w) = \{X_\text{r},Y_\text{q}\}$,
- operation write ports $W(u) = \{X_\text{p}\}$, $W(v) = \{Y_\text{p}\}$, $W(w) = \varnothing$,
- iterator bound vectors $\boldsymbol{I}(u) = \boldsymbol{I}'$, $\boldsymbol{I}(v) = \boldsymbol{I}(w) = \boldsymbol{I}''$,
- index matrices $\boldsymbol{A}(X_\text{p}) = \boldsymbol{A}'$, $\boldsymbol{A}(X_\text{q}) = \boldsymbol{A}(X_\text{r}) = -\boldsymbol{A}''$, $\boldsymbol{A}(Y_\text{p}) = \boldsymbol{A}(Y_\text{q}) = \mathbf{I}$,
- index offset vectors $\boldsymbol{b}(X_\text{p}) = \mathbf{0}$, $\boldsymbol{b}(X_\text{q}) = \boldsymbol{b}(Z_\text{r}) = \boldsymbol{b}$, $\boldsymbol{b}(Y_\text{p}) = \boldsymbol{b}(Y_\text{q}) = \mathbf{0}$,
- a fully set-associative cache with size $|\mathcal{E}|$ and block size one, and

- cost bound $K = 1$.

We then have to prove that

$$\exists_{\boldsymbol{i} \in \mathbb{Z}^{\delta'+\delta''}} \mathbf{0} \leq \boldsymbol{i} \leq \begin{bmatrix} \boldsymbol{I}' \\ \boldsymbol{I}'' \end{bmatrix} \wedge \begin{bmatrix} \boldsymbol{A}' & \boldsymbol{A}'' \end{bmatrix} \boldsymbol{i} = \boldsymbol{b} \Leftrightarrow \neg \exists_\sigma \sigma \text{ feasible} \wedge c_{\mathcal{G},\mathcal{C}}(\sigma) \leq 1.$$

This proof goes along the same lines as the proof of theorem 4.12. First of all, checking ($\Rightarrow$) again means proving that at least two different addresses are necessary for a feasible schedule, which is done in an analogous way. For the implication ($\Leftarrow$) we choose time assignment $\tau = (\boldsymbol{p}, s)$ and address assignment $\mu = (\boldsymbol{c}, o)$ with

- period vectors $p_i(u) = -\prod_{j=i+1}^{\delta'-1}(I'_j + 1)$, $p_i(v) = p_i(w) = 2\prod_{j=i+1}^{\delta''-1}(I''_j + 1)$.
- start times $s(u) = 0$, $s(v) = 1$, and $s(w) = 2$,
- address coefficient vectors $\boldsymbol{c}(X) = \boldsymbol{c}(Y) = \mathbf{0}$, and
- address offsets $o(X) = o(Y) = 0$.

All constraints are met and as exactly one address is used, this schedule has cost one.

So, $\mathcal{I}_{\text{co-mpcsd}}$ has a solution if and only if $\mathcal{I}_{\text{sle-ll}'}$ has one. Note that the size of $\mathcal{I}_{\text{co-mpcsd}}$ is polynomially bounded in the size of $\mathcal{I}_{\text{sle-ll}'}$. Furthermore, the largest number in $\mathcal{I}_{\text{co-mpcsd}}$ is polynomially bounded in the largest number in $\mathcal{I}_{\text{sle-ll}'}$ and the size of $\mathcal{I}_{\text{sle-ll}'}$. Therefore, co-MPCSD is NP-hard in the strong sense. □

4.4 Discussion

In Figure 4.3 the complexity results of this chapter have been depicted graphically. The arrows denote reductions between the problems that have been given throughout this chapter.

Clout [1994] has studied the problem of scheduling programs that can be represented as synchronous data flow graphs [Lee & Messerschmitt, 1987] on a processor with separate local memories for instructions and data. He shows that minimising the bandwidth between main memory and the two local memories for this kind of problems is NP-complete in the strong sense. Kennedy & McKinley [1993] prove that the problem of fusing loops for optimal reuse is NP-hard. Gupta, Malloy & McRae [1997] prove that the problem of instruction reordering within a basic block for data cache optimisation is NP-complete. The problems *register sufficiency* and *register sufficiency for loops* [Garey & Johnson, 1979] handle the case where all data dependencies have been given explicitly. For these problems, deciding whether the number of data items to be stored is bounded by a given number has been shown to be NP-complete.

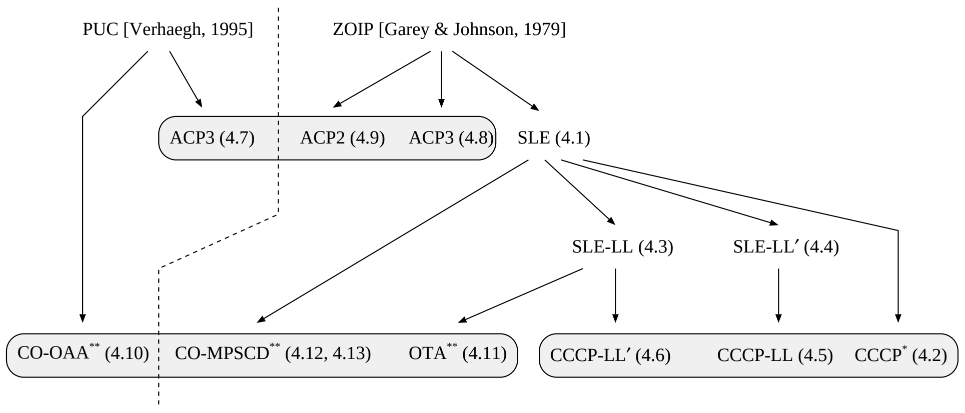


Figure 4.3. An overview of the complexity results of this chapter. Between brackets the numbers of the theorems are given. Problems to the left of the line are NP-hard. To the right of the dotted line the problems are hard in the strong sense. The problem tagged with one asterisk is not NP-hard but cannot be solved in polynomial time unless P = NP. Problems tagged with two asterisks are not known to be in NP.

5

Sequentialisability

In this chapter we show that not all schedules can be executed efficiently by a processor. To restrict the set of schedules we consider for the scheduling problem to schedules that can be executed efficiently, we introduce a property of schedules, called *sequentialisability*. The problem of inefficient execution of schedules stems from the single thread of execution that is common for processors. In the model of multidimensional periodic operations the time assignment determines independent threads of control for every operation in the program graph. In hardware these threads can be implemented by separate pieces of logic, but in a processor we typically have only one thread of control. In this chapter we give conditions on schedules that ensure that these separate operations can *efficiently* be mapped onto a processor. Such a mapping we call an *implementation* of a schedule. In Section 5.1 we define what we mean by an implementation, and under which conditions we speak of an efficient implementation. Section 5.2 gives sufficient conditions on schedules for an efficient implementation.

5.1 Sequential programs

It is not difficult to see that every schedule σ can be translated into a sequential program, as the number of executions is finite, and thus they can be written as a single sequence. Of course, such an implementation results in very long programs. The

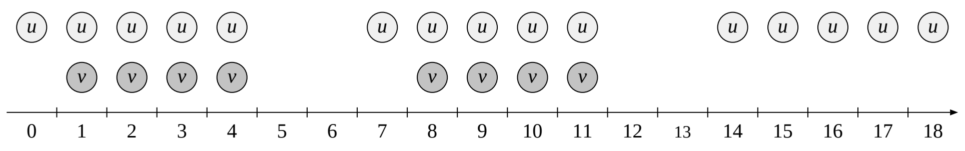


Figure 5.1. A schedule for two operations u and v with iterator bound vectors $\boldsymbol{I}(u) = \begin{bmatrix}2 & 4\end{bmatrix}^{\mathrm{T}}$ and $\boldsymbol{I}(v) = \begin{bmatrix}1 & 3\end{bmatrix}^{\mathrm{T}}$ having start times $s(u) = 0$, $s(v) = 1$ and period vectors $\boldsymbol{p}(u) = \boldsymbol{p}(v) = \begin{bmatrix}7 & 1\end{bmatrix}^{\mathrm{T}}$.

repetitive nature of operations often allows a compact implementation by means of repetition statements. However, there are choices to be made in order to obtain an efficient implementation of a schedule. In this section, we look at two cost measures for implementations and define what we mean by an implementation of a schedule.

First observe that we want the processor to spend as few machine cycles as possible on overhead. All repetition statements and condition statements do not contribute to the actual computation and can thus be viewed as overhead. On the other hand, if no repetition statements are used the code of the sequential program will become very large, causing, besides large code size, a vast number of instruction cache misses. Because of these observations we choose two cost measures for an implementation of a schedule, being

- the number of guard evaluations as a measure of time not spent on the computation proper, and
- the number of operation labels appearing in the program, as an indication of the size of the program.

These operation labels can be thought of as function calls in a programming language like C. At least one operation label occurs in a program for each operation.

By means of an example we explore several programs that all implement the schedule given in Figure 5.1. The programs vary in their numbers of operation labels and numbers of guard evaluations.

First of all, let us take an 'easy' approach by stepping through time as depicted in the program of Figure 5.2. For each time slot we check if an operation exists that 'wants' to execute at that time. In general for each schedule with weak lexicographical executions for every operation, we can construct such a program.

Clearly, this program results in a large number of guard evaluations. Furthermore, note that several guards have to be evaluated for the *increase* statements. The total number of guard evaluations exceeds 88. This number consists of 20 evaluations for the outer **do**-statement, and of $15 \cdot 4 + 4 \cdot 2 = 68$ evaluations for the inner **do**-statement. Additional guard evaluations are needed for lexicographically

```
$t := 0;\ \boldsymbol{i} := \begin{bmatrix}0 & 0\end{bmatrix}^{\mathrm{T}};\ \boldsymbol{j} := \begin{bmatrix}0 & 0\end{bmatrix}^{\mathrm{T}};$
**do** $t < 19 \rightarrow$
    **do** $\boldsymbol{p}^{\mathrm{T}}(u)\boldsymbol{i} + s(u) = t \rightarrow u(\boldsymbol{i})$; ‘increase $\boldsymbol{i}$’
    [] $\boldsymbol{p}^{\mathrm{T}}(v)\boldsymbol{j} + s(v) = t \rightarrow v(\boldsymbol{j})$; ‘increase $\boldsymbol{j}$’
    **od**;
    $t := t + 1$
**od**
```

Figure 5.2. First implementation of the schedule of Figure 5.1. The operation labels for operations u and v are $u(\boldsymbol{i})$ and $v(\boldsymbol{j})$ respectively. The ‘increase $\boldsymbol{i}$’-statement and ‘increase $\boldsymbol{j}$’ statements increase the variables $\boldsymbol{i} \in \mathcal{I}(u)$ and $\boldsymbol{j} \in \mathcal{I}(v)$ lexicographically.

```
**for** $i_0 := 0 \ldots 2 \rightarrow$
    **for** $i_1 := 0 \ldots 4 \rightarrow$
        $u(i_0, i_1)$;
        **if** $i_1 \geq 1 \wedge i_0 \leq 1 \rightarrow v(i_0, i_1 - 1)$ **fi**
    **rof**
**rof**
```

Figure 5.3. Second implementation of the schedule of Figure 5.1.

increasing the variables $\boldsymbol{i}$ and $\boldsymbol{j}$. On the other hand, for each operation only one operation label appears in the program, which is optimal with respect to the second cost measure.

Another implementation is possible because of the following observation. As the period vectors of the operations are equal, the operations execute ‘synchronously’. Hence, we can use the same repetition statements to code the executions of u and v, resulting in the program in Figure 5.3.

By the removal of time, i.e., program variable t, from the program, the number of guard evaluations has been reduced to $4 + 6 \cdot 3 + 3 \cdot 5 \cdot 2 = 52$ with the same number of operation labels. The number of guard evaluations can be reduced at the expense of more operation labels in the program, as given in the program in Figure 5.4.

At the expense of one operation label the number of guard evaluations has again been reduced, now to $3 + 2 \cdot 6 + 2 \cdot 5 + 6 = 31$. The **if**-statements can all be removed from the program, resulting in the program in Figure 5.5.

Again, at the expense of one operation label, we are left with $3 + 5 \cdot 2 + 6 = 19$ guard evaluations. Now we have reached the situation where each removal of an evaluation results in one additional operation label in the program. In the extreme

```
**for** $i_0 := 0 \ldots 1 \rightarrow$
    **for** $i_1 := 0 \ldots 4 \rightarrow$
        $u(i_0, i_1)$;
        **if** $i_1 \geq 1 \rightarrow v(i_0, i_1 - 1)$ **fi**
    **rof**
**rof**;
**for** $i_1 := 0 \ldots 4 \rightarrow u(2, i_1)$ **rof**
```

Figure 5.4. Third implementation of the schedule of Figure 5.1.

```
**for** $i_0 := 0 \ldots 1 \rightarrow$
    $u(i_0, 0)$;
    **for** $i_1 := 0 \ldots 3 \rightarrow u(i_0, i_1 + 1); v(i_0, i_1)$ **rof**
**rof**;
**for** $i_1 := 0 \ldots 4 \rightarrow u(2, i_1)$ **rof**
```

Figure 5.5. Fourth implementation of the schedule of Figure 5.1.

case no guard evaluations are left resulting in 23 operation labels, and the program in Figure 5.6.

$u(0,0)$; $u(0,1)$; $v(0,0)$; $u(0,2)$; $v(0,1)$; $u(0,3)$; $v(0,2)$; $u(0,4)$; $v(0,3)$;
$u(1,0)$; $u(1,1)$; $v(1,0)$; $u(1,2)$; $v(1,1)$; $u(1,3)$; $v(1,2)$; $u(1,4)$; $v(1,3)$;
$u(2,0)$; $u(2,1)$; $u(2,2)$; $u(2,3)$; $u(2,4)$

Figure 5.6. Fifth implementation of the schedule of Figure 5.1.

Both the first and the fifth program are extremes in the sense that the first has a minimum of operation labels but many guard evaluations, and the fifth program has a minimum of guard evaluations but a maximum of operation labels. The programs in between can all be achieved by rewriting the second program, which makes it possible to make a trade-off between guard evaluations and operation labels. As this trade-off falls outside the scope of this thesis, we choose for the approach of the second implementation, where one operation label is used for every operation in the program graph. Hence, in its general form we define a sequential program as follows.

Definition 5.1 (sequential program). The syntax of sequential programs is given in Figure 5.7. A sequential program is a statement S. A statement can either be the sequential composition of two statements, a **for**-statement, or an **if**-statement. The guard of an **if**-statement consists of a conjunction of bounds on iterator variables. The iterator variables x are taken from $i, j, \ldots$ and i_0, j_0, i_1, j_1 and so forth. Parameter N stands for an integer constant, and parameter E is a comma-separated list

$$\begin{array}{lll} \langle \mathrm{S} \rangle & \mathbf{::=} & \langle \mathrm{S} \rangle \; ; \langle \mathrm{S} \rangle \;\Big|\; \mathbf{for}\ x := 0 \ldots N \rightarrow \langle \mathrm{S} \rangle\ \mathbf{rof} \;\Big|\; \mathbf{if}\ \langle \mathrm{G} \rangle \rightarrow\ u(E)\ \mathbf{fi} \\ \langle \mathrm{G} \rangle & \mathbf{::=} & \langle \mathrm{G} \rangle \wedge \langle \mathrm{G} \rangle \;\Big|\; x \leq N \;\Big|\; x \geq N \end{array}$$

Figure 5.7. Sequential program syntax in BNF (Backus Naur Form).

of integer expressions of the form $c_0 \cdot i_0 + \ldots + c_{n-1} \cdot i_{n-1} + d$ for integer constants d, c_l, and n and iterator variables i_l. □

Next, we must state precisely under which conditions a program does exactly what we expected from a schedule, which means that the program executes exactly those operation executions as in the schedule, and that it executes them in the same order as demanded by the schedule. If we look at the schedule as a specification of the sequential program that we want to write, we can define an implementation of a schedule as a program that satisfies the schedule.

For the preservation of execution order by an implementation we introduce a *step number assignment* by means of an example.

```
**for** $j_0 := 0 \ldots 1 \rightarrow$
    **for** $j_1 := 0 \ldots 2 \rightarrow$
        $u(j_0, j_1)$
    **rof**
**rof**;
**for** $j_0 := 0 \ldots 2 \rightarrow v(j_0)$ **rof**
```

In this program $u(0,0)$ executes before $u(0,1)$, which is executed before $u(0,2)$, and so on. If we assign the step numbers $n(u, \boldsymbol{j}) = 3j_0 + j_1$ and $n(v, \boldsymbol{j}) = 6 + j_0$, we get the desired property that $u(i_0, i_1)$ is executed before $u(j_0, j_1)$ if and only if $n(u, \boldsymbol{i}) < n(v, \boldsymbol{j})$. The same holds for $u(i_0, i_1)$ and $v(j)$. In this example the step numbers start at 0 and are numbered successively. In general a step number assignment is not unique and may be chosen at random as long as it correctly represents the order of execution of the operation labels.

Definition 5.2 (step number assignment). For a sequential program, which executes every operation label $u(i_0, \ldots, i_{\delta(u)-1})$ at most once, a step number assignment is given by an integer $n(u, \boldsymbol{i})$ for each execution of an operation label $u(i_0, \ldots, i_{\delta(u)-1})$. Furthermore we demand that $n(u, \boldsymbol{i}) < n(v, \boldsymbol{j})$ if and only if $u(i_0, \ldots, i_{\delta(u)-1})$ is executed before $v(j_0, \ldots, j_{\delta(v)-1})$ for each pair of executions of operation labels. □

Now, we can define under which conditions a sequential program is an implementation of a schedule.

Definition 5.3 (implementation of a schedule). Given are a program graph $\mathcal{G}$ and a schedule σ. A sequential program is called an implementation of the schedule if

- for each operation u and each vector $\boldsymbol{i} \in \mathbb{Z}^{\delta(u)}$, $\boldsymbol{i}$ is an execution of operation u, i.e., $\boldsymbol{i} \in \mathcal{I}(u)$ if and only if there is exactly one execution of an operation label $u(i_0, \ldots, i_{\delta(u)-1})$ during execution of the program,
- the order of execution of operations imposed by the schedule is preserved by the program, i.e., $t(u,\boldsymbol{i}) < t(v,\boldsymbol{j}) \Rightarrow n(u,\boldsymbol{i}) < n(v,\boldsymbol{j})$ for each pair of executions $\boldsymbol{i},\boldsymbol{j}$ of operations u and v and for some step number assignment n of the sequential program.

$\square$

Except for the first one, all programs from the earlier example are implementations of the schedule of Figure 5.1. For the number of operation labels, that we introduced earlier as a cost measure for an implementation, we introduce the term *compact-sequentialisability*.

Definition 5.4 (n-compact-sequentialisability). A schedule is called n-compact-sequentialisable if an implementation of the schedule exists in which the total number of operation labels is at most n times the number of operations in the schedule. 1-compact-sequentialisable schedules are also called compact-sequentialisable. $\square$

Next, we look at sufficient conditions for compact-sequentialisable schedules.

5.2 Sufficient conditions for compactness

In the remainder of this chapter we shall assume all periods to be non-negative and ordered decreasingly, i.e., $p_0 \geq p_1 \geq \ldots \geq p_{\delta-1}$. Periods can always be made non-negative by renumbering the iterations of dimensions with negative period, i.e., by making the substitution $i_l = I_l - i'_l$.

First we give a sufficient condition for compact-sequentialisability for a single operation. After that we give three sufficient conditions for two operations and show how they generalise to more operations.

5.2.1 One operation

A single operation u can be implemented by the sequential program of Figure 5.8 if u has weak lexicographical executions, i.e., $\text{wlex}(\boldsymbol{p}(u),\boldsymbol{I}(u))$ holds. Every execution of an operation $(u,\boldsymbol{i})$ corresponds to exactly one execution of an operation label $u(i_0, \ldots, i_{\delta(u)-1})$, and vice versa. As every step number assignment for the given program has the property $\boldsymbol{i} <_{\text{lex}} \boldsymbol{j} \Leftrightarrow n(u,\boldsymbol{i}) < n(u,\boldsymbol{j})$, and operation u has weak lexicographical executions, i.e., $\boldsymbol{i} \leq_{\text{lex}} \boldsymbol{j} \Rightarrow t(u,\boldsymbol{i}) \leq t(u,\boldsymbol{j})$, and hence $t(u,\boldsymbol{i}) < t(u,\boldsymbol{j}) \Rightarrow \boldsymbol{i} <_{\text{lex}} \boldsymbol{j}$, we have

$$t(u,\boldsymbol{i}) < t(u,\boldsymbol{j}) \Rightarrow n(u,\boldsymbol{i}) < n(u,\boldsymbol{j}),$$

for all executions $\boldsymbol{i},\boldsymbol{j} \in \mathcal{I}(u)$ of operation u.

```
for $i_0 := 0 \ldots I_0(u) \rightarrow$
    ...
        for $I_{\delta(u)-1} := 0 \ldots I_{\delta(u)-1} \rightarrow$
            $u(i_0, \ldots, i_{\delta(u)-1})$
        rof
    ...
rof
```

Figure 5.8. Implementation of an operation with the weak lexicographical ordering property.

5.2.2 Disjoint lifetimes

Two operations u and v with $s(u) \leq s(v)$ are said to have disjoint lifetimes if the last execution of operation u takes place before the first execution of v, i.e.,

$$s(u) + \sum_{l=0}^{\delta(u)-1} p_l(u) I_l(u) \leq s(v).$$

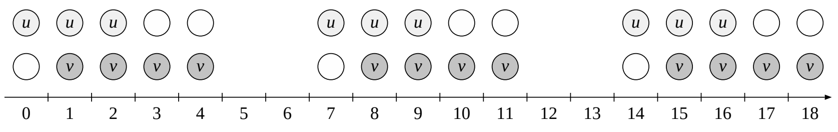
For a set V of operations and δ the maximum dimension of any operation, i.e., $\max_{u \in V} \delta(u)$, we can check in $\mathcal{O}(|V| \log(|V|)\delta)$ time whether the operations have disjoint lifetimes. This is done by sorting the operations on start times, followed by checking the disjoint lifetime condition for each operation and its successor. Hence, provided that the schedules for every single operation are compact-sequentialisable, checking compact-sequentialisability for a set of operations with disjoint lifetimes can be done efficiently.

The disjoint lifetime condition, however, is too strict. It only allows a sequence of loop nestings with one operation label per nesting. The schedule in the example of Figure 5.1, for instance, cannot be implemented by such a sequential program.

5.2.3 Equal periods

At the beginning of this chapter we indicated that compact-sequentialisable schedules may exist for two operations with equal period vectors, i.e., $\boldsymbol{p}(u) = \boldsymbol{p}(v)$. In Figure 5.9(a) we give another example. Here, we have added *ghost* executions by means of empty circles. With the six additional ghost executions for operation u and three for v, operations u and v have the same pattern of repetition, which leads to the sequential program of Figure 5.9(b), where ghost executions are discarded by means of the **if**-statements.

We can look at these operations in another way. If we *shift* the operations of u to the right with a vector $\boldsymbol{d} = \begin{bmatrix} 0 & 1 \end{bmatrix}^{\mathrm{T}}$, which results in a shift in time with $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{d} = 1$,



(a)

```
for $i_0$ := $0 \ldots 2 \rightarrow$
    for $i_1$ := $0 \ldots 4 \rightarrow$
        if $0 \leq i_1 \leq 2 \rightarrow u(i_0, i_1)$ fi;
        if $1 \leq i_1 \leq 4 \rightarrow v(i_0, i_1 - 1)$ fi
    rof
rof
```

(b)

Figure 5.9. A schedule (a) for two operations u and v with iterator bound $\boldsymbol{I}(u) = \begin{bmatrix} 2 & 2 \end{bmatrix}^{\mathrm{T}}$ and $\boldsymbol{I}(v) = \begin{bmatrix} 2 & 3 \end{bmatrix}^{\mathrm{T}}$ having start times $s(u) = 0$, $s(v) = 1$ and period vectors $\boldsymbol{p}(u) = \boldsymbol{p}(v) = \begin{bmatrix} 8 & 1 \end{bmatrix}^{\mathrm{T}}$. The sequential program in (b) is a possible implementation of the schedule.

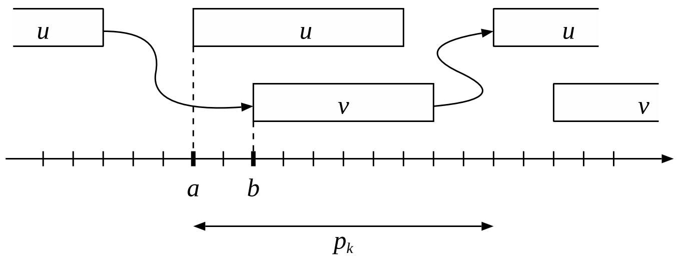


Figure 5.10. The edges represent conditions for *equal periods*, which assure that the group of executions of u in dimensions $k+1 \ldots \delta - 1$ starting at $a = s(u) + \sum_{l=0}^{k} p_l d_l$ and the group of executions of v in dimensions $k+1 \ldots \delta - 1$ starting at $b = s(v)$ can be implemented by the same loop nest.

```
for $i_0 := 0 \ldots J_0 \rightarrow$
   $\ldots$
     for $i_{\delta-1} := 0 \ldots J_{\delta-1} \rightarrow$
        if $B_u \rightarrow u(i_0 + d_0^-, \ldots, i_{\delta-1} + d_{\delta-1}^-)$ fi;
        if $B_v \rightarrow v(i_0 - d_0^+, \ldots, i_{\delta-1} - d_{\delta-1}^+)$ fi
     rof
   $\ldots$
rof
```

with

$$B_u = 0 \leq i_0 + d_0^- \leq I_0(u) \wedge \ldots \wedge 0 \leq i_{\delta-1} + d_{\delta-1}^- \leq I_{\delta-1}(u)$$
$$B_v = 0 \leq i_0 - d_0^+ \leq I_0(v) \wedge \ldots \wedge 0 \leq i_{\delta-1} - d_{\delta-1}^+ \leq I_{\delta-1}(v)$$
$$J_l = \max(I_l(u) - d_l^-, I_l(v) + d_l^+), \text{ for } l = 0, \ldots, \delta - 1$$

Figure 5.11. Program for operations u and v that satisfy the *equal periods* conditions.

we get a regular pattern of executions, in which only the number of executions in each dimension can vary for the operations. The same effect is achieved by moving operation v to the left with vector $-\boldsymbol{d}$, which results in a shift in time with $-\boldsymbol{p}^{\mathrm{T}}\boldsymbol{d} = -1$. With such a displacement vector $\boldsymbol{d} \in \mathbb{Z}^{\delta}$, where $\delta = \delta(u) = \delta(v)$, we shift the first execution of u towards the first execution of v.

As is depicted in Figure 5.10, every group of executions inside dimension k of operation u may have an overlap with at most one such group of v and vice versa. In the figure these conditions are depicted by the two arrows. Formally, they are defined as

$$s(u) + \sum_{l=0}^{k} p_l d_l + \sum_{l=k+1}^{\delta-1} p_l I_l(u) - p_k \leq s(v) \tag{5.1}$$

$$s(v) + \sum_{l=k+1}^{\delta-1} p_l I_l(v) \leq s(u) + \sum_{l=0}^{k} p_l d_l + p_k, \tag{5.2}$$

which has to hold for all $k = 0, \ldots, \delta - 1$.

In order to make a loop nest in which u executes before v, we must furthermore demand that the *shifted* execution of u is executed not later than the first execution of v, i.e.,

$$s(u) + \sum_{l=0}^{\delta-1} p_l d_l \leq s(v). \tag{5.3}$$

The next theorem shows that these conditions are sufficient for a schedule to be compact-sequentialisable.

Theorem 5.1. *Given are two operations u and v with iterator bound vectors $\boldsymbol{I}(u)$ and $\boldsymbol{I}(v)$, start times $s(u)$ and $s(v)$, and equal period vectors $\boldsymbol{p} = \boldsymbol{p}(u) = \boldsymbol{p}(v)$. Both operations satisfy the weak lexicographical property, i.e.,* $\mathrm{wlex}(\boldsymbol{p}, \boldsymbol{I}(u))$ *and* $\mathrm{wlex}(\boldsymbol{p}, \boldsymbol{I}(v))$ *hold. Let $\boldsymbol{d} \in \mathbb{Z}^\delta$ be a vector that satisfies (5.1), (5.2), and (5.3). Then the schedule for these operations is compact-sequentialisable, and can be implemented by the program of Figure 5.11.*

Proof. First of all, showing that the program in Figure 5.11 is an implementation of the schedule is sufficient for proving compact-sequentialisability, as the operation labels u and v appear only once in the program. The guards of the operation labels and the iterator bounds of the **for**-statements assure that each execution of an operation label corresponds to exactly one execution of an operation, and vice versa. What remains to be shown is that the order of execution of operations is maintained by the sequential program. Therefore, we introduce the following step number assignment.

$$n(u, \boldsymbol{i}) = b(u) + \boldsymbol{q}^{\mathrm{T}} \boldsymbol{i} \quad n(v, \boldsymbol{j}) = b(v) + \boldsymbol{q}^{\mathrm{T}} \boldsymbol{j}$$

$$b(u) = -\sum_{l=0}^{\delta-1} q_l d_l^- \quad b(v) = 1 + \sum_{l=0}^{\delta-1} q_l d_l^+$$

$$q_k = 2 \prod_{l=k+1}^{\delta-1} (J_l + 1) \text{ for all } k = 0, \ldots, \delta - 1.$$

First, we show that the executions of operation u are correctly ordered by step number assignment n by deriving

$$\begin{aligned}
& t(u, \boldsymbol{i}) < t(u, \boldsymbol{j}) \\
\Rightarrow \quad & \{ \ \mathrm{wlex}(\boldsymbol{p}, \boldsymbol{I}(u)) \ \} \\
& \boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{j} \\
\equiv \quad & \{ \ \mathrm{lex}(\boldsymbol{q}, \boldsymbol{I}(u)) \ \} \\
& n(u, \boldsymbol{i}) < n(u, \boldsymbol{j}),
\end{aligned}$$

for all executions $\boldsymbol{i}, \boldsymbol{j} \in \mathcal{I}(u)$ of operation u. The same argument is used for executions of operation v. In order to prove preservation of order between executions of operations u and v we show

$$t(u,\boldsymbol{i}) < t(v,\boldsymbol{j}) \;\Rightarrow\; n(u,\boldsymbol{i}) < n(v,\boldsymbol{j})$$
$$t(u,\boldsymbol{i}) > t(v,\boldsymbol{j}) \;\Rightarrow\; n(u,\boldsymbol{i}) > n(v,\boldsymbol{j}).$$

From here, we split the proof into two parts. First, we show that the order of executions of operation labels $u(i_0,\ldots,i_{\delta-1})$ and $v(j_0,\ldots,j_{\delta-1})$ can be expressed in terms of vectors $\boldsymbol{i},\boldsymbol{j}$, and $\boldsymbol{d}$, namely

$$\boldsymbol{i} \leq_{\text{lex}} \boldsymbol{j}+\boldsymbol{d} \;\Rightarrow\; n(u,\boldsymbol{i}) < n(v,\boldsymbol{j}) \tag{5.4}$$
$$\boldsymbol{i} >_{\text{lex}} \boldsymbol{j}+\boldsymbol{d} \;\Rightarrow\; n(u,\boldsymbol{i}) > n(v,\boldsymbol{j}), \tag{5.5}$$

for all executions $\boldsymbol{i} \in \mathcal{I}(u)$ and $\boldsymbol{j} \in \mathcal{I}(v)$. After that we show preservation of order of executions by proving

$$t(u,\boldsymbol{i}) < t(v,\boldsymbol{j}) \;\Rightarrow\; \boldsymbol{i} \leq_{\text{lex}} \boldsymbol{j}+\boldsymbol{d}, \text{ and} \tag{5.6}$$
$$t(u,\boldsymbol{i}) > t(v,\boldsymbol{j}) \;\Rightarrow\; \boldsymbol{i} >_{\text{lex}} \boldsymbol{j}+\boldsymbol{d}. \tag{5.7}$$

For (5.4), we show that $\boldsymbol{i} = \boldsymbol{j}+\boldsymbol{d}$ is a sufficient condition for $n(u,\boldsymbol{i}) < n(v,\boldsymbol{j})$, by deriving

$$\begin{aligned}
& n(u,\boldsymbol{j}+\boldsymbol{d}) \\
= & \\
& -\sum_{l=0}^{\delta-1} q_l d_l^- + \sum_{l=0}^{\delta-1} q_l(j_l+d_l) \\
= & \qquad \{\, d_l = d_l^- + d_l^+ \,\} \\
& \sum_{l=0}^{\delta-1} q_l d_l^+ + \sum_{l=0}^{\delta-1} q_l j_l \\
< & \\
& n(v,\boldsymbol{j}).
\end{aligned}$$

Next, for $\boldsymbol{i} <_{\text{lex}} \boldsymbol{j}+\boldsymbol{d}$, we have an $0 \leq m < \delta$ with $i_m < j_m + d_m$ and $i_l = j_l + d_l$ for all $0 \leq l < m$. Hence,

$$\begin{aligned}
& n(u,\boldsymbol{i}) \\
< & \qquad \{\, \boldsymbol{i} <_{\text{lex}} \boldsymbol{j}+\boldsymbol{d} \,\} \\
& n(u,\boldsymbol{j}+\boldsymbol{d}) \\
< & \qquad \{ \text{ previous derivation } \} \\
& n(u,\boldsymbol{j}).
\end{aligned}$$

A similar argument is used for proving (5.5).

For (5.6) we must prove, assuming that $t(u,\boldsymbol{i}) < t(v,\boldsymbol{j})$, either $\boldsymbol{i} = \boldsymbol{j}+\boldsymbol{d}$, or $\boldsymbol{i} \lhd_{\text{lex}} \boldsymbol{j}+\boldsymbol{d}$. Assume that there exists an $l \in \{0,\ldots,\delta-1\}$ such that $i_l \neq j_l + d_l$. Then there also exists a smallest such an l, say m. Hence $i_m \neq j_m + d_m$ and $i_l = j_l + d_l$

for all $l = 0, \ldots, m-1$. Now, we only have to show that $i_m < j_m + d_m$. To this end we derive

$$
\begin{aligned}
& t(u,\boldsymbol{i}) < t(v,\boldsymbol{j}) \\
\equiv & \\
& s(u) + \sum_{l=0}^{m-1} p_l i_l + p_m i_m + \sum_{l=m+1}^{\delta-1} p_l i_l < s(v) + \sum_{l=0}^{m-1} p_l j_l + p_m j_m + \sum_{l=m+1}^{\delta-1} p_l j_l \\
\Rightarrow & \quad \{\ 0 \leq p_l,\ 0 \leq i_l,\ j_l \leq I_l(v)\ \} \\
& s(u) + \sum_{l=0}^{m-1} p_l i_l + p_m i_m < s(v) + \sum_{l=0}^{m-1} p_l j_l + \sum_{l=m+1}^{\delta-1} p_l I_l(v) + p_m j_m \\
\Rightarrow & \quad \{\ (5.2)\ \} \\
& s(u) + \sum_{l=0}^{m-1} p_l i_l + p_m i_m < s(u) + \sum_{l=0}^{m-1} p_l j_l + \sum_{l=0}^{m} p_l d_l + p_m + p_m j_m \\
\Rightarrow & \quad \{\ i_l = j_l + d_l \text{ for all } l = 0, \ldots, m-1\ \} \\
& p_m i_m < p_m j_m + p_m d_m + p_m \\
\Rightarrow & \quad \{\ 0 \leq p_m\ \} \\
& i_m \leq j_m + d_m \\
\equiv & \quad \{\ i_m \neq j_m + d_m\ \} \\
& i_m < j_m + d_m.
\end{aligned}
$$

Finally, for (5.7) we must show that $\boldsymbol{i} >_{\text{lex}} \boldsymbol{j} + \boldsymbol{d}$, if we assume $t(u,\boldsymbol{i}) > t(v,\boldsymbol{j})$. There always is an $m \in \{0, \ldots, \delta - 1\}$ with $i_l = j_l + d_l$ for all $l = 0, \ldots, m-1$. Hence, a largest such m also exists. If we assume $m = \delta$, then

$$
\begin{aligned}
& s(u) + \sum_{l=0}^{\delta-1} p_l i_l > s(v) + \sum_{l=0}^{\delta-1} p_l j_l \\
\equiv & \quad \{\ i_l = j_l + d_l \text{ for all } l = 0, \ldots, \delta-1\ \} \\
& s(u) + \sum_{l=0}^{\delta-1} p_l d_l > s(v),
\end{aligned}
$$

which contradicts (5.3). So, $m < \delta$ and $i_m \neq j_m + d_m$ and $i_l = j_l + d_l$ for all $l = 0, \ldots, m-1$. Now, only $i_m > j_m + d_m$ remains to be shown, which can be proved in a similar way as we did for (5.6). □

Having established sufficient conditions for sequentialisability, we must find a way of producing a vector $\boldsymbol{d}$ that satisfies (5.1), (5.2), and (5.3). If both (5.1) and (5.2) are satisfied for a vector $\boldsymbol{d}$, but (5.3) is not, a valid vector $\boldsymbol{d}$ can be found by negating $\boldsymbol{d}$ and switching the roles of u and v in the three conditions. If we assume that all periods are divisible, the following theorem shows that a vector $\boldsymbol{d}$ can be found in polynomial time.

Theorem 5.2. *The problem of deciding whether a vector* **d** *exists that satisfies (5.1) and (5.2) for equal period vectors with the property $p_{l+1} \mid p_l$ for all $l = 0, \ldots, \delta-2$ can be solved in polynomial time.*

Proof. When the two conditions are written out for $k = 0, \ldots, \delta-1$, we have

$$L_0 \le p_0 d_0 \le U_0 \tag{5.8}$$

$$L_1 \le p_0 d_0 + p_1 d_1 \le U_1 \tag{5.9}$$

$$\vdots$$

$$L_{\delta-1} \le \sum_{l=0}^{\delta-1} p_l d_l \le U_{\delta-1}, \tag{5.10}$$

with

$$L_k = s(v) - s(u) - p_k + \sum_{l=k+1}^{\delta-1} p_l(v) I_l(v), \text{ and}$$

$$U_k = s(v) - s(u) + p_k - \sum_{l=k+1}^{\delta-1} p_l(u) I_l(u).$$

Here, a possible value for d_0 can be extracted from (5.8), after which we determine a value for d_1 from (5.9), and so forth. Note that this algorithm does not yield a unique solution $\boldsymbol{d}$. Nevertheless, we can show that no *wrong* choice can be made. Suppose that we have a solution $\boldsymbol{e}$ for (5.8) … (5.10) and suppose that we have a solution $\boldsymbol{d}$ up to entry k, i.e., $d_0 \ldots d_{k-1}$ have been determined. Then we can show that it is possible to find a solution for d_k. Since $\boldsymbol{e}$ is a solution, we know that

$$\begin{aligned}
& L_k \le \sum_{l=0}^{k} p_l e_l \le U_k \\
\equiv & \\
& L_k \le p_k e_k + \sum_{l=0}^{k-1} p_l(e_l - d_l) + \sum_{l=0}^{k-1} p_l d_l \le U_k \\
\equiv & \quad \{\, p_l = p_k n_l \text{ for some } n_l \text{ and } l < k \,\} \\
& L_k \le p_k e_k + \sum_{l=0}^{k-1} p_k n_l(e_l - d_l) + \sum_{l=0}^{k-1} p_l d_l \le U_k,
\end{aligned}$$

so by choosing $d_k = e_k + \sum_{l=0}^{k-1} n_l(e_l - d_l)$ we find a solution if one exists. □

The complexity of finding a solution $\boldsymbol{d}$ for (5.1) and (5.2) for arbitrary period vectors remains unknown. The algorithm proposed in the theorem above cannot be used for arbitrary period vectors as is shown in the next example, where two operations u and v are given with iterator bound vectors $\boldsymbol{I}(u) = \begin{bmatrix}10 & 1 & 3\end{bmatrix}^{\mathrm{T}}$, $\boldsymbol{I}(v) = \begin{bmatrix}10 & 0 & 5\end{bmatrix}^{\mathrm{T}}$, start times $s(u) = 0$, $s(v) = 10$, and period vectors $\boldsymbol{p}(u) = \boldsymbol{p}(v) = \boldsymbol{p} =$

$\begin{bmatrix}20 & 6 & 1\end{bmatrix}^{\mathrm{T}}$. Rewriting (5.1) and (5.2) gives

$$\begin{gathered} -5 \le 20d_0 \le 21 \\ 9 \le 20d_0 + 6d_1 \le 13 \\ 9 \le 20d_0 + 6d_1 + d_2 \le 11. \end{gathered}$$

There are three solutions for these inequalities, namely with $d_0 = 0$, $d_1 = 2$, and $d_2 \in \{-1, -2, -3\}$. If we determine the entries d_k one by one, we can choose $d_0 = 1$, which does not leave a solution for d_1.

So far we have looked at the sequentialisability of two operations with equal periods. In order to generalise these conditions for more than two operations we look at the example of Figure 5.9, where the two operations can be seen as a single operation with start time $s = 0$, period vector $\boldsymbol{p} = \begin{bmatrix}7 & 1\end{bmatrix}^{\mathrm{T}}$, and iterator bound vector $\boldsymbol{I} = \begin{bmatrix}2 & 4\end{bmatrix}^{\mathrm{T}}$.

In general, any pair of operations that satisfy the *equal periods* conditions can be written as one single operation. Hence, sufficient conditions for multiple operations u_l are verified by alternately checking two operations and, if sequentialisable, composing them into one new operation. Unfortunately, whether or not we find that the operations are sequentialisable may depend on the order in which the operations are checked. This is a result of the nondeterminism in the algorithm of Theorem 5.2. The formal complexity of determining whether or not a set of operations is sequentialisable with respect to the *equal periods* conditions is unknown.

5.2.4 Gaps

The *equal period* conditions for compact-sequentialisability can be weakened even further. Without loss of generality we assume that $s(u) \le s(v)$. In Figure 5.12 two operations u and v have equal period for dimension 0, but a different period for dimension 1. Nevertheless, all executions of operation u in dimension 1 occur before executions of operation v in dimension 1. In general, the last execution of u that occurs not later than the first execution of v is $\boldsymbol{d}$ with

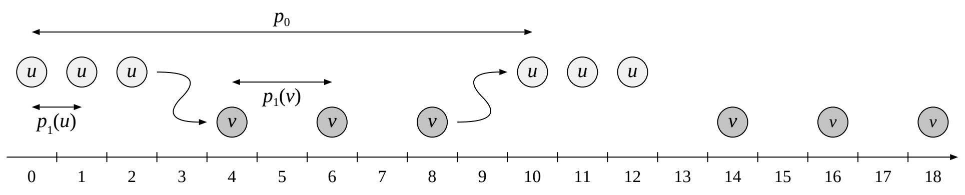
$$d_k = \min(I_k(u), (s(v) - s(u) - \sum_{l=0}^{k-1} p_l(u) d_l) \operatorname{div} p_k(u)), \tag{5.11}$$

for all $k = 0, \ldots, \delta(u) - 1$.

We define δ to be the first dimension in which u and v do not execute synchronously (dimension 1 in our example), i.e., we define

$$\delta = \min\{\, k \mid k \in \{0, \ldots, \delta(u) - 1\} \,\wedge\, \forall_{l \in \{k, \ldots, \delta(u)-1\}}\ d_l = I_l(u) \,\} \tag{5.12}$$

Now, we can give the conditions for compact-sequentialisability. These are denoted in Figure 5.12 by means of two arrows. The arrow that points towards



(a)

```
for $i_0$ := $0 \ldots 1 \rightarrow$
    for $i_1$ := $0 \ldots 2 \rightarrow$
        $u(i_0, i_1)$
    rof;
    for $i_1$ := $0 \ldots 2 \rightarrow$
        $v(i_0, i_1)$
    rof
rof
```

(b)

Figure 5.12. Operations *u* and *v* in (a) run synchronously in dimension 0 and have disjoint lifetimes for dimension 1. The loop nest in (b) correctly implements these operations.

the first execution of v originates from execution $\boldsymbol{d}$ of operation u by construction of vector $\boldsymbol{d}$. The second arrow is the same arrow as we saw for equal periods, corresponding to the condition

$$s(v) + \sum_{l=k+1}^{\delta(v)-1} p_l(v)I_l(v) \leq s(u) + \sum_{l=0}^{k} p_l(u)d_l + p_k(u), \tag{5.13}$$

for all $k = 0, \ldots, \delta - 1$.

These conditions are called *gap* conditions. In Figure 5.12 we see that the executions of u within dimension 0 leave a gap, in which the executions of v take place. In general, this holds for executions of u and v within dimension $\delta - 1$.

One can look at (5.13) as a generalisation of (5.2). Inequality (5.3) and a generalisation of (5.1) are met by the construction of vector $\boldsymbol{d}$ in (5.11).

Theorem 5.3. *Given are two operations u and v with iterator bound vectors $\boldsymbol{I}(u)$ and $\boldsymbol{I}(v)$, start times $s(u) \leq s(v)$, and period vectors $\boldsymbol{p}(u)$, $\boldsymbol{p}(v)$. Both operations satisfy the weak lexicographical property, i.e.,* $\mathrm{wlex}(\boldsymbol{p}(u), \boldsymbol{I}(u))$ *and* $\mathrm{wlex}(\boldsymbol{p}(v), \boldsymbol{I}(v))$ *hold. Let $\boldsymbol{d} \in \mathbb{Z}^{\delta(u)}$ be a vector that is defined by (5.11) and that satisfies (5.13) for all $k = 0, \ldots, \delta(u) - 1$. Furthermore, we assume that $p_k(u) = p_k(v)$ for all $k = 0, \ldots, \delta - 1$. Then the schedule for these operations is compact-sequentialisable, and can be implemented by the program of Figure 5.13.*

Proof. For dimensions $k = 0, \ldots, \delta - 1$ we follow the proof for equal periods. For dimensions $k \geq \delta$ the conditions for disjoint lifetimes are satisfied. □

The above conditions can be checked in $\mathcal{O}(\delta(u)^2)$ time. A generalisation for more operations is not straightforward as the composition of two operations can in general not easily be modelled as one operation, with a start time and period vector.

5.3 Discussion

Compact-sequentialisable schedules allow for an *efficient* implementation, in the sense that the number of operation labels is minimal, i.e., equal to the number of operations. Such a schedule may result in a large number of guard evaluations, but a trade-off can be made between guard evaluations and operation labels. In this chapter we have given sufficient conditions for compact-sequentialisability. For each individual operation we demand weak lexicographical executions. The only conditions for two operations that generalise to more operations are the disjoint time conditions and the conditions for equal periods. The conditions for equal periods can be checked in polynomial time if all periods are divisible.

In general, we may assume that a schedule has equal period vectors for all operations as we may insert missing periods using an iterator bound zero. Furthermore, if all operations have equal period vectors, we can increase every period to

```
for $i_0 := 0 \ldots J_0 \rightarrow$
    $\ldots$
        for $i_{\delta-1} := 0 \ldots J_{\delta-1} \rightarrow$
            for $i_\delta := 0 \ldots I_\delta(u) \rightarrow$
                $\ldots$
                    for $i_{\delta(u)-1} := 0 \ldots I_{\delta(u)-1}(u) \rightarrow$
                        if $B_u \rightarrow u(i_0, \ldots, i_{\delta(u)-1})$ fi
                    rof
                $\ldots$
            rof;
            for $i_\delta := 0 \ldots I_\delta(v) \rightarrow$
                $\ldots$
                    for $i_{\delta(v)-1} := 0 \ldots I_{\delta(v)-1}(v) \rightarrow$
                        if $B_v \rightarrow v(i_0 - d_0, \ldots, i_{\delta-1} - d_{\delta-1}, i_\delta, \ldots, i_{\delta(v)-1})$ fi
                    rof
                $\ldots$
            rof
        rof
    $\ldots$
rof
```

with

$B_u = i_0 \leq I_0(u) \ \wedge \ \ldots \ \wedge \ i_{\delta-1} \leq I_{\delta-1}(u)$

$B_v = 0 \leq i_0 - d_0 \leq I_0(v) \ \wedge \ \ldots \ \wedge \ 0 \leq i_{\delta-1} - d_{\delta-1} \leq I_{\delta-1}(v)$

$J_l = \max(I_l(u), d_l + I_l(v))$, for $l = 0, \ldots, \delta - 1$

Figure 5.13. Implementation of a schedule that satisfies the *gap* conditions.

the next power of two. Hence, without losing optimal solutions we may assume that all operations have equal period vectors, all of which elements are powers of two. Checking the sufficient conditions for compact-sequentialisability can then be performed in polynomial time.

6

Cost Calculation

As is pointed out in Chapter 4, it is difficult to efficiently compute the number of cache misses for an execution of a schedule. In this chapter we develop an algorithm for estimating the number of cache misses for a given program graph, schedule, and cache. The objective is to find an efficient algorithm for computing upper bounds on the number of cache misses, thus providing upper bounds on the time required for the execution of a schedule. In Section 6.1 we decompose the cache cost computation problem by introducing two functions, being a *reuse length* function and a *filling* function. These functions give some insight on how to construct an algorithm for an efficient and effective estimation of the number of cache misses. Reuse length is the subject of Sections 6.2 and 6.3. Determining the filling is the subject of Section 6.4. The cache cost computation algorithm that is proposed in Section 6.5 has been implemented and experimental results are shown in Section 6.6.

6.1 Decomposition

In Chapter 3 we categorise cache misses for LRU caches as follows. A compulsory miss occurs for a port execution $e \in \mathcal{E}$ if it is the first to access block address $a_b(e)$. An expiration miss occurs if too many block addresses have been accessed since the previous access to block address $a_b(e)$. Formally, a port execution e causes

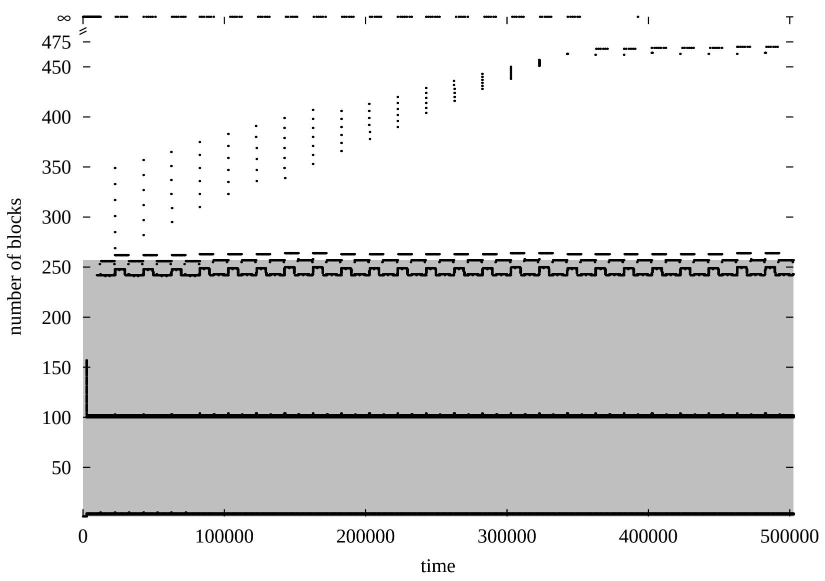


Figure 6.1. A visualisation of cache misses for the 50×50 matrix multiplication example of Figure 3.1 For each port execution the number of different blocks accessed since the previous access to the same block is denoted with a dot. All dots at the height denoted with ∞ represent port executions that access a block address for the first time. The grey area in the graph has a height of 256, the cache size. All dots within the grey area represent cache hits; all dots above this area represent cache misses.

a compulsory miss if $M(e) = \emptyset$ as introduced in (3.1), and an expiration miss if $M(e) \neq \emptyset \wedge |S(e)| > s_s$, as introduced in (3.2). Hence, we are interested in computing the number of different blocks $|S(e)|$ that are accessed between e and its predecessor that accessed the same block $a_b(e)$.

In Figure 6.1 we have visualised all hits and misses for the 50×50 matrix multiplication example of Figure 3.1. Here we plot a dot at position $(t(e), |S(e)|)$ for each port execution $e \in \mathcal{E}$. For a port execution e that accesses a block address first, a dot has been plotted at $(t(e), \infty)$. The grey area in the graph has a height equal to the cache size, in this case 256. By definition all dots within the grey area represent cache hits, and all dots above the area represent cache misses.

For this example we used a time assignment with start times $s(l) = 0$, $s(m) = 2500$, and period vectors $\boldsymbol{p}(l) = \begin{bmatrix}50 & 1\end{bmatrix}^{\mathrm{T}}$, $\boldsymbol{p}(m) = \begin{bmatrix}1 & 2500 & 50\end{bmatrix}^{\mathrm{T}}$. The address assignment for the three arrays is given by address offsets $o(X) = 0$, $o(Y) = 2504$, $o(Z) = 5008$ and coefficient vectors $\boldsymbol{c}(X) = \boldsymbol{c}(Y) = \begin{bmatrix}50 & 1\end{bmatrix}^{\mathrm{T}}$, $\boldsymbol{c}(Z) = \begin{bmatrix}50 & 1 & 0\end{bmatrix}^{\mathrm{T}}$.

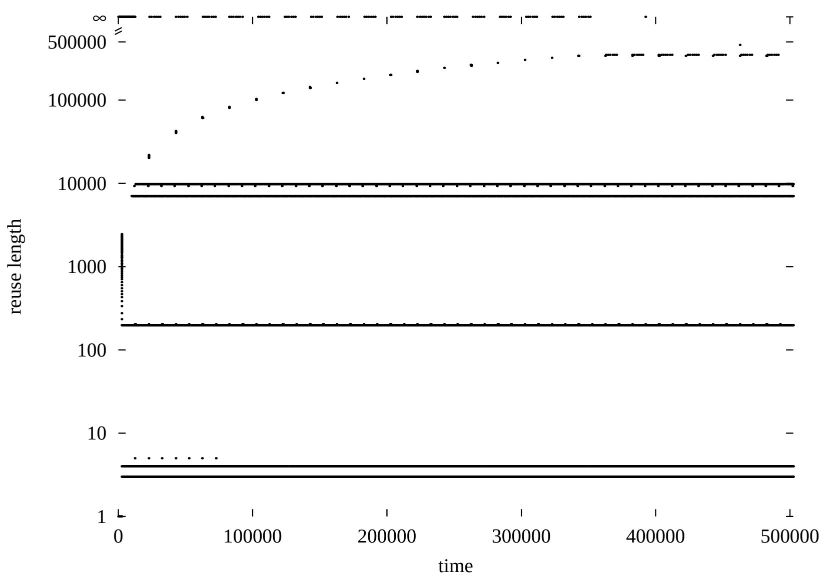


Figure 6.2. Reuse length for each port execution for the 50 × 50 matrix multiplication example of Figure 3.1.

Furthermore, we use a fully set-associative cache of 256 blocks with a block size of 16 bytes.

When computing a single point in the graph of Figure 6.1, the previous access to the same block is determined. After that the number of blocks accessed by all other ports in this interval is counted. In general it will prove difficult to find the previous access to a block as is discussed later on in this chapter. Furthermore, the contributions to the number of distinct block addresses of *all* ports have to be taken into account to compute $|S(e)|$.

As there are as many points in the graph as there are port executions for a given schedule, computing them all separately cannot be done within running time polynomial in the size of an instance of CCCP, i.e., a program graph, a schedule, and a cache. Hence, we are not interested in finding all individual points, but rather in finding a compact representation for them.

In order to tackle this problem, we split it into two separate problems. First we concentrate on finding the previous port execution that accesses the same block address as a function of execution $e \in \mathcal{E}(p)$ for each port $p \in P$. Second, we determine the so-called filling of the cache at the time of this port execution e.

One way or another we have to determine *reuse* for each port execution, i.e., to determine the previous access to the same block address. Whether a port execution

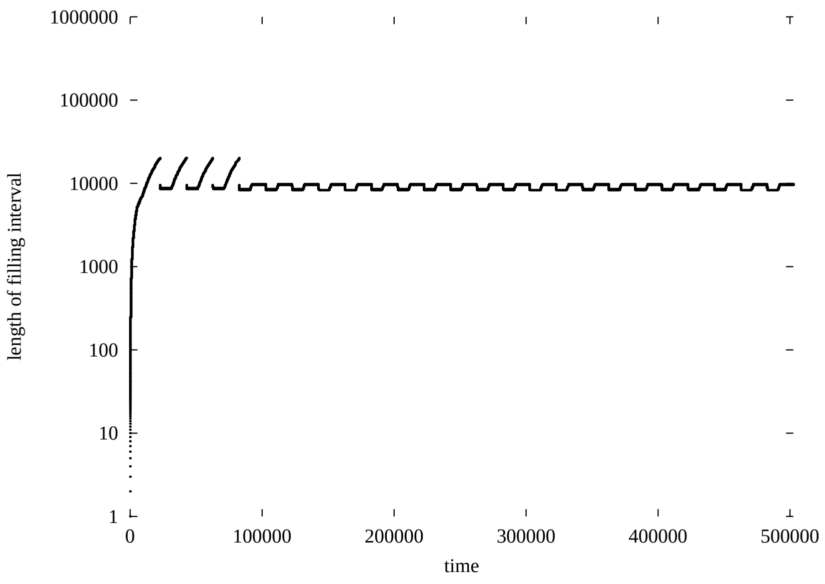


Figure 6.3. Length of filling interval for each port execution for the 50×50 matrix multiplication example of Figure 3.1.

causes a miss or a hit is determined by the number of different block addresses that are referenced between the previous reference of a block address and the current one.

Rather than determining the number of different block addresses between the previous and current block access, however, we are going to determine how much *time* has elapsed between them, which is reflected in the reuse length function, and from how long ago all accessed block addresses are still in the cache, which is reflected in the filling function. Based on these two functions, which can be approximated rather efficiently, we can determine whether we have a cache hit or a cache miss.

Formally, we define reuse length $r(e) \in \mathbb{Z}_\infty$ for a port execution $e \in \mathcal{E}$ by

$$r(e) = \min\{\, t(e) - t(f) \mid f \in \mathcal{E} \wedge t(f) < t(e) \wedge a_\mathrm{b}(f) = a_\mathrm{b}(e)\,\}.$$

For a port execution $e \in \mathcal{E}$, the reuse length is the time between the previous access to the block address that e accesses and e itself. If no previous access exists, the reuse length is the minimum over an empty domain, i.e., the reuse length is infinite. The reuse length for all executions of the matrix multiplication example of Figure 3.1 are depicted in Figure 6.2.

The filling interval for a port execution $e \in \mathcal{E}$ is the maximum interval ending at time $t(e)$ in which at most s_{s} different blocks are accessed. The filling interval $\{t(e)-f(e),\dots,t(e)\}$ for port execution $e \in \mathcal{E}$ is defined by its length

$$f(e) = \max\{\, t(e) - t(f) \mid f \in \mathcal{E} \;\wedge\; R(t(f),e) \leq s_{\mathrm{s}} \,\},$$

where $R(j,e) \in \mathbb{Z}$ is the number of different blocks in the same set as $a(e)$ that are accessed in the time interval $\{j,\dots,t(e)\}$, or formally

$$R(j,e) = |\{\, a_{\mathrm{b}}(f) \mid f \in \mathcal{E} \;\wedge\; s(a(e)) = s(a(f)) \;\wedge\; j \leq t(f) \leq t(e) \,\}|.$$

For a fully set-associative cache the length of the filling interval is

$$f(e) = \max\{\, t(e) - t(f) \mid f \in \mathcal{E} \;\wedge\; |\{\, a_{\mathrm{b}}(g) \mid g \in \mathcal{E} \;\wedge\; t(f) \leq t(g) \leq t(e) \,\}| \leq c_{\mathrm{s}} \,\}.$$

The length of the filling interval for all executions of the matrix multiplication example of Figure 3.1 are depicted in Figure 6.3.

In Figure 6.4 we show for an example that a miss can indeed be formulated in terms of the reuse length $r(e)$ and the length of the filling interval $f(e)$ for all port executions $e \in \mathcal{E}$. Before proving that the proposed decomposition is correct, we need an additional result.

Lemma 6.1. *$M(e) \neq \emptyset \Rightarrow r(e) = t(e) - m(e)$ for all port executions $e \in \mathcal{E}$.*

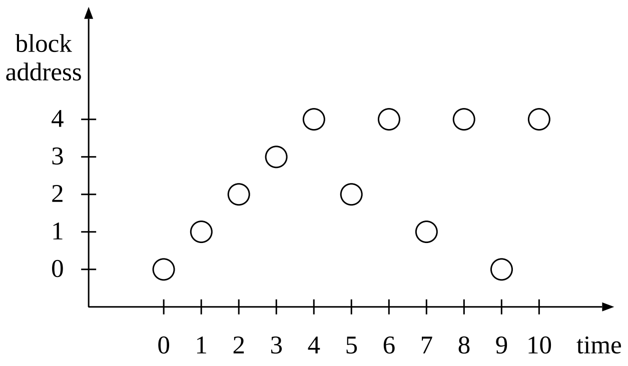
Proof.

$$\begin{array}{ll}
 & t(e) - m(e) \\
= & \quad \{\text{ definition of } m(e) \ \} \\
 & t(e) - \max\ \{\, t(f) \mid f \in M(e) \,\} \\
= & \quad \{\ M(e) \neq \emptyset \ \} \\
 & \min\ \{\, t(e) - t(f) \mid f \in M(e) \,\} \\
= & \quad \{\text{ definition of } M(e) \ \} \\
 & \min\{\, t(e) - t(f) \mid f \in \mathcal{E} \;\wedge\; a_{\mathrm{b}}(f) = a_{\mathrm{b}}(e) \;\wedge\; t(f) < t(e) \,\} \\
= & \quad \{\text{ definition of } r(e) \ \} \\
 & r(e)
\end{array}$$

□

Theorem 6.1. *A cache miss for a port execution $e \in \mathcal{E}$ occurs if and only if $r(e) > f(e)$.*

Proof. The proof is by case analysis. First, we assume $M(e) = \emptyset$ for port execution $e \in \mathcal{E}$. By definition, e generates a compulsory cache miss, and hence we have to prove that $r(e) > f(e)$. The reuse length $r(e)$ is infinite as no port execution $f \in \mathcal{E}$ exists that executes before $t(e)$ and that accesses the same block address as e. As positive values for $f(e)$ are finite, we have that $r(e) > f(e)$.



(a)

time $t(e)$	cache content	block $a_b(e)$	miss?	reuse $r(e)$	filling $f(e)$	$r(e) > f(e)$
0	empty	0	yes	∞	0	true
1	0	1	yes	∞	1	true
2	0 1	2	yes	∞	2	true
3	0 1 2	3	yes	∞	3	true
4	0 1 2 3	4	yes	∞	3	true
5	1 2 3 4	2	no	3	4	false
6	1 2 3 4	4	no	2	5	false
7	1 2 3 4	1	no	6	6	false
8	1 2 3 4	4	no	2	7	false
9	1 2 3 4	0	yes	9	5	true
10	0 1 2 4	4	no	2	6	false

(b)

Figure 6.4. An example showing two equivalent descriptions of a cache miss. The cache is fully set-associative of size $c_s = 4$ and it has block size $b_s = 1$. In (a) each circle represents an execution of a port. At the left side in (b) the cache content *before* the corresponding port execution and the corresponding address determine whether or not a miss occurs. At the right side a miss is characterised by means of the reuse length $r(e)$ and length of the filling interval $f(e)$.

Next, for $M(e) \neq \emptyset$ we know that a port execution $e' \in \mathcal{E}$ exists with $t(e') = m(e)$. Now, $ex(e) \Leftrightarrow r(e) > f(e)$ remains to be shown. If we assume that a port execution $g \in \mathcal{E}$ exists with $R(t(g), e) \leq s_s$, we have that

$$
\begin{aligned}
& r(e) > f(e) \\
\equiv \quad & \{ \text{ Lemma 6.1 and definition of } f(e) \ \} \\
& t(e) - m(e) > \max\{ t(e) - t(f) \mid f \in \mathcal{E} \wedge R(t(f), e) \leq s_s \} \\
\equiv \quad & \{ \ \exists_{g \in \mathcal{E}} \ R(t(g), e) \leq s_s \text{ and } m(e) = t(e') \ \} \\
& t(e') < \min\{ t(f) \mid f \in \mathcal{E} \wedge R(t(f), e) \leq s_s \} \\
\equiv \quad & \{ \ e' \in \mathcal{E}, \exists_x \ R(x, e) \leq s_s, \text{ and } \forall_x \ (R(x, e) > s_s \Rightarrow \forall_{y<x} \ R(y, e) > s_s) \ \} \\
& R(t(e'), e) > s_s \\
\equiv \quad & \{ \ t(e') = m(e) \ \} \\
& R(m(e), e) > s_s \\
\equiv \quad & \{ \ |S(e)| = R(m(e), e) \ \} \\
& ex(e).
\end{aligned}
$$

If we find that for all $g \in \mathcal{E}$ we have $R(t(g), e) > s_s$, then $f(e) = -\infty$, so $t(e) - m(e) > f(e)$. In particular, we then also know that $R(t(e'), e) > s_s$, or equivalently, $R(m(e), e) > s_s$, i.e., $ex(e)$ holds. □

As a result of the previous theorem, we have obtained a decomposition for the problem of determining whether a cache miss occurs for an individual port execution. However, the cache cost computation problem is the problem of determining the total number of cache misses for *all* port executions of a schedule.

In the next sections we first look at the computation of the reuse length for ports instead of individual port executions by expressing the reuse length for every port as a function of its iterator vector, at the complexity of this computation, and at estimating the reuse length. After that we address the computation of the length of the filling interval.

6.2 Reuse graphs

In the previous section we have reasoned about reuse for individual port executions. As the number of port executions is in general not bounded by a polynomial in the size of an instance $\mathcal{I}_{\text{cccp}}$ of CCCP, we attempt to characterise reuse for ports in terms of their iterator vector. In general, ports reuse data from more than one port. Hence, for an accurate computation or estimation of the reuse length, we have to find out from which ports data are reused. To that end we introduce the notion of a *reuse graph*, which we first explain by means of an example.

Suppose that we have three operations with one port each, being p, q, and r, with respective schedules that are given in Figure 6.5(a). Here the definition of a

```
port $p$:
for $i_0 := 0 \ldots 9$ period $1 \rightarrow$
    $\langle\ i_0\ \rangle$ start at 0

port $q$:
for $i_0 := 0 \ldots 4$ period $1 \rightarrow$
    $\langle\ i_0 + 10\ \rangle$ start at 10

port $r$:
for $i_0 := 0 \ldots 14$ period $1 \rightarrow$
    $\langle\ i_0\ \rangle$ start at 20
```

(a)

$$\begin{tikzcd}
 & p \arrow[dl, "\{(p, \boldsymbol{i}) \mid 0 \leq i_0 \leq 9\ \}"'] & \\
\bot & & r \arrow[ul, "\{(r, \boldsymbol{i}) \mid 0 \leq i_0 \leq 9\ \}"'] \arrow[dl, "\{(r, \boldsymbol{i}) \mid 10 \leq i_0 \leq 14\ \}"] \\
 & q \arrow[ul, "\{(q, \boldsymbol{i}) \mid 0 \leq i_0 \leq 4\ \}"] &
\end{tikzcd}$$

(b)

Figure 6.5. Example of a reuse graph. In (a) we give the definitions of the ports. The edges of the reuse graph in (b) are annotated with port executions that reuse data. For example, all executions of port r with $0 \leq i_0 \leq 9$ reuse data from port p. All executions of port r with $10 \leq i_0 \leq 14$ reuse from port q.

port has been extended with its schedule by giving the respective periods for all dimensions and its start time. The address assignment is specified by the address expression as a linear expression in the iterators. All such address expressions are surrounded by *address brackets* $\langle\ \ \rangle$. For the cache we choose a block size $b_\ell = 1$. The first ten executions of port r reuse data that were most recently used by port p, and the remaining five executions reuse data from port q. In Figure 6.5(b) this is depicted by a reuse graph. Here, the nodes correspond to the ports p, q, and r. Node $\bot$ is a special node that can be thought of as a port with exactly one execution that takes place before any other port execution, and that accesses all block addresses. By inserting this node in the reuse graph, reuse is always defined.

Formally, a reuse graph is defined as follows.

Definition 6.1 (reuse graph). For a program graph $\mathcal{G}$, a schedule σ, and a cache $\mathcal{C}$, a reuse graph is a 3-tuple $\mathcal{R} = (U, A, I)$. The set of vertices $U = P \cup \{\perp\}$ consists of the set of ports of the program graph and a special port $\perp$. The set of directed edges of the reuse graph is denoted by $A \subseteq U \times U$. Each edge $a = (p, q) \in A$ is labelled with a non-empty subset of the executions $I(a) \subseteq \mathcal{E}(p)$ of port p. There are three additional constraints on a reuse graph.

- For every edge $a = (p,q) \in A$ and all executions in $I(a)$ of port p, there must exist reuse from port q, i.e.,
$$q = \perp \vee \forall_{(p,\boldsymbol{i}) \in I(a)} \exists_{\boldsymbol{j} \in \mathcal{I}(q)}\ a_{\mathrm{b}}(p,\boldsymbol{i}) = a_{\mathrm{b}}(q,\boldsymbol{j}) \wedge t(q,\boldsymbol{j}) < t(p,\boldsymbol{i}).$$
- The labels $I(a)$ of the outgoing edges $a = (p,q)$ for each $p \in P$ must be a partition of all executions $\mathcal{E}(p)$ of p.
- Port $\perp$ has no outgoing edges.

The reuse length for port execution $(p,\boldsymbol{i}) \in \mathcal{E}$ induced by this graph is the minimum reuse length that can be found on $a = (p,q)$ with $(p,\boldsymbol{i}) \in I(a)$, or infinite if $q = \perp$, i.e.,

$$r_{\mathcal{R}}(p,\boldsymbol{i}) = \min\{\, t(p,\boldsymbol{i}) - t(q,\boldsymbol{j}) \mid q \neq \perp \wedge \boldsymbol{j} \in \mathcal{I}(q) \wedge a_{\mathrm{b}}(p,\boldsymbol{i}) = a_{\mathrm{b}}(q,\boldsymbol{j}) \wedge t(q,\boldsymbol{j}) < t(p,\boldsymbol{i}) \,\}.$$

□

For each combination of a program graph, schedule, and cache, a reuse graph exists. For example, the reuse graph with edges $A = \{\, (p,\perp) \mid p \in P \}$ and $I(a) = \mathcal{E}(p)$ for each edge $a = (p,\perp)$ meets all constraints. This graph, however, has rather poor quality as all reuse lengths induced by the graph are infinite. Using the induced reuse length of this graph to compute the number of cache misses results in a cache miss for each port execution.

However, not all overestimations of the reuse length have to result in cache misses. We say that a reuse graph is called *weakly optimal* if

$$r(e) > f(e) \Leftrightarrow r_{\mathcal{R}}(e) > f(e),$$

for all port executions $e \in \mathcal{E}$.

A reuse graph is called *optimal* if for every port execution $e \in \mathcal{E}$ with reuse length $r(e)$, this reuse length is induced by the reuse graph, i.e., $r(e) = r_{\mathcal{R}}(e)$. Such a graph always exists, but does not have to be unique as a block address may be accessed by two different ports at the same time.

For establishing the exact reuse length for every port execution, we must establish an optimal reuse graph. We show that the problem of constructing such a graph cannot be done in polynomial time unless P = NP.

Definition 6.2 (optimal reuse graph construction problem (ORGCP)).

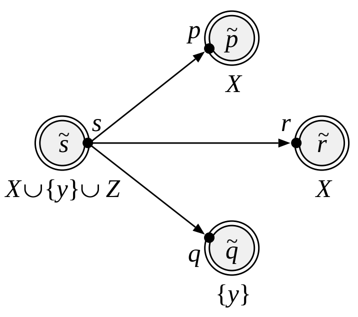


Figure 6.6. The created program graph of the complexity proof of ORGCP. Below every operation we give the set of block addresses that are accessed by the corresponding port.

Given a program graph $\mathcal{G} = (V, R, W, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, a schedule $\sigma = (\tau, \mu)$, and a cache $\mathcal{C}$, construct an optimal reuse graph. □

Theorem 6.2. *ORGCP cannot be solved in polynomial time unless P = NP.*

Proof. For the proof we use a reduction from SLE. Let an instance $\mathcal{I}_{\text{sle}}$ of SLE be given by a matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$, a vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$, and a vector $\boldsymbol{I} \in \mathbb{N}^{\delta}$. We construct an instance of ORGCP as follows. For the cache we choose block size $b_s = 1$. Other cache parameters are not used in the proof. Furthermore, we choose a program graph $\mathcal{G} = (V, R, W, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, which is depicted in Figure 6.6, with

- four operations $V = \{\tilde{p}, \tilde{q}, \tilde{r}, \tilde{s}\}$,
- operation read ports $R(\tilde{p}) = \{p\}$, $R(\tilde{q}) = \{q\}$, $R(\tilde{r}) = \{r\}$, $R(\tilde{s}) = \varnothing$,
- operation write ports $W(\tilde{p}) = \varnothing$, $W(\tilde{q}) = \varnothing$, $W(\tilde{r}) = \varnothing$, $W(\tilde{s}) = \{s\}$,
- iterator bound vectors $\boldsymbol{I}(\tilde{p}) = \boldsymbol{I}$, $\boldsymbol{I}(\tilde{q}) = []$, $\boldsymbol{I}(\tilde{r}) = \boldsymbol{I}$, $\boldsymbol{I}(\tilde{s}) = \boldsymbol{w} - \boldsymbol{v}$,
- data dependencies $E = \{(s,p), (s,q), (s,r)\}$,
- index matrices $\boldsymbol{A}(p) = \boldsymbol{A}$, $\boldsymbol{A}(q) = []$, $\boldsymbol{A}(r) = \boldsymbol{A}$, $\boldsymbol{A}(s) = \mathbf{I}$, and
- index offset vectors $\boldsymbol{b}(p) = \mathbf{0}$, $\boldsymbol{b}(q) = \boldsymbol{b}$, $\boldsymbol{b}(r) = \mathbf{0}$, $\boldsymbol{b}(s) = \boldsymbol{v}$,

where $\mathbf{I}$ is the $\delta \times \delta$ identity matrix, and where vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{Z}^{\delta}$ have been chosen such that $\boldsymbol{v} \leq \boldsymbol{z} \leq \boldsymbol{w}$ holds for all $\boldsymbol{z} \in \{\boldsymbol{A}\boldsymbol{i} \mid \boldsymbol{i} \in \mathbb{Z}^{\delta} \wedge \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{I}\} \cup \{\boldsymbol{b}\}$. In this way, port s produces all data consumed by ports p, q, and r. We choose a feasible time assignment for $\mathcal{G}$ with first all executions of $\tilde{s}$, followed by the executions of $\tilde{p}$. After that the only execution of $\tilde{q}$ takes place, followed by the executions of $\tilde{r}$. For the address assignment we choose a unique address assignment as given in Theorem 3.1. In Figure 6.6 each operation has been annotated with the set of block addresses accessed by its port. The set of block addresses accessed by port

p is denoted by X. As port r visits exactly the same array elements as port p, port r also accesses block addresses X. Next, the one block address accessed by port q is denoted by y. We observe that port s accesses block address y and block addresses X by construction of its index matrix and its index offset vector. Apart from these, it accesses an additional set of block addresses, which is denoted by Z. When constructing a reuse graph for the given program graph and schedule, the question must be answered whether or not block address y is visited by port p, i.e., whether $y \in X$. If the answer to this question is positive then Figure 6.7(a) depicts the optimal reuse graph. If block address y is not visited by port p, then Figure 6.7(b) is the optimal choice.

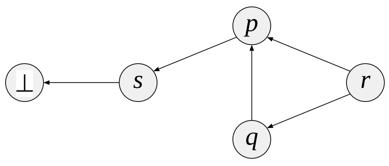
Giving an answer to the question whether $y \in X$, however, is as difficult as giving an answer to our original SLE instance, which is shown by the following derivation.

$$\begin{array}{ll}
 & y \in X \\
\equiv & \quad \{ \text{ cache block size } b_\text{s} = 1 \} \\
 & a(q,[\,]) \in \{ a(p,\boldsymbol{i}) \mid \boldsymbol{i} \in \mathbb{Z}^\delta \wedge \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I} \} \\
\equiv & \quad \{ \mu \text{ is a unique address assignment, Theorem 3.1 } \} \\
 & \boldsymbol{n}(q,[\,]) \in \{ \boldsymbol{n}(p,\boldsymbol{i}) \mid \boldsymbol{i} \in \mathbb{Z}^\delta \wedge \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I} \} \\
\equiv & \\
 & \boldsymbol{b} \in \{ \boldsymbol{A}\boldsymbol{i} \mid \boldsymbol{i} \in \mathbb{Z}^\delta \wedge \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I} \} \\
\equiv & \\
 & \exists_{\boldsymbol{i} \in \mathbb{Z}^\delta} \; \boldsymbol{0} \le \boldsymbol{i} \le \boldsymbol{I} \wedge \boldsymbol{A}\boldsymbol{i} = \boldsymbol{b}
\end{array}$$

Hence, $y \in X$ if and only if $\mathcal{I}_\text{sle}$ has a solution. So, if it is possible to solve the optimal reuse graph construction problem in polynomial time, we can solve SLE in polynomial time, implying P = NP. □

As an optimal reuse graph is difficult to find, an approximation of the reuse graph is introduced. The goal of the scheduling problem of Definition 3.10 is to minimise the number of cache misses. As we want to use an approximation of the reuse length in order to determine whether or not a cache miss occurs, we want to make sure that we do not count misses as hits. A cache miss occurs if the reuse length $r(e)$ is greater than the filling function $f(e)$. Hence, in order to get an upper bound on the number of cache misses we have to derive a lower bound on the reuse length.

Definition 6.3 (approximate reuse graph). Given a program graph $\mathcal{G}$, a schedule σ, and a cache $\mathcal{C}$, an approximate reuse graph is a 5-tuple $\mathcal{R} = (U, A, I, \boldsymbol{x}, y)$, where $U = P \cup \{\bot\}$ is the set of vertices of the graph and $A \subseteq U \times U$ is the set of edges of the graph. For every directed edge $a = (p,q) \in A$, an *approximate reuse length*


(a) $y \in X$

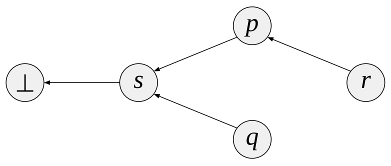

(b) $y \notin X$

Figure 6.7. Reuse graphs for the complexity proof of ORGCP. In (a) the reuse graph is depicted for the case that block address y accessed by port q is also accessed by port p. In (b) the reuse graph is depicted for the case that y is not accessed by p.

vector $\boldsymbol{x}(a) \in \mathbb{Z}^{\delta(p)}$ and an *approximate reuse length offset* $y(a) \in \mathbb{Z}_\infty$ are given for all executions $I(a) \subseteq \mathcal{E}$ of port p. The following constraints must be satisfied for an approximate reuse graph.

- For every edge $a = (p,q) \in A$, either $q = \perp$, or the reuse length must be overestimated by the approximate reuse length vector and approximate reuse length offset, i.e.,

  $$\exists_{\boldsymbol{j} \in \mathcal{I}(q)}\, a_\text{b}(p,\boldsymbol{i}) = a_\text{b}(q,\boldsymbol{j}) \;\wedge\; t(q,\boldsymbol{j}) < t(p,\boldsymbol{i}) \le t(q,\boldsymbol{j}) + \boldsymbol{x}^\text{T}(a)\boldsymbol{i} + y(a),$$

  for all executions $(p,\boldsymbol{i}) \in I(a)$ of port p.
- The executions $I(a)$ of the outgoing edges $a = (p,q)$ of port $p \in P$ must be a partition of all executions $\mathcal{E}(p)$ of port p.
- Port $\perp$ has no outgoing edges.

□

Here, we aim at finding linear approximations of the reuse length. There are two reasons for choosing a linear approximation. First, as all times at which ports execute and all addresses that ports access are linear expressions in the iterators, it is likely that reuse length can also be expressed as a linear expression in the iterators. Second, by choosing a linear expression we get a compact representation of reuse. These approximations are denoted by the *approximate reuse length* function $r_\mathcal{R}$. We say that port execution $(p,\boldsymbol{i})$ has approximate reuse length

$$r_\mathcal{R}(p,\boldsymbol{i}) = \boldsymbol{x}^\text{T}(a)\boldsymbol{i} + y(a)$$

for all $\boldsymbol{i} \in I(a)$ with $a = (p,q) \in A$.

In the same way as we defined optimal reuse graphs and weakly optimal reuse graphs, we introduce *optimal approximate reuse graphs* and *weakly optimal approximate reuse graphs*. An approximate reuse graph is called *optimal* if the approximate reuse length is equal to the exact reuse length for all port executions, i.e., $r_{\mathcal{R}}(e) = r(e)$ for all port executions $e \in \mathcal{E}$. We say that an approximate reuse graph is *weakly optimal* if

$$r(e) > f(e) \Leftrightarrow r_{\mathcal{R}}(e) > f(e),$$

for all port executions $e \in \mathcal{E}$.

Quality of an approximate reuse graph can be measured in several ways. Differences between the approximate reuse length and the exact reuse length that are too large may result in hits counted as misses, leading to an overestimation of the number of cache misses. Hence, the sum of differences between reuse length and approximate reuse length for all executions of ports is an indication for the quality of a reuse graph.

However, not all reuse lengths that are too large result in an erroneous count of the number of cache misses. Hence, if we assume that we can compute the filling function f exactly, the difference between the number of cache misses and the estimated number of cache misses using the approximate reuse length and filling f also measures the quality of the reuse graph. In Section 6.6 we come back to these quality measures.

Next, we present some heuristics that can be used for the construction of an approximate reuse graph.

6.3 Reuse heuristics

Two port executions $e, f \in \mathcal{E}$ are said to exhibit reuse if they both access the same block address and do not execute simultaneously, i.e., $a_{\mathrm{b}}(e) = a_{\mathrm{b}}(f)$ and $t(e) < t(f)$. If executions e and f take place at the same time, then we must assume that no reuse takes place. This is a direct result of the definition that we chose for a cache miss, namely if two port executions take place at the same time, one may not assume that one takes place before the other.

We need efficient means for computing and storing the reuse length $r(e)$ for all $e \in \mathcal{E}$. In order to do so, we must be able to characterise all executions $\boldsymbol{i}$ and $\boldsymbol{j}$ of respective ports p of array A and port q of array B that access the same block address, i.e., $a_{\mathrm{b}}(p, \boldsymbol{i}) = a_{\mathrm{b}}(q, \boldsymbol{j})$, which is equivalent to

$$(\boldsymbol{c}^{\mathrm{T}}(A)(\boldsymbol{A}(p)\boldsymbol{i} + \boldsymbol{b}(p)) + o(A)) \operatorname{div} b_{\mathrm{s}} = (\boldsymbol{c}^{\mathrm{T}}(B)(\boldsymbol{A}(q)\boldsymbol{j} + \boldsymbol{b}(q)) + o(B)) \operatorname{div} b_{\mathrm{s}}, \quad (6.1)$$

and can be rewritten into

$$(\boldsymbol{c}^{\mathrm{T}}(p)\boldsymbol{i} + o(p)) \operatorname{div} b_{\mathrm{s}} = (\boldsymbol{c}^{\mathrm{T}}(q)\boldsymbol{j} + o(q)) \operatorname{div} b_{\mathrm{s}}, \quad (6.2)$$

for *port address coefficient vector* $\boldsymbol{c}(p) = \boldsymbol{A}^{\mathrm{T}}(p)\boldsymbol{c}(A)$ and *port address offset* $o(p) = \boldsymbol{c}^{\mathrm{T}}(A)\boldsymbol{b}(p) + o(A)$.

We want to determine as many pairs $(p, \boldsymbol{i})$ and $(q, \boldsymbol{j})$ that exhibit reuse as pos-

$$\begin{tikzcd}
\text{self temporal} \arrow[r] \arrow[d] & \text{self spatial} \arrow[d] \\
\text{group temporal} \arrow[r] & \text{group spatial}
\end{tikzcd}$$

Figure 6.8. Relationship between different kinds of reuse. The arrows denote that one kind of reuse is a special case of another kind of reuse. The most general kind of reuse is group spatial reuse.

sible. To this end, (6.1) can be simplified if we make some assumptions. For that, we use the classification of reuse that was also used by Wolf & Lam [1991]. Two executions of the same port are said to possess *self temporal reuse* if they access the same address. *Self spatial reuse* occurs if two executions of the same port access the same cache block. Two executions of two ports in general may also access the same address. *Group temporal reuse* occurs if the executions access the same address, and *group spatial reuse* if the executions access the same block address. By definition, temporal reuse is a special case of spatial reuse, and self reuse is a special case of group reuse, as depicted in the diagram of Figure 6.8.

Expression (6.1) describes the most general case, i.e., group spatial reuse. For group temporal reuse it is simplified to

$$\boldsymbol{c}^{\mathrm{T}}(A)\,(\boldsymbol{A}(p)\boldsymbol{i}+\boldsymbol{b}(p))+o(A)=\boldsymbol{c}^{\mathrm{T}}(B)\,(\boldsymbol{A}(q)\boldsymbol{j}+\boldsymbol{b}(q))+o(B),$$

or, if p and q belong to the same array ($p,q \in A$), to

$$\boldsymbol{c}^{\mathrm{T}}(A)\,(\boldsymbol{A}(p)\boldsymbol{i}-\boldsymbol{A}(q)\boldsymbol{j}+\boldsymbol{b}(p)-\boldsymbol{b}(q))=0.$$

For self temporal reuse, the expression can be reduced even further, namely to

$$\boldsymbol{c}^{\mathrm{T}}(A)\boldsymbol{A}(p)(\boldsymbol{i}-\boldsymbol{j})=0.$$

In the remainder of this section we give heuristic rules for finding self temporal reuse, group temporal reuse, and self spatial reuse. A heuristic for group spatial reuse is the topic of the Section 6.3.5, and consists of a combination of the heuristics in this section. We start by discussing temporal reuse in general.

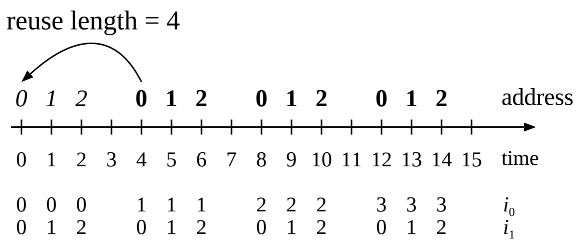
6.3.1 Introduction to temporal reuse

In this section we give some tools that we use to find self temporal reuse and group temporal reuse in the following sections. In Figure 6.9 we give an example of (self) temporal reuse, where, for example, executions $\begin{bmatrix}1 & 0\end{bmatrix}^{\mathrm{T}}$ and $\begin{bmatrix}2 & 1\end{bmatrix}^{\mathrm{T}}$ reuse data from executions $\begin{bmatrix}0 & 0\end{bmatrix}^{\mathrm{T}}$ and $\begin{bmatrix}1 & 1\end{bmatrix}^{\mathrm{T}}$, respectively.

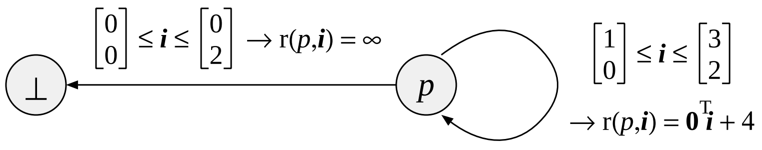
We want to express temporal reuse compactly. As all times at which ports

```
for $i_0 := 0\ldots3$ period $4 \rightarrow$
   for $i_1 := 0\ldots2$ period $1 \rightarrow$
      $\langle\ i_1\ \rangle$ start at $0$
```

(a)



(b)



$\begin{bmatrix}0\\0\end{bmatrix} \le \boldsymbol{i} \le \begin{bmatrix}0\\2\end{bmatrix} \rightarrow \mathrm{r}(p,\boldsymbol{i}) = \infty$
$\perp$
p
$\begin{bmatrix}1\\0\end{bmatrix} \le \boldsymbol{i} \le \begin{bmatrix}3\\2\end{bmatrix}$
$\rightarrow \mathrm{r}(p,\boldsymbol{i}) = \mathbf{0}^{\mathrm{T}}\boldsymbol{i} + 4$

(c)

Figure 6.9. Example of (self) temporal reuse. The definition of the port is given in (a). For all executions of the port that reuse data, the addresses have been typeset bold in (b). All executions of the port for which no reuse can be found have been typeset italic. The reuse graph for this port p is given in (c).

execute and all addresses that ports access are linear expressions in the iterators, we aim at expressing reuse length as a linear expression in the iterators. We say that executions $J \subseteq \mathcal{I}(q)$ of port q reuse from port p with *reuse matrix* $\boldsymbol{R} \in \mathbb{Z}^{\delta(p)\times\delta(q)}$ and *reuse vector* $\boldsymbol{r} \in \mathbb{Z}^{\delta(p)}$ if

$$\boldsymbol{i} = \boldsymbol{R}\boldsymbol{j} + \boldsymbol{r} \;\Rightarrow\; a(p,\boldsymbol{i}) = a(q,\boldsymbol{j}), \tag{6.3}$$

for all $\boldsymbol{i} \in \mathcal{I}(p), \boldsymbol{j} \in J \subseteq \mathcal{I}(q)$, and $t(p,\boldsymbol{i}) < t(q,\boldsymbol{j})$.

Assuming that $\boldsymbol{i} = \boldsymbol{R}\boldsymbol{j} + \boldsymbol{r}$, the next computation derives sufficient conditions for $\boldsymbol{R}$ and $\boldsymbol{r}$ to satisfy (6.3). The three steps in this derivation that strengthen the expression are discussed after this derivation.

$$\begin{aligned}
&\quad a(p,\boldsymbol{i}) = a(q,\boldsymbol{j})\\
&\equiv\\
&\quad \boldsymbol{c}^{\mathrm{T}}(A)(\boldsymbol{A}(p)\boldsymbol{i} + \boldsymbol{b}(p)) + o(A) = \boldsymbol{c}^{\mathrm{T}}(B)(\boldsymbol{A}(q)\boldsymbol{j} + \boldsymbol{b}(q)) + o(B)\\
&\Leftarrow \qquad \{ \text{(i): assume } A = B \;\}\\
&\quad \boldsymbol{c}^{\mathrm{T}}(A)((\boldsymbol{A}(p)\boldsymbol{R} - \boldsymbol{A}(q))\boldsymbol{j} + \boldsymbol{b}(p) - \boldsymbol{b}(q) + \boldsymbol{A}(p)\boldsymbol{r}) = 0\\
&\Leftarrow \qquad \{ \text{(ii): assume independent indices} \}\\
&\quad (\boldsymbol{A}(p)\boldsymbol{R} - \boldsymbol{A}(q))\boldsymbol{j} + \boldsymbol{b}(p) - \boldsymbol{b}(q) + \boldsymbol{A}(p)\boldsymbol{r} = \boldsymbol{0}\\
&\Leftarrow \qquad \{ \text{(iii): assume independent iterators} \}\\
&\quad \boldsymbol{A}(p)\boldsymbol{R} = \boldsymbol{A}(q) \;\wedge\; \boldsymbol{A}(p)\boldsymbol{r} = \boldsymbol{b}(q) - \boldsymbol{b}(p)
\end{aligned}$$

In this derivation we made three important assumptions. First of all, in Step (i), we assume that both ports belong to the same array. We return to this topic later on in this section. In Step (ii) we ignore reuse that results from the chosen address assignment. The decision in Step (iii) ignores possible reuse generated within the columns of $\boldsymbol{A}$.

Another way of obtaining sufficient conditions for reuse is found by deriving

$$\begin{aligned}
&\quad a(p,\boldsymbol{i}) = a(q,\boldsymbol{j})\\
&\equiv\\
&\quad \boldsymbol{c}^{\mathrm{T}}(p)(\boldsymbol{R}\boldsymbol{j} + \boldsymbol{r}) + o(p) = \boldsymbol{c}^{\mathrm{T}}(q)\boldsymbol{j} + o(q)\\
&\equiv\\
&\quad (\boldsymbol{c}^{\mathrm{T}}(p)\boldsymbol{R} - \boldsymbol{c}^{\mathrm{T}}(q))\boldsymbol{j} + \boldsymbol{c}^{\mathrm{T}}(p)\boldsymbol{r} + o(p) - o(q) = 0\\
&\Leftarrow \qquad \{ \text{assume independent iterators} \}\\
&\quad \boldsymbol{c}^{\mathrm{T}}(p)\boldsymbol{R} = \boldsymbol{c}^{\mathrm{T}}(q) \;\wedge\; \boldsymbol{c}^{\mathrm{T}}(p)\boldsymbol{r} = o(q) - o(p).
\end{aligned}$$

These conditions on $\boldsymbol{R}$ and $\boldsymbol{r}$ are weaker than the previous conditions as there is only one equation for each column of $\boldsymbol{R}$. Therefore, these weak conditions are less guiding towards a good reuse matrix, a good reuse vector, and a good set J of executions of port q that exhibit reuse, in the sense of resulting in a good estimation of the reuse length for as many executions (of port q) as possible.

Independent of the above derivation of conditions for the reuse matrix and reuse vector, the set of executions J of port q that exhibit reuse is constrained by

$$\mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(p) \tag{6.4}$$

$$t(p,\boldsymbol{i}) < t(q,\boldsymbol{j}) \tag{6.5}$$

$$\mathbf{0} \leq \boldsymbol{j} \leq \boldsymbol{I}(q), \tag{6.6}$$

for all integer vectors $\boldsymbol{j} \in J$ and $\boldsymbol{i} = \boldsymbol{R}\boldsymbol{j} + \boldsymbol{r}$.

These conditions follow from (6.3) in the following way. Condition (6.4) takes care that all executions $\boldsymbol{j}$ of port q reuse data from existing executions $\boldsymbol{i} = \boldsymbol{R}\boldsymbol{j} + \boldsymbol{r}$ of port p. Condition (6.6) follows naturally as $\boldsymbol{j}$ must be an execution of port q. For reuse to take place, execution $\boldsymbol{j}$ of q must take place after the associated execution $\boldsymbol{i}$ of port p, which is stated in (6.5). The difference in time between executions $(p, \boldsymbol{R}\boldsymbol{j} + \boldsymbol{r})$ and $(q, \boldsymbol{j})$ is an upper bound on the reuse length $r(q, \boldsymbol{j})$. As this estimation of the reuse length is used more often in this section, we introduce the following notation.

Definition 6.4 (estimated reuse length). The estimated reuse length for ports p, q, reuse matrix $\boldsymbol{R}$, reuse vector $\boldsymbol{r}$, and executions $J \subseteq \mathcal{I}(q)$ of port q is defined as the time between executions $\boldsymbol{j} \in J$ of port q and executions $\boldsymbol{i} = \boldsymbol{R}\boldsymbol{j} + \boldsymbol{r}$ of port p, and denoted by $\rho(q, \boldsymbol{j}) = t(q, \boldsymbol{j}) - t(p, \boldsymbol{R}\boldsymbol{j} + \boldsymbol{r})$. □

With this definition we reformulate conditions (6.4), (6.5), and (6.6) as

$$\mathbf{0} \leq \boldsymbol{R}\boldsymbol{j} + \boldsymbol{r} \leq \boldsymbol{I}(p) \tag{6.7}$$

$$\rho(q, \boldsymbol{j}) > 0 \tag{6.8}$$

$$\mathbf{0} \leq \boldsymbol{j} \leq \boldsymbol{I}(q), \tag{6.9}$$

for all integer vectors $\boldsymbol{j} \in J$.

Next, we consider self temporal reuse and group temporal reuse in more detail.

6.3.2 Self temporal reuse

If two different executions of the same port access the same address, we say that they exhibit self temporal reuse. For self temporal reuse we choose the weaker conditions on reuse matrix and reuse vector, i.e.,

$$\boldsymbol{c}^{\mathrm{T}}(p)\boldsymbol{R} = \boldsymbol{c}^{\mathrm{T}}(p) \,\wedge\, \boldsymbol{c}^{\mathrm{T}}(p)\boldsymbol{r} = 0.$$

In order to limit the number of port executions for which we find reuse as little as possible, we choose $\boldsymbol{R} = \mathbf{I}$. In this way only the reuse vector $\boldsymbol{r}$ limits the search space because of (6.7) and (6.8). Next, we consider self temporal reuse for two cases.

Self temporal reuse carried by one dimension

We say that self temporal reuse is carried by dimension l if executions $\boldsymbol{i}$ and $\boldsymbol{j}$ of a port p access the same address and differ only in index l. We recall that self temporal reuse exists if and only if $a(p,\boldsymbol{i}) = a(p,\boldsymbol{j})$, which is equivalent to saying that $\boldsymbol{c}^{\mathrm{T}}(p)(\boldsymbol{j}-\boldsymbol{i}) = 0$. Hence $c_l(p)$ must be zero. We already established that the reuse matrix $\boldsymbol{R} = \mathbf{I}$. Thus, we must choose a reuse vector $\boldsymbol{r} \neq \mathbf{0}$ and set $J \subseteq \mathbb{Z}^{\delta}$ such that

$$\mathbf{0} \le \boldsymbol{j}+\boldsymbol{r} \le \boldsymbol{I}(p) \tag{6.10}$$

$$\rho(p,\boldsymbol{j}) > 0 \tag{6.11}$$

$$\mathbf{0} \le \boldsymbol{j} \le \boldsymbol{I}(p), \tag{6.12}$$

for all executions $\boldsymbol{j} \in J$, where

$$\rho(p,\boldsymbol{j}) = t(p,\boldsymbol{j}) - t(p,\boldsymbol{j}+\boldsymbol{r}) = s(p) + \boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{j} - s(p) - \boldsymbol{p}^{\mathrm{T}}(p)(\boldsymbol{j}+\boldsymbol{r}) = -\boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{r}. \tag{6.13}$$

First of all we observe that we may choose $r_m = 0$ for all $m \neq l$, hence only r_l remains to be defined. Without loss of generality we may assume that the corresponding period $p_l(p)$ is positive. Because of (6.11) and (6.13), r_l needs to be negative. The only restriction that (6.10) and (6.12) impose on the size of J with respect to the value of r_l is $-r_l \le j_l \le I_l(p)$. As we want to choose $\rho(p,\boldsymbol{j})$ as small as possible and J as large as possible, $r_l = -1$ is the best choice.

Using the above for the example in Figure 6.9 we find reuse for dimension 0. The reuse vector is $\boldsymbol{r} = \begin{bmatrix}-1 & 0\end{bmatrix}^{\mathrm{T}}$ and estimated reuse length $\rho(p,\boldsymbol{i}) = -\boldsymbol{p}^{\mathrm{T}}\boldsymbol{r} = 4$.

Self temporal reuse carried by two dimensions

We say that self temporal reuse is carried by dimensions l and m if executions $\boldsymbol{i}$ and $\boldsymbol{j}$ access the same address and are different only in dimensions l and m.

In Figure 6.10 we give an example of self temporal reuse that is carried by dimensions 0 and 1. In this figure, all executions that have been typeset bold have reuse with a reuse vector $\boldsymbol{r} = \begin{bmatrix}-2 & 1\end{bmatrix}^{\mathrm{T}}$, with an estimated reuse length

$$\rho(p,\boldsymbol{j}) = -\boldsymbol{p}^{\mathrm{T}}\boldsymbol{r} = -\begin{bmatrix}6 & 1\end{bmatrix}\begin{bmatrix}-2\\ 1\end{bmatrix} = 11.$$

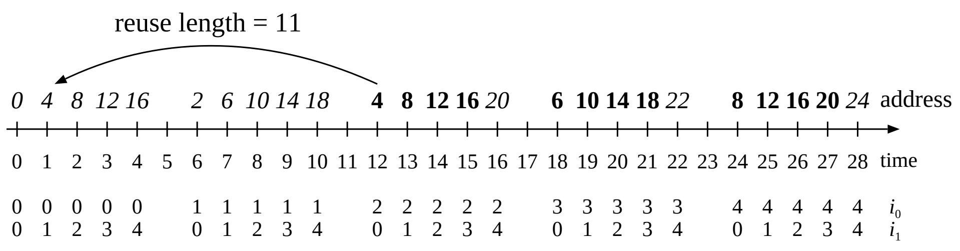
In general, for self temporal reuse we must find r_l and r_m such that

$$c_l r_l + c_m r_m = 0 \,\wedge\, r_l \neq 0 \,\wedge\, r_m \neq 0. \tag{6.14}$$

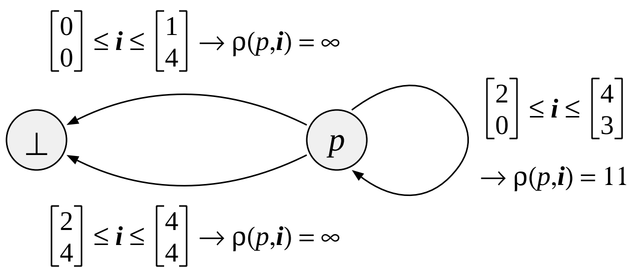
We may assume $c_l \neq 0$ and $c_m \neq 0$ as the cases $c_l = 0$ and $c_m = 0$ are covered by self temporal reuse that is carried by one dimension. As c_l and c_m may contain

```
for $i_0 := 0 \ldots 4$ period 6 $\rightarrow$
    for $i_1 := 0 \ldots 4$ period 1 $\rightarrow$
        $\langle\ 2i_0 + 4i_1\ \rangle$ start at 0
```

(a)


reuse length = 11
0 4 8 12 16 2 6 10 14 18 4 8 12 16 20 6 10 14 18 22 8 12 16 20 24 address
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 time
0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 i_0
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 i_1

(b)


$\begin{bmatrix}0\\0\end{bmatrix} \le \boldsymbol{i} \le \begin{bmatrix}1\\4\end{bmatrix} \rightarrow \rho(p,\boldsymbol{i}) = \infty$
$\perp$
p
$\begin{bmatrix}2\\0\end{bmatrix} \le \boldsymbol{i} \le \begin{bmatrix}4\\3\end{bmatrix}$
$\rightarrow \rho(p,\boldsymbol{i}) = 11$
$\begin{bmatrix}2\\4\end{bmatrix} \le \boldsymbol{i} \le \begin{bmatrix}4\\4\end{bmatrix} \rightarrow \rho(p,\boldsymbol{i}) = \infty$

(c)

Figure 6.10. Example of self temporal reuse for a port *p* generated by two dimensions. The definition (a) of port *p* includes its schedule, which is shown in (b). The reuse graph of port *p* is given in (c).

common factors (6.14) is restated as

$$\frac{c_l}{g} r_l + \frac{c_m}{g} r_m = 0 \ \text{ for } \ g = \gcd(|c_l|, |c_m|) \ \wedge \ r_l, r_m \neq 0.$$

Now, $r_l = k\frac{c_m}{g}$ and $r_m = -k\frac{c_l}{g}$ for $k \in \mathbb{Z} \setminus \{0\}$ characterise all integer solutions of (6.14). The search space is again limited to one parameter, k. The set of executions J is limited by

$$\begin{gathered} \mathbf{0} \leq \boldsymbol{j} + \boldsymbol{r} \leq \boldsymbol{I}(p) \\ \rho(p, \boldsymbol{j}) = -\boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{r} > 0 \\ \mathbf{0} \leq \boldsymbol{j} \leq \boldsymbol{I}(p), \end{gathered}$$

for all $\boldsymbol{j} \in J$. The restricting dimensions with inequalities in which k appears are l and m, giving

$$\begin{gathered} 0 \leq j_l + r_l \leq I_l(p) \\ 0 \leq j_m + r_m \leq I_m(p) \\ k \cdot (c_l(p)p_m(p) - c_m(p)p_l(p)) > 0 \\ 0 \leq j_l \leq I_l(p) \\ 0 \leq j_m \leq I_m(p), \end{gathered}$$

for all $\boldsymbol{j} \in J$. Hence, we must choose k, and thereby J, in such a way that for all $\boldsymbol{j} \in J$

$$\begin{gathered} 0 \leq j_n \leq I_n(p) \text{ if } n \notin \{l, m\} \\ (-r_l)^+ \leq j_l \leq I_l(p) + (-r_l)^- \\ (-r_m)^+ \leq j_m \leq I_m(p) + (-r_m)^- \\ k \cdot (c_l(p)p_m(p) - c_m(p)p_l(p)) > 0, \end{gathered} \tag{6.15}$$

and in such a way that the reuse length is as small as possible and $|J|$ is as large as possible. This is achieved by choosing $|k| = 1$ with the same sign as $q(p)p_m(p) - c_m(p)p_l(p)$. The situation with $c_l(p)p_m(p) - c_m(p)p_l(p) = 0$ is not an interesting case to look at for this heuristic as no reuse exists, i.e., this case would imply reuse length $\rho(p, \boldsymbol{j}) = 0$ for all $\boldsymbol{j} \in J$.

Using the above for the example of Figure 6.10 and choosing $l = 0$ and $m = 1$ results in $r_0 = 2k$, $r_1 = -k$, and $k(c_0(p)p_1(p) - c_1(p)p_0(p)) = k(2 \cdot 1 - 4 \cdot 6) = -22k$. The constraints in (6.15) say that k must be negative, and that reuse with estimated reuse length $-\boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{r} = -(6 \cdot 2k + 1 \cdot -k) = -11k$ is found for executions $\boldsymbol{j} \in J \subseteq \mathbb{Z}^2$ with $0 - 2k \leq j_0 \leq 4$ and $0 \leq j_1 \leq 4 + k$. Choosing $k = -1$ results in the shortest estimated reuse length $\rho(p, \boldsymbol{j}) = 11$ for the largest set of executions J.

6.3.3 Group temporal reuse

Group temporal reuse exists between executions of two (different) ports p and q that access the same address. The problem of deciding whether two such executions exist, is NP-complete. Hence, we give heuristic rules for finding reuse. Here, we are guided by situations that occur often in video algorithms. As mentioned earlier we aim at finding a reuse matrix $\boldsymbol{R}$ and reuse vector $\boldsymbol{r}$, such that $a(p, \boldsymbol{Rj} + \boldsymbol{r}) = a(q, \boldsymbol{j})$, and a set J of executions that satisfy (6.7), (6.8), and (6.9). Below we give two methods for obtaining a valid reuse matrix and reuse vector. The first method can only be applied for index matrices that have the property that each iterator appears in at most one index. The second method uses a property of index matrices that is often found in video algorithms, namely that the number of iterators that appears in each index is limited.

Each iterator appears in at most one index

The first method for finding group temporal reuse uses the property that an iterator rarely appears in more than one index, which is often the case in video algorithms. Assuming that ports p and q belong to the same array and hence $\alpha(p) = \alpha(q)$, we showed that finding a reuse matrix $\boldsymbol{R}$ and reuse vector $\boldsymbol{r}$ that satisfy

$$\boldsymbol{A}(q) = \boldsymbol{A}(p)\boldsymbol{R} \wedge \boldsymbol{A}(p)\boldsymbol{r} = \boldsymbol{b}(q) - \boldsymbol{b}(p) \tag{6.16}$$

is a first step to find executions $\boldsymbol{i} = \boldsymbol{Rj} + \boldsymbol{r}$ that access the same address. Below we discuss choices for the reuse matrix $\boldsymbol{R}$ and the reuse vector $\boldsymbol{r}$ separately. For this discussion we assume that every iterator appears in exactly one index. If an iterator does not appear in any index, it generates self temporal reuse, which can be dealt with using the methods described earlier in this chapter. The dimension in which iterator $k = 0, \ldots, \delta(p) - 1$ for port p appears is denoted by $\pi_k(p)$.

Rewriting the first term of (6.16), we must have

$$A_{lm}(q) = \sum_{k=0}^{\delta(p)-1} A_{lk}(p) R_{km} \text{ for all } l = 0, \ldots, \alpha - 1$$

for each iterator dimension $m = 0, \ldots, \delta(q) - 1$. Here we use $\alpha = \alpha(p) = \alpha(q)$. For dimension m we assumed exactly one non-zero in column A_m, so we get

$$A_{\pi_m(q)m}(q) = \sum_{k=0}^{\delta(p)-1} A_{\pi_m(q)k}(p) R_{km}$$

$$0 = \sum_{k=0}^{\delta(p)-1} A_{lk}(p) R_{km} \text{ for all } l = 0, \ldots, \alpha - 1 \wedge l \neq \pi_m(q).$$

Both sums contain several $A_{lk} = 0$, but we know that only those k for which $\pi_k(p) =$

l yield $A_{lk} \neq 0$, and they can therefore be rewritten into

$$A_{\pi_m(q)m}(q) = \sum_{\substack{0 \le k < \delta(p) \\ \pi_k(p) = \pi_m(q)}} A_{\pi_m(q)k}(p) R_{km} \tag{6.17}$$

$$0 = \sum_{\substack{k=0 \\ \pi_k(p)=l}}^{\delta(p)-1} A_{lk}(p) R_{km} \text{ for all } l = 0, \ldots, \alpha - 1 \wedge l \neq \pi_m(q). \tag{6.18}$$

Both (6.17) and (6.18) restrict the values that R_{km} can take, the former for $\pi_k(p) = \pi_m(q)$, the latter for $\pi_k(p) \neq \pi_m(q)$. As these two domains are disjoint, we may choose values for R_{km} according to (6.17) and (6.18) independently.

An obviously correct choice for the R_{km} values that appear in (6.18) is $R_{km} = 0$. A reason for wanting small $|R_{km}|$ is the following. The number of executions that have reuse is bounded by

$$\mathbf{0} \le \boldsymbol{Rj} + \boldsymbol{r} \le \boldsymbol{I}(p).$$

Choosing values for $R_{km} = 0$ limits the number of executions $\boldsymbol{j}$ with reuse as little as possible.

The restrictions in (6.17) can be formulated as a special case of the problem PUC [Verhaegh, 1995], where we aim at finding small solutions, i.e., solutions with small $|R_{km}|$.

Reuse vector $\boldsymbol{r}$ must satisfy

$$\sum_{\substack{m=0 \\ l=\pi_m(p)}}^{\delta(p)-1} A_{lm}(p) r_m = b_l(q) - b_l(p) \tag{6.19}$$

for all indices $l = 0, \ldots, \alpha - 1$. Again, these independent equations are special cases of the problem PUC.

In the example of Figure 6.11 equations (6.17) and (6.18) result in $5 = 1 \cdot R_{00}$ and $1 = 1 \cdot R_{01}$. Reuse vector $\boldsymbol{r}$ must satisfy $1 \cdot r_0 = 0 - 0$ because of (6.19). So, we find that reuse exists with reuse matrix $\boldsymbol{R} = \begin{bmatrix} 5 & 1 \end{bmatrix}$, reuse vector $\boldsymbol{r} = \begin{bmatrix} 0 \end{bmatrix}$. Because of (6.7), (6.8), and (6.9) we find that all executions $\boldsymbol{j} \in J = \mathcal{I}(q)$ of port q have group temporal reuse with estimated reuse length

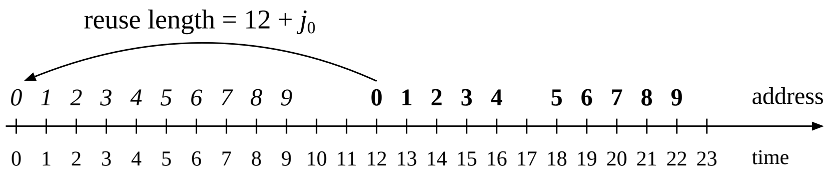
$$\begin{aligned} \rho(q, \boldsymbol{j}) &= t(q, \boldsymbol{j}) - t(p, \boldsymbol{Rj} + \boldsymbol{r}) \\ &= s(q) + \boldsymbol{p}^{\mathrm{T}}(q)\boldsymbol{j} - s(p) - \boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{Rj} - \boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{r} \\ &= 12 + 6j_0 + j_1 - 0 - 5j_0 - j_1 - 0 = 12 + j_0. \end{aligned}$$

Unfortunately, there are situations in which reuse exists but no reuse matrix and vector can be found. For example, assume an array X with unique address assignment and two ports $p, q \in X$ as defined below.

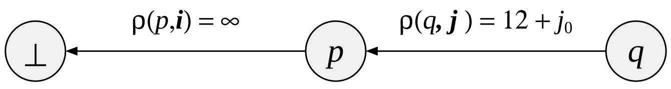
```
port $p$:
for $i_0 := 0 \ldots 9$ period $1 \rightarrow$
    $X[i_0]$ start at 0

port $q$:
for $j_0 := 0 \ldots 1$ period $6 \rightarrow$
    for $j_1 := 0 \ldots 4$ period $1 \rightarrow$
        $X[5j_0 + j_1]$ start at 12
```

(a)

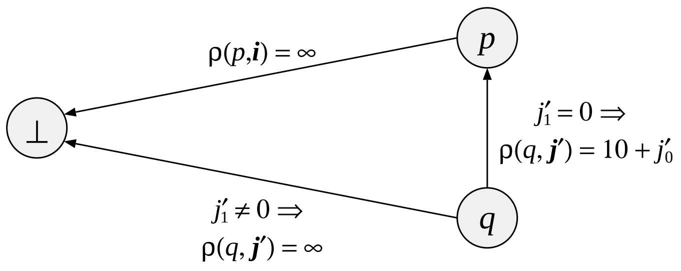


(b)


$\rho(p,\boldsymbol{i}) = \infty$
$\rho(q,\boldsymbol{j}) = 12 + j_0$
$\perp$
p
q

(c)

Figure 6.11. Example of group temporal reuse. We assume address coefficient $c_0(X) = 1$ and address offset $o(X) = 0$. Here, each execution $\boldsymbol{j} \in \mathcal{I}(q)$ of port q reuses data that was previously accessed by port p with reuse length $12 + j_0$. The resulting reuse graph is depicted in (c).



$\rho(p,\boldsymbol{i}) = \infty$
p
$\perp$
$j'_1 = 0 \Rightarrow$
$\rho(q,\boldsymbol{j'}) = 10 + j'_0$
$j'_1 \neq 0 \Rightarrow$
$\rho(q,\boldsymbol{j'}) = \infty$
q

Figure 6.12. A dimension split may be necessary in order to find reuse. In this case port q is obtained by splitting the only dimension of port q. Even iterations of port q are represented by executions $j'_0 = 0$ of port q'. They have reuse with approximate reuse length $10 + j'_0$. For the odd iterations of port q, represented by $j'_0 \neq 0$ of port q', no reuse exists.

```
port $p$:
for $i_0 := 0 \ldots 3$ period $1 \rightarrow$
   $X[4i_0]$ start at 0

port $q$:
for $j_0 := 0 \ldots 5$ period $1 \rightarrow$
   $X[2j_0]$ start at 10
```

Clearly, reuse exists between p and q for all *even* executions j_0 of q. However, no integer solution exists to (6.17). We apply techniques called *dimension splitting* [Verhaegh, 1995] and *domain splitting* to separate the odd and even executions j_0 of port q. By performing a dimension split, of which the formal definition follows, we obtain the following port q'.

```
port $q'$:
for $j'_0 := 0 \ldots 2$ period $2 \rightarrow$
   for $j'_1 := 0 \ldots 1$ period $1 \rightarrow$
      $X[4j'_0 + 2j'_1]$ start at 10
```

The approximate reuse graph for this example is given in Figure 6.12. In this graph, the executions in the inner dimension of port q' have been split by means of a domain split, resulting in the lower edge with executions that correspond to odd executions j_0 of port q, and the edge from q to p with even executions j_0 of port q.

More formally, given a port q, a dimension split is specified by a number $k \in \{0,\ldots,\delta(q)-1\}$ and a number $I \in \{1,\ldots,I_k(q)-1\}$ for which $(I+1)|(I_k(q)+1)$. The dimension split results in a new port q' with

- $\delta(q') = \delta(q) + 1$, $\alpha(q') = \alpha(q) + 1$,

- $I_l(q') = \begin{cases} I_l(q) & \text{for } l = 0,\ldots,k-1 \\ (I_k(q)+1)/(I+1)-1 & \text{for } l = k \\ I & \text{for } l = k+1 \\ I_{k-1}(q) & \text{for } l = k+2,\ldots,\delta(q')-1, \end{cases}$
- $A_{\cdot l}(q') = \begin{cases} A_{\cdot l}(q)(I+1) & \text{for } l = k \\ A_{\cdot l}(q) & \text{otherwise,} \end{cases}$
- $\boldsymbol{b}(q') = \boldsymbol{b}(q)$,
- $p_l(q') = \begin{cases} p_l(q)(I+1) & \text{for } l = k \\ p_l(q) & \text{otherwise,} \end{cases}$
- $s(q') = s(q)$.

Furthermore, given a port q, a domain split is specified by a number $k \in \{0,\ldots,\delta(q)-1\}$ and a number $I \in \{1,\ldots,I_k(q)-1\}$. The dimension split results in two new ports q' and q'' with

- $\delta(q') = \delta(q'') = \delta(q)$, $\alpha(q') = \alpha(q'') = \alpha(q)$,
- $I_l(q') = \begin{cases} I & \text{for } l = k \\ I_l(q) & \text{for } l \neq k, \end{cases}$
- $I_l(q'') = \begin{cases} I_k(q) - I - 1 & \text{for } l = k \\ I_l(q) & \text{for } l \neq k, \end{cases}$
- $\boldsymbol{A}(q') = \boldsymbol{A}(q'') = \boldsymbol{A}(q)$,
- $\boldsymbol{b}(q') = \boldsymbol{b}(q)$,
- $\boldsymbol{b}(q'') = \boldsymbol{b}(q) + \boldsymbol{A}_{\cdot k}(q)(I+1)$,
- $s(q') = s(q)$, and
- $s(q'') = s(q) + p_l(q)(I+1)$.

In general, the splitting techniques can be used for every dimension m of port q for which (6.17) has no integer solution. In Figure 6.13 we propose an algorithm for finding group temporal reuse between ports p and q. In Step (ii) we perform dimension splitting for every dimension of port q as shown in the preceding example. Step (vi) creates an edge in the reuse graph for executions $\mathcal{I}(q) \setminus J$ of port q for which no reuse is found. Step (vii) creates an edge in the reuse graph for all executions J of port q that reuse data from port p with reuse length

$$t(q,\boldsymbol{j}) - t(p,\boldsymbol{Rj}+\boldsymbol{r}) = (\boldsymbol{p}^{\mathrm{T}}(q) - \boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{R})\boldsymbol{j} + s(q) - s(p) - \boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{r}.$$

i. Determine $\pi_k(p)$ such that $A_{\pi_k(p)k}(p) \neq 0$ and $A_{\pi_k(p)k}(p) = 0$ for all $k = 0, \ldots, \delta(p) - 1$. The $\pi_k(q)$ are determined analogously.

ii. For each array dimension k of q, split the corresponding dimension $\pi_k(q)$ in such a way that $A_{k\pi_k(p)}(p) \mid A_{k\pi_k(q)}(q)$.

iii. Assign R_{km} according to (6.17) and (6.18).

iv. Assign r_m according to (6.19).

v. Determine $J \subseteq \mathcal{I}(q)$ such that (6.7), (6.8), and (6.9) hold for every $\boldsymbol{j} \in J$.

vi. Create an edge $a = (q, \perp)$ in the reuse graph with $I(a) = \mathcal{I}(q) \setminus J$, $\boldsymbol{x}(a) = \boldsymbol{0}$, and $y(a) = \infty$.

vii. Create an edge $a = (q, p)$ in the reuse graph with $I(a) = J$, $\boldsymbol{x}(a) = \boldsymbol{p}(q) - \boldsymbol{R}^{\mathrm{T}}\boldsymbol{p}(p)$, and $y(a) = s(q) - s(p) - \boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{r}$.

Figure 6.13. An algorithm for determining group temporal reuse between ports p and q if each iterator appears in exactly one index.

Limited number of iterators in each index

Another method for finding group temporal reuse uses the property that the number of iterators that appear in each index is limited, which is often the case in video algorithms. Therefore, it is useful to look at the equation $\boldsymbol{A}(p)\boldsymbol{R} = \boldsymbol{A}(q)$ for one row of the matrices $\boldsymbol{A}(p)$ and $\boldsymbol{A}(q)$. Without loss of generality we may assume that for index l the first m iterators have non-zero entries in the index matrix $\boldsymbol{A}(p)$ of port p, and that the first m' iterators have non-zero entries in the index matrix $\boldsymbol{A}(q)$ for port q. Now we can write $\boldsymbol{A}(p)\boldsymbol{R} = \boldsymbol{A}(q)$ for array dimension l as

$$\begin{bmatrix} \\ A_{l0}(p)\ldots A_{lm}(p) \quad 0\ldots 0 \\ \\ \end{bmatrix} \begin{bmatrix} R_{00}\ldots R_{0m'} & 0\ldots 0 \\ \vdots & \vdots \\ R_{m0}\ldots R_{mm'} & 0\ldots 0 \\ & \end{bmatrix} = \begin{bmatrix} \\ A_{l0}(q)\ldots A_{lm'}(q) \quad 0\ldots 0 \\ \\ \end{bmatrix}. \tag{6.20}$$

We propose a heuristic for finding a solution to (6.20) based on two observations. The first observation is that if the rows $\boldsymbol{A}_{l\cdot}(p)$ and $\boldsymbol{A}_{l\cdot}(q)$ for an index $0 \leq l < \alpha$ contain the same number of non-zeroes, it is likely that reuse can be determined by relating the contributing iterators at both sides one to one. Such a relationship between the iterators of both ports is found by sorting them with respect to their absolute values.

If one iterator of port q relates to two iterators of port p, we split the corresponding dimension of q. We show this by means of the following example.

```
port $p$:
for $i_0 := 0 \ldots 3$ period $6 \rightarrow$
   for $i_1 := 0 \ldots 1$ period $1 \rightarrow$
      $X[4i_0 + i_1]$ start at 0

port $q$:
for $j_0 := 0 \ldots 15$ period $1 \rightarrow$
   $X[j_0]$ start at 30
```

In order to find reuse, we have to split the only dimension of port q in order to get

```
port $q'$:
for $j'_0 := 0 \ldots 3$ period $4 \rightarrow$
   for $j'_1 := 0 \ldots 3$ period $1 \rightarrow$
      $X[4j'_0 + j'_1]$ start at 30
```

After this split we are able to give a one-to-one relation $\nu_n(p)$ of dimensions of p to dimensions of q. Now, we can find a solution to (6.20) by choosing

$$\sum_{\substack{n=0 \\ m=\nu_n(p)}}^{\delta(p)-1} A_{ln}(p)R_{nm} = A_{lm}(q), \tag{6.21}$$

for every index $l = 0, \ldots, \alpha - 1$ of port q. This equation can be solved in the same way as (6.18). For the reuse vector $\boldsymbol{r}$ we have to find a solution to

$$\sum_{n=0}^{\delta(p)-1} A_{ln}r_n = b_l(q) - b_l(p), \tag{6.22}$$

in a similar way to (6.19).

The second observation is that finding a solution to (6.20) and hence to (6.21) is easier for cases with small m and m'.

These observations lead to the algorithm of Figure 6.14. In Step (i) we sort the rows based on the first observation and we use the second observation as a tie breaker. In Steps (ii) and (iii) we determine reuse matrix $\boldsymbol{R}$ and reuse vector $\boldsymbol{r}$. Steps (iv), (v), and (vi) create the approximate reuse graph.

The assumptions for this heuristic are strong and the method is rather crude, but in practice this heuristic has been able to find most of the group temporal reuse for the experiments that we discuss in Section 6.6.

6.3.4 Self spatial reuse

Self spatial reuse for a port p occurs if the same cache block is accessed by two or more executions of port p. Before going into details we give an example.

i. Sort the rows $\boldsymbol{A}(p)$ and $\boldsymbol{A}(q)$ in increasing order on
   - the difference in the number of non-zeroes of $A_{l\cdot}(p)$ and the number of non-zeroes of $A_{l\cdot}(q)$, and
   - in case of a tie, on the number of non-zeroes of $A_{l\cdot}(p)$.

ii. Using the order established in Step (i), perform the next steps for each row $m = 0, \ldots, \alpha - 1$.
   - Find relationship $\nu_l(p)$ for every dimension $l \in \{0, \ldots, \delta(p) - 1\}$ of port p, possibly after a dimension split.
   - Choose reuse matrix $\boldsymbol{R}$ according to (6.21).

iii. Assign $\boldsymbol{r}$ according to (6.22).

iv. Determine $J \subseteq \mathcal{I}(q)$ such that (6.7), (6.8), and (6.9) hold for every $\boldsymbol{j} \in J$.

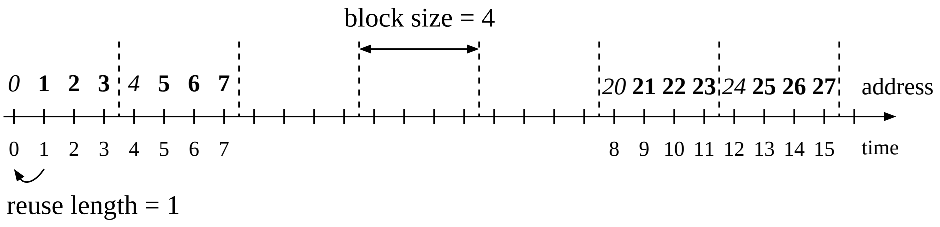
v. Create an edge $a = (q, \perp)$ in the reuse graph with $I(a) = \mathcal{I}(q) \setminus J$, $\boldsymbol{x}(a) = \mathbf{0}$, and $y(a) = \infty$.

vi. Create an edge $a = (q, p)$ in the reuse graph with $I(a) = J$, $\boldsymbol{x}(a) = \boldsymbol{p}(q) - \boldsymbol{R}^{\mathrm{T}}\boldsymbol{p}(p)$, and $y(a) = s(q) - s(p) - \boldsymbol{p}^{\mathrm{T}}(p)\boldsymbol{r}$.

Figure 6.14. An algorithm for determining group temporal reuse between ports p and q assuming that each index expression contains few iterators.

```
for $i_0 := 0 \ldots 1$ period 8 $\rightarrow$
  for $i_1 := 0 \ldots 7$ period 1 $\rightarrow$
    $\langle\ 20i_0 + i_1\ \rangle$ start at 0
```

(a)



(b)

$$\begin{tikzcd}
\perp & p \arrow[l, "i_1 \bmod 4 = 0 \rightarrow \rho(p{,}\boldsymbol{i}) = \infty"'] \arrow[loop right, "i_1 \bmod 4 \neq 0 \rightarrow \rho(p{,}\boldsymbol{i}) = 1"]
\end{tikzcd}$$

(c)

Figure 6.15. Example of a port with self spatial reuse for a cache with block size $b_s = 4$. Different from earlier examples, addresses have been taken as the unit on the axis in (b).

In Figure 6.15 we give the reuse graph for a port p that exhibits self spatial reuse. Here, all executions that are typeset bold reuse data directly from their predecessor. For example, executions $\begin{bmatrix}0 & 1\end{bmatrix}^{\mathrm{T}}$, $\begin{bmatrix}0 & 2\end{bmatrix}^{\mathrm{T}}$, and $\begin{bmatrix}0 & 3\end{bmatrix}^{\mathrm{T}}$ reuse data that were used most recently by executions $\begin{bmatrix}0 & 0\end{bmatrix}^{\mathrm{T}}$, $\begin{bmatrix}0 & 1\end{bmatrix}^{\mathrm{T}}$, and $\begin{bmatrix}0 & 2\end{bmatrix}^{\mathrm{T}}$ respectively. Hence, if we split the second dimension of the port in the first executions that access a cache block and all remaining executions by means of a dimension split followed by a domain split, we end up with

```
port $p'$:
for $i_0 := 0 \ldots 1$ period 8 $\rightarrow$
  for $i_1 := 0 \ldots 1$ period 4 $\rightarrow$
    for $i_2 := 0 \ldots 2$ period 1 $\rightarrow$
      $\langle\ 20i_0 + 4i_1 + i_2 + 1\ \rangle$ start at 1

port $p''$:
for $i_0 := 0 \ldots 1$ period 8 $\rightarrow$
  for $i_1 := 0 \ldots 1$ period 4 $\rightarrow$
    $\langle\ 20i_0 + 4i_1\ \rangle$ start at 0
```

All executions of p' have reuse length $p_0 = 1$. For none of the executions of p'' we can determine self spatial reuse. Not all self spatial reuse can be found this easily, as can be seen in the example of Figure 6.16, where not all executions with $i_1 = 0$ start on a cache block boundary. In general, we use the following algorithm, of which the steps are explained below.

---

i. Determine a dimension l that carries the reuse.

ii. Split all other dimensions ($m \neq l$) in order to compensate for address coefficients that are not an integer multiple of the block size.

iii. Split dimension l into a head, body, and tail.

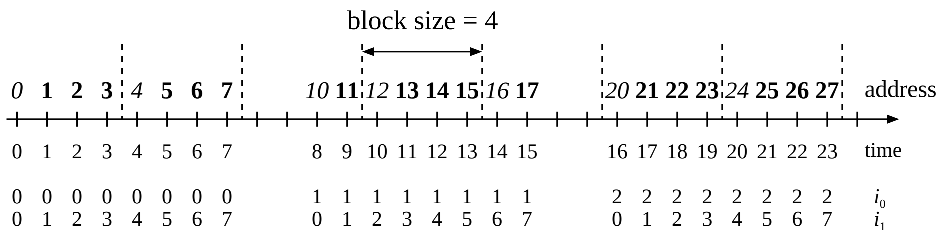
iv. Split the body in order to compensate for address coefficients that are not divisors of the block size.

v. Split the head, bodies, and tail into executions with the first access to a block and executions with subsequent accesses to the same block.

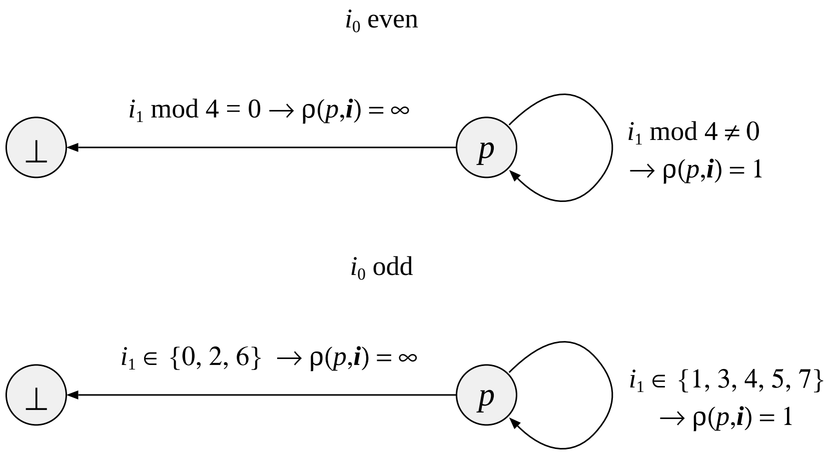
In Step (i), we want to find a dimension l that carries most reuse. By fixing the values of the other iterators, we find that on average $b_S/|c_l(p)|$ executions access the same cache block in dimension l. As we want to maximise the number of executions that exhibit reuse, we choose the dimension with smallest $|q(p)|$. For the remainder of the discussion on self spatial reuse we assume non-negative $c_m(p)$ for all dimensions $0 \leq m < \delta(p)$. For non-positive values analogous results can be

```
for $i_0 := 0 \dots 2$ period 8 $\rightarrow$
    for $i_1 := 0 \dots 7$ period 1 $\rightarrow$
        $\langle\ 10 i_0 + i_1\ \rangle$ start at 0
```

(a)



(b)


i_0 even
$i_1 \bmod 4 = 0 \rightarrow \rho(p,\boldsymbol{i}) = \infty$
$\perp$
p
$i_1 \bmod 4 \neq 0$
$\rightarrow \rho(p,\boldsymbol{i}) = 1$
i_0 odd
$i_1 \in \{0, 2, 6\} \rightarrow \rho(p,\boldsymbol{i}) = \infty$
$\perp$
p
$i_1 \in \{1, 3, 4, 5, 7\}$
$\rightarrow \rho(p,\boldsymbol{i}) = 1$

(c)

Figure 6.16. Example of a port with self spatial reuse given a cache with block size $b_s = 4$. For reasons of readability, the reuse graph (c) has been split into two parts, but should be looked at as one graph.

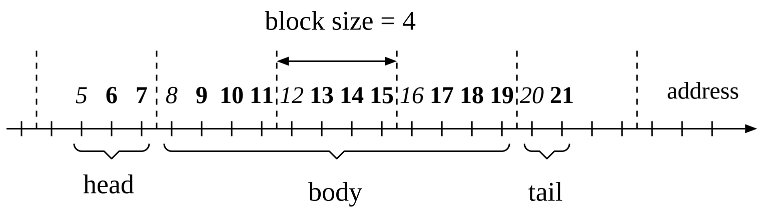


Figure 6.17. Executions of a port are split into a head, a body, and a tail.

found.

In Step (ii) we create groups of executions that have the same reuse pattern for executions in dimension l. In the example of Figure 6.16 there were two such groups: one with i_0 even and one with i_0 odd. In general the executions are split into $b_s / \gcd(b_s, |c_m(p)|)$ groups (unless $c_m(p) = 0$, in which case no split has to be performed) for all dimensions $m \neq l$. This split consists of a dimension split followed by a domain enumeration of the newly created dimension.

After having performed Step (ii) we have obtained groups of executions with the same reuse pattern in dimension l. In general, such a group can be divided into three parts: a head, a body, and a tail, as the first and last executions may produce a different reuse pattern than the other executions. In Figure 6.17 an example is given where the first three and the last two executions have a reuse pattern that is different from the reuse pattern of the executions that access addresses 8 to 19.

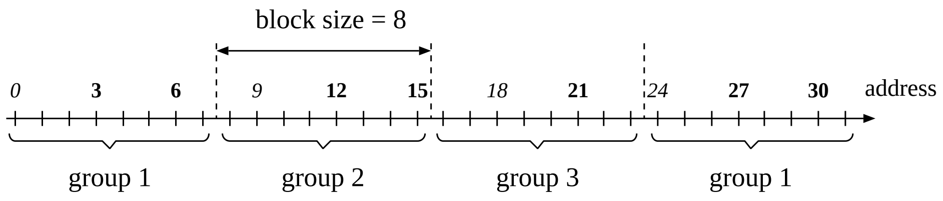
As depicted in Figure 6.18 the reuse pattern for executions may not be equal for all cache blocks that are accessed within the body. In Step (iv) groups of executions are created in such a way that for each group all first accesses to a cache block are on the same position within a cache block. The number of groups that have to be created equals $b_s / \gcd(b_s, |c_l(p)|)$.

Finally, in Step (v), all first executions to a cache block are separated from the successors in the same block. For the former ones no reuse is found, whereas for the latter ones we find reuse with reuse length $|p_l(p)|$. For example, the executions tagged *group 2* of Figure 6.18 are divided as follows, where we find no reuse for the first set of executions and we find reuse for the second set with reuse length $p_1(\tilde{p}'') = 1$.

```
port $p''$:
for $i_0 := 0 \ldots 1$ period 8 $\rightarrow$
   for $i_1 := 0 \ldots 0$ period 1 $\rightarrow$
      $\langle\ 24i_0 + 3i_1 + 9\ \rangle$ start at 0
```

```
port $p$:
for $i := 0 \ldots 23$ period 1 $\rightarrow$
    $\langle\ 3i\ \rangle$ start at 0
```

(a)



(b)

```
port $p'$:
for $i_0 := 0 \ldots 1$ period 8 $\rightarrow$
    for $i_1 := 0 \ldots 2$ period 1 $\rightarrow$
        $\langle\ 24i_0 + 3i_1\ \rangle$ start at 0

port $p''$:
for $i_0 := 0 \ldots 1$ period 8 $\rightarrow$
    for $i_1 := 0 \ldots 2$ period 1 $\rightarrow$
        $\langle\ 24i_0 + 3i_1 + 9\ \rangle$ start at 3

port $p'''$:
for $i_0 := 0 \ldots 1$ period 8 $\rightarrow$
    for $i_1 := 0 \ldots 1$ period 1 $\rightarrow$
        $\langle\ 24i_0 + 3i_1 + 18\ \rangle$ start at 6
```

(c)

Figure 6.18. Example of a port with an address coefficient that does not divide the block size of the cache. The port is split into three ports, in each of which the same reuse pattern is exhibited for all executions of the outer loop.

```
port $\tilde{p}''$:
for $i_0 := 0 \ldots 1$ period $8 \rightarrow$
   for $i_1 := 0 \ldots 1$ period $1 \rightarrow$
      $\langle\ 24i_0 + 3i_1 + 12\ \rangle$ start at 0
```

The five steps of the algorithm produce a number of sets of executions that can be written in the way we did above. As a result of this algorithm, all address coefficients of dimensions ($m \neq l$) that do not carry reuse are an integer multiple of the block size. This also means that for reuse analysis, we can forget about the exact addresses within a cache block. For reuse, the positions of addresses within a cache block are unimportant, only whether they fall in the same cache cache block determines if there is reuse.

Therefore, after having performed the five steps for self spatial reuse we can simplify the address coefficient vector and the address offset in such a way that all executions in dimension l access a uniquely chosen representative in each cache block. The choice for this representative is arbitrary. For this discussion we choose the first address in the cache block. We perform such a simplification of the address expressions in view of group spatial reuse, which we discuss in the following section. For the above example this results in the following sets of executions.

```
port $p''$:
for $i_0 := 0 \ldots 1$ period $8 \rightarrow$
   for $i_1 := 0 \ldots 0$ period $1 \rightarrow$
      $\langle\ 24i_0 + 0i_1 + 8\ \rangle$ start at 0

port $\tilde{p}''$:
for $i_0 := 0 \ldots 1$ period $8 \rightarrow$
   for $i_1 := 0 \ldots 1$ period $1 \rightarrow$
      $\langle\ 24i_0 + 0i_1 + 8\ \rangle$ start at 0
```

6.3.5 Reuse graph construction

The remaining kind of reuse that we have not yet considered is *group spatial* reuse. Group spatial reuse occurs whenever executions of two (distinct) ports access the same cache block. In general group spatial reuse is characterised by

$$(\boldsymbol{c}^{\mathrm{T}}(A)(\boldsymbol{A}(p)\boldsymbol{i}+\boldsymbol{b}(p))+o(A)) \operatorname{div} b_{\mathrm{s}} = (\boldsymbol{c}^{\mathrm{T}}(B)(\boldsymbol{A}(q)\boldsymbol{j}+\boldsymbol{b}(q))+o(B)) \operatorname{div} b_{\mathrm{s}}. \tag{6.23}$$

Solutions to this equation are hard to find as can be shown by a reduction from SLE (see Definition 4.3). Nevertheless, after we have found self spatial reuse, and have changed the address coefficient vector and address offset as described above, every execution accesses the first address within a cache block. With the changed address coefficient vector and address offset, (6.23) is simplified to the expression

i. Initialise the approximate reuse graph with edges $\{(q,\perp) \mid q \in P\}$. Choose $y(a) = \infty$ and $\boldsymbol{x}(a) = \boldsymbol{0}$ for all edges $a \in A$.

ii. For every edge $a \in A$, determine self temporal reuse for executions $I(a)$, and change the approximate reuse graph accordingly.

iii. For every edge $a \in A$, determine self spatial reuse for executions $I(a)$, and change the approximate reuse graph accordingly.

iv. For every edge $a = (q,r) \in A$ and every port p, determine group spatial reuse between executions $I(a)$ of q and executions of port p, and change the approximate reuse graph accordingly.

Figure 6.19. An algorithm to construct an approximate reuse graph.

for group temporal reuse, i.e.,

$$\boldsymbol{c}'^{\mathrm{T}}(A)(\boldsymbol{A}(p)\boldsymbol{i}+\boldsymbol{b}(p)) + o'(A) = \boldsymbol{c}'^{\mathrm{T}}(B)(\boldsymbol{A}(q)\boldsymbol{j}+\boldsymbol{b}(q)) + o'(B).$$

Hence, by first finding self spatial reuse, we can use the heuristics for group temporal reuse to find group spatial reuse. This leads to the eventual algorithm of Figure 6.19. Here we have chosen to determine self temporal reuse before self spatial reuse, as the address transformation that we have defined for self spatial reuse maps all addresses within a cache block to the same address, which would cause self temporal reuse, but only self temporal reuse of a kind that the algorithm for self spatial reuse has already found.

By applying this algorithm, different reuse lengths may be found for sets of port executions of the same port p. So, assume reuse for a port p has been found for executions I' with approximate reuse length $\boldsymbol{x}'^{\mathrm{T}}\boldsymbol{i}+y'$ and for executions I'' with approximate reuse length $\boldsymbol{x}''^{\mathrm{T}}\boldsymbol{i}+y''$. If no overlap exists between executions I' and I'', then the approximate reuse graph gets two edges corresponding to executions I' and I'' and their respective approximate reuse lengths. If overlap exists then we create three edges in the approximate reuse graph, one for executions $I' \setminus I''$ with approximate reuse length $\boldsymbol{x}'^{\mathrm{T}}\boldsymbol{i}+y'$, one for executions $I'' \setminus I'$ with approximate reuse length $\boldsymbol{x}''^{\mathrm{T}}\boldsymbol{i}+y''$, and one for executions $I' \cap I''$. How the approximate reuse length for this last set of executions must be chosen is an open issue. A correct choice that we used in the experiments of Section 6.6 is using the minimum reuse length found for any execution $i \in I' \cap I''$, i.e., $\min\{m', m''\}$ with

$$\begin{aligned} m' &= \max\{\boldsymbol{x}'^{\mathrm{T}}\boldsymbol{i}+\boldsymbol{y}' \mid \boldsymbol{i} \in I' \cap I''\} \\ m'' &= \max\{\boldsymbol{x}''^{\mathrm{T}}\boldsymbol{i}+\boldsymbol{y}'' \mid \boldsymbol{i} \in I' \cap I''\}. \end{aligned}$$

In Figure 6.20, the approximate reuse graphs are given for the two cases discussed

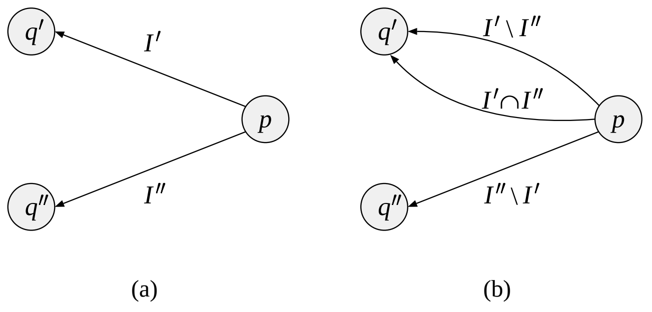


Figure 6.20. If different reuse lengths are found for two sets I' and I'' of port executions of port p, (a) gives an approximate reuse graph if I' and I'' have no overlap. Otherwise, (b) gives an approximate reuse graph given that the minimum reuse length found for any execution in $I' \cap I''$ is found in I', i.e., $m' \leq m''$.

above.

Especially for the heuristic for self spatial reuse, the number of edges in the reuse graph may become large. A worst case upper bound for this number is $6b_s^{\delta(p)}$. This number is reached for a port with all address coefficients $c_n(p)$ relatively prime to the block size b_s. The factor 6 is the result of splitting into three groups in Step (iii) and two groups in Step (iv). Note that this number may even be larger than the total number of executions of port p. In order to limit the explosion of the number of edges in the reuse graph, we limit the number of outgoing edges for every node in the reuse graph. Once this number is reached, we no longer accept that new approximations of the reuse length are added to the approximate reuse graph.

So far, we have given a characterisation of a cache miss in terms of reuse length and filling in Section 6.1. In this section we discussed heuristics for estimating the reuse length for operations by linear expressions in their iterator vectors. In the next section we present a way of estimating the filling.

6.4 Filling function

In this section we aim at finding a good estimation of the length of the filling interval $f(e)$ defined in Section 6.1 and given by

$$f(e) = \max\{\, t(e) - t(f) \mid f \in \mathcal{E} \;\wedge\; R(t(f), e) \leq s_s \,\},$$

where

$$R(j,e) = |\{\, a_b(f) \mid f \in \mathcal{E} \;\wedge\; s(a(f)) = s(a(e)) \;\wedge\; j \leq t(f) \leq t(e) \,\}|.$$

First we look at computing $f(e)$ for a fully set-associative cache and a single port execution $e \in \mathcal{E}$. As $R(j,e)$ is a monotonously descending function in $j \in \mathbb{Z}$, $f(e)$ can be found by means of binary search. Now, we need an efficient method of computing $R(j,e)$ for $j \in \mathbb{Z}$ and port execution $e \in \mathcal{E}$.

Lemma 6.2. *The number of different block addresses $R(j,e)$ that are accessed in a time interval $\{j,\dots,t(e)\}$, can be bounded from above by*

$$|\{ f \in \mathcal{E} \mid t(f) - r(f) < j \wedge j \le t(f) \le t(e) \}|.$$

Proof. Of all port executions that access a certain block address, at least one has a reuse length that extends outside the interval $\{j,\dots,t(e)\}$. If no port executions may occur simultaneously, the first access to each block address in the interval is defined uniquely, and is the only execution with a reuse length that extends outside the interval. In that case the upper bound equals $R(j,e)$ provided that the cache is fully set-associative. □

Instead of counting all executions that have reuse outside the interval $\{j,\dots,t(e)\}$, we can also count all executions that have reuse within the interval. As these executions are hits, we count a considerable number of hits as a side effect of computing the filling function for one port execution. The number of executions that have reuse within the interval $\{j,\dots,t(e)\}$ is given by

$$|\{ f \in \mathcal{E} \mid j \le t(f) \le t(e) \}| - |\{ f \in \mathcal{E} \mid j \le t(f) - r(f) \wedge j \le t(f) \le t(e) \}|,$$

which equals

$$\sum_{p \in P} |\{ \boldsymbol{i} \in \mathcal{I}(p) \mid j \le t(p,\boldsymbol{i}) \le t(e) \}| - \sum_{p \in P} |\{ \boldsymbol{i} \in \mathcal{I}(p) \mid j \le t(p,\boldsymbol{i}) - r(p,\boldsymbol{i}) \wedge j \le t(p,\boldsymbol{i}) \le t(e) \}|. \tag{6.24}$$

In case of lexicographical executions for all ports, the first term of (6.24) can be computed in $\mathcal{O}(|P|\delta)$ time, where $\delta = \max_{p \in P} \delta(p)$. The second term of (6.24) is bounded from above by using an estimation of the reuse length $\rho(p,\boldsymbol{i})$ instead of $r(p,\boldsymbol{i})$. For every edge $a = (p,q)$ in an approximate reuse graph $\mathcal{R} = (V,A,I,\boldsymbol{x},y)$, we have such an estimation of the reuse length, i.e., $\rho(p,\boldsymbol{i}) = \boldsymbol{x}^{\mathrm{T}}(a)\boldsymbol{i} + y(a)$ for all $\boldsymbol{i} \in I(a)$. Hence the second term of (6.24) is restated as

$$\sum_{a \in A} |\{ \boldsymbol{i} \in I(a) \mid j - s(p) + y(a) \le (\boldsymbol{p}^{\mathrm{T}}(p) - \boldsymbol{x}^{\mathrm{T}}(a))\boldsymbol{i} \wedge j \le t(p,\boldsymbol{i}) \le t(e) \}|.$$

If the vectors $\boldsymbol{p}(p) - \boldsymbol{x}(a)$ and $\boldsymbol{I}(p)$ have lexicographic executions, i.e., $\mathrm{lex}(\boldsymbol{p}(p) - \boldsymbol{x}(a), \boldsymbol{I}(p))$, this sum can be computed in $\mathcal{O}(|A|\delta)$ time. Otherwise we use an estimation of the reuse length that is linear in time and given by

$$\alpha \cdot t + \beta \geq \boldsymbol{x}^{\mathrm{T}}(a)\boldsymbol{i} + y(a) \quad \text{for all } \boldsymbol{i} \in I(a) \text{ and } t = t(p,\boldsymbol{i}),$$

which leads to the following lower bound for the second term of (6.24)

$$\sum_{a \in A} |\{\, \boldsymbol{i} \in I \mid j \leq t(p,\boldsymbol{i}) - \alpha t(p,\boldsymbol{i}) - \beta \,\wedge\, j \leq t(p,\boldsymbol{i}) \leq t(e) \,\}|,$$

which equals

$$\sum_{a \in A} |\{\, \boldsymbol{i} \in I \mid j + \beta \leq (1-\alpha)t(p,\boldsymbol{i}) \,\wedge\, j \leq t(p,\boldsymbol{i}) \leq t(e) \,\}|.$$

Again, based on the lexicographical executions of port p, i.e., $\mathrm{lex}(\boldsymbol{p}(p), \boldsymbol{I}(p))$, this sum can be computed in $\mathcal{O}(|A|\delta)$ time. This leaves the problem of finding an estimation of the reuse length that is linear in time.

Definition 6.5 (linear time approximation problem (LTAP)). Given are a period vector $\boldsymbol{p} \in \mathbb{N}_{+}^{\delta}$, an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_{+}^{\delta}$, and a vector $\boldsymbol{x} \in \mathbb{Z}^{\delta}$. Determine $a, b \in \mathbb{Q}$ such that

$$a\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + b \geq \boldsymbol{x}^{\mathrm{T}}\boldsymbol{i} \quad \text{for all } \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}, \tag{6.25}$$

minimising

$$\sum_{\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}} a\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + b - \boldsymbol{x}^{\mathrm{T}}\boldsymbol{i}. \tag{6.26}$$

$\square$

Theorem 6.3. *LTAP can be solved in $\mathcal{O}(\delta^2)$ time.*

Proof. The minimisation term (6.26) is rewritten into

$$\prod_{l=0}^{\delta-1} (I_l + 1) \left(\frac{1}{2}\boldsymbol{p}^{\mathrm{T}}\boldsymbol{I}a + b - \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{I} \right)$$

and therefore, minimising

$$\frac{1}{2}\boldsymbol{p}^{\mathrm{T}}\boldsymbol{I}a + b \tag{6.27}$$

is sufficient and necessary for optimality.

If we fix the value of a, a feasible (with respect to (6.25)), and minimal (with respect to (6.27)) value for b is the smallest solution of $b \geq \boldsymbol{x}^{\mathrm{T}}\boldsymbol{i} - a\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}$ for all $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}$, i.e.,

$$b = \max_{\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}} (\boldsymbol{x}^{\mathrm{T}} - a\boldsymbol{p}^{\mathrm{T}})\boldsymbol{i} = \sum_{l=0}^{\delta-1} (x_l - a p_l)^{+} I_l.$$

```
filling($z,t$)::
 $r,s,n := z,t+1,0$
 { invariant :  $R(s,t) \leq c_s \wedge n \leq |\{e \in \mathcal{E} \mid s \leq t(e) - r(e) \wedge t(e) \leq t\}| \wedge$
                $t - s \leq f(t)\}$
;do $(r+1 < s) \rightarrow$
    $h := (r+s)\,\text{div}\,2$
    ;$x := |\{e \in \mathcal{E} \mid h \leq t(e) \leq t\}|$
    ;$y :=$ lower bound of $|\{e \in \mathcal{E} \mid h \leq t(e) - \rho(e) \wedge t(e) \leq t\}|$
    ;if $(x-y) > c_s \rightarrow r := h$
    [] $(x-y) \leq c_s \rightarrow s,n := h,y$
    fi
 od
;return($(n,s)$)
```

Figure 6.21. Assuming a fully set-associative cache, the function filling(z,t) returns a pair (n,s) where $t-s$ is a lower bound on the length of the filling interval at t, and n is a lower bound on the number of hits in the interval $\{s,\dots,t\}$. Here, the filling function f and R have been expressed as function of time, instead of port executions.

Substituting this value for b into (6.27) leaves the minimisation of

$$\frac{1}{2}\boldsymbol{p}^{\mathrm{T}}\boldsymbol{I}a + \sum_{l=0}^{\delta-1} (x_l - ap_l)^+ I_l. \tag{6.28}$$

As each term $(x_l - ap_l)^+$ is linear in a for $a \leq \frac{x_l}{p_l}$ and $a \geq \frac{x_l}{p_l}$, we know that an optimum for (6.28) is found either for $a = \frac{x_l}{p_l}$ for some $0 \leq l < \delta$, or $|a| = \infty$. However, as (6.28) goes to infinity for both $a \rightarrow \infty$ and $a \rightarrow -\infty$, the optimum is found by evaluating (6.28) for $a = \frac{p_l}{x_l}$ for all $0 \leq l < \delta$. □

In Figure 6.21 we now give a binary search algorithm for determining the filling for a port execution $e \in \mathcal{E}$ with $t(e) = t$. The additional parameter z to the function *filling* gives the earliest time at which any port execution can occur. The function returns a pair (n,s) where $t-s$ is a lower bound on the length of the filling interval and n is a lower bound on the number of hits in the interval $\{s,\dots,t\}$.

6.5 Counting cache misses

As discussed earlier, a lower bound on the number of cache hits, and hence an upper bound on the number cache misses, is obtained as a side effect of the algorithm of Figure 6.21. In Figure 6.22 we give an algorithm for the determination of a lower bound on the number of hits for a schedule. The algorithm works as follows. For

```
 $s,t := \min\{\, f \in \mathcal{E} \mid t(f)\,\}, \max\{\, f \in \mathcal{E} \mid t(f)\,\}$
;$h := 0$
 { invariant : $h \leq |\{\, e \in \mathcal{E} \mid t \leq t(e) \wedge r(e) \leq f(e)\,\}|$ }
;**do** $(t > s) \rightarrow$
     $(n,g) := \text{filling}(s,t)$
    ;$t,h := \min(g, t-1), h+n$
 **od**
 { $h \leq |\{\, e \in \mathcal{E} \mid r(e) \leq f(e)\,\}|$ }
```

Figure 6.22. An algorithm for the determination of a lower bound on the number of hits for a set $\mathcal{E}$ of port executions.

the port execution with the latest scheduled time, say $e \in \mathcal{E}$, we determine a lower bound $t(e) - g$ on the length of the filling interval $f(e)$ and a lower bound on the number of hits, say n, in the interval $\{t(e) - g, \dots, g\}$ using the algorithm of Figure 6.21. Next, we determine a lower bound on the length of the filling interval at time $t(e) - g$, and a lower bound on the number of hits in the corresponding interval, and so forth. By summing these number of hits in the successive intervals we get a lower bound on the total number of cache hits for the schedule.

Progress of the algorithm is guaranteed as the value of variable t decreases by at least one in each iteration. Furthermore, in the interval $\{\min(g, t-1), \dots, t\}$ at least c_s port executions take place. Hence, the number of steps of the algorithm is bounded from above by $\frac{|\mathcal{E}|}{c_s}$.

This algorithm only counts hits for which the reuse falls within the interval $\{t - g, \dots, t\}$, i.e., hits for executions e with both $t(e)$ and $t(e) - \rho(e)$ within this interval, which results in a pessimistic estimation of the number of cache hits. Instead of choosing the decrement $\max(t - g, 1)$ for variable t, we may choose any value between $t - g$ and 1. Choosing a smaller decrement reduces the chance of not counting a hit, but increases computation time of the algorithm. If we choose such a decrement, we have to take care that hits are not counted multiple times by successive iterations of the algorithm. This problem can be solved by an administration of executions that generate hits using the approximation of the reuse length as given by Definition 6.5.

6.6 Experiments

The algorithms of Figures 6.21 and 6.22 have been implemented and tests have been performed for two characteristic algorithms. In the implementation of the reuse graph construction algorithm we have chosen that the iteration spaces $I(a)$

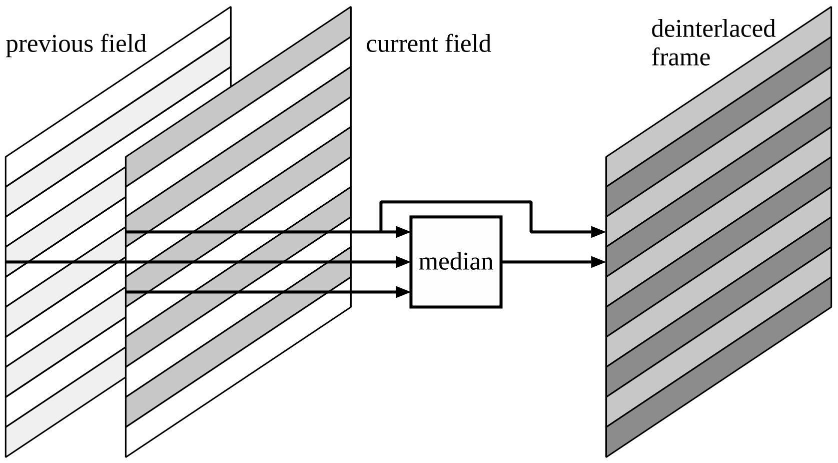


Figure 6.23. Median filter deinterlacing algorithm. Pixels in missing lines of the deinterlaced frame are calculated by taking the median of two pixels from neighbouring lines in the current field, and of one pixel from the line on the same vertical position in the previous field [Riemens, Schutten & Vissers, 1997].

on the edges a of the reuse graph are hypercubes, i.e., each iteration space $I(a)$ of an edge $a = (p,q)$ is a set $\{\boldsymbol{i} \mid \boldsymbol{i} \in \mathbb{Z}^{\delta(p)} \wedge \boldsymbol{y} \leq \boldsymbol{i} \leq \boldsymbol{z}\}$ for some vectors $\boldsymbol{y},\boldsymbol{z} \in \mathbb{Z}^{\delta(p)}$.

The first algorithm is a median filter deinterlacing algorithm and it converts an interlaced video signal to a non-interlaced signal [Riemens, Schutten & Vissers, 1997]. Every picture (frame) in an interlaced video signal consists of two consecutive fields, one containing all even lines of the image, the other all odd lines. The median filter deinterlace algorithm converts such a signal to a signal where every image contains all lines. The operation of the algorithm is given schematically in Figure 6.23. The program graph of this algorithm consists of seven operations with a total of 24 ports and 34 edges. For these tests we have chosen fields consisting of 20 lines of 400 pixels each. We have done experiments with three interlaced input frames (three even fields and three odd fields) and six non-interlaced output frames. These experiments are denoted by M3. M7 denotes experiments with 14 output frames.

The other algorithm is matrix multiplication of 64×64 matrices, denoted by m64. For all experiments we have chosen a fully set-associative cache with 256 blocks of 16 bytes each.

All experiments have been performed on a 200 MHz Pentium MMX with 128 MB of RAM, running Linux kernel 2.2.5-15. All programs have been compiled with release 1.1.2 of egcs with optimisation flag -O.

	#access	#miss	rate	sim (s)	sim+gen (s)
M3	475200	24000	5.05%	11.16	13.85
M7	1108800	56000	5.05%	27.92	32.54
m64	1052672	20928	1.99%	29.18	34.18

Table 6.1. Simulation results show the number of cache misses, the cache miss rate, and the amount of time necessary to compute these numbers.

As a reference for our cache cost estimation algorithms we have performed cache simulations, for which we used DineroIII (version 3.4). This simulator is based on the work of Hill [1987]. Simulation results are listed in Table 6.1. In column '#access' the total number of port executions is given. The next columns give the number of cache misses, and this number as a percentage of the total number of port executions. The last two columns give the time for cache simulation, where the first of the two columns gives the time the cache simulation program has spent. The second also includes the time needed to generate the address trace by executing the video algorithm.

In Table 6.2 we give numbers produced by our algorithm for reuse graph construction. As the number of edges in the reuse graph can become very large, we stop estimation of reuse for a port if the out-degree of the port in the reuse graph has exceeded a certain threshold. In our experiments we have limited the out-degree in the reuse graph to 4 and 8 for the median filter algorithm. For the matrix multiplication example, the out-degree does not exceed 4. The out-degree of any vertex in the reuse graph has been limited to the number in column 'max'. In the column 'time' the time needed for the construction of the reuse graph is listed.

As discussed before, the quality of a reuse graph can be measured in several ways. One way is by comparing the approximated reuse length with the (exact) reuse length. The two columns of Table 6.2 tagged 'overestimated' give the number of port executions for which the approximate reuse length is larger than the reuse length and this number as a percentage of the total number of port executions. The exact reuse length has been determined by computing the reuse length for every port execution individually by means of simulation. Column 'inf' gives the number of port executions for which no reuse was found in the reuse graph, despite the fact that reuse existed. The 'avg error' is the average overestimation of reuse, i.e., the average difference between approximated reuse length and exact reuse length.

Another way of measuring the quality of a reuse graph is to determine for how many port executions a miss is predicted using the reuse graph and the (exact) length of the filling interval, but where a hit occurs. The last two columns of Table 6.2 give the number of port executions that result in a hit, but where a miss was

	max	time (s)	overestimated		inf	avg error	error	
M3	4	2.20	61450	12.9%	2500	19564	2775	0.58%
	8	3.51	51450	10.8%	2500	12452	75	0.02%
M7	4	2.68	148050	13.4%	2500	22113	6475	0.58%
	8	2.98	138600	12.5%	2500	23068	175	0.02%
m64	4	0.11	274176	26.0%	0	229	0	0.00%

Table 6.2. Quality of the reuse graph for a median filter algorithm and a matrix multiplication algorithm. For an explanation of this table, see the text.

	max	time (s)	#int	#miss	miss rate
M3	4	0.55	107	27007	5.68%
	8	0.69	100	25343	5.33%
M7	4	1.82	249	63016	5.68%
	8	1.81	233	59141	5.33%
m64	4	0.15	129	32738	3.11%

Table 6.3. Cache miss estimation results for a median filter algorithm and a matrix multiplication algorithm using estimated reuse lengths and estimated filling function.

estimated, and this number as a percentage of the total number of port executions.

In Table 6.2, we see that despite overestimation of the reuse length for a considerable number of port executions the error for the estimation of the miss rate is relatively small for these experiments. For example, for m64 the reuse length was overestimated for 26% of the port executions, yet this does not result in errors in the estimation of the reuse length. Furthermore, we see that choosing a larger out-degree for the vertices in the reuse graph results in a better estimation of reuse reuse length and consequently in a better estimated cache miss rate.

Estimation of the number of cache misses based on the reuse graph and *estimated* length of the filling interval is given in Table 6.3. The maximum out-degree of any edge in the reuse graph is again in the column tagged 'max'. The time for computing the number of cache misses is in the column tagged 'time'. Here the time for the construction of the reuse graph is not included. The number of iterations of the cache miss estimation algorithm of Figure 6.22 is in column '#int'. The estimated number of misses and the estimated miss rate are in the last columns.

In Figure 6.4 we have summarised the results of the previous tables. The first column gives the exact cache miss rate. The last two columns give an estimation of the cache miss rate using an estimation of the reuse length and the exact filling on the one hand and an estimation of the filling on the other hand. Between brackets

	exact reuse exact filling	max	est. reuse exact filling	est. reuse est. filling
M3	5.05% (13.85s)	4	5.63%	5.68% (2.75s)
		8	5.07%	5.33% (4.20s)
M7	5.05% (32.54s)	4	5.63%	5.68% (4.50s)
		8	5.07%	5.33% (4.79s)
m64	1.99% (34.18s)	4	1.99%	3.11% (0.26s)

Table 6.4. Summarised cache miss estimation results.

are the computation times for the cache miss rate for simulation in the leftmost column and for estimation in the rightmost column. We observe that estimation of the miss rate can be improved by allowing a larger out-degree of the vertices in the approximate reuse graph. The increase in computation time is caused mainly by the time needed for the construction of the reuse graph. Furthermore we observe that increasing the problem size of the median filter algorithm from M3 to M7 does not change the estimation results. In the simulation results, we see that the difference in number of accesses between M3 and M7 results in a proportional increase in simulation time. For our method, however, the influence of the values of the loop bounds, and thus the number of accesses, has only a small effect on the time needed for the construction of the reuse graph, and the time needed for computing the filling function is small compared to the simulation times. In general we see that good estimation results can be obtained in computation times much smaller than the corresponding times needed for cache simulation.

The last row shows, however, relatively many uncounted hits. This is a result of the effect described in Section 6.5. In that section we have proposed a solution to this problem at the cost of computation time, but better estimation results can probably also be achieved by using the following property. During the experiments we observed that the value of the filling function is almost constant over long time intervals. In order to find out whether the filling function may be estimated on only a limited number of points in time, we have done some experiments on some video algorithms. In Table 6.5 results are listed for five algorithms, which have all been executed and the resulting address traces have been analysed. Applications I is matrix multiplication on 10×10 matrices. Applications II, III, and IV are video deinterlacing algorithms like the median filtering algorithm of Figure 6.23. Application V is an MPEG2-decoder.

In the table we find the number of accesses, the number of cache misses and the cache miss rate, which all follow directly from the address trace. Furthermore, we have calculated the length of the filling interval at a limited number of points in time, the length between which is given in the column tagged ‘interval length’.

program	number of accesses	#misses	miss rate	interval length	estimated miss rate
I	5295	626	11.82%	100	11.84%
				1000	12.63%
				5000	15.60%
II	134400	6504	4.84%	10000	4.84%
				100000	4.98%
				500000	5.22%
III	387200	18508	4.78%	10000	4.78%
				100000	4.78%
				500000	5.23%
IV	5102006	97328	1.91%	10000	1.91%
				100000	1.92%
				500000	1.87%
V	45166957	191721	0.42%	10000	0.42%
				100000	0.42%
				500000	0.43%

Table 6.5. Estimation of $f(t)$ by linear interpolation between $f(t_0)$ and $f(t_1)$ for interval $\{t_0,\dots,t_1\}$, where the interval length $t_1 - t_0$ is varied. For the calculation of the estimated miss rate we use the exact reuse length and the interpolated filling.

For all points inside this interval we estimate the length of the filling function as a linear interpolation between the end points of the interval. Using this interpolated value and the exact reuse length from simulation we obtain an estimation for the miss rate. Note that large errors in the miss rate are due to a limited number of estimations of the filling function. In some cases we have only one or two such estimations. A disadvantage of this method is that we lose the property of overestimating the miss rate, as an interpolation of the filling function does not yield a lower bound on this function.

6.7 Discussion

In this chapter we have considered the problem of computing the number of cache misses for a given program graph, a given schedule, and a cache. We have split this problem into two parts. In the first part we determine for every execution of a port the most recent execution of a port that accessed the same cache block. The difference in time between these executions is called the reuse length. In the second part we determine the so-called filling interval for every execution of a port. By combining the length of the filling interval and the reuse length we have found an expression for cache misses.

In Theorem 6.2 we have shown that the problem of finding reuse is a formally difficult problem. Hence, we have given heuristics for finding an approximate reuse graph. This graph is used next for finding the length of the filling interval. We have implemented an algorithm for the computation of the number of cache misses and have obtained good results for a number of relevant programs.

The algorithm for finding an approximate reuse graph yields reuse information for sets of port executions. If all executions in such a set result in cache misses, this information may be used to change the schedule, in order to diminish the number of cache misses. Minimising the number of cache misses is the subject of the next chapter.

Besides cache miss computation, reuse graphs may have other applications, for example for prefetching. Prefetching is a well known technique for avoiding latency for cache misses. A prefetch operation fetches data from memory into the cache before it is actually needed, avoiding a miss penalty. Such an operation may be inserted in the instruction stream by a special statement in a program, either manually or by a compiler. This is known as *software prefetching*. In another approach, known as *hardware prefetching*, the cache determines autonomously which data to prefetch from memory. Struik, Van der Wolf & Pimentel [1998] combine the advantages of both software prefetching and hardware prefetching by introducing special prefetch instructions that prefetch a stream of cache blocks over time. Such a stream can be represented as a one-dimensional periodic operation. As we have represented sets of port executions as periodic operations on the edges of the reuse graph, we can find good candidates for prefetching from the reuse graph.

7

Towards Global Cache-Miss Minimisation

In this chapter we make a first step towards an optimisation algorithm based on our cache cost calculation approach of the previous chapter. The multidimensional periodic cache scheduling problem of Definition 3.10 cannot be solved in polynomial time unless P = NP as we proved in Chapter 4. As we are interested in finding good approximate solutions to this optimisation problem in reasonable running times we aim at a local search approach, a well known method that often works well for a large range of scheduling problems. In order to reduce the search space beforehand, we decompose the optimisation problem into two parts, where we aim at finding a good time assignment and a part of the address assignment in the first part of the decomposition, and where the fine-tuning of the schedule takes place in the second part by completing the address assignment. Section 7.1 proposes the decomposition strategy. Section 7.2 focuses on a local search approach for finding a good time assignment. Possibilities for neighbourhoods are discussed in Section 7.3. Incremental cost computation for finding the cost of neighbouring solutions is handled in Section 7.4.

7.1 Decomposition

The multidimensional periodic cache scheduling problem of Definition 3.10 has two kinds of decision variables, on one hand being the start times and period vectors, that determine the execution times of the operations, and on the other hand the offsets and coefficient vectors, that determine the memory layout of the arrays.

We propose a decomposition of the optimisation problem into two sub-problems. The first sub-problem, which we call the *extended time assignment problem*, takes intra-array reuse into account. The second problem, the *constrained address assignment problem*, focuses on inter-array reuse and prevention of so-called *conflict misses*.

The decision variables that we determine in the first sub-problem are the start times and period vectors of all operations, and for the arrays only those coefficients that are smaller than the cache block size. The decision variables that are determined by the second sub-problem are the remaining address coefficients.

The reason for choosing this decomposition is threefold. First, the feasibility of an address assignment depends on a choice for a time assignment. For every time assignment it is possible to find a feasible address assignment, but the opposite is not necessarily true. Hence, in the first stage of the optimisation process we try to find a feasible time assignment and complete the optimisation with proper values for the address assignment.

Second, we assume that most cache hits are caused by *intra-array reuse*, i.e., reuse between port executions that belong to the same array. By definition, effective reuse of the values stored in a cache can only occur between elements of the same array. For inter-array reuse we have to take the address offsets and the address coefficients with values larger than the block size into consideration.

Third, the cost calculation algorithm of Chapter 6 only takes into account fully set-associative caches. Caches with lower associativity suffer from *conflict* misses, as discussed in Chapter 2.

Hill [1987] introduced the 3-C's model, where he makes a distinction between *compulsory misses*, *capacity misses*, and *conflict misses*. Conflict misses and capacity misses together constitute what we call expiration misses in the following way. Conflict misses are misses that occur in *n*-way set-associative caches but not in a fully set-associative cache, and capacity misses are misses that result in a miss in a fully set-associative cache. In this way, conflict misses occur because of limited associativity of the cache, and capacity misses occur because of limited size of the cache. For scientific programs, Rivera & Tseng [1998] have shown that conflict misses can effectively be dealt with appropriate choices for the address coefficients and address offsets. Therefore, for the first sub-problem we limit the discussion to capacity misses.

In this chapter we focus on the extended time assignment problem. In the next section we give the ingredients for a local search approach to this problem. For the second sub-problem we refer to techniques proposed by Rivera & Tseng [1998], Calder, Krintz, John & Austin [1998], and Strout, Carter, Ferrante & Simon [1998].

7.2 Local search strategy

In Chapter 4 we have proved the optimal time assignment problem (OTA) NP-complete. Therefore, it is likely that the extended time assignment problem cannot be solved to optimality in computation times that are bounded by a polynomial in the size of an instance of the problem. In order to find near-optimal solutions within reasonable computation time there are two approaches that we can follow; *greedy* algorithms or *local search* algorithms.

There are several examples of greedy algorithms for the time assignment optimisation problem. Most of them are based on one or more types of *loop transformations*, which are discussed in the next section. For example, McKinley, Carr & Tseng [1996] propose a *compound* algorithm, which combines the advantages of four such transformations.

Local search algorithms explore neighbourhoods of solutions. *Iterative improvement* is one such algorithm that starts with an initial solution, after which it repeatedly replaces the solution with a neighbouring solution with lower cost. The algorithm terminates with a solution that does not have lower-cost neighbours, i.e., a local optimum.

The cache optimisation problem (MPCS) as well as the problems optimal time assignment (OTA) and optimal address assignment (OAA) that we considered in Chapter 4 are combinatorial optimisation problems. We follow the definition of a combinatorial optimisation problem given by Aarts & Lenstra [1997].

Definition 7.1 (combinatorial optimisation problem). An *instance of a combinatorial optimisation problem* is a pair $(\mathcal{S}, f)$, where the *solution set* $\mathcal{S}$ is a finite or countably infinite set of *feasible* solutions, and the *cost function* f is a mapping $f : \mathcal{S} \to \mathbb{R}$. The problem is to find a *globally optimal* solution. □

For a local search approach the notion of a neighbourhood is essential.

Definition 7.2 (neighbourhood structure). A neighbourhood structure for an instance $(\mathcal{S}, f)$ of a combinatorial optimisation problem is a mapping $\mathcal{N} : \mathcal{S} \to \mathcal{P}(\mathcal{S})$. The set $\mathcal{N}(s)$ for a solution $s \in \mathcal{S}$ is called the *neighbourhood* of s. A solution $s \in \mathcal{S}$ is a *local minimum* of $(\mathcal{S}, f)$ for $\mathcal{N}$ if $f(s) \leq f(s')$ for all $s' \in \mathcal{N}(s)$. □

As loop transformations have proved their effectiveness for cache usage, we base the neighbourhood structure on them. Before elaborating on a neighbourhood structure for the extended time assignment problem we give five aspects that have to be taken into account when designing a local search algorithm. These are a *representation* of solutions, a neighbourhood structure, a way to compute the cost of neighbour solutions efficiently, a stop criterion for the local search method, and a way to construct an initial solution. The last two aspects are not discussed in this thesis.

In the next section we discuss possibilities for representation of solutions and possibilities for neighbourhoods that we base on existing techniques for reducing cache misses.

In general, neighbouring solutions are only different in a limited number of decision variables. In Section 7.4 we indicate how the cost of a neighbouring solution can be found efficiently, without having to apply the algorithm of Chapter 6 for the entire newly created schedule, but only for the parts of the schedule that changed. This is called *incremental cost calculation*.

7.3 Neighbourhoods

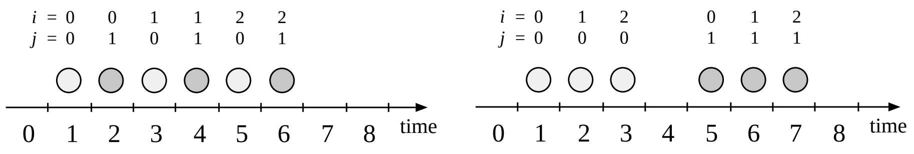
Solutions to our problem are defined in terms of the decision variables start times, period vectors, and address coefficients. The size of the solution space is large, but many solutions result in the same program and, therefore, in the same number of cache misses. For example, multiplying all periods and start times by the same factor leads to a schedule with the same partial order on the execution of operations, and hence it leads to the same program with the same number of cache misses.

In order to reduce the solution space, we apply some restrictions. We already have given restrictions on time assignments for sequentialisability purposes in Chapter 5 and for cost calculation purposes in Chapter 6. Based on these restrictions we limit the periods to powers of a chosen base, e.g., 2. The number of values that can be chosen for the exponents can be limited, thereby making the search space for periods finite. Also the number of start times can be limited by choosing a left-justification of the executions of the operations.

As loop transformations have proved their effectiveness for cache usage, we base moves in the neighbourhood structure on them. Unfortunately, the neighbourhood of a solution that we obtain by considering the following moves can become large. Nevertheless, this neighbourhood can be reduced by only considering those moves that involve ports that are responsible for cache misses. This kind of information is available in the (approximate) reuse graph.

7.3.1 Loop transformations

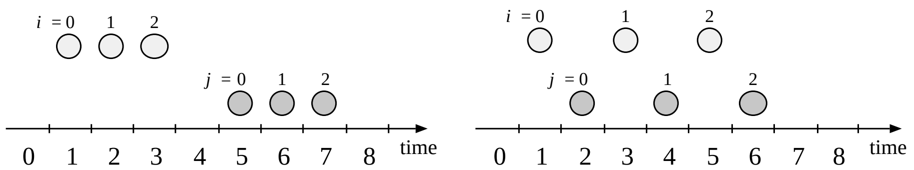
In the field of compiler optimisations, the use of loop transformations is very common. Bacon, Graham & Sharp [1994] give an overview of many transformations that are used for minimisation of the number of cache misses, for fine grained parallelism (instruction level parallelism and vectorisation), and for coarse grained parallelism. In this section we give an overview of transformation techniques that are often used to enhance cache performance. In addition we give the corresponding formulation of the transformation in terms of multidimensional periodic operations.



```
for $i := 0 \ldots 2$ period 2 $\rightarrow$
   for $j := 0 \ldots 1$ period 1 $\rightarrow$
      $\ldots = f(Y[i+10j])$ start at 1
```

```
for $i := 0 \ldots 2$ period 1 $\rightarrow$
   for $j := 0 \ldots 1$ period 4 $\rightarrow$
      $\ldots = f(Y[i+10j])$ start at 1
```

Figure 7.1. Loop interchange.



```
for $i := 0 \ldots 2$ period 1 $\rightarrow$
   $X[i] = f(Y[i])$ start at 1
for $i := 0 \ldots 2$ period 1 $\rightarrow$
   $Z[j] = g(X[j])$ start at 5
```

```
for $i := 0 \ldots 2$ period 2 $\rightarrow$
   $X[i] = f(Y[i])$ start at 1
for $i := 0 \ldots 2$ period 2 $\rightarrow$
   $Z[j] = g(X[j])$ start at 2
```

Figure 7.2. Loop fusion.

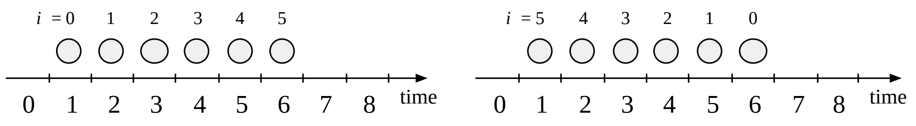
Loop interchange

Loop interchange (or loop permutation) interchanges the order of execution of loops within a loop nest. Interchanging loops may have positive effect on the number of cache misses and on the amount of parallelism that a compiler can find [Bacon, Graham & Sharp, 1994]. The positive effect on possible reuse is depicted in Figure 7.1, where the spatial reuse between iterations i and $i+1$ is shortened. In terms of operations we see that only the period vector is affected.

Loop fusion and loop fission

Loop fusion combines the loop bodies of two loop nests into a single loop nest. Loop fusion can, amongst others, be used to increase instruction level parallelism, and to improve the locality of reference. In Figure 7.2, the reuse length of all executions of the input port of operation g are shortened from 4 to 1. In terms of periods and start times this transformation is a ‘stretch’ of the periods of the outer dimensions and a left-justification of start times.

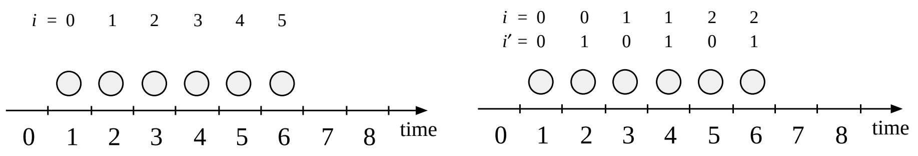
The inverse transformation is called loop distribution, loop fission, or loop splitting.



```
for $i := 0 \ldots 5$ period $1 \rightarrow$
    $X[i] = Y[i] + 1$ start at 1

for $i := 0 \ldots 5$ period $-1 \rightarrow$
    $X[i] = Y[i] + 1$ start at 6
```

Figure 7.3. Loop reversal.



```
for $i := 0 \ldots 5$ period $1 \rightarrow$
    $X[i] = Y[i]$ start at 1

for $i := 0 \ldots 2$ period $2 \rightarrow$
    for $i' := 0 \ldots 1$ period $1 \rightarrow$
        $X[2i + i'] = Y[2i + i']$ start at 1
```

Figure 7.4. Loop tiling.

Loop reversal

Loop reversal switches the direction in which the iterations of a loop are executed. For a program consisting of a single loop nest, loop reversal will not change the number of cache misses. Nevertheless, it is used to *enable* other loop transformations such as loop permutation. In Figure 7.3 it is shown that loop reversal negates the period for one iterator and increases the start time of the operation by $\underset{\sim}{p}I_l$, with l being the changed iterator.

Tiling

Tiling divides the iteration space into smaller blocks (or *tiles*). Tiling (or blocking) is used to break computations for large arrays into several computations on sub-arrays. In Figure 7.4 we give a small example. In this case the array has been split in tiles of size 2. In terms of periods and start times, blocking or tiling can be described as a dimension split, as introduced in the previous chapter.

Research on the selection of the size of the tiles has, amongst others, been done by Lam, Rothberg & Wolf [1991, Coleman & McKinley [1995, Kennedy & McKinley [1992].

Neighbourhoods based on transformations

We have given four common types of loop transformations and we have discussed how they can be described as manipulations of period vectors and start times. Hence, for our neighbourhood we choose these transformations. Probably many more manipulations are possible using periods and start times than the ones discussed above.

7.3.2 Moves for address coefficients

Besides changes in the time assignments, we also have to determine which address coefficients are assigned values smaller than the cache block size. These coefficients are responsible for self spatial reuse. Therefore, we keep a set of address coefficients for every array. Every address coefficient in this set we choose as small as possible in order to maximise the number of executions for which we find self spatial reuse.

As we saw in the matrix multiplication example of Chapter 6, we may even choose address coefficients zero for some array dimensions. By choosing coefficients zero, memory addresses are reused and in this way cache locations can be reused for multiple array elements.

As we have not explored neighbourhoods for the address assignment further, we conclude this short discussion by saying that possible moves consist of changing the set of address coefficients for which we choose small values.

7.4 Incremental cost calculation

A main assumption for adopting local search is fast evaluation of (changes in) the cost function. As proved in Chapter 4, a polynomial time algorithm for computation of the cost function is unlikely to exist. In Chapter 6 we have given a heuristic approach for estimating the cost of a schedule. The time complexity of the heuristic approach, which is fast in practice, is however not bounded by a polynomial in the size of the schedule.

Fortunately, two observations may save a lot of work for the recalculation of the cost function when considering neighbours. Local changes in a solution have only local effects on both parts of the decomposition of the cost evaluation, since only array clusters that were changed have to be looked at for the reuse analysis, and local changes in the time assignment result in a limited time interval for which the number of cache misses must be recalculated.

As we postpone the greater part of the address assignment until after the time assignment, estimations for the reuse length can only be based on reuse within array clusters. As the number of array clusters that are affected by a local transformation between neighbours is limited, the amount of computation time for an

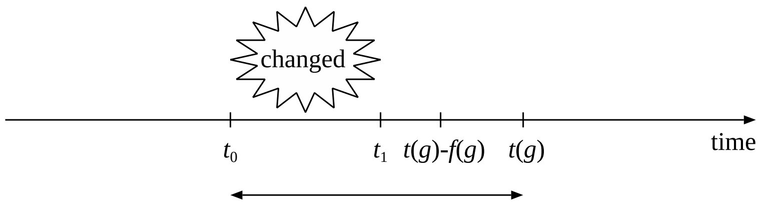


Figure 7.5. For two solutions that differ only in the time interval $\{t_0,\ldots,t_1\}$, the set of cache misses differs only in the time interval $\{t_0,\ldots,t(g)\}$ for some port execution g with $t(g) - f(g) \geq t_1$.

estimation of a new reuse graph is also limited.

In Figure 7.5 we show a local change in the time assignment and the resulting time interval in which the number of cache misses may change. Theorem 7.1 shows the validity of the interval.

Theorem 7.1. *Given are two time assignments τ and τ' and two integers $t_0 < t_1$, such that the assignments differ only in the time interval $\{t_0,\ldots,t_1\}$, i.e., $t(p,\boldsymbol{i}) \neq t'(p,\boldsymbol{i}) \Rightarrow t_0 \leq t(p,\boldsymbol{i}), t'(p,\boldsymbol{i}) \leq t_1$ for all executions $\boldsymbol{i} \in \mathcal{I}(p)$ of operations $p \in P$. Also given are a port execution $g \in \mathcal{E}$ with $t_1 < t(g) - f(g)$ and a fully set-associative cache $\mathcal{C}$. Then for all port executions $e \in \mathcal{E}$ with $t(e) < t_0$ or $t(g) \leq t(e)$ a miss occurs for time assignment τ if and only if it occurs for time assignment τ'.*

Proof. As proved in Theorem 6.1, a cache miss for a port execution $e \in \mathcal{E}$ occurs if and only if $r(e) > f(e)$.

As both the reuse length $r(e)$ and the filling $f(e)$ are defined only in terms of port executions that take place before e, and the time assignment has not changed for all port executions preceding t_0, the reuse length and filling have not changed for all port executions preceding t_0. Hence, we know that $r(e) = r'(e)$ and $f(e) = f'(e)$ for port executions e with $t(e) < t_0$ and consequently that a miss occurs for time assignment τ if and only if it occurs for time assignment τ'.

For the other case, $t(g) \leq t(e)$ we first prove that $f(e) = f'(e)$. This equality is based on the following monotony argument for fully set-associative caches.

$$t(e) \leq t(e') \;\Rightarrow\; t(e) - f(e) \leq t(e') - f(e') \text{ for all } e \in \mathcal{E} \tag{7.1}$$

From this, we derive $t_1 < t(g) - f(g) \leq t(e) - f(e)$. As the value of $f(e) = \max\{\, t(e) - t(f) \mid f \in \mathcal{E} \wedge R(t(f), e) \leq c_s \,\}$ is determined only by terms $t(f)$ larger than t_1, we know that $f(e) = f'(e)$.

Next, we apply case analysis by considering the cases $t(g) - f(g) \leq t(e) - r(e)$ and $t(e) - r(e) < t(g) - f(g)$. First we look at the case $t_1 < t(g) - f(g) \leq t(e) - r(e)$. As the value of $r(e) = \min\{\, t(e) - t(f) \mid f \in \mathcal{E} \;\wedge\; t(f) < t(e) \;\wedge\; a_b(f) =$

$a_b(e)\}$ is determined only by port executions that take place after t_1, we know that $r(e) = r'(e)$. We already established that the filling was equal for both time assignments and hence we have shown that a miss for time assignment τ occurs if and only if it occurs for τ'.

For the other case, $t(e) - r(e) < t(g) - f(g)$, we prove that a miss occurs both for time assignment τ as well as for τ'. Using the monotony property (7.1), we derive that $t(e) - r(e) < t(g) - f(g) \leq t(e) - f(e)$, and thus $r(e) > f(e)$. Hence a miss occurs for port execution e for time assignment τ. As the time assignment did not change for executions that take place after t_1, especially those that take place after $t(g) - f(g)$, we know that $t(e) - r'(e) < t(g) - f(g) \leq t(e) - f(e) = t(e) - f'(e)$, which implies $r'(e) > f'(e)$, and thus a miss occurs also for time assignment τ'. □

7.5 Discussion

In this chapter we have given ingredients for a local search algorithm for the multidimensional periodic cache scheduling problem. We have aimed at a global approach in the sense of optimising for all operations together, instead of optimising for individual operations. We have proposed a decomposition of the problem into two sub-problems, where we aim at exploiting intra-array reuse in the first part, and where completing address assignments is left to the second part. For the first part we use a local search approach with a neighbourhood structure based on loop transformation techniques that are well known from literature. For the second part of the decomposition, where we determine the address offsets and most of the address coefficients, we refer to existing techniques such as padding [Rivera & Tseng, 1998].

8

Conclusion

In this thesis we studied the multidimensional periodic cache scheduling problem. This problem originates from the field of video signal processing algorithms, where operations have to be performed repeatedly. In our case, we want to execute such algorithms on fast processors. Because of the widening gap between the speed of processors and the speed of memory, these processors are equipped with caches, but their optimal use is still an open problem.

In this problem, the order of execution of operations is modelled by a time assignment, consisting of periods and start times. Operations read data from memory locations and write data to memory locations. An address assignment, consisting of address coefficient vectors and address offsets, determines these memory locations for every execution of an operation. The problem is to find a time assignment and an address assignment that obey precedence constraints and address constraints, and that minimise the number of cache misses. Contrary to other approaches found in literature we aim at a global approach, by which we mean that we look at all operations or loop nestings at the same time, instead of optimising a single operation or loop nesting.

The multidimensional periodic cache scheduling problem has been shown to be formally hard. In addition to the cache scheduling problem, we have also looked at the problem of minimising the number of cache misses if we fix either the time assignment or the address assignment. These problems are also proved NP-hard.

Even if we fix both the time assignment and the address assignment we have shown that computing the number of cache misses can only be done in polynomial time if P = NP.

The time assignment orders the executions of each operation. If operations must be executed on a processor that has a single thread of control, we need to map the periods and start times of all operations onto a sequential program. Not all time assignments, however, can be mapped efficiently onto a such a program. We have given conditions for the time assignments that allow an efficient implementation on a processor.

At the heart of this thesis is a discussion on the problem of computing the number of cache misses for a given schedule consisting of a time assignment and an address assignment. Since this problem is formally difficult, we have aimed at finding an efficient and effective algorithm to estimate the number of cache misses. This algorithm consists of two parts. In the first part we create a so-called reuse graph. In such a graph we collect information about reuse of cache blocks, i.e., we give a compact formulation for the time between two successive accesses to the same cache block. Also, the construction of such a graph cannot be done in polynomial time unless P = NP. As we want a fast evaluation of the number of cache misses for a schedule, we have introduced an approximation of the reuse graph and we have given heuristics for finding edges in such an approximate reuse graph. These heuristics are based on situations that often occur in video algorithms.

In the second part of the computation of cache misses, we estimate the filling of the cache, which reflects from how long ago all accessed blocks are still in the cache. We have given an algorithm for computing this filling as a function of time. This algorithm uses the constructed approximate reuse graph and computes the number of cache misses as a result. From experiments we have learned that the filling of the cache as a function of time is rather constant, and can well be approximated by computing the filling at only a few points in time.

The algorithm for the estimation of the number of cache misses has been implemented and experimental results have been compared to cache simulations. We have obtained good estimations with computation times much smaller than cache simulations.

Finally, as a first step towards a solution approach based on our cache cost calculations, we have presented ingredients for a local search approach for the multidimensional periodic cache scheduling problem. Moves for the local search algorithm are based on loop transformations that are well known from literature. As the reuse graph contains information about sets of operations that cause cache misses, we can use this information to reduce the number of considered moves in the local search approach. Furthermore, we have shown a way of reducing the amount of work needed to compute cost of neighbour solutions.

Bibliography

AARTS, E.H.L., AND J.K. LENSTRA (eds.) [1997], *Local Search in Combinatorial Optimization,* Wiley, Chichester.

AHO, A.V., R. SETHI, AND J.D. ULLMAN [1986], *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, Massachusetts.

BACON, D.F., S.L. GRAHAM, AND O.J. SHARP [1994], Compiler transformations for high-performance computing, *ACM Computing Surveys* **26**, 345–420.

BELADY, L.A. [1966], A study of replacement algorithms for a virtual-storage computer, *IBM Systems Journal* **5**, 78–101.

CALDER, B., C. KRINTZ, S. JOHN, AND T. AUSTIN [1998], Cache-conscious data placement, *Proceedings of the eighth International Conference on Architectural Support for Programming Languages,* San Jose, 139–149.

CALLAHAN, D., K. KENNEDY, AND A. PORTERFIELD [1991], Software prefetching, *Computer Architecture News* **19**, 40–52.

CHEN, T.-F. [1997], Reducing memory penalty by a programmable prefetch engine for on-chip caches, *Microprocessors and Microsystems* **21**, 121–130.

CHEN, T.-F., AND J.L. BAER [1992], Reducing memory latency via non-blocking and prefetching caches, *SIGPLAN Notices* **27**, 51–61.

CHROBAK, M., AND J. NOGA [1999], LRU is better than FIFO, *Algorithmica* **23**, 180–185.

CLAUSS, PH. [1996], Counting solutions to linear and nonlinear constraints through Ehrhart polynomials, *Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, 278–285.

CLOUT, R.A.W. [1994], Off-line scheduling for cache optimization, Master's thesis, Eindhoven University of Technology.

COLEMAN, S., AND K.S. MCKINLEY [1995], Tile size selection using cache organization and data layout, *Proceedings of the 1995 Conference on Programming Language Design and Implementation*, 279–290.

DIJKSTRA, E.W., AND C.S. SCHOLTEN [1990], *Predicate Calculus and Program Semantics*, Springer-Verlag, Berlin.

GANNON, D., W. JALBY, AND K. GALLIVAN [1988], Strategies for cache and local memory management by global program transformation, *Journal of*

Parallel and Distributed Computing **5**, 587–616.

GAREY, M.R., AND D.S. JOHNSON [1979], *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York.

GHOSH, S., M. MARTONOSI, AND S. MALIK [1998], Precise miss analysis for program transformations with caches of arbitrary associativity, *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages*, San Jose, California, 228–239.

GUPTA, D., B. MALLOY, AND A. MCRAE [1997], The complexity of scheduling for data cache optimization, *Information Sciences* **100**, 27–48.

HENNESSY, J.L., AND D.A. PATTERSON [1996], *Computer Architecture: A Quantitative Approach* (second ed.), Morgan Kauffman Publishers, Inc., San Francisco, California.

HILL, M.D. [1987], *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. thesis, University of California at Berkeley.

HILL, M.D. [1988], A case for direct-mapped caches, *Computer* **21**, 25–40.

HILL, M.D., AND A.J. SMITH [1984], Experimental evaluation of on-chip microprocessor cache memories, *Proceedings of the 11th Annual International Symposium on Computer Architecture*, Ann Arbor, 158–164.

JOUPPI, N.P. [1990], Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 364–373.

KANDEMIR, M., J. RAMANUJAM, AND A. CHOUDHARY [1999], Improving cache locality by a combination of loop and data transformations, *IEEE Transactions on Computers* **48**, 159–167.

KENNEDY, K., AND K.S. MCKINLEY [1992], Optimizing for parallelism and data locality, *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington D.C., 323–334.

KENNEDY, K., AND K.S. MCKINLEY [1993], Maximizing loop parallelism and improving data locality via loop fusion and distribution, *Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, 301–320.

KOCK, E. DE [1999], *Video Signal Processor Mapping*, Ph.D. thesis, Eindhoven University of Technology.

KORST, J.H.M. [1992], *Periodic Multiprocessor Scheduling*, Ph.D. thesis, Eindhoven University of Technology.

LAM, M.S., E.E. ROTHBERG, AND M.E. WOLF [1991], The cache performance and optimizations of blocked algorithms, *Computer Architecture News* **19**, 63–74.

LEE, E.A., AND D.G. MESSERSCHMITT [1987], Static scheduling of synchronous data flow programs for digital signal processing, *IEEE Transactions on Computers* **C-36**, 24–35.

MANJIKIAN, N., AND T.S. ABDELRAHMAN [1997], Fusion of loops for parallelism and locality, *IEEE Transactions on Parallel and Distributed Systems* **8**, 193–209.

MCKINLEY, K.S., S. CARR, AND C.-W. TSENG [1996], Improving data locality with loop transformations, *ACM Transactions on Programming Languages and Systems* **18**, 424–453.

MEERBERGEN, J.L. VAN, P.E.R. LIPPENS, W.F.J. VERHAEGH, AND A. VAN DER WERF [1995], PHIDEO: high-level synthesis for high throughput applications, *Journal of VLSI Processing* **9**, 89–104.

MOWRY, T.C., M.S. LAM, AND A. GUPTA [1992], Design and evaluation of a compiler algorithm for prefetching, *SIGPLAN Notices* **27**, 62–73.

PALACHARLA, S. [1994], Evaluating stream buffers as a secondary cache replacement, *Proceedings of the 21th Annual International Symposium on Computer Architecture*, Chicago, Illinois, 24–33.

PANDA, P.R., AND N.D. DUTT [1999], Augmenting loop tiling with data alignment for improved cache performance, *IEEE Transactions on Computers* **48**, 142–149.

PAPADIMITRIOU, C.H. [1995], *Computational Complexity*, Addison-Wesley, Reading, Massachusetts.

PAPADIMITRIOU, C.H., AND K. STEIGLITZ [1982], *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs.

PHILBIN, J., J. EDLER, O.J. ANSHUS, C.C. DOUGLAS, AND K. LI [1996], Thread scheduling for cache locality, *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages*, Cambridge, Massachusetts, 60–71.

PINEDO, M. [1995], *Scheduling: Theory, Algorithms, and Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

PRZYBYLSKI, S.A. [1990], *Cache and Memory Design: A Performance-Directed Approach*, Morgan Kauffman Publishers, Inc., San Mateo, California.

PRZYBYLSKI, S.A., M. HOROWITZ, AND J. HENNESSY [1988], Performance tradeoffs in cache design, *Computer Architecture News* **16**, 290–299.

RIEMENS, A.K., R.J. SCHUTTEN, AND K.A. VISSERS [1997], High speed video de-interlacing with a programmable TriMedia VLIW core, *Proceedings of the 1997 International Conference on Signal Processing Applications and Technology*, San Diego, California, 1375–1380.

RIVERA, G., AND C.-W. TSENG [1998], Data transformations for eliminating conflict misses, *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, 38–49.

SCHRIJVER, A. [1986], *Linear and Integer Programming,* Wiley, Chichester.

SCHUTTE, K., AND G.M.P. VAN KEMPEN [1997], Optimal cache usage for separable image processing algorithms on general purpose workstations, *Signal Processing* **59**, 113–122.

SINGHAI, S.K., AND K.S. MCKINLEY [1997], Algorithm for improving parallelism and cache locality, *The Computer Journal* **40**, 340–355.

SLAVENBURG, G., S. RATHNAM, AND H. DIJKSTRA [1996], The Trimedia TM-1 PCI VLIW mediaprocessor, *Eigth Symposium on High-Performance Chips, Hot Chips 8,* Palo Alto, California, 171–177.

SMITH, A.J. [1982], Cache memories, *Computing Surveys* **14**, 473–530.

SMITH, J.E., AND J.R. GOODMAN [1983], A study of instruction cache organizations and replacement policies, *Computer Architecture News* **11**, 132–137.

STROUT, M.M., L. CARTER, J. FERRANTE, AND B. SIMON [1998], Schedule-independent storage mapping for loops, *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages*, San Jose, California, 24–33.

STRUIK, P., P. VAN DER WOLF, AND A.D. PIMENTEL [1998], A combined hardware/software solution for stream prefetching in multimedia applications, *Proceedings of SPIE Multimedia Hardware Architectures 1998,* San Jose, California, 120–130.

TOPHAM, N., AND A. GONZÁLEZ [1999], Randomized cache placement for eliminating conflicts, *IEEE Transactions on Computers* **48**, 185–192.

VERHAEGH, W.F.J. [1995], *Multidimensional Periodic Scheduling,* Ph.D. thesis, Eindhoven University of Technology.

VISSERS, K.A., G. ESSINK, P.H.J. VAN GERWEN, P.J.M. JANSSEN, O. POPP, E. RIDDERSMA, W.J.M. SMITS, AND H.J.M. VEENDRICK [1995], Architecture and programming of two generations video signal processors, *Microprocessing and Microprogramming* **41**, 373–390.

WERF, A. VAN DER, F. BRÜLS, R. KLEIHORST, E. WATERLANDER, M. VERSTRAELEN, AND T. FRIEDRICH [1997], I.McIC: A single-chip MPEG2 video encoder for storage, *Proceedings of the International Solid-State Circuits Conference*, San Francisco, California, 254–255.

WOLF, M.E., AND M.S. LAM [1991], A data locality optimizing algorithm, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, 30–44.

WOLFE, M.J. [1996], *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, California.

ZUCKER, D.F., M.J. FLYNN, AND R.B. LEE [1995], *A Comparison of Hardware Prefetching Techniques for Multimedia Benchmarks*, Technical Report CSL-TR-95-683, Stanford University.

Symbol Index

The numbers refer to the pages of first occurrence.

Sets and matrices

$\mathbb{Z}$	set of integers	23
$\mathbb{Z}_\infty$	set of integers including ∞	78
$\mathbb{N}$	set of non-negative integers	23
$\mathbb{N}_+$	set of positive integers	29
$\mathbb{Q}$	set of rational numbers	112
$\mathcal{P}(X)$	powerset of set X	123
$\mathbf{0}$	zero vector	23
$\mathbf{1}$	one vector	37
$[\,]$	empty vector, empty matrix	23
$\mathbf{O}$	zero matrix	40
$\mathbf{I}$	identity matrix	40

Program graphs

$\mathcal{G} = (V, R, W, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$	program graph	22
V	set of multidimensional periodic operations	23
$u, v, w \in V$	multidimensional periodic operations	23
$R(v)$	set of read ports of operation v	23
$W(v)$	set of write ports of operation v	23
$\boldsymbol{I}(v)$	iterator bound vector of operation v	23
E	set of edges	23
$\boldsymbol{A}(p)$	index matrix of port p	23
$\boldsymbol{b}(p)$	index offset vector of port p	23
$P(v) = R(v) \cup W(v)$	set of all ports of an operation	23
P	set of all ports	23
$p, q, r \in P$	ports	23
$\mathcal{E}(p)$	set of all executions of port p	24
$\mathcal{E}$	set of executions of all ports	24
$e, f, g \in \mathcal{E}$	port executions	29
$\mathcal{I}(v)$	iterator space of operation v	23
$\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k} \in \mathcal{I}(v)$	iterator vector of operation v	23

$\boldsymbol{n}(p,\boldsymbol{i})$	index vector at execution $\boldsymbol{i}$ of port p	24
$\delta(v)$	dimension of operation v	23
$\alpha(p)$	array dimension of port p	24

Schedules

$\sigma = (\tau, \mu)$	schedule	27
$\tau = (\boldsymbol{p}, s)$	time assignment	26
$\boldsymbol{p}$	period vector	26
s	start time	26
$t(v,\boldsymbol{i})$	time of execution $\boldsymbol{i}$ of operation v	26
$t(p,\boldsymbol{i})$	time of execution $\boldsymbol{i}$ of port p	27
$\mu = (\boldsymbol{c}, o)$	address assignment	27
$\boldsymbol{c}$	coefficient vector	27
o	offset	27
$a(p,\boldsymbol{i})$	address of execution $\boldsymbol{i}$ of port p	27
$(p \to q, r)$	address constraint for ports p, q, and r	28

Array clusters

$\mathcal{A}$	set of array clusters	25
$\bowtie$	join	25

Lexicographical properties

$<_{\text{lex}}$	lexicographically smaller than	32
lex	lexicographical execution	32
wlex	weak lexicographical execution	32
lio	lexicographical index ordering	33

Caches

$\mathcal{C} = (c_{\text{s}}, s_{\text{s}}, b_{\text{s}})$	cache	29
c_{s}	cache size in blocks	29
s_{s}	set size in blocks	29
b_{s}	block size in bytes	29
$a_{\text{b}}(a)$	block address of address a	29
$comp(e)$	compulsory miss for port execution e	29
$ex(e)$	expiration miss for port execution e	30
$c_{\mathcal{G},\mathcal{C}}(\sigma)$	cost of schedule σ for cache $\mathcal{C}$ and program graph $\mathcal{G}$	30
$\varphi_{\mathcal{G},\mathcal{C}}(\sigma)$	cache miss rate	30

Cache cost calculation

$r(e)$	reuse length for port execution e	78
R	number of different addresses between j and e	79

$f(e)$	length of filling interval for port execution e	79

Reuse graph

$\mathcal{R} = (U, A, I, \boldsymbol{x}, y)$	approximate reuse graph	85
U	vertices of approximate reuse graph	85
A	edges of approximate reuse graph	85
$I(a)$	executions for edge a	85
$\boldsymbol{x}(a)$	approximate reuse length vector for edge a	85
$y(a)$	approximate reuse length offset for edge a	85
$r_{\mathcal{R}}(p, \boldsymbol{i}) = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{i} + y$	approximate reuse length for $(p, \boldsymbol{i})$	86
$\boldsymbol{R}$	reuse matrix	90
$\boldsymbol{r}$	reuse vector	90
$\rho(p, \boldsymbol{i})$	estimated reuse length for $(p, \boldsymbol{i})$	91

Combinatorial optimisation problems

$(\mathcal{S}, f)$	instance of a combinatorial optimisation problem	123
$\mathcal{S}$	solution set	123
$f(s)$	cost of solution s	123
$\mathcal{N}(s)$	neighbourhood of solution s	123

Notational convention

Proofs that contain manipulation of large expressions are written in a format based on Dijkstra & Scholten [1990]. When we have to prove that $P \Rightarrow R$, we can do this, for instance, by proving $P \equiv Q$ and $Q \Rightarrow R$. Such a proof is written down as follows:

$$\begin{array}{ll} & P \\ \equiv & \{ \text{ hint why } P \equiv Q \ \} \\ & Q \\ \Rightarrow & \{ \text{ hint why } Q \Rightarrow R \ \} \\ & R \end{array}$$

Author Index

Subject Index

Samenvatting

In dit proefschrift behandelen we een meerdimensionaal periodiek planningsprobleem. Dit probleem vindt zijn oorsprong in de videosignaalbewerking. Algoritmen voor de bewerking van videosignalen beschouwen we als verzamelingen operaties die periodiek uitgevoerd moeten worden met zeer hoge frequentie. Daartoe moeten snelle processoren gebruikt worden. De verwerkingssnelheid van deze processoren is in de afgelopen jaren toegenomen met meer dan 50% op jaarbasis. De geheugens waarvan deze processoren gebruik maken zijn jaarlijks slechts 7% sneller geworden, waardoor beide uit de pas gaan lopen. Een standaard oplossing om dit verschil in snelheid te overbruggen is het toevoegen van cachegeheugen tussen de processor en het geheugen. Een cachegeheugen is een snel maar klein buffer dat tussenresultaten van een berekening kan opslaan. De vraag hoe een cachegeheugen optimaal benut wordt kan gezien worden als een planningsprobleem. Het effectief oplossen van dit probleem is echter nog een open vraagstuk.

Het planningsprobleem bestaat eruit om twee toekenningen te vinden, te weten een tijdstoekenning en een geheugentoekenning. De tijdstoekenning representeert de volgorde waarin de operaties uitgevoerd moeten worden. Van deze toekenning eisen we dat operaties in een zodanige volgorde plaatsvinden dat tussenresultaten berekend worden voordat ze worden gebruikt. De geheugentoekenning legt voor ieder tussenresultaat in de berekening een geheugenplaats vast. Van deze toekenning eisen we dat een geheugenplaats waar een tussenresultaat opgeslagen is niet overschreven wordt voordat dit tussenresultaat voor de laatste maal gebruikt is. Verder eisen we van de toekenningen dat het cachegeheugen optimaal gebruikt wordt, hetgeen we vertalen in een minimalisatie van het aantal *cache misses*.

We hebben aangetoond dat dit planningsprobleem formeel lastig is. De lastigheid wordt enerzijds veroorzaakt door de beperkingen die opgelegd worden aan de tijdstoekenning en geheugentoekenning. Anderzijds blijkt het lastig om voor een gegeven tijdstoekenning, geheugentoekenning en cachegeheugen, het aantal *cache misses* uit te rekenen.

Elke tijdstoekenning bepaalt een volgorde waarin de operaties uitgevoerd moeten worden. Helaas kunnen niet alle tijdstoekenningen op een efficiënte wijze uitgevoerd worden worden door een processor. Om de verzameling tijdstoekenningen in te perken tot toekenningen die een efficiënte afbeelding toestaan, leggen we aanvullende beperkingen op.

De kern van het proefschrift wordt gevormd door een discussie over het berekenen van het aantal *cache misses* voor een gegeven tijdstoekenning en geheugentoekenning. Deze berekening doet later dienst als een middel om verschillende toekenningen met elkaar te kunnen vergelijken. Aangezien het bepalen van het aantal *cache misses* een formeel lastig probleem is, beschouwen we hiervoor een benaderingsalgoritme. Daartoe hebben we het probleem in twee delen gesplitst. In het eerste deel proberen we zo goed mogelijk het hergebruik van tussenresultaten te bepalen, hetgeen gemeten wordt in de tijd die verstrijkt tussen twee opeenvolgende momenten waarop een tussenresultaat gebruikt wordt. In het tweede deel benaderen we de vulling van het cachegeheugen. De vulling geeft een tijdspanne aan waarin alle tussenresultaten zich nog in het cachegeheugen bevinden. We laten zien dat op basis van het hergebruik en de vulling het aantal *cache misses* berekend kan worden. Gebaseerd op deze opsplitsing hebben we een benaderingsalgoritme ontworpen.

Het resulterende algoritme voor het benaderen van het aantal *cache misses* hebben we geïmplementeerd en gebruikt voor een aantal experimenten. We hebben laten zien dat we het aantal *cache misses* goed kunnen benaderen in rekentijden die veel kleiner zijn dan de tijden die nodig zijn voor een zogenaamde cachesimulatie.

Tenslotte hebben we een eerste stap gedaan in de richting van een lokaal zoekalgoritme voor het planningsprobleem. De basis voor het algoritme wordt gevormd door het benaderingsalgoritme voor het aantal *cache misses*. De zoekruimte, die nodig is voor lokaal zoeken, wordt opgespannen door veranderingen in de tijdstoekenning en geheugentoekenning. Voor deze veranderingen stellen wij technieken voor die bekend zijn uit de literatuur.

Curriculum Vitae

Ramon Clout was born on November 22, 1970, in Oosterhout, the Netherlands. He attended the Sint-Oelbertgymnasium, in Oosterhout, where he obtained his Gymnasium diploma in 1989. From 1989 to 1994 he studied computing science at Eindhoven University of Technology. He graduated with honours in April 1994, on the subject of off-line scheduling for cache optimisation. His Master's thesis was written under the supervision of M. Rem and F.W. Sijstermans.

After his graduation, he started his work as a Ph.D. student at the Section Parallel Systems at Eindhoven University of Technology. The work was carried out at the Philips Research Laboratories in Eindhoven under the supervision of E.H.L. Aarts and W.F.J. Verhaegh.

Since January 1st, 2000, Ramon works as a research scientist at the Philips Research Laboratories in Eindhoven.

Stellingen

behorende bij het proefschrift

Periodic Scheduling for Cache-Miss Minimisation

van

Ramon Clout

I

Het aantal *cache misses* voor een gegeven programma geschreven in een imperatieve programmeertaal kan niet bepaald worden in een rekentijd die polynomiaal begrensd is in de grootte van de programmatekst, tenzij P = NP.

Dit proefschrift.

II

a. Een *afhankelijkheidstest* voor lustransformaties moet minstens alle data-afhankelijkheden vinden. Voor het tellen van *cache misses* is daarentegen een test nodig die hoogstens alle data-afhankelijkheden vindt.

b. Daarom zou de eerste test eigenlijk *onafhankelijkheidstest* genoemd moeten worden.

Dit proefschrift.

III

Het aantal *instruction cache misses* kan gereduceerd worden door herplaatsing van code in de *object file* gestuurd door *profiling* van het programma.

VERHOEVEN, M.G.A., R.A.W. CLOUT, AND A. AUGUSTEIJN [1998], *Method for reducing the frequency of cache misses in a computer*, European Patent WO9838579.

IV

De kwaliteit van het universitaire programmeeronderwijs aan informaticastudenten laat wereldwijd te wensen over.

GRIES, D. [1996], Eliminating the Chaff – Again, Banquet Speech at Marktoberdorf 1996, *Proceedings of the NATO Advanced Study Institute on Mathematical Methods in Program Development*, Marktoberdorf, Germany.

V

Gezien het dynamische karakter van het *World Wide Web* moet men er in het algemeen van afzien om een *Uniform Resource Locator* (URL) te gebruiken als literatuurreferentie.

VI

Het niet toestaan van matrices waarvan het aantal rijen of kolommen nul is, is onnodig.

STOER, J. AND C. WITZGALL [1970], *Convexity and Optimization in Finite Dimensions I*, Springer-Verlag, Berlin.

NETT, C.N. AND W.M. HADDAD [1993], A system-theoretic appropriate realization of the empty matrix concept, *IEEE Transactions on Automatic Control* **38**, 771–775.

VII

Gezien het grote aantal treinen dat met vertraging rijdt, is het beter om de geplande frequentie op elk traject op te nemen in het spoorboekje dan de geplande vertrektijd.

Nog anderhalf jaar ellende bij de NS, *de Volkskrant*, 23 mei 2001.

VIII

Artikelen die *online* gepubliceerd worden, tellen voor twee.

LAWRENCE, S. [2001], Free online availability substantially increases a paper's impact, *Nature* **411**, 521.

IX

Stellingen behorende bij een proefschrift zijn vaak taalkunstig geconstrueerde meningen.

HERBERGS, B. (ed.) [1995], *De beste stellingen zijn van hout*, TU Delft.

X

Op het universitaire schaakbord zijn aio's de pionnen.