CS812 : High Level Design and Modelling of Digital Systems

Memory Synthesis

Term paper

by

Arun Kejariwal (98172) & Bhuvan Middha (csu98133)

Under the guidance

of

Prof. M. Balakrishnan & Prof. Anshul Kumar

Department of Computer Science and Engineering

Indian Institute of Technology, Delhi

Contents

List of Figures

Chapter 1

Introduction

As new processor families and processor cores begin to push the limits of high performance, the traditional processor-memory gap widens and often becomes the dominant bottleneck in achieving high performance.

Traditionally, much attention has been paid to the role of memory system design in the compiler, architecture, and CAD domains. Many of these techniques, while applicable to some extent, do not fully exploit the optimization opportunities in embedded system design. From an application viewpoint, embedded systems are special-purpose, and so are amenable to aggressive optimization techniques that can fully utilize knowledge of the applications. Whereas many traditional memory-related hardware and software optimizations had to account for variances due to general- purpose applications, memory optimizations for embedded systems can be tailored to suit the expected profile of code and data. Furthermore, from an architectural viewpoint, the embedded system designer pays great attention to the customization of the memory subsystem (both on-chip, as well as off-chip): this leads to many nontraditional memory organizations, with a standard cache hierarchy being only one of many memory architectural options. Finally, from a constraint viewpoint, the embedded system designer needs to meet not only system performance goals, but also has to do this within a power budget (especially for mobile applications), and meet real-time constraints. The system performance should account for not only the processor's speed but also the system bus load to the shared board-level storage units such as main memory and disk. Even the L2 cache is shared in a multiprocessor context. As a result of all this, the memory and bus subsystem costs become a significant contributor to overall system costs, and thus the embedded system designer attempts to minimize memory requirements with the goal of lowering overall system costs.

In this report, we present a variety of optimization techniques for data and memory used in embedded systems. Chapter 2 discusses the issues associated with memory design.

Chapter 3 focuses on several global optimizations that are independent of the target architectural platform, and which, more often than not, always result in improved performance, cost, and power. These optimizations take the form of source-to-source code transformations that precede many traditional compiler and synthesis steps, and which move the design to a superior starting point in the design space exploration of alternative embedded system realizations.

Chapter 4 highlights optimization opportunities in the context of specific memory modules, customized memory architectures, and their use in both a hardware (or behavioral) synthesis context, as well as in a software (or traditional compiler) context. We then discuss optimization techniques for increasingly distant memory structures: SRAM, cache, and DRAM.

Chapter 2

Issues in Memory Design

The three most important metrics guiding optimizations in memory architectures are

- Area
- Power
- Performance

These optimizations can be applied with different levels of granularity, ranging from register files to on-chip memory, data caches, and dynamic memory (DRAM).

Applications like image processing, signal processing are data intensive. In such applications data storage techniques have a direct consequence on performance and power. The hardware parameters of concern in memory design are as follows

- Number of distributed registers
- Number of register files
- Number of register file ports
- On-chip cache or off-chip memory

- Cache parameters
  1. Cache size, Cache line size
  2. Cache associativity
  3. Split vs Unified Cache
- Cache vs Scratch Pad
- Victim Caches
- Number of memory ports
- Memory Bus Bandwidth

The basic design flow is given in figure 2.1.

Application specific optimizations boost the performance and the power

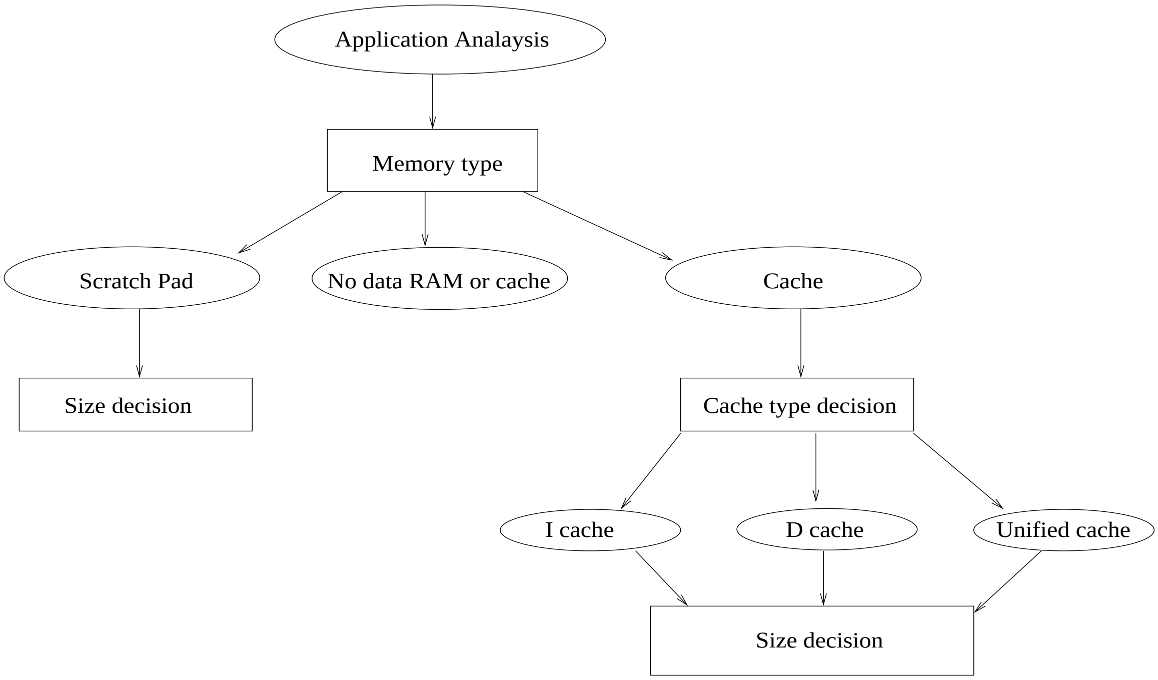

Figure 2.1: Decision flow of memory synthesis

sector a great deal. A detailed analysis of these is carried out in Chapter 4.

Chapter 3

Code Transformations

The role of source-to-source code transformations in the solution to the data transfer and storage bottleneck problem especially important for embedded applications since performance is not the only goal; cost issues such as memory footprint and power consumption are also crucial.

Code-transformations carried out in platform-independent order allow us to apply them to a given application code without having any prior knowledge of parameters such as memory sizes, communication scheme, and datapath type. The resulting optimized code can then be passed through a platform-dependent stage to obtain further cost-performance improvements and trade-offs.

For example, consider the following code

```
for (i=0 ; i<N; ++i)
  for (j=0 ; j<=N-L ; ++j)
    b[i][j] = 0;
for (i=0; i<N; ++i)
  for (j=0; j<= N-L; ++j)
    for (k=0; k<L; ++k)
      b[i][j] += a[i][j+k];
```

The code if implemented directly would result in high storage and bandwidth requirements (assuming that N is large), since all entries in the array b have to be written to an off-chip background memory in the first loop and read back in the second loop. Rewriting the code using a loop-merging transformation, gives the following:

```
for (i=0 ; i<N; ++i)
  for (j=0 ; j<=N-L ; ++j){
    b[i][j] = 0;
    for (k=0; k<L; ++k)
      b[i][j] += a[i][j+k];
  }
```

Here the entries in the array b can be stored in registers up to the end of the accumulation, since they are immediately consumed after they have been produced. This reduces memory bandwidth requirements significantly, since L is typically small. hence, we observe that such platform independent optimizations can increase the performance significantly.

Various techniques have been developed to minimize the memory requirements like

- **Clustering** of scalar variables into off-chip memory pages.
- **Loop Transformations**
    1. Loop Interchange
    2. Loop Fusion
    3. Loop Splitting
    4. Loop Unrolling
    5. Loop Blocking(or Tiling)
- **Reordering** such as Read-Modify-Write
- **Hoisting**

3.1 Clustering of scalar variables

Scalar variables are normally assigned to on-chip registers. However, if the number of such variables is large, it might be necessary to store these variables in memory. The variables stored in off-chip memory have to be assigned memory address. Clustering groups the variables into clusters of size P (to reside in the same memory page), so that the number of accesses to the same memory page is maximised. This achieved by carrying out clique-partitioning, where each node in the graph corresponds to a variable and the edge weight is directly related to the frequencies of its terminal nodes.

3.2 Loop Transformations

3.2.1 Loop Interchange

Consider the following

```
int a[N][N]
.
.
.

for i = 0 to N - 1
  for j = 0 to N - 1
    a[j][i] = a[j][i] + 1;
```

(a) Original Loop

```
int a[N][N]
.
.
.

for j = 0 to N - 1
  for i = 0 to N - 1
    a[j][i] = a[j][i] + 1;
```

(b) Transformed Loop

We observe that the original code(assuming row-major data storage) exibhits poor spatial locality because the inner loop access elements along one column of the array, and consecutive elements do not occur in array. In contrast to above, the transformed loop has better cache performance as it ensures that the elements along the same row are accessed in the inner loop.

3.2.2 Loop Fusion

Consider the following

```
for i = 0 to 1000 step 1          for i = 0 to 1000 step 1
  b[i] = a[i] + 1;                b[i] = a[i] + 1;
for i = 0 to 1000 step 1          c[i] = a[i]*3;
  c[i] = a[i]*3;

(a) Original Loop                 (b) Transformed Loop
```

Here we observe that the transformation improves cache performance by preventing multiple occurences of cache misses to the same elements of the same array a. Further, a[i] can be register allocated , therby eliminating the second memory access to array a in the loop.

3.2.3 Loop Splitting

Loop splitting increases the total number of operations slightly due to the repeated index increments and comparisons, but the overall time saved by exploiting page mode operations larger.

```
for (i=0; i<N; i++)               for (i=0; i<N; i++)
  A[i] = 0                          A[i] = 0
  B[i] = 0                        end for
end for                           for (i=0; i<N; i++)
                                    B[i] = 0
                                  end for

Original code                         Split code
```

In the above example, we observe the page made access can be effectively used by splitting the write inti arrays anad b into separate loops.

3.2.4 Loop Unrolling

Loop Unrolling improves cache performance by reducing the number of accesses to the cache.

```
for i = 0 to 99 step 1
  b[i] = a[i] + a[i+1];


(a) Original Loop
```

```
for i = 0 to 99 step 1
  b[i] = a[i] + a[i+1];
  b[i+1] = a[i+1] + a[i+2];

(b) Transformed Loop
```

In the above example we note that a[i+1] is accessed twice in the rolled loop body, it can be register allocated, thereby saving an access to the cache. Unrolling also reduces the control overhead of the loop.

3.2.5 Loop Blocking

Loop Blocking(Tiling) is a compiler transformation that divides the iteration space into blocks of tiles to ensure that the data set involved in a computation fits unto the data cache. The motivation of the transformation is to reduce capacity misses resulting due to large array sizes. Stratergies for selection of tile size are discussed in Chapter 4.

3.3 Reordering

It aims at exploiting memory access modes like R-M-W(Read-Modify-Write). The R-M-W mode is illustrated in figure 3.1. The schedule requires 10 cycles. If the operation were to be modeled as separate read and write cycles it would have consumed 2x7 = 14 cycles.

The possibility of utilizing R-M-W mode can be identified by starting from each "Read(x)" memory accesss (whee x is a scalr variable or an array element), and proceeding down the DFG to find a path to the next "Write(x)" access, if it exists. If multiple path exist, we combine them all together and

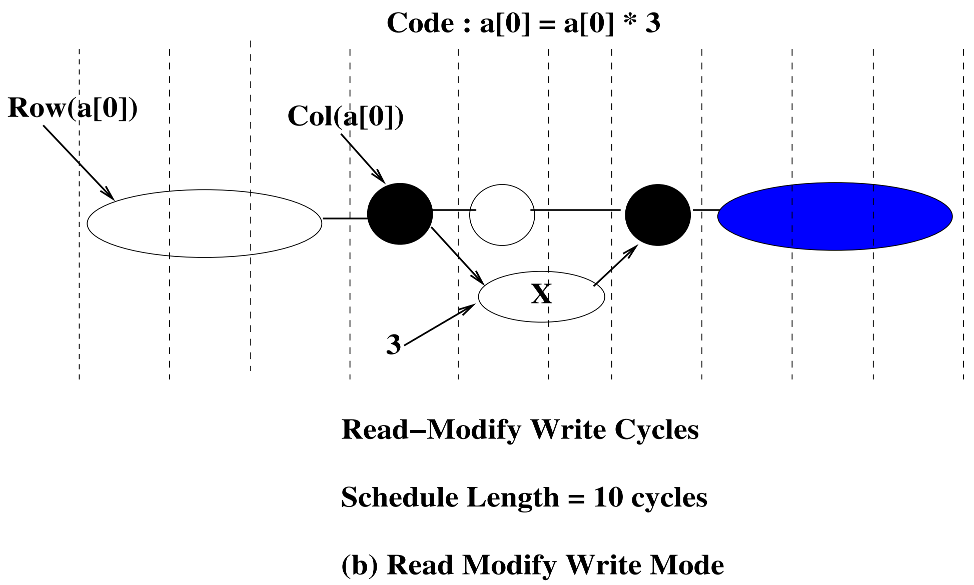

Figure 3.1: Read-Modify-Write mode

still call together a "path". However, there could be conflicting R-M-W paths that cannot be simultaneously satisfied. In such a case we select a maximally non-intersecting subset from a set of possible n different "Read(x) - Write(x)" paths.

3.4 Hoisting

Due to time-multiplexing of the memory address bus between row and column address, a scheduling optimization is possible when two addresses in the same memory page are accessed from different paths of a conditional branch. Consider the following sample code

```
p = d * 2
c = p + 1
if (c>0)
   y = a[0]
else
   y = a[1]
```

```
endif
```

Assume that variables p,c,d and y are stored in on-chip registers, whereas array a is stored in off-chip memory. Either a[0] or a[1] is fetched from memory, depending on the result of the conditional evaluation (c¿0). A simple schedule using the 3-stage read cycle model given in Figure 3.2. and the assumption

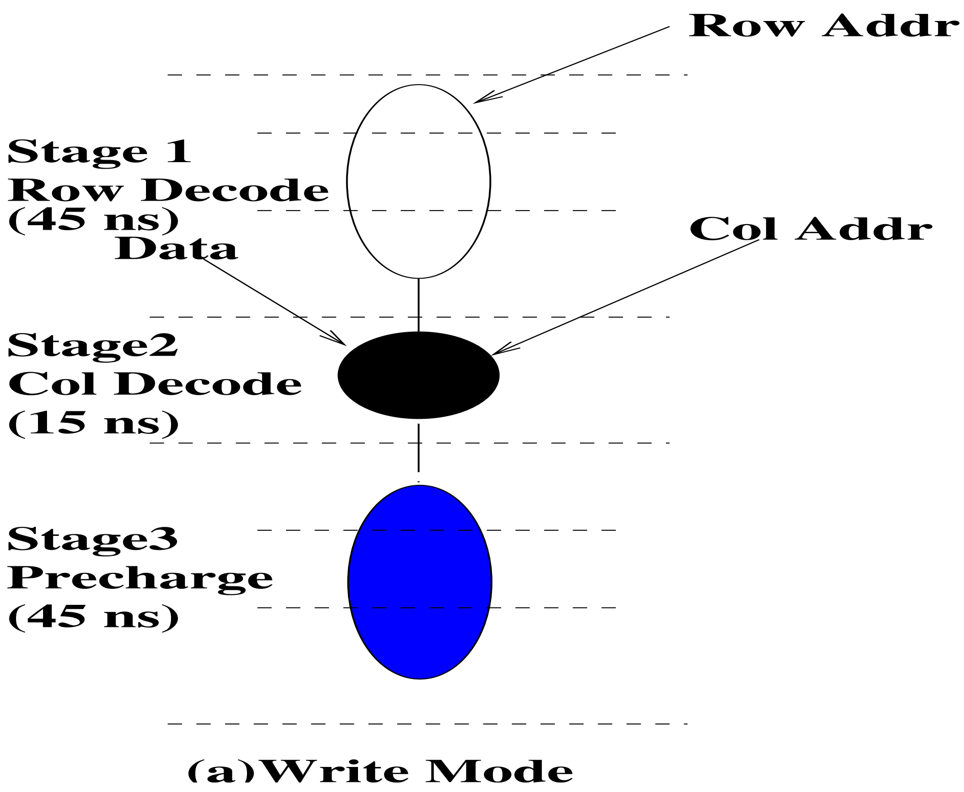

Figure 3.2: 3-stage model of memory access

that +,* and ¿ operations require 1 cycle, results in a schedule of length of 10 cycles. However, the knowledge that a[0] and a[1] reside in the same memory page (memory address assignments are statically computed) allow us to infer that both have the same row address, and hence the read cycle could be initiated before the comparison operation. The schedule resulting from this optimization results in a length of 7 cycles only. The original and the transformed DFG are given in Figure 3.3.

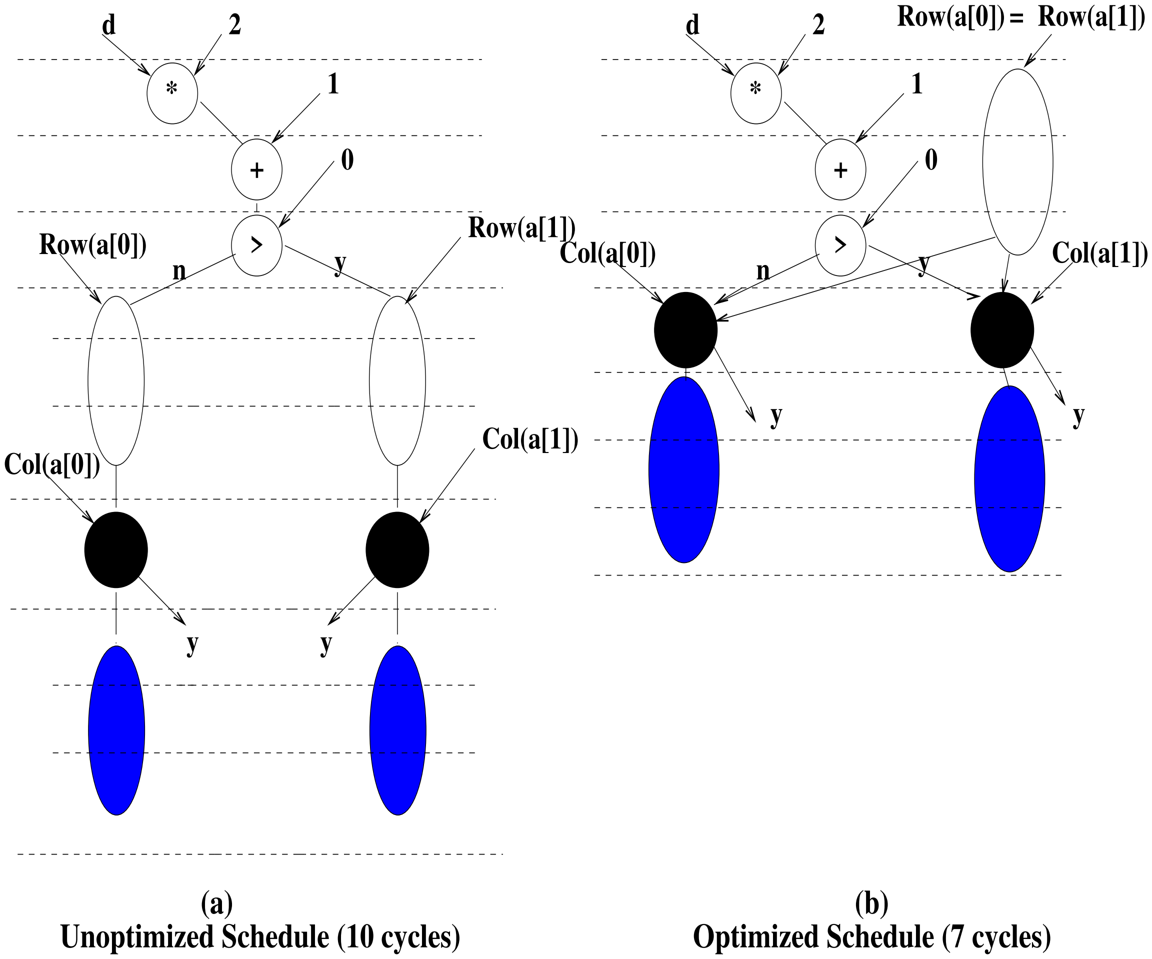

Figure 3.3: DFG transformation due to Hoisting

Chapter 4

Memory Customizaton and Optimization

Various techniques have been developed to minimize the memory requirements like

- Data Organization
- Allocating scalar variables to single and multiport memories
- Memory assignment of array variables
- Memory Banking
- Scratch Pad Memory
- Register Allocation by Graph Coloring

4.1 Data organisation

Data organisation can be studied for structured data types and arrays. In this we discuss the data organisation for the arrays which are frequently found in data intensive applications like image processing and digital signal processing.

Mapping of arrays is done to memory when the appilcation program is being executed. Effective mapping of arrays into memory can be done in three different ways. They are

- Row major
- Column major
- Tile based

In row major form, the elements of the first row are placed in consecutive memory locations in order of increasing column index. For an arbitrary element u[i][j] in an address generator the index expression for generating the logical address is (A + (i x MAX) +j)

In column major mapping, the elements of two dimensional array are stored column by column. The index expression is (A + (j x MAX) +i)

In tile based mapping style it first partitions array into tiles of smaller rectangles. Here, the elements of a row (or column) of tiles arestored in consecutive memory locations, with the tiles themselves being stored in either row or column major format. The index expression is (A + (i x MAX) + (N x j) - (i mod N) x (MAX -1)). The above is illustrated by the example in Figure 4.1

4.2 Memory assignment of array variables

The organization of the array variables in memory has a direct affect on the number data cache conflicts when the arrays are accessed at run time. The memory organization problem for arrays is resolved by first constructing the *Interference Graph* among the arrays in the code, and then assigning memory addresses to each array by minimizing the possibility of cache conflicts with other arrays in the code. Consider the example in figure 4.2

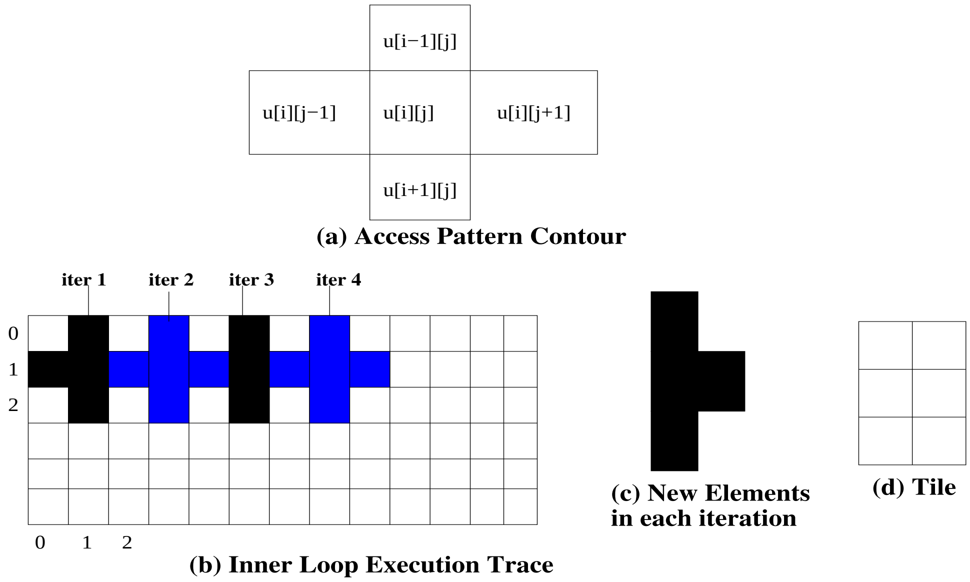

Figure 4.1: Tile arrangement

We observe that padding the arrays with empty slots (which can be used for scalar variables) helps to get over the conflict problem. Potential for reuse of the memory location is determined by carrying out the lifetime analysis of the arrays.

4.3 Memory Banking

The core memory is divided into multiple banks, each with its own independent page buffer, so that two separate memory pages can be simultaneously active in the multiple page buffers. Memory bank assignment is performed by creating an interference graph between arrays and partitioning it into subgraphs so that data in each part is assigned to a different memory bank.

If we have a loop that in succession accesses data from three large arrays A, B, and C, each of which is much larger than a page, then each memory access leads to a fresh page being read from the storage, effectively cancelling the benefits of the page buffer. This page buffer interference problem cannot

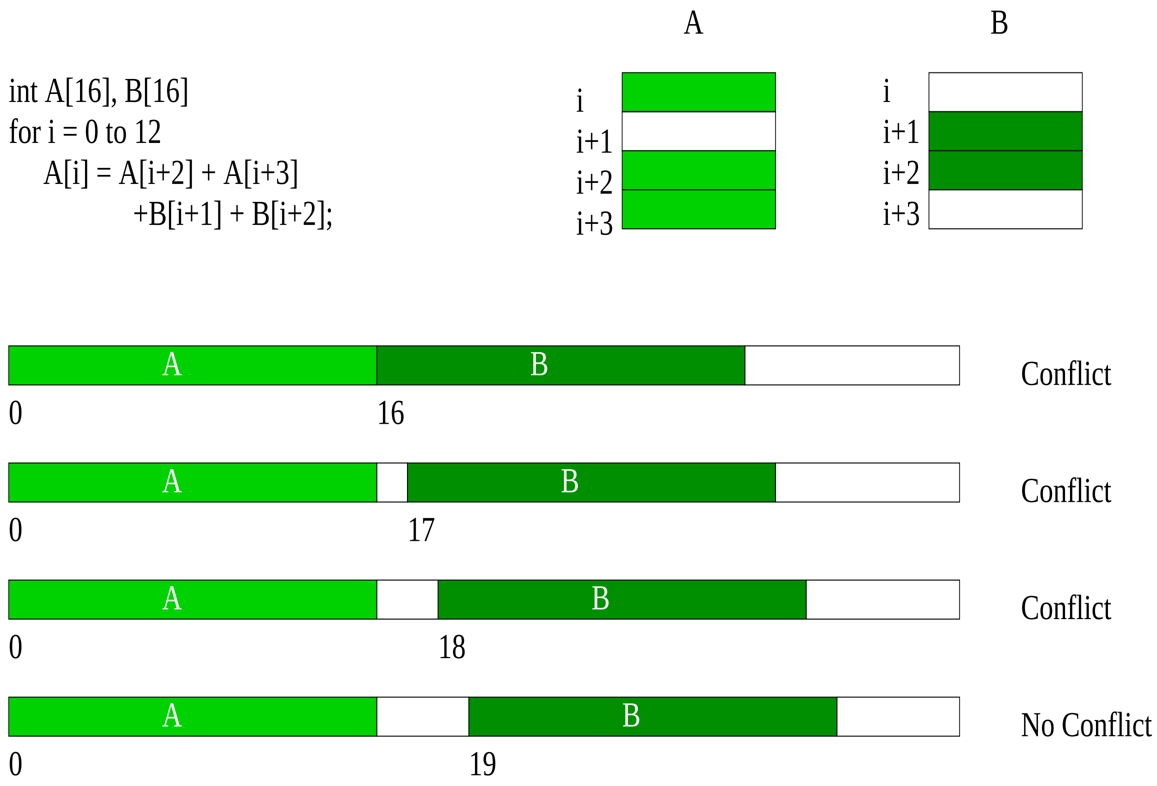

Figure 4.2: Padding of arrays

be avoided if a fixed architecture DRAM is used. However, the arrays are assigned to separate banks as shown in Figure 4.3 each with its own private page buffer, there is no interference between the arrays and the memory accesses do not represent a bottleneck.

4.4 Scratch Pad Memory

Partitioning of program variables into Sratch-Pad memory and DRAM memory aims at minimization of the interference between different variables in the data cache. The features affecting the partitioning process are

1. Scalar variables and constants

2. Size of arrays

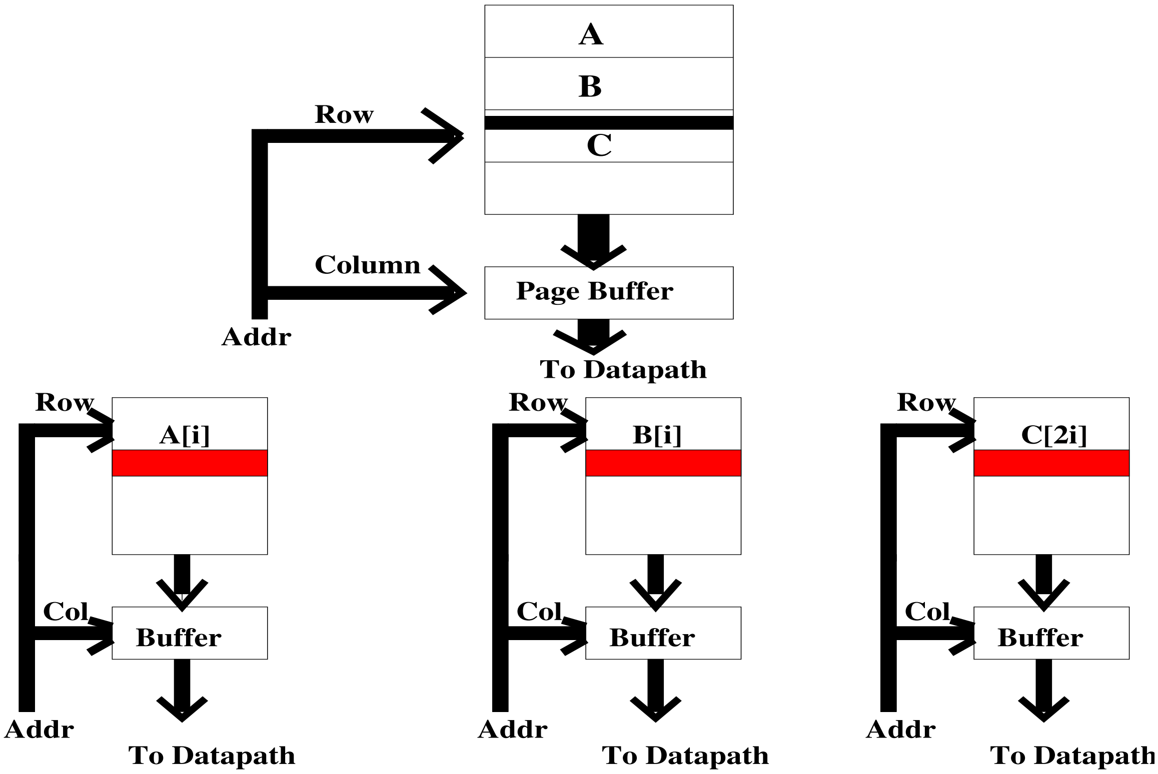

Figure 4.3: Memory Banking Arrangement

3. Life-times of variables

4. Access frequency of variables

5. Conflicts in groups

Figure 4.4 shows the position of Scratch-Pad memory in the hierarchy. In applications where context switch between different programs is required, the SRAM is partitioned(assuming threaded execution) among the different programs. If the execution order of the programs is known to be sequentialthen the entire SRAM is available to the entire program.

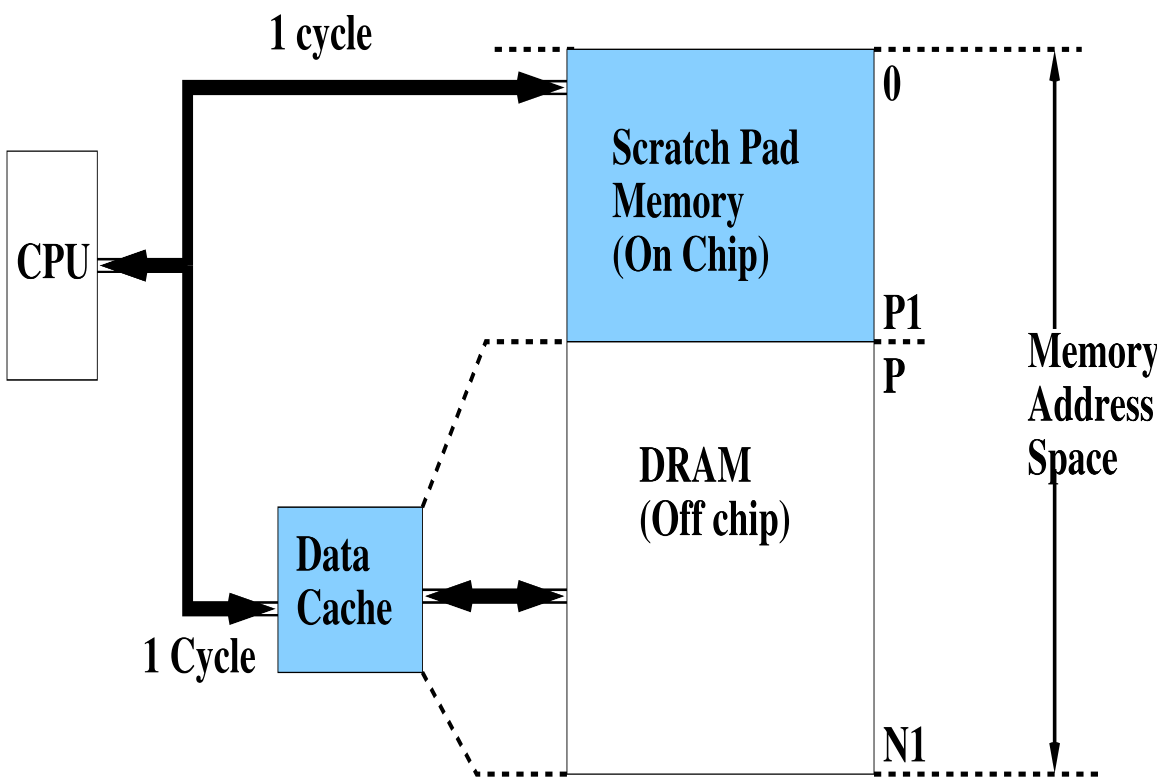

Figure 4.4: Scratchpad Arrangement

Bibliography

[1] P.R.Panda et.al. *Memory Issues in Embedded Systems-on-chip* Kluwer Academic Publication 1999

[2] Krovski et. al. *Synthesis of memory intensive systems-0n-chip* IEEE CAD Sep. 99.

[3] Li et. al *Hardware-software cosynthesis with memory heirarchies* IEEE CAD Oct. 1999.

[4] P.R.Panda, N. D. Dutt : *Behavioral array mapping into multiport memories targeting low power* In Proceedings of the 10th International Conference on VLSI Design, Hyderabad, India 1997.