Embedded Memory Systems

16 / 03 / 2022

Motivation

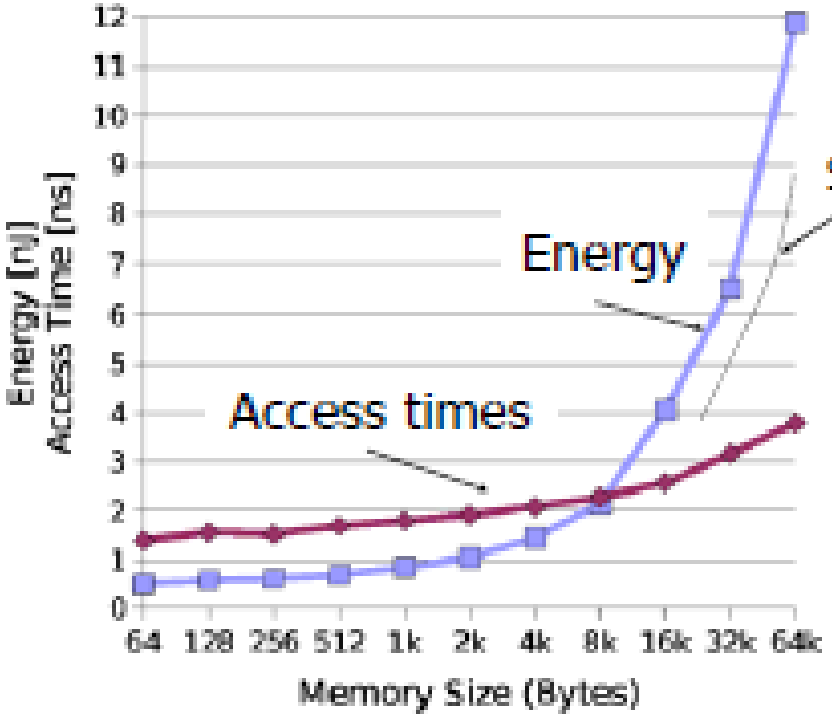
- Memories are the limiting performance factor
  - System-on-Chip memories and SRAMs embedded in FPGAs are fast (1-2 cycles access) but:
    - On-chip memory might not be enough
    - eDRAM or eFLASH may be coming into the picture
- Memories are a key energy consumer on SoC
- Memory systems in embedded systems can be customised
  - Large design space to exploit for optimisation
  - SoC and FPGA technologies support to a good extent "irregular" memory systems

Importance of Memory in SoCs

Some rough rule-of-thumb figures:

- Area
  - 50-70% of chip area may be memory
- Performance
  - 10-90% of system performance may be memory related
- Power
  - 25-40% of system power may be memory related

Things are getting worse

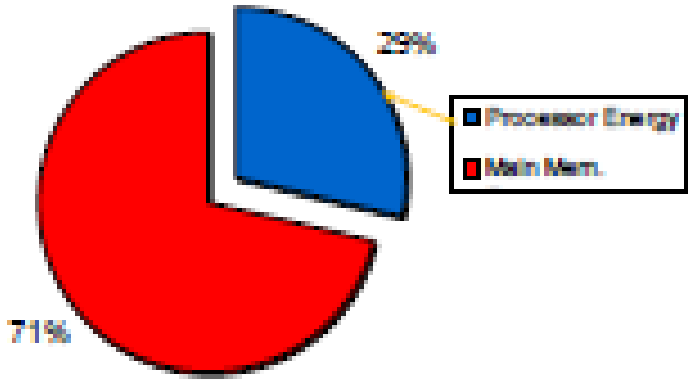


- Applications are getting larger and larger
- The energy cost of keeping access times low is very high

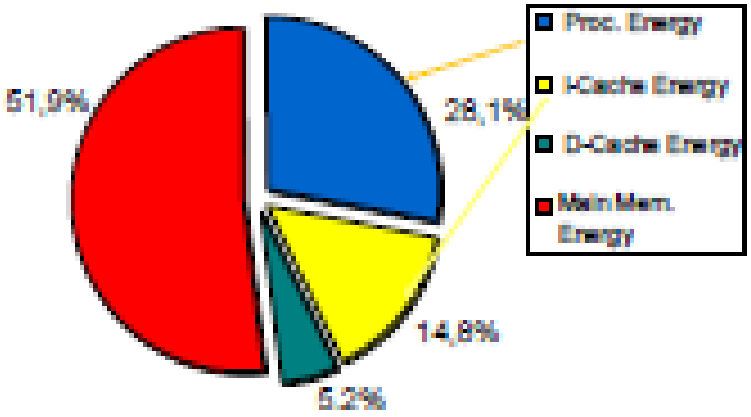
Source: Marwedel, © 2007

Recent Estimations

Cacheless monoprocessor



Multiprocessor with I and D caches



Average of over 200 benchmarks

Outline

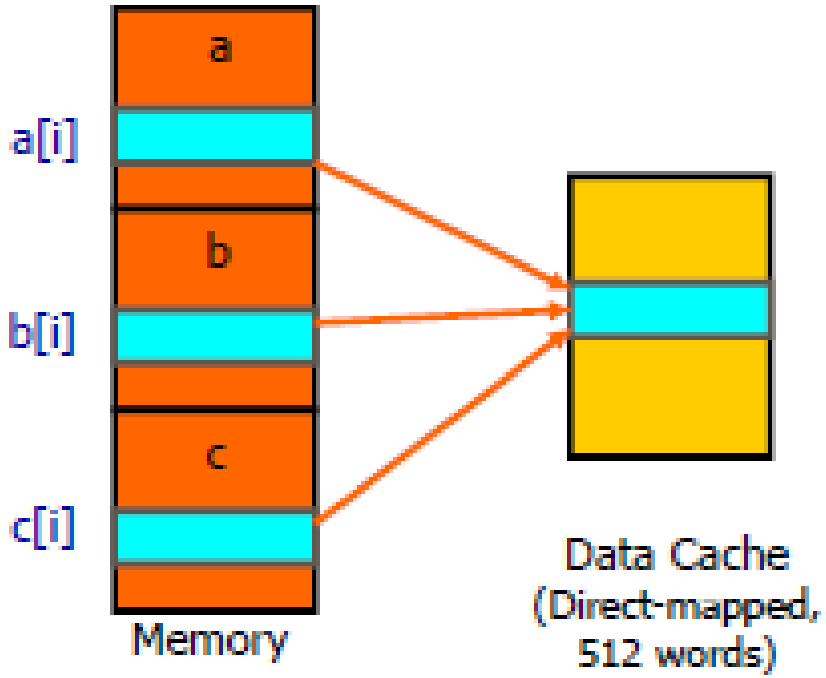
- Memory data layout
- Scratchpad Memory
- Custom memory architectures

Memory Data Layout

- Problem statement
  - Optimise the placement of data in memory to maximise cache effectiveness with minimal resources
- Normally compilers place data following language conventions and program order
  - Can they do any better if they see the complete embedded application?
- Similarity
  - This reminds of the study of placement for DSP variables (but problem and strategy was different)

Array Layout and Data Cache

```
int a[1024];
int b[1024];
int c[1024];
...
for (i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```



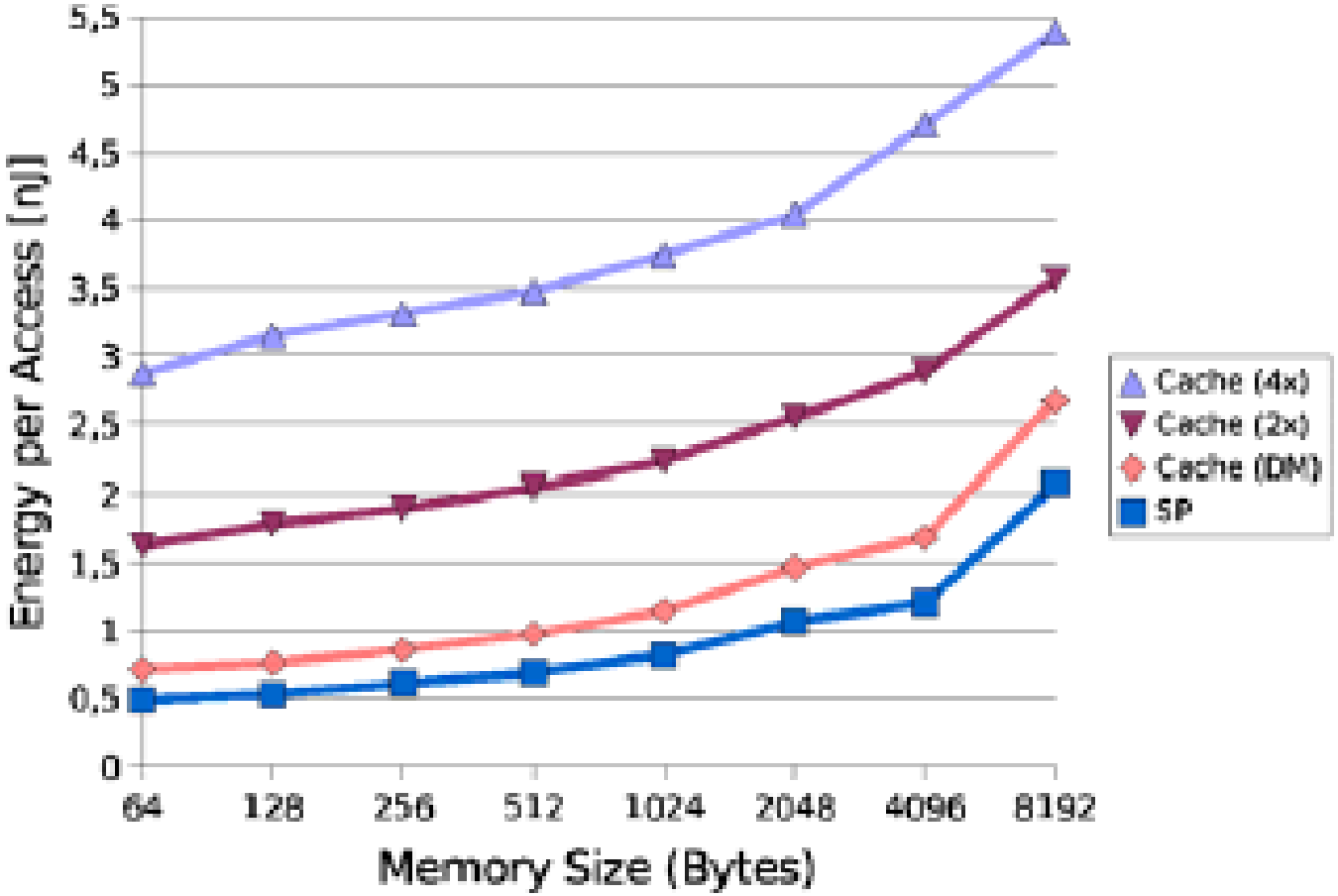
Problem: Every access leads to cache miss!

Aliasing Example

- Cache size → C, line size → M, array size → N
- Addresses and cache position:
  - a[i]: i → (i mod C) / M
  - b[i]: i + N → ((i + N) mod C) / M
  - c[i]: i + 2N → ((i + 2N) mod C) / M
- If N = kC → all cache positions identical!
  - C is normally a power of 2
  - N is **often** a power of 2, too...
- Solutions?
  - Set-associative cache
  - Make C larger than N

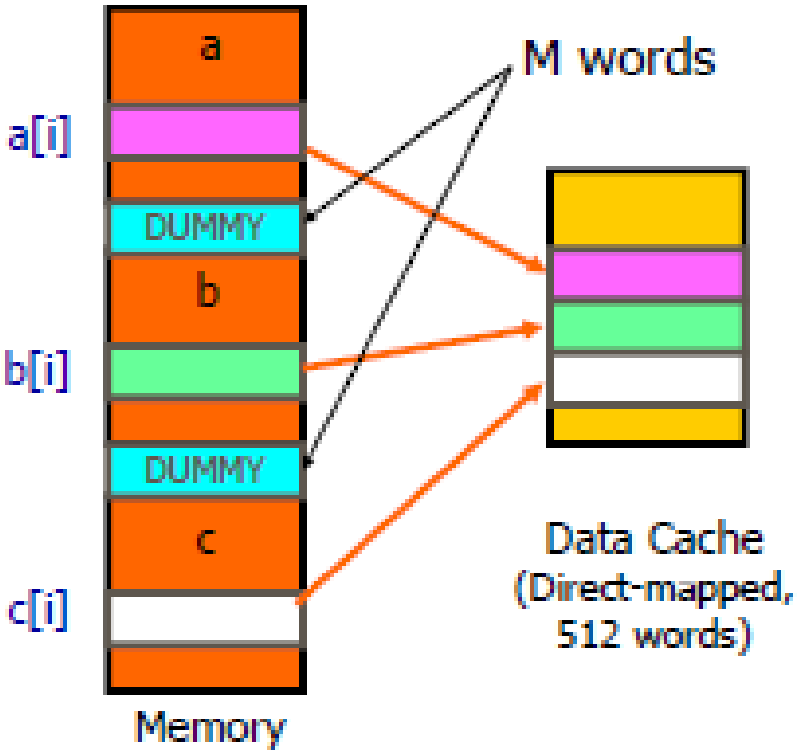
} Costly!

Energy cost of Associativity



A solution: Array Padding

```
int a[1024];
int b[1024];
int c[1024];
...
for (i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```



Data alignment avoids cache conflicts

Transformation: Loop Blocking

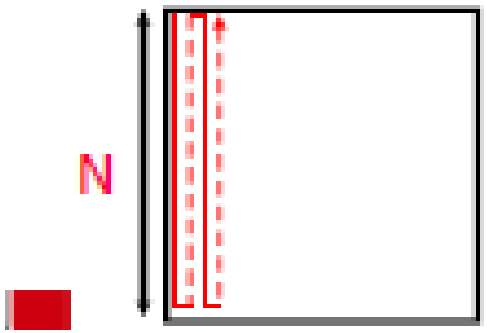
- Modify loop exploration space in blocks (or "tiles") so that all elements accessed at once fit the cache

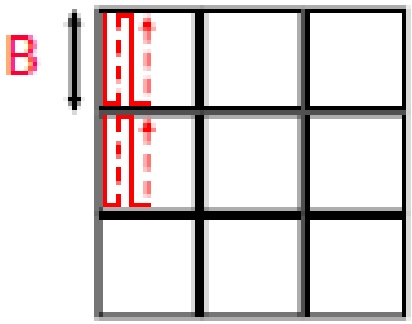
Original Code

```
for i = 1 to N
  for k = 1 to N
    r = X [i,k]
    for j = 1 to N
      Z[i,j] = r * Y[k,j]
```

Blocked Code

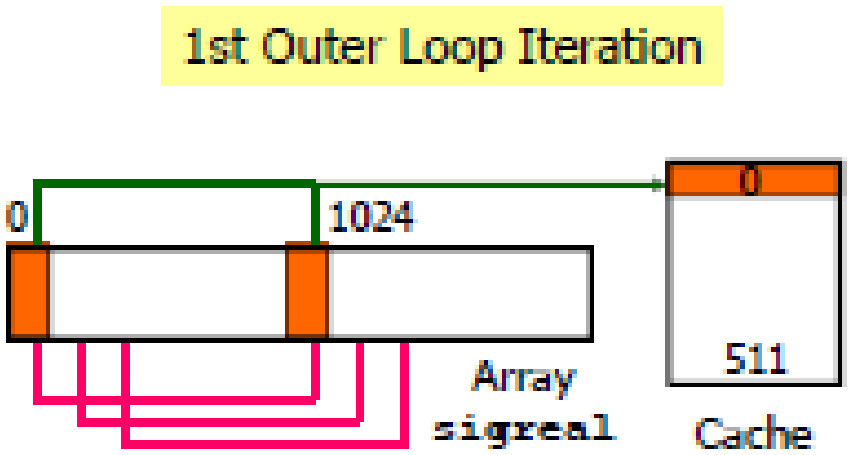
```
for kk = 1 to N step B
  for jj = 1 to N step B
    for i = 1 to N
      for k = kk to min (kk+B-1, N)
        r = X [i,k]
        for j = jj to min (jj+B-1, N)
          Z[i,j] = r * Y[k,j]
```





Real World Example: FFT

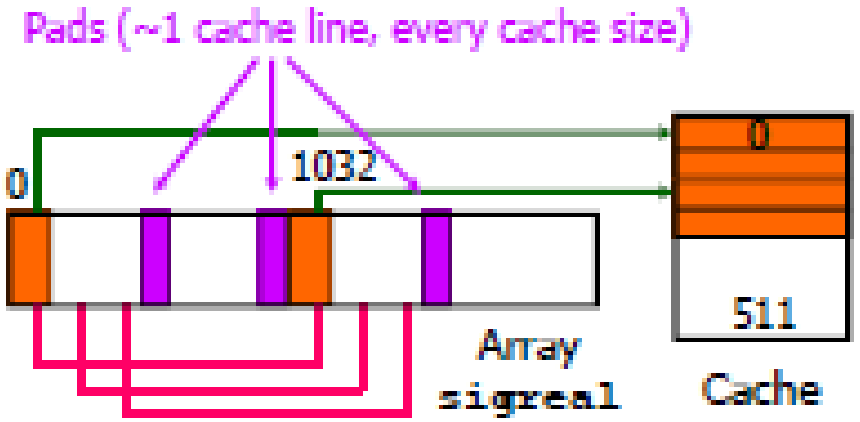
```
double sigreal[2048]
…
le = le / 2
for (i = j; i < 2048; i += 2*le)
   {
   … = sigreal[i]
   … = sigreal[i + le]
   …
   sigreal[i] = …
   sigreal[i + le] = …
}
```



Padded FFT

```
double sigreal[2048 + 16]
…
le = le / 2; le' = le + le / 128
for (i = j; i < 2048; i += 2*le) {
   i' = i + i / 128
   … = sigreal[i']
   … = sigreal[i' + le']
   …
   sigreal[i'] = …
   sigreal[i' + le'] = …
}
```

1st Outer Loop Iteration

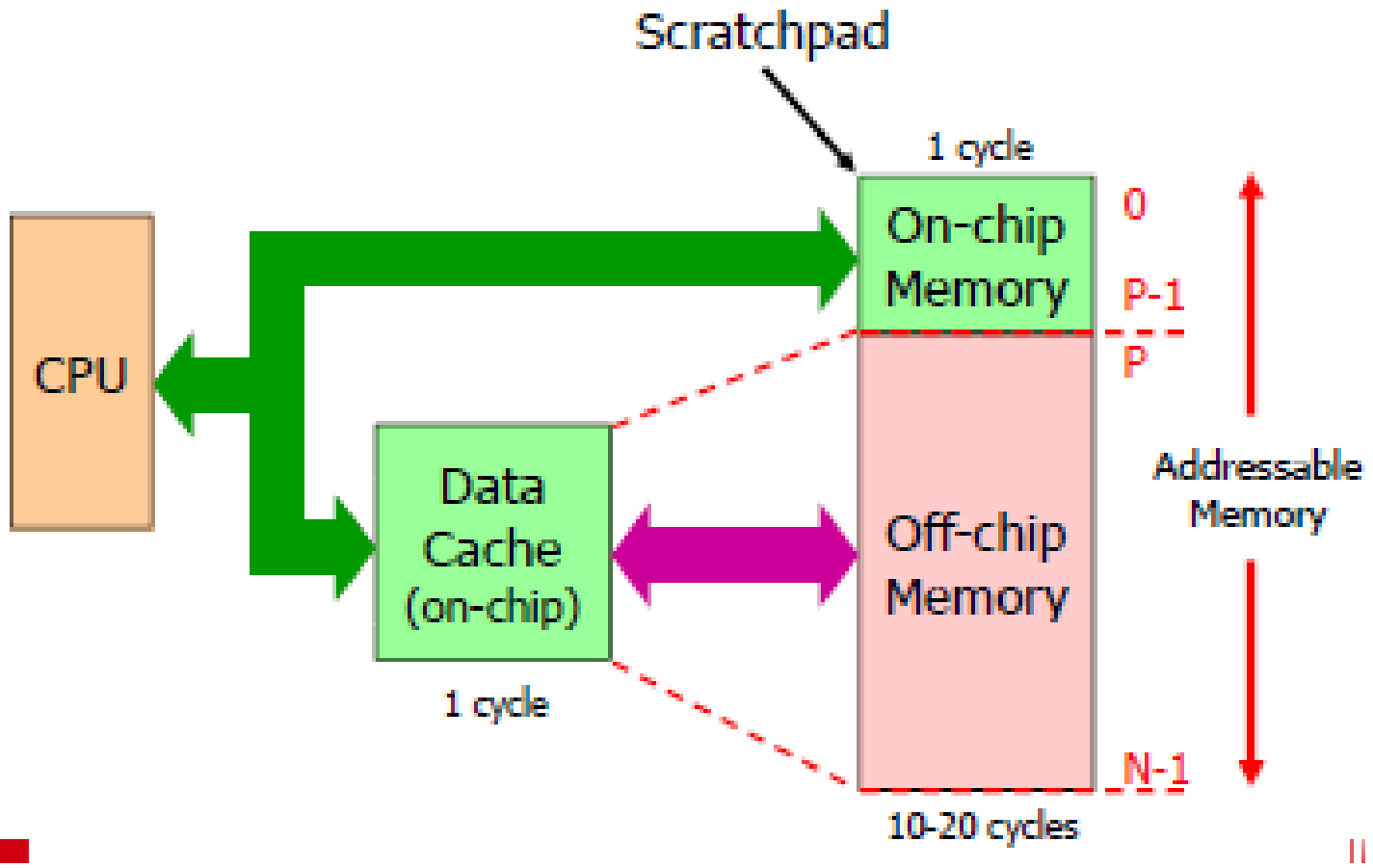


Algorithms to decide data layout

Need to make the following decisions:

- Tile Size Computation
  - Largest possible tiles such that working set fits the cache
- Pad Size Computation
  - Minimum pad size which eliminates aliasing
- Interleaving of Tiled Arrays
  - Arrangement of multiple arrays so that there is no aliasing among arrays and all working sets fit the cache

Scratchpad Memory

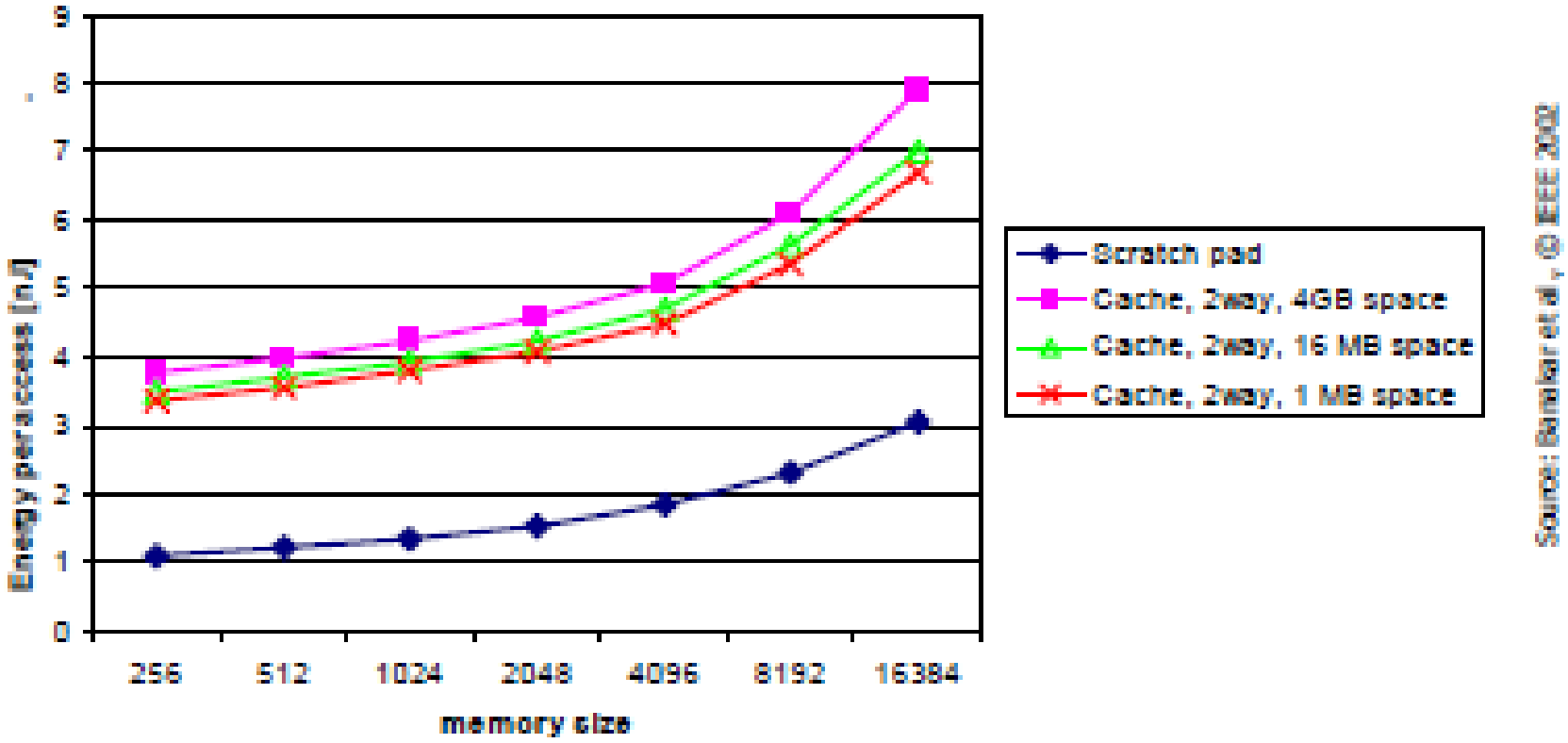


Scratchpad Memory Advantages

"Architecturally visible static software-managed cache"

- Avoid aliasing problems
- Decide explicitly which data will be reused
  - Avoid evicting useful data from the cache
- Increase determinism
  - Data are always "in the cache" when needed
- Save power
  - Avoid energy cost of high associativity
  - Avoid caching irrelevant data
  - Avoid using 2 levels of the memory system for temporaries

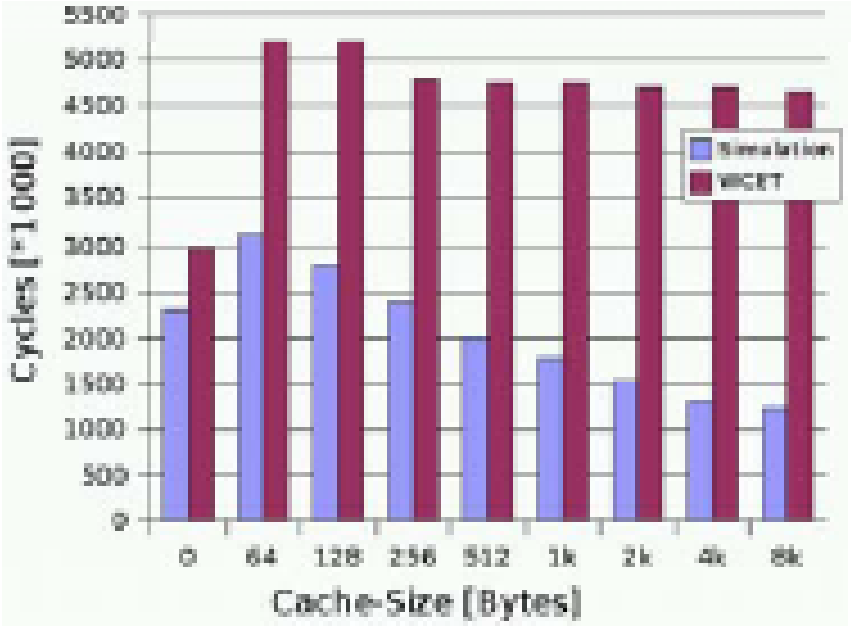
Energy Cost


Energy per access [nJ]
0
1
2
3
4
5
6
7
8
9
256
512
1024
2048
4096
8192
16384
memory size
Scratch pad
Cache, 2way, 4GB space
Cache, 2way, 16 MB space
Cache, 2way, 1 MB space
Source: Banakar et al., © IEEE 2002

Timing Predictability

Many embedded systems are real-time systems: computations must be finished in a given amount of time

Most memory hierarchies (i.e., caches) for PC-like systems designed for good average case, not for good worst case behavior



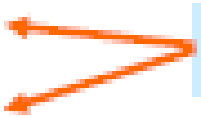
Worst case execution time (WCET) larger than without cache

G.721 using unified cache on a ARM7TDMI

Source: Marwedel, © 2007

Scratchpad Memory

- Embedded processor-based system
  - Processor core
  - Embedded memory
    - Instruction and Data Cache
    - Embedded SRAM
    - Embedded DRAM



Scratch Pad Memory

- **Design problems**
  1. How much on-chip memory?
  2. Partitioning of on-chip memory in cache and scratchpad?
  3. Which variables/arrays in the scratchpad?
- Goals
  - Improve performance
  - Save power

Data Partitioning

```
procedure Histogram_Evaluation

char   BrightnessLevel[512][512];
int    Hist [256];

for (i = 0; i < 512; i++)
       for (j = 0; j < 512; j++) {
              /* for each pixel (i,j) in image */

              level = BrightnessLevel[i][j];

              Hist[level] += 1;
       }
```

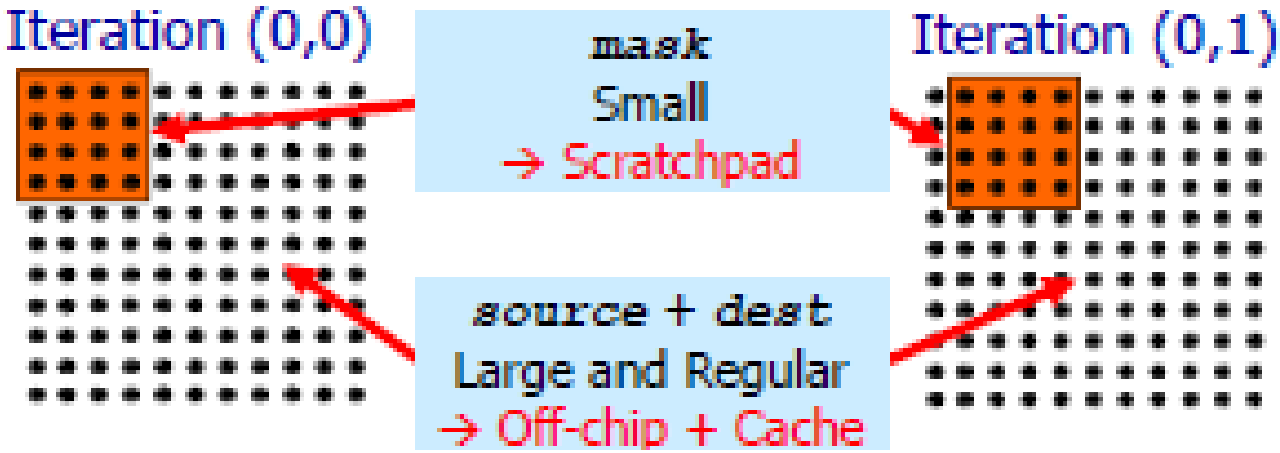
Regular Access
→ Off-chip + Cache

Irregular Access
→ Scratchpad

Data Partitioning 2

```
procedure Convolution

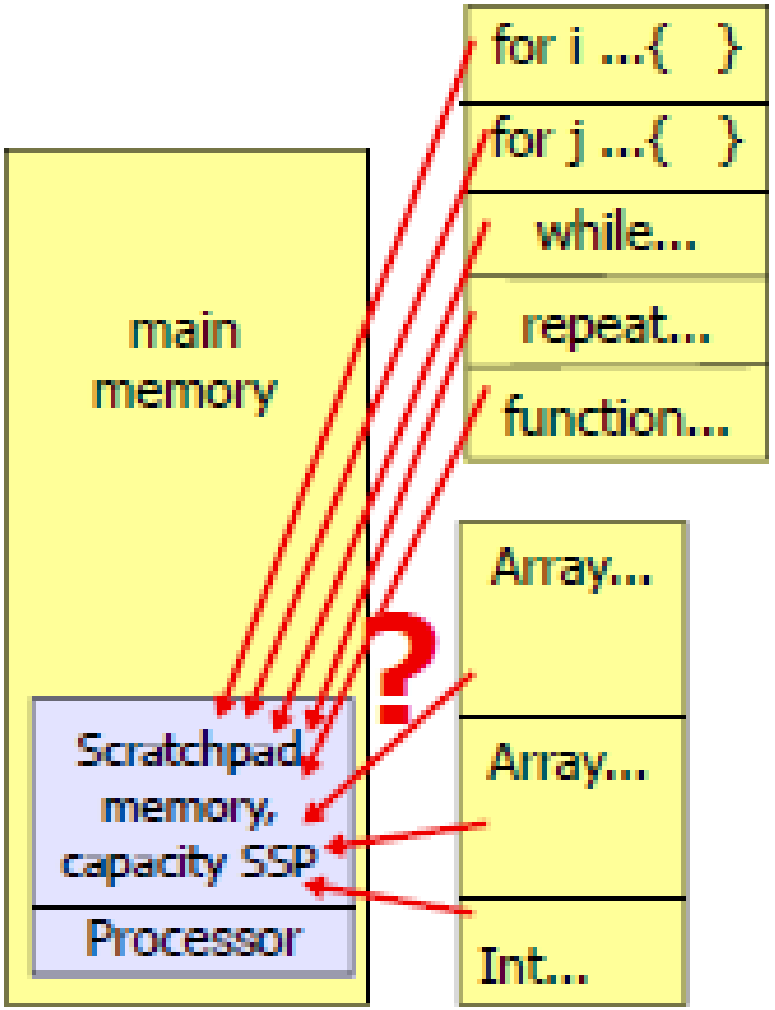
int source[128][128], dest[128][128];
int mask[4][4];
for (all points x,y of source)
  new = 0;
  for (i scanning the mask horizontally)
    for (j scanning the mask vertically)
      new += source[x+i][y+j] * mask[i,j];
  dest[x][y] = new / norm;
```



Data Partitioning

- Pre-Partitioning Scratchpad/Off-chip
  - Scalar variables and constants to scratchpad
  - Large arrays to off-chip memory
- Detailed Partitioning
  - Identify "critical" data for scratchpad
  - Criteria:
    - Life-times of arrays
    - Access frequency of arrays
    - Loop conflicts
- Similar reasoning can be applied to code

Global Placement



Which object (array, loop, etc.) to be stored in a scratchpad?

Non-overlaying allocation

Gain g_k + size s_k for each object k

Maximise gain $G = \Sigma g_k$ respecting the scratchpad size $SSP \geq \Sigma\ s_k$

Solution: knapsack algorithm

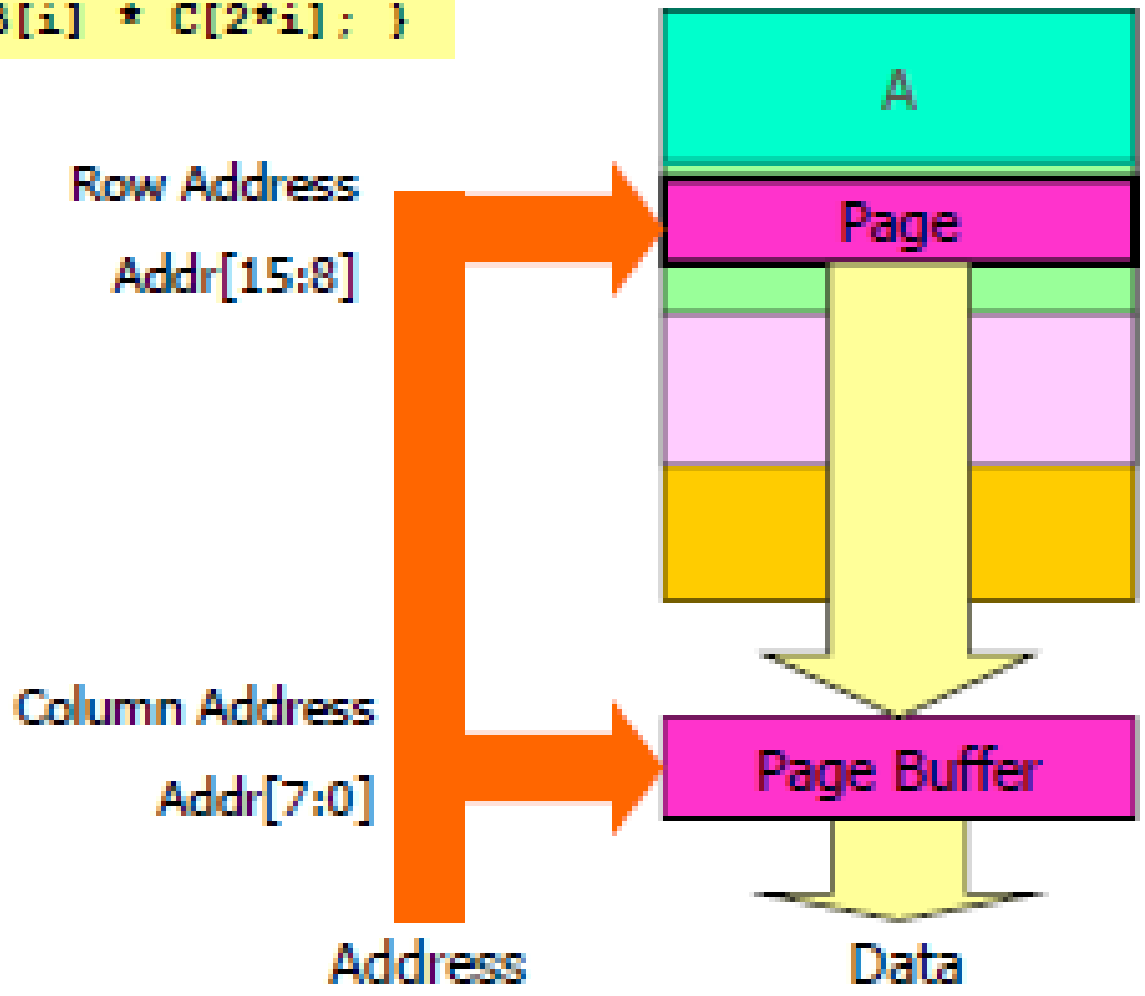
Overlaying allocation

Moving objects back and forth between hierarchy levels

Solution: more complex...

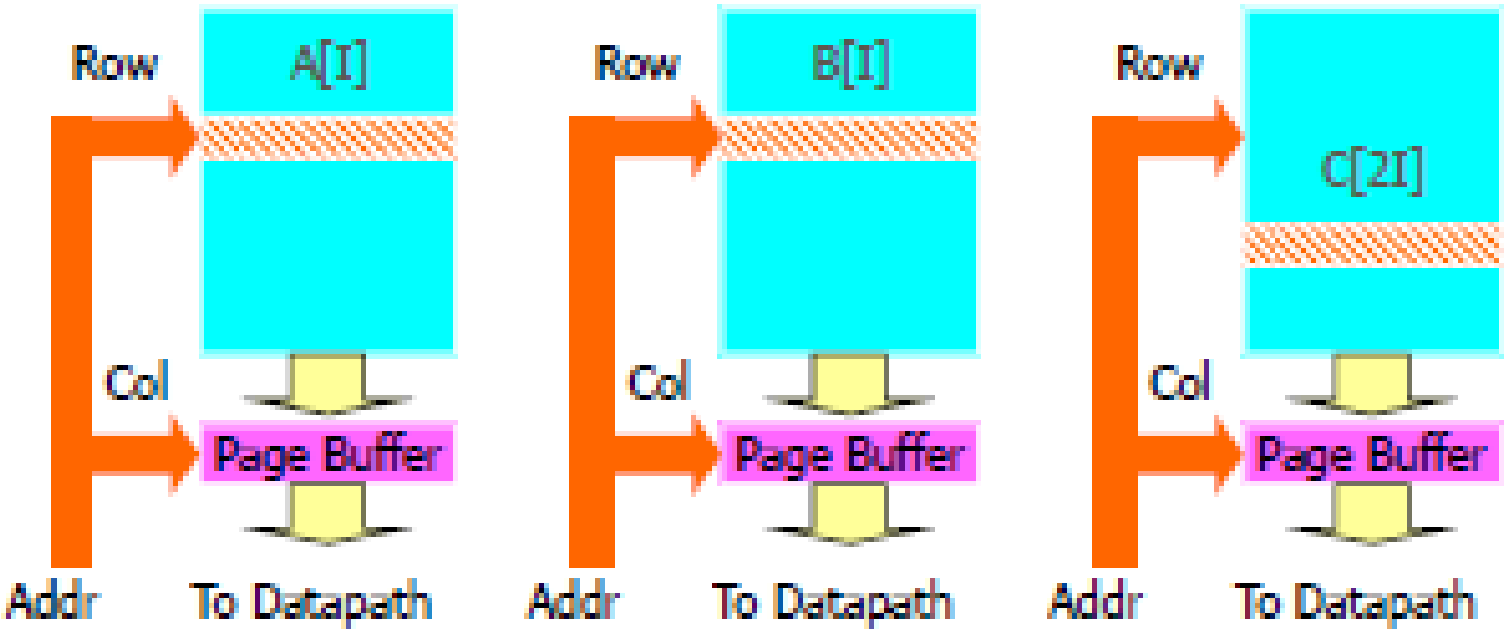
Custom: Memory Banking

```
for (i = 0,  i < 1000; i++)
  { A[i] += B[i] * C[2*i]; }
```



Memory Banking (e)DRAM

```
for (i = 0,  i < 1000; i++)
  {
  A[i] += B[i] * C[2*i];
  }
```



DRAM tradeoffs

DRAMs are complex objects:

- Multiple interleaved DRAM banks in a system
- Large premium for burst accesses
- Tradeoff between leaving a page open and having neighbouring accesses faster and closing a page and not requiring precharge time for far accesses

In **servers**, optimizations are usually of dynamic nature performed by the memory controller subsystem and controlled by the BIOS/OS

In **embedded computers**, similar optimizations can be done statically and application-specific

Summary

- In SoCs and FPGAs situation is different from general purpose computers
  - Different design space (fast memories almost as fast as logic)
  - Less constraints to use standard components: any size possible, more types of memory available (e.g., dual port), etc.
  - More bandwidth exploitable (no pins)
- Hence different world where many more things are possible (whereas in classic computing or normal chip based embedded systems there is not much freedom)
- Companion situation to the customisation of processors:
  - Optimizations tailored to Data Cache
    - Memory Data Layout
  - Memory architecture customized to a given application
    - Scratchpad Memory
    - Memory Banking