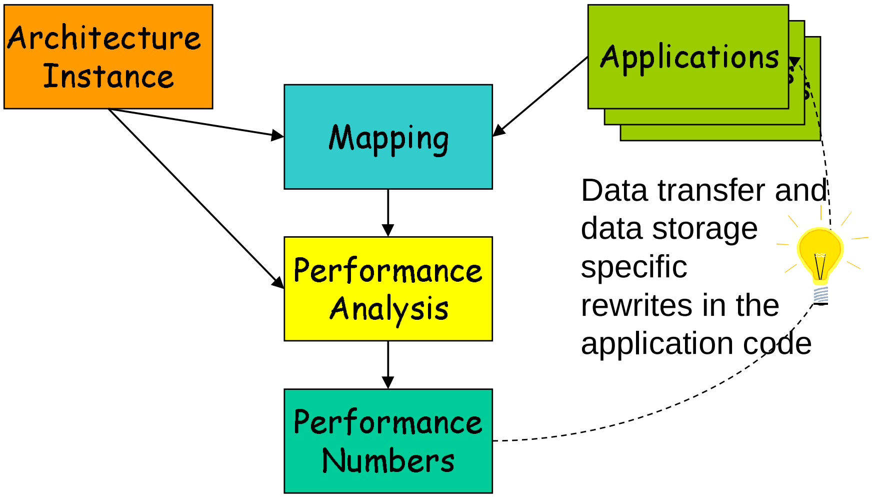
Memory Organization in Embedded Multimedia Platforms

Diederik.Verkest@imec.be

Positioning in the Y-chart



Architecture Instance

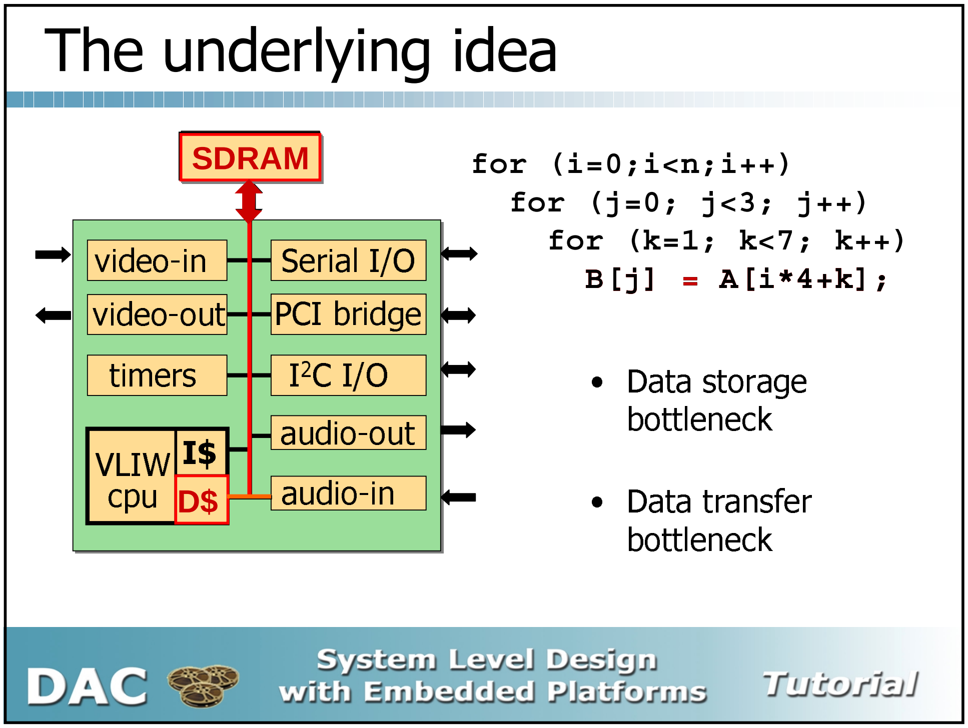
Applications

Mapping

Performance Analysis

Performance Numbers

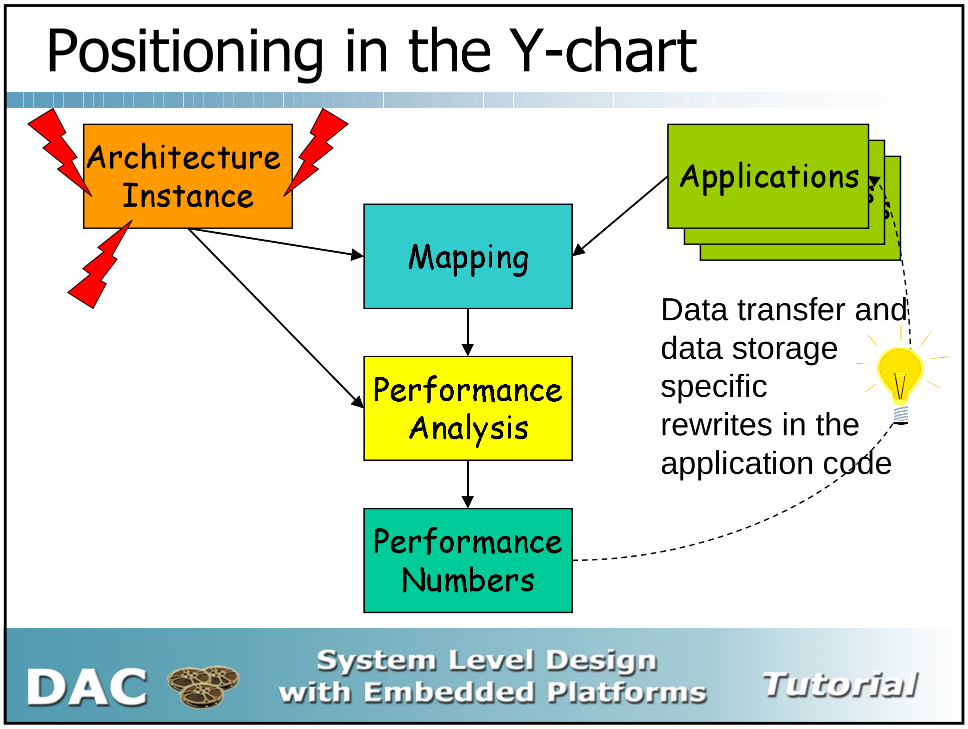
Data transfer and data storage specific rewrites in the application code



The underlying idea

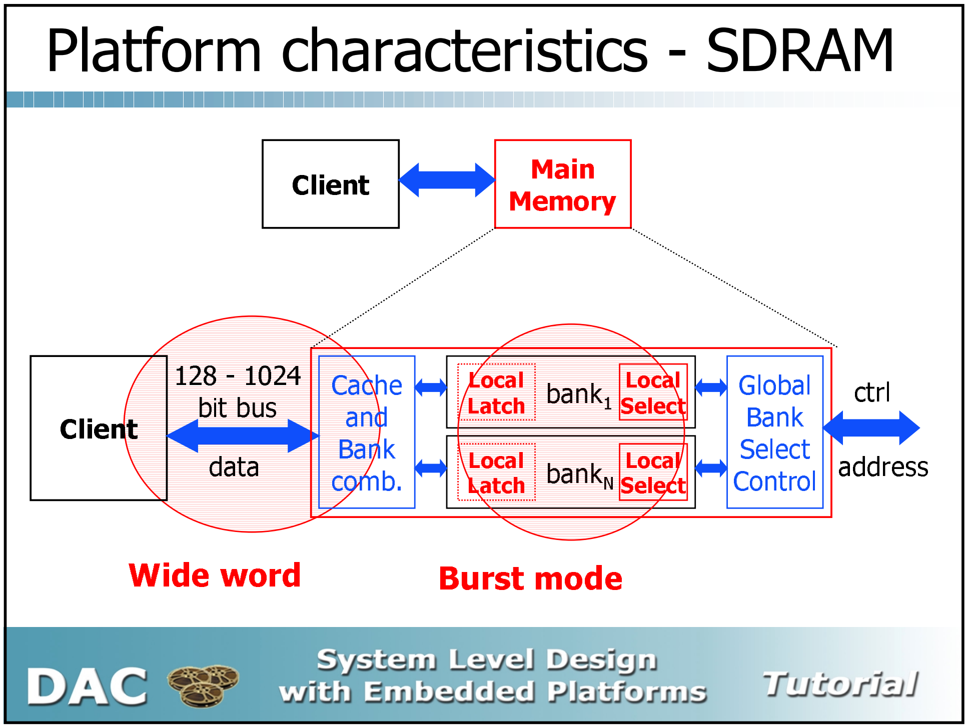
```
for (i=0;i<n;i++)
  for (j=0; j<3; j++)
    for (k=1; k<7; k++)
      B[j] = A[i*4+k];
```

- Data storage bottleneck
- Data transfer bottleneck



Positioning in the Y-chart

Data transfer and data storage specific rewrites in the application code



Platform characteristics - SDRAM

Client

Main Memory

Client

128 - 1024 bit bus

data

Cache and Bank comb.

Local Latch

bank_1

Local Select

Local Latch

bank_N

Local Select

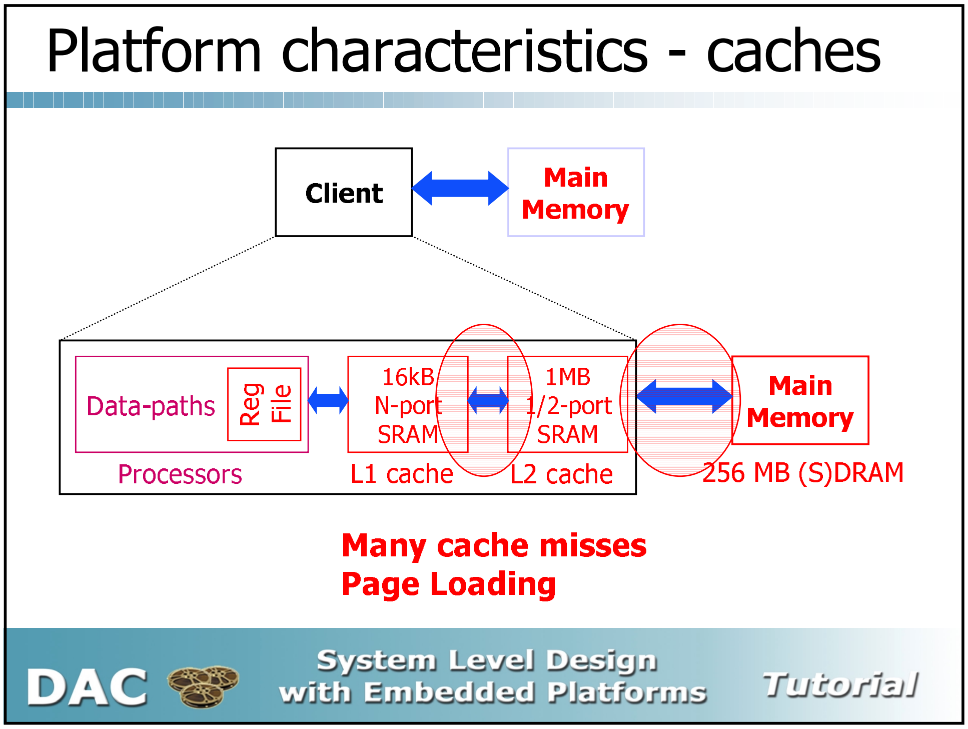
Global Bank Select Control

ctrl

address

Wide word

Burst mode



Platform characteristics - caches

Client

Main Memory

Data-paths

Reg File

Processors

16kB N-port SRAM

L1 cache

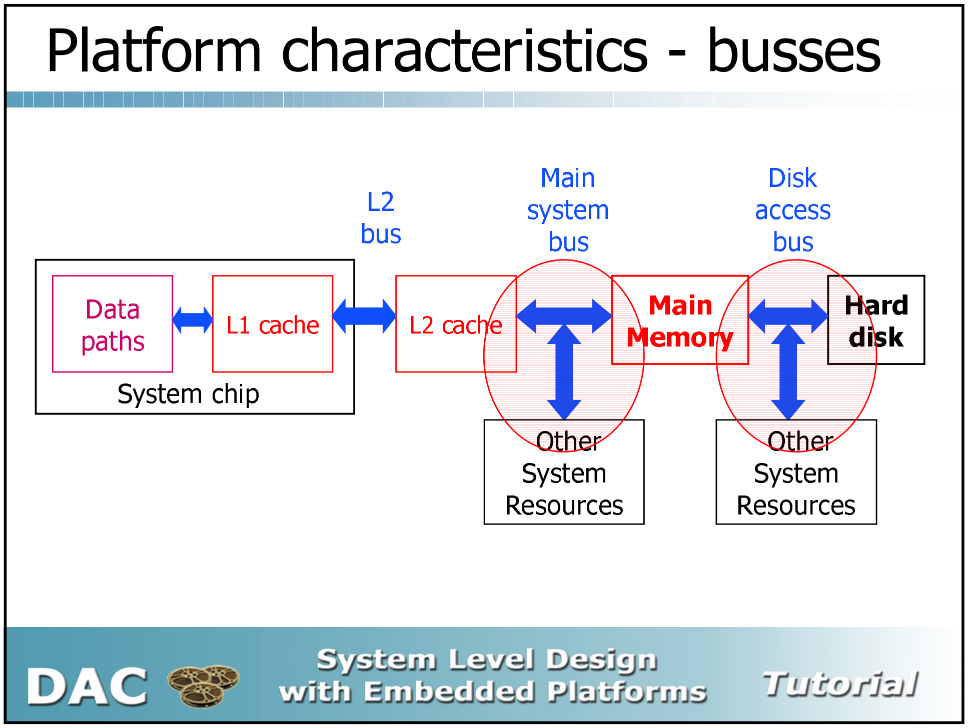
1MB 1/2-port SRAM

L2 cache

Main Memory

256 MB (S)DRAM

Many cache misses
Page Loading



Platform characteristics - busses

L2 bus

Main system bus

Disk access bus

Data paths

L1 cache

System chip

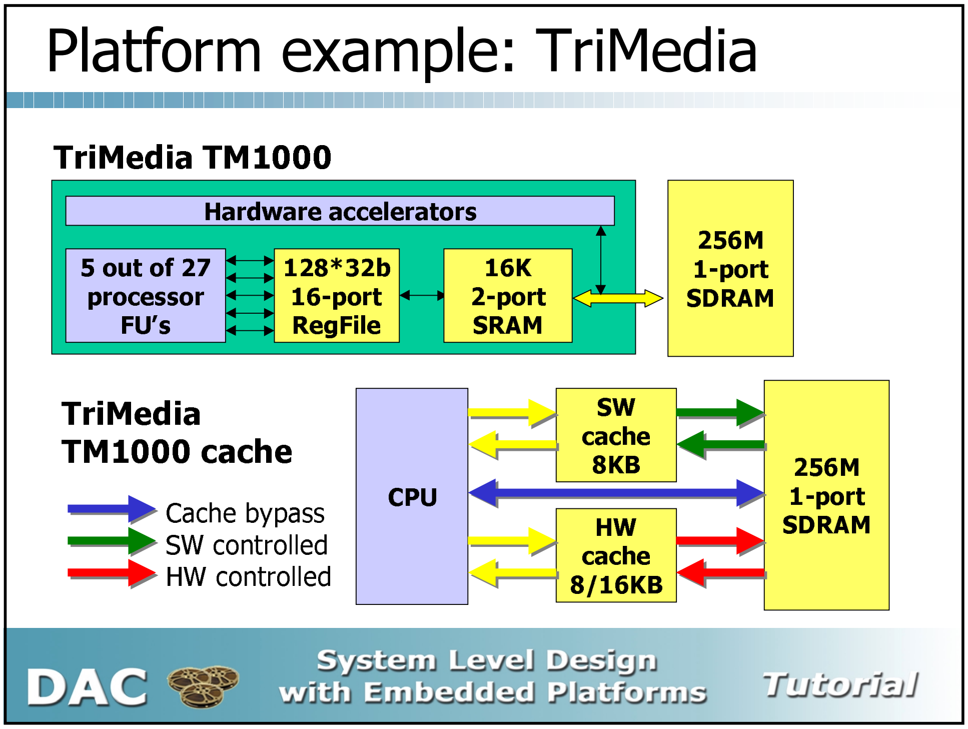
L2 cache

Main Memory

Hard disk

Other System Resources

Other System Resources



Platform example: TriMedia

TriMedia TM1000

Hardware accelerators

5 out of 27 processor FU's

128*32b 16-port RegFile

16K 2-port SRAM

256M 1-port SDRAM

TriMedia TM1000 cache

Cache bypass

SW controlled

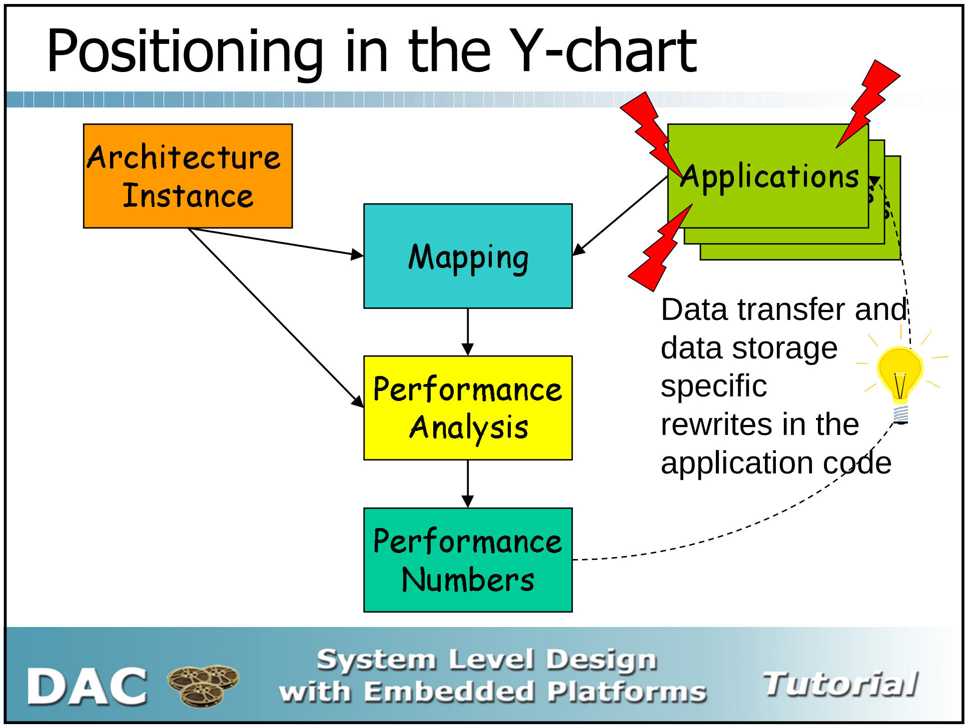
HW controlled

CPU

SW cache 8KB

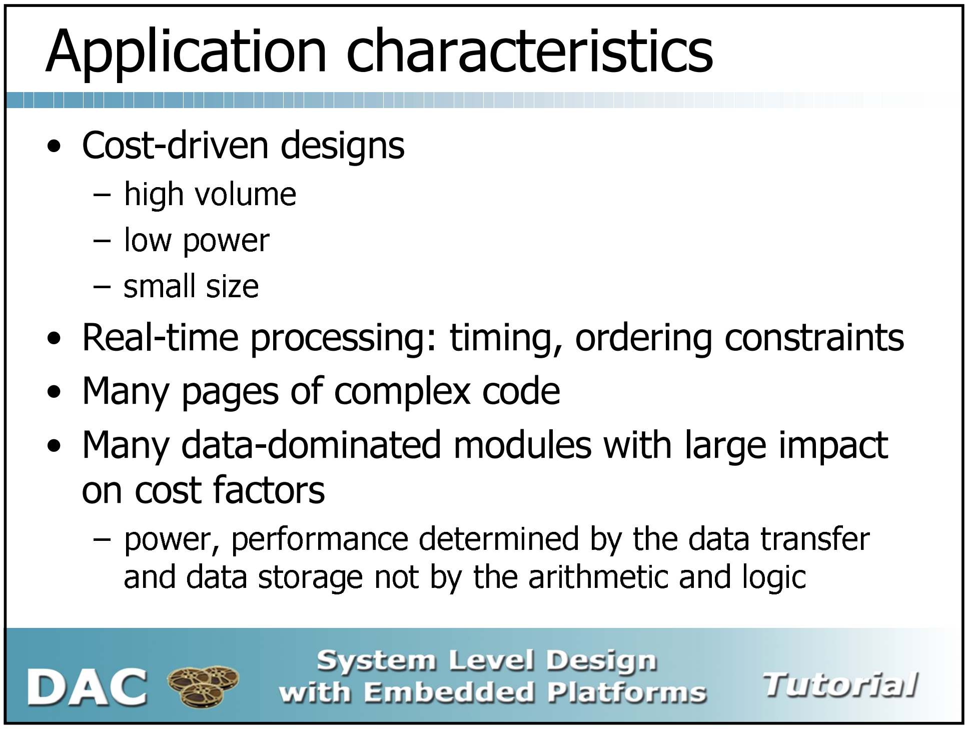
HW cache 8/16KB

256M 1-port SDRAM



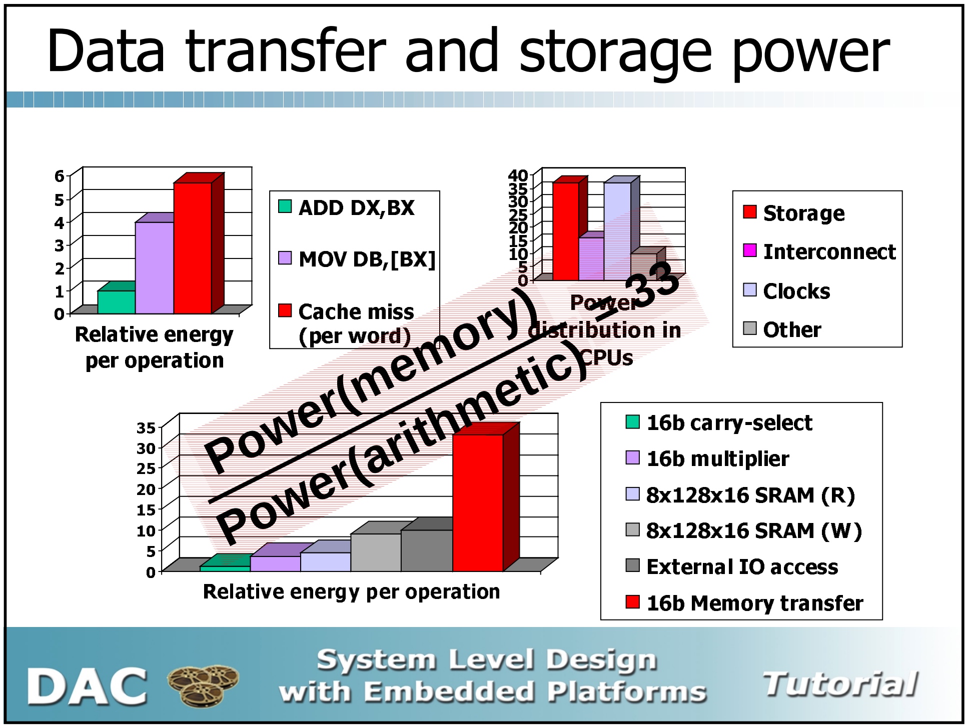
Positioning in the Y-chart

Data transfer and data storage specific rewrites in the application code



Application characteristics

- Cost-driven designs
  - high volume
  - low power
  - small size
- Real-time processing: timing, ordering constraints
- Many pages of complex code
- Many data-dominated modules with large impact on cost factors
  - power, performance determined by the data transfer and data storage not by the arithmetic and logic



Data transfer and storage power

Relative energy per operation

- 🟩 ADD DX,BX
- 🟪 MOV DB,[BX]
- 🟥 Cache miss (per word)

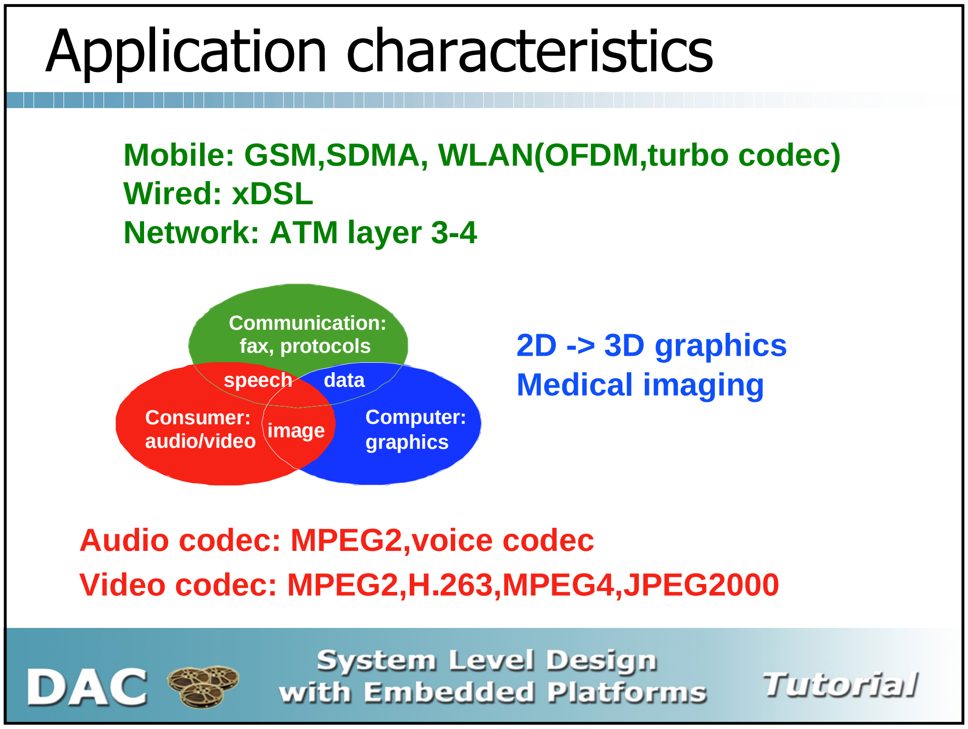
Power distribution in CPUs

- 🟥 Storage
- 🟪 Interconnect
- ⬜ Clocks
- ⬜ Other

$$\frac{Power(memory)}{Power(arithmetic)} = 33$$

Relative energy per operation

- 🟩 16b carry-select
- 🟪 16b multiplier
- ⬜ 8x128x16 SRAM (R)
- ⬜ 8x128x16 SRAM (W)
- ⬜ External IO access
- 🟥 16b Memory transfer



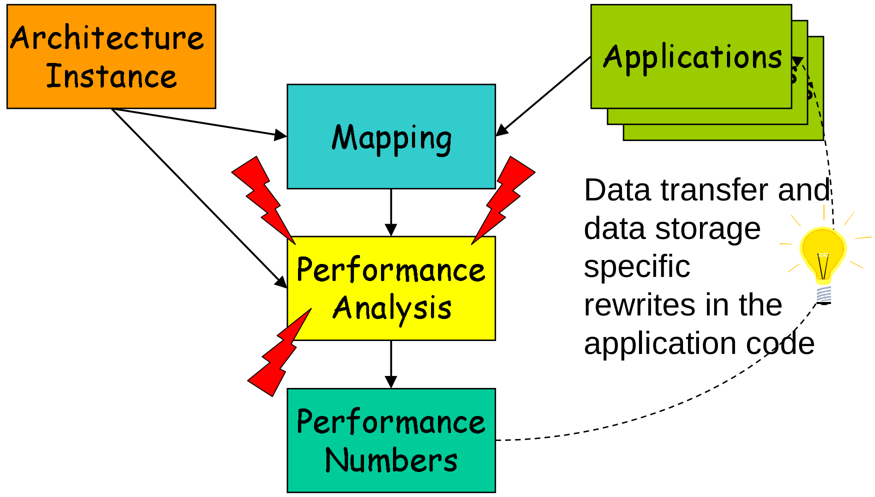
Application characteristics

Mobile: GSM,SDMA, WLAN(OFDM,turbo codec)
Wired: xDSL
Network: ATM layer 3-4

2D -> 3D graphics
Medical imaging

Audio codec: MPEG2,voice codec
Video codec: MPEG2,H.263,MPEG4,JPEG2000

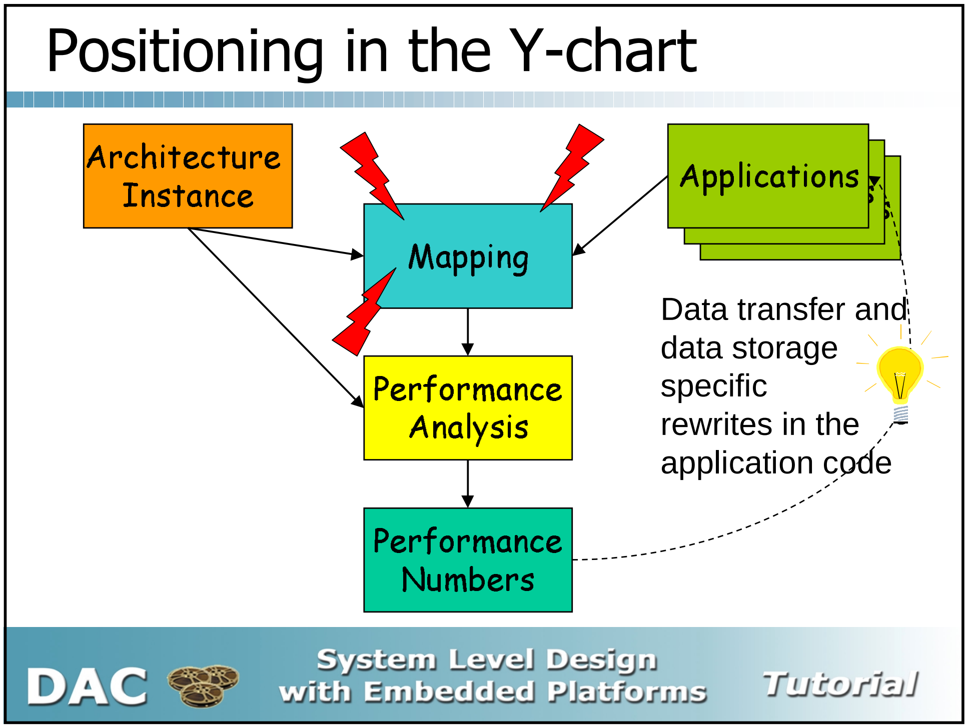
Positioning in the Y-chart



Data transfer and data storage specific rewrites in the application code

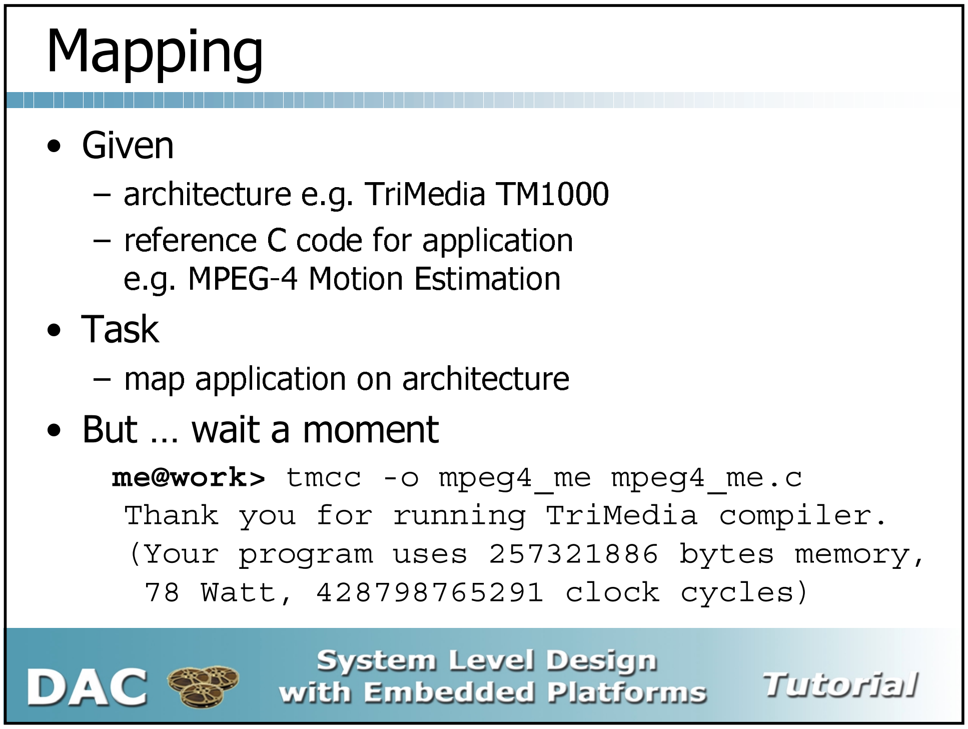
Performance analysis

- Feedback using estimation of
  - Data transfers
  - Memory size
- Derived performance parameters
  - System bus load
  - Power dissipation in memories
    - size of memory
    - frequency of access
    - technology (VDD, on-chip/off-chip)
- Measured performance parameter
  - Clock cycles



Positioning in the Y-chart

Data transfer and data storage specific rewrites in the application code



Mapping

- Given
  - architecture e.g. TriMedia TM1000
  - reference C code for application e.g. MPEG-4 Motion Estimation
- Task
  - map application on architecture
- But … wait a moment

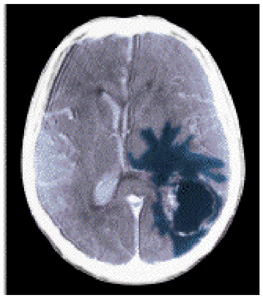
```
me@work> tmcc -o mpeg4_me mpeg4_me.c
 Thank you for running TriMedia compiler.
 (Your program uses 257321886 bytes memory,
  78 Watt, 428798765291 clock cycles)
```

Let's help the compiler ...

- DTSE is a methodology to explore data-transfer and data-storage in multi-media applications
  - Transforms C-code of the application
  - By focusing on multi-dimensional signals (arrays)
  - To better exploit platform capabilities
- Tools give feedback about the effect of transformations for a given target platform
- In this tutorial we'll cover four major steps to improve power, area, performance trade-off in the context of platform based design

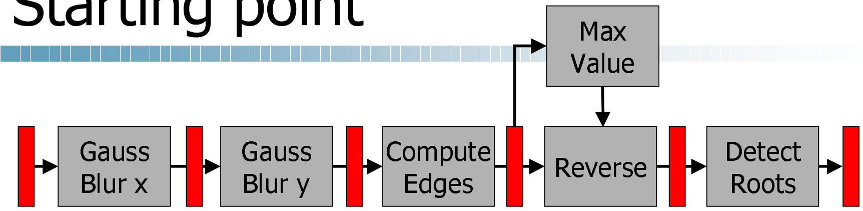
Application example

- Application domain:
  - Computer Tomography in medical imaging
- Algorithm:
  - Cavity detection in CT-scans
  - Detect dark regions in successive images
  - Indicate cavity in brain



⇨ **Bad news for owner of brain**

Starting point



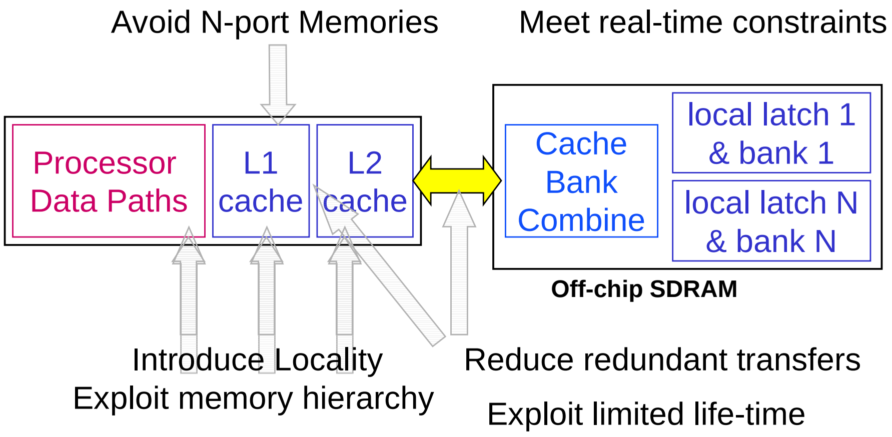
- Reference (conceptual) C code for the algorithm
  - all functions: image_in[N x M]$_{t-1}$ -> image_out[N x M]$_t$
  - new value of pixel depends on its neighbors
  - neighbor pixels read from background memory
  - approximately 110 lines of C code (ignoring file I/O etc)
  - experiments with N x M = 640 x 400 pixels
  - straightforward implementation: 6 image buffers

Reference code structure

```
main(){                                /* Layer 1 code */
  read_image(IN_NAME, image_in);
  cav_detect();
```

```
void cav_detect() {             /* Layer 2 code */
  for (x=GB; x<=N-1-GB; ++x) {
    for (y=GB; y<=M-1-GB; ++y) {
      gauss_x_tmp = 0;
      for (k=-GB; k<=GB; ++k) {
        gauss_x_tmp += in_image[x+k][y] * Gauss[abs(k)];
      }
      gauss_x_image[x][y] = foo(gauss_x_tmp);
    }
  }                        /* Makes code for data access */
}                          /* and data transfer explicit */
```

DTSE principles



The major steps in DTSE

❶ Data flow transformations

- Eliminate redundant transfers and storage

❷ Loop and control flow transformations

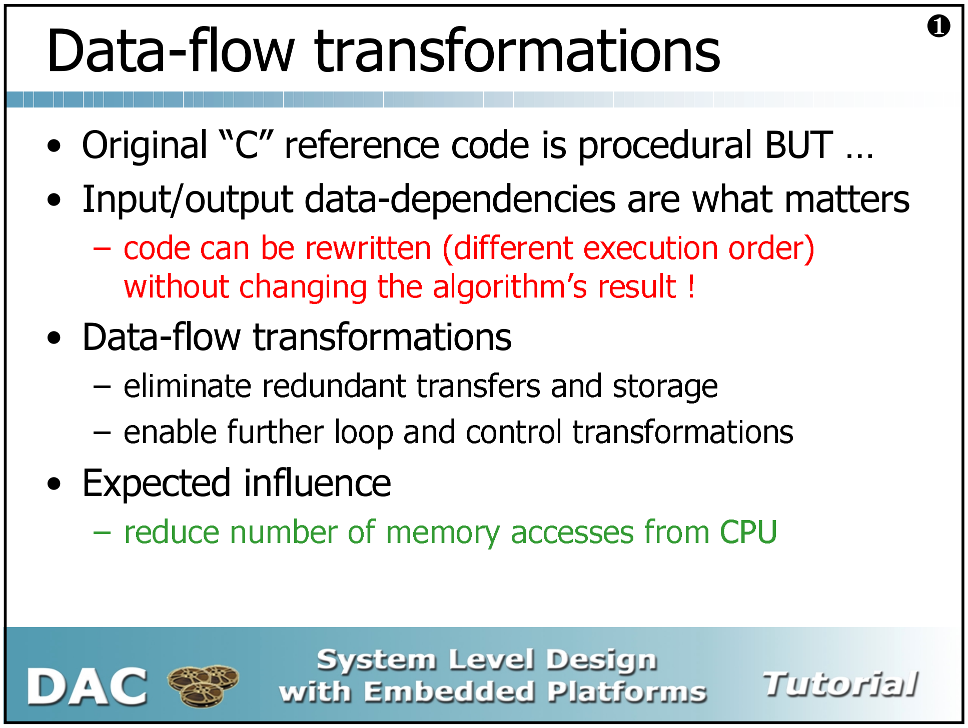
- Improve regularity of accesses and data locality

❸ Data re-use and memory hierarchy exploitation

- Determine when to move which data between memories to meet the cycle budget of the application with low cost

❹ Data layout optimization

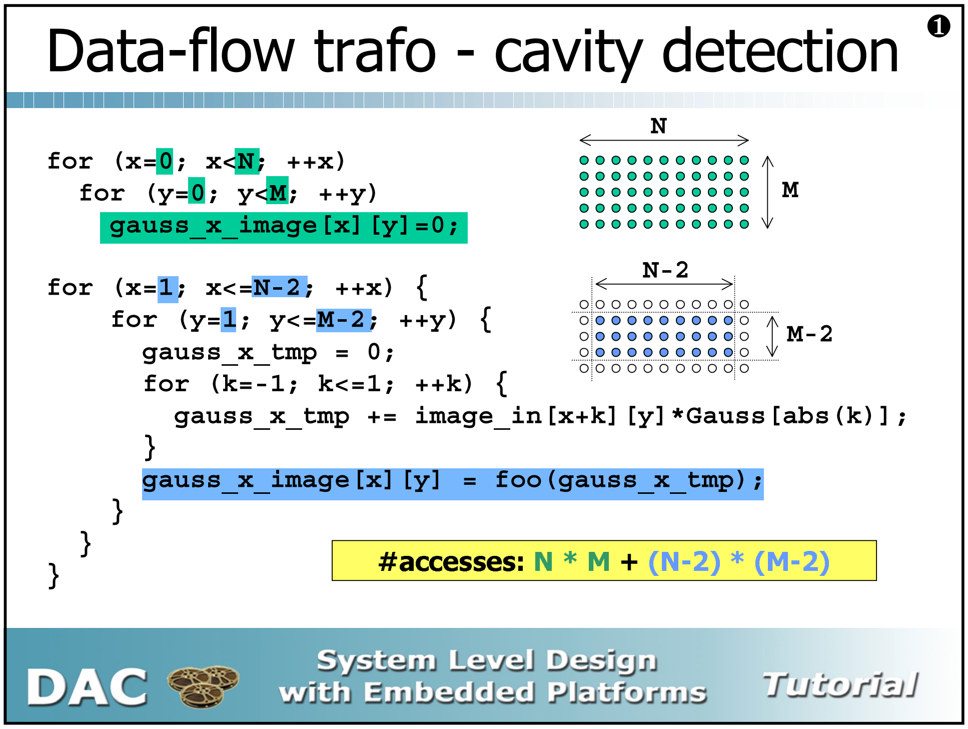
- Arrange data in the correct place in memory to (1) minimize memory size and (2) make efficient use of memory features such as cache locking and associativity



Data-flow transformations

❶

- Original "C" reference code is procedural BUT …
- Input/output data-dependencies are what matters
  - code can be rewritten (different execution order) without changing the algorithm's result !
- Data-flow transformations
  - eliminate redundant transfers and storage
  - enable further loop and control transformations
- Expected influence
  - reduce number of memory accesses from CPU

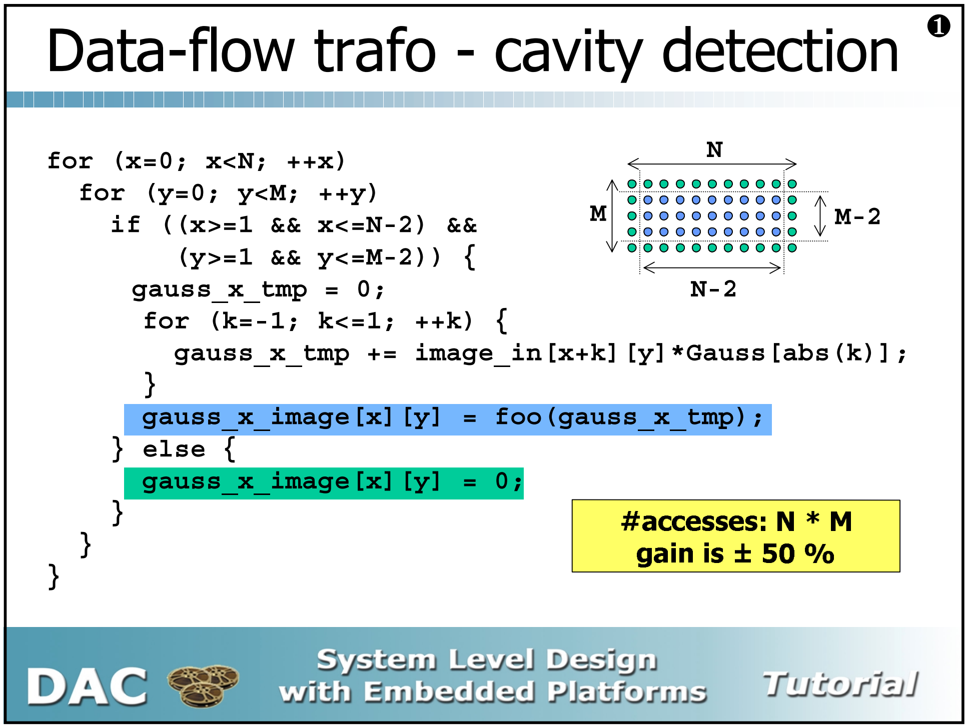


Data-flow trafo - cavity detection

❶

```
for (x=0; x<N; ++x)
  for (y=0; y<M; ++y)
    gauss_x_image[x][y]=0;

for (x=1; x<=N-2; ++x) {
    for (y=1; y<=M-2; ++y) {
      gauss_x_tmp = 0;
      for (k=-1; k<=1; ++k) {
        gauss_x_tmp += image_in[x+k][y]*Gauss[abs(k)];
      }
      gauss_x_image[x][y] = foo(gauss_x_tmp);
    }
  }
}
```

#accesses: N * M + (N-2) * (M-2)

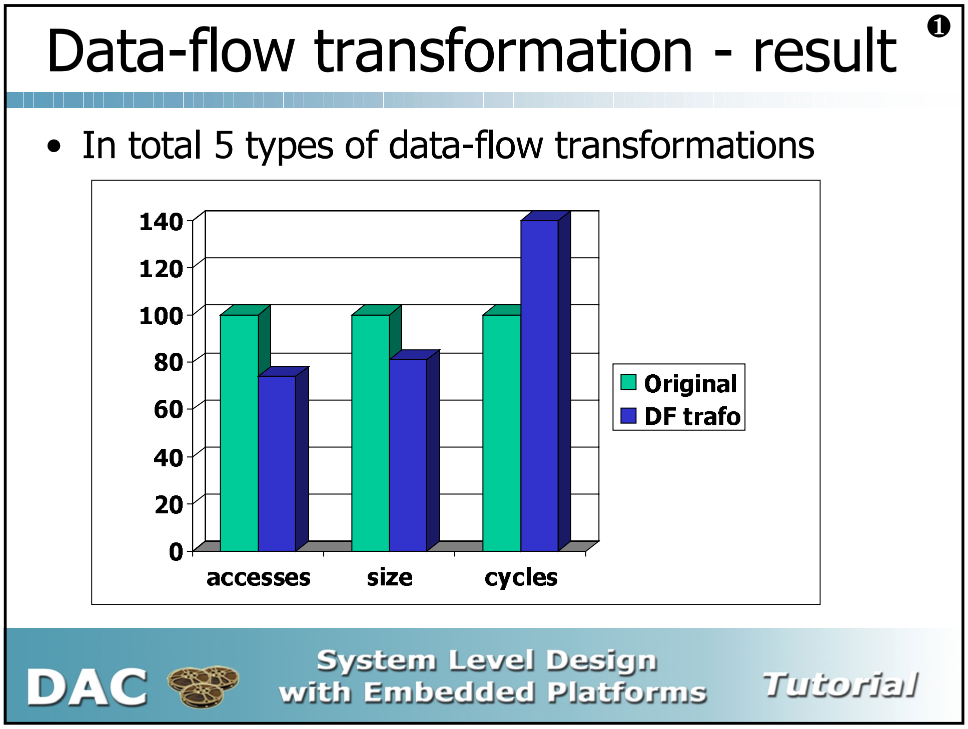


Data-flow trafo - cavity detection

❶

```
for (x=0; x<N; ++x)
  for (y=0; y<M; ++y)
    if ((x>=1 && x<=N-2) &&
        (y>=1 && y<=M-2)) {
      gauss_x_tmp = 0;
       for (k=-1; k<=1; ++k) {
         gauss_x_tmp += image_in[x+k][y]*Gauss[abs(k)];
       }
      gauss_x_image[x][y] = foo(gauss_x_tmp);
    } else {
      gauss_x_image[x][y] = 0;
    }
  }
}
```

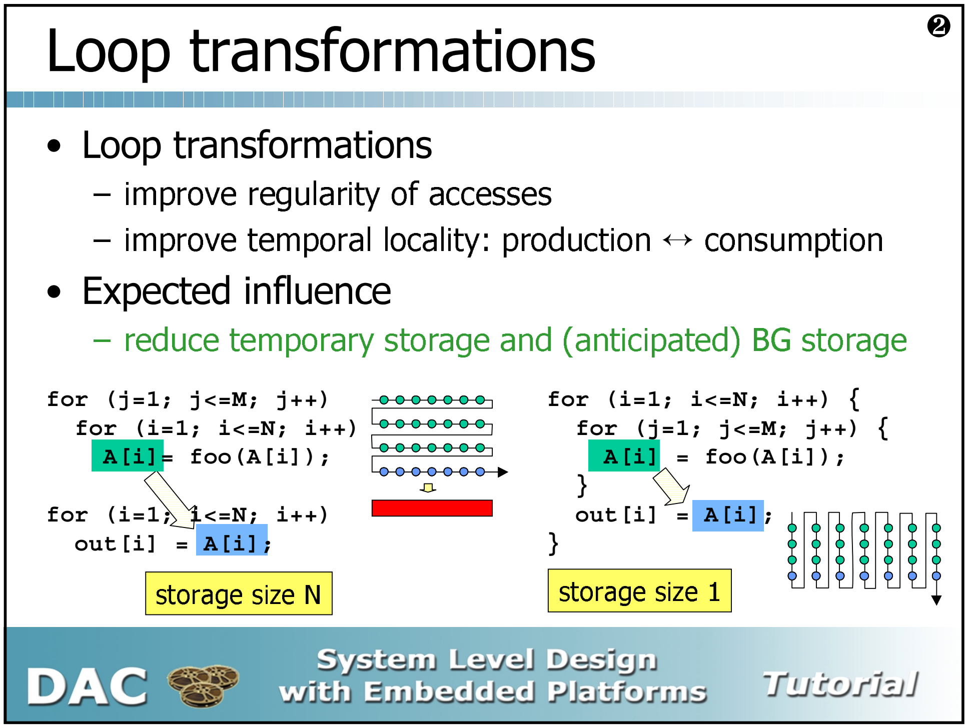
#accesses: N * M
gain is ± 50 %



Data-flow transformation - result

❶

- In total 5 types of data-flow transformations



Loop transformations

❷

- Loop transformations
  - improve regularity of accesses
  - improve temporal locality: production ↔ consumption
- Expected influence
  - reduce temporary storage and (anticipated) BG storage

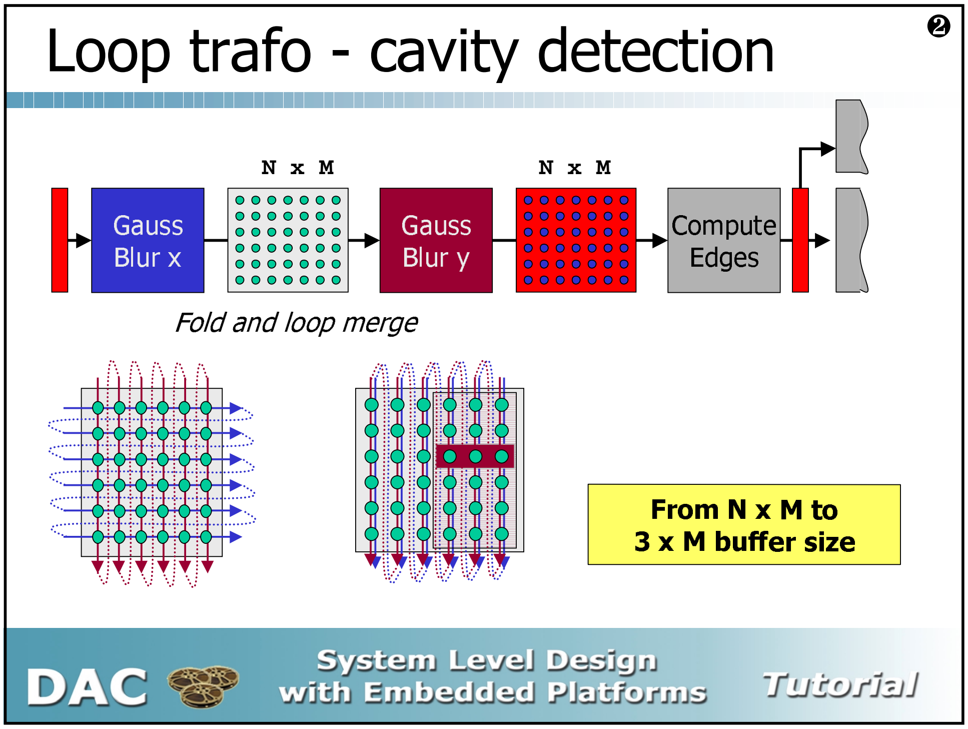
```
for (j=1; j<=M; j++)
  for (i=1; i<=N; i++)
    A[i]= foo(A[i]);

for (i=1; i<=N; i++)
  out[i] = A[i];
```

storage size N

```
for (i=1; i<=N; i++) {
  for (j=1; j<=M; j++) {
    A[i] = foo(A[i]);
  }
  out[i] = A[i];
}
```

storage size 1

DAC System Level Design with Embedded Platforms Tutorial



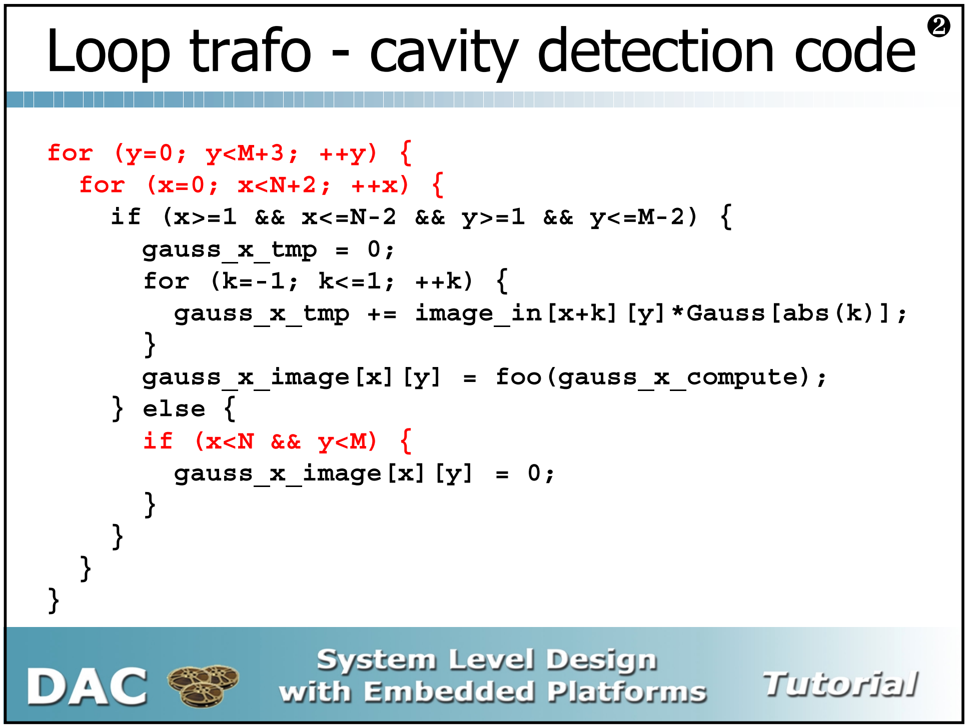
Loop trafo - cavity detection

❷

Fold and loop merge

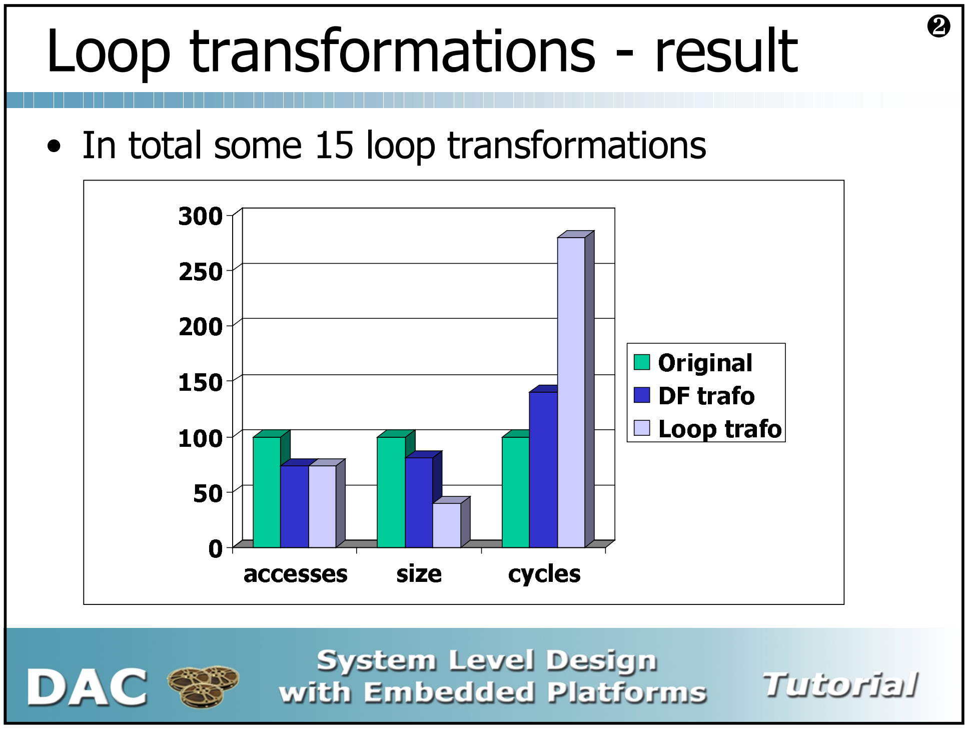
From N x M to 3 x M buffer size

DAC System Level Design with Embedded Platforms Tutorial



Loop trafo - cavity detection code ❷

```
for (y=0; y<M+3; ++y) {
  for (x=0; x<N+2; ++x) {
    if (x>=1 && x<=N-2 && y>=1 && y<=M-2) {
      gauss_x_tmp = 0;
      for (k=-1; k<=1; ++k) {
        gauss_x_tmp += image_in[x+k][y]*Gauss[abs(k)];
      }
      gauss_x_image[x][y] = foo(gauss_x_compute);
    } else {
      if (x<N && y<M) {
        gauss_x_image[x][y] = 0;
      }
    }
  }
}
```

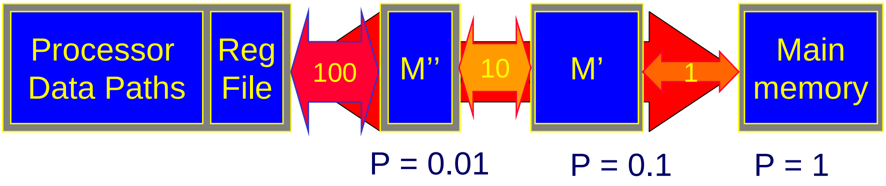


Loop transformations - result ❷

- In total some 15 loop transformations

❸

Data re-use & memory hierarchy



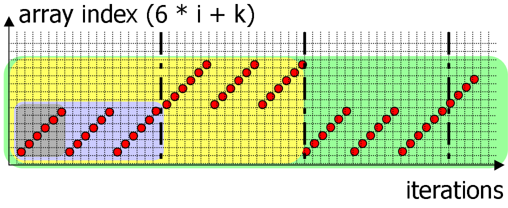
- P (original) = # access x power/access = 100
- P (after) = 100 x 0.01 + 10 x 0.1 + 1 x 1 = 3
- Introduce memory hierarchy
  - reduce number of reads from main memory
  - heavily accessed arrays stored in smaller memories

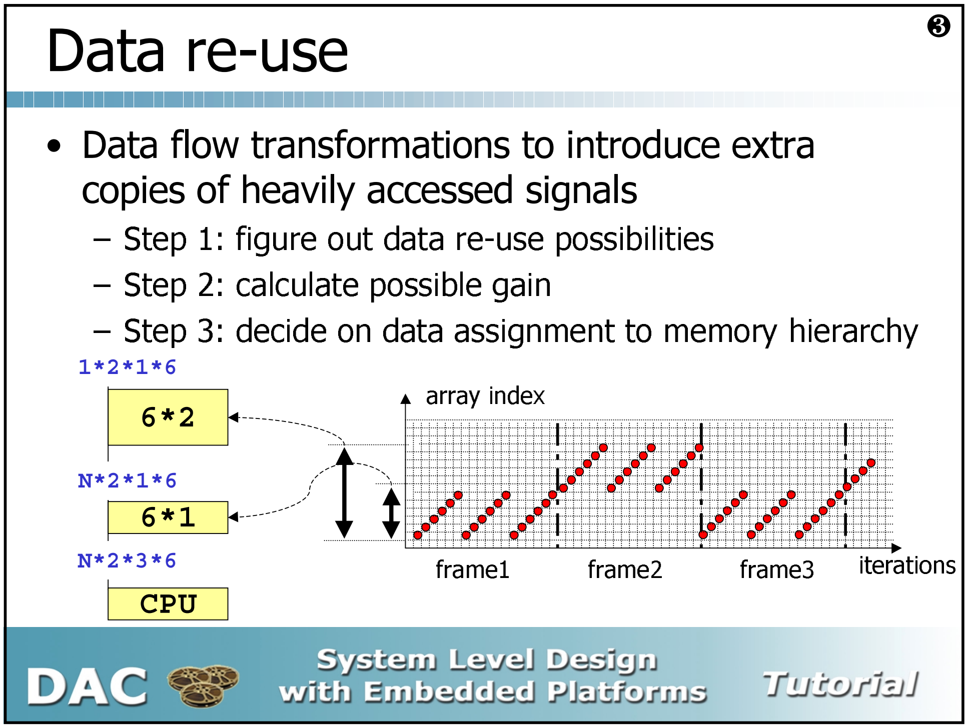
❸

Data re-use

- Data flow transformations to introduce extra copies of heavily accessed signals
  - Step 1: figure out data re-use possibilities
  - Step 2: calculate possible gain
  - Step 3: decide on data assignment to memory hierarchy

```
int[2][6] A;

for (h=0; h<N; h++)
 for (i=0; i<2; i++)
  for (j=0; j<3; j++)
   for (k=1; k<7; k++)
    B[j] = A[i][k];
```

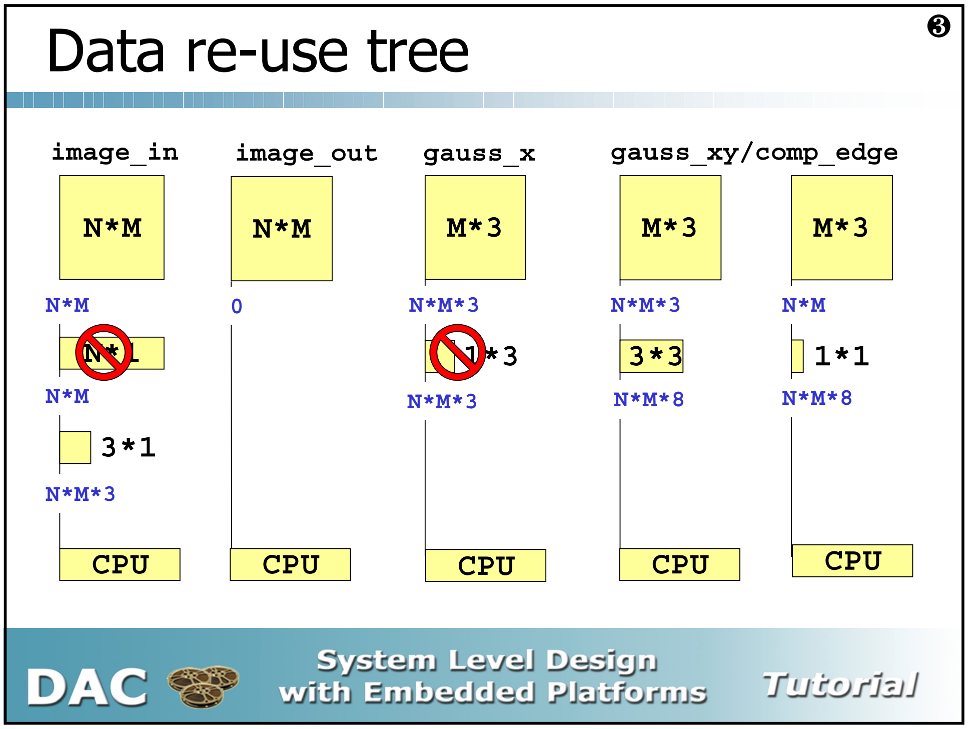




Data re-use

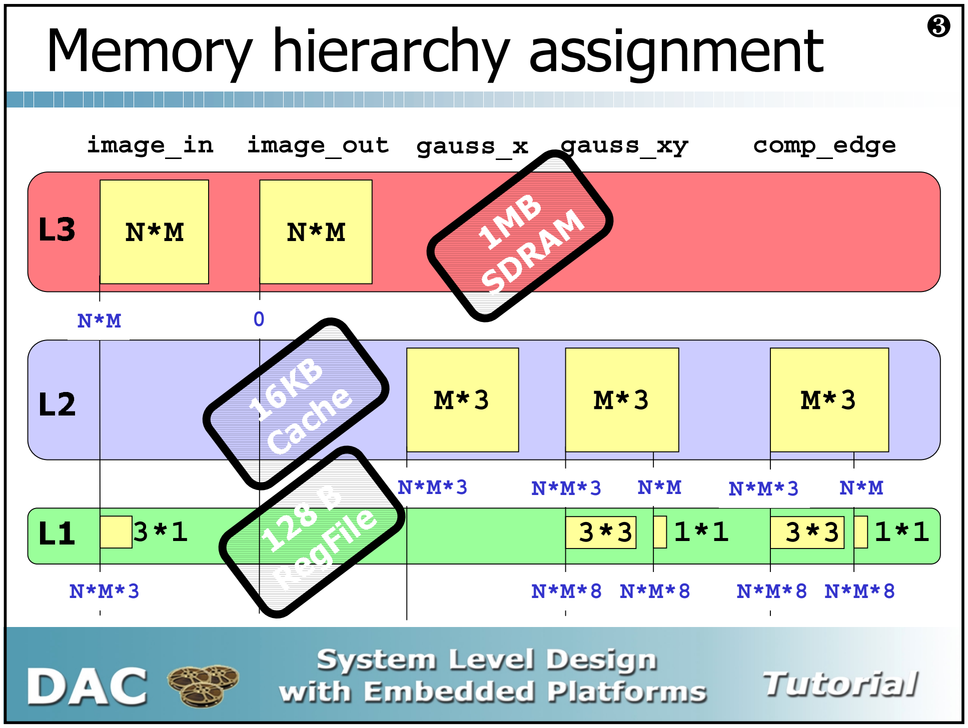
❸

- Data flow transformations to introduce extra copies of heavily accessed signals
  - Step 1: figure out data re-use possibilities
  - Step 2: calculate possible gain
  - Step 3: decide on data assignment to memory hierarchy



Data re-use tree

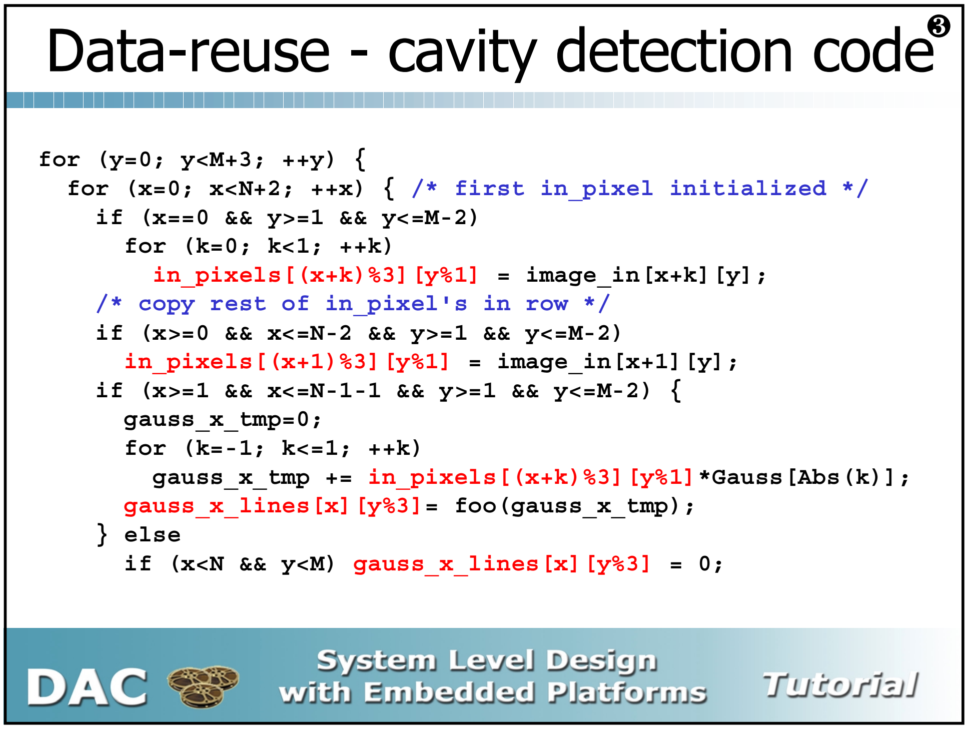
❸



Memory hierarchy assignment

❸

DAC System Level Design with Embedded Platforms Tutorial

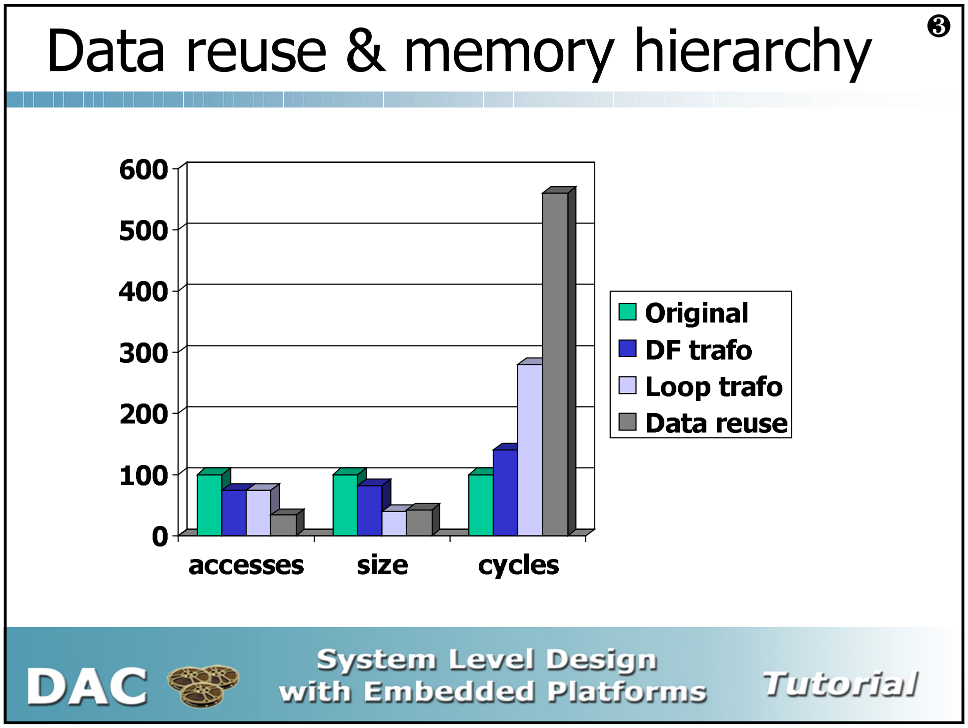


Data-reuse - cavity detection code

❸

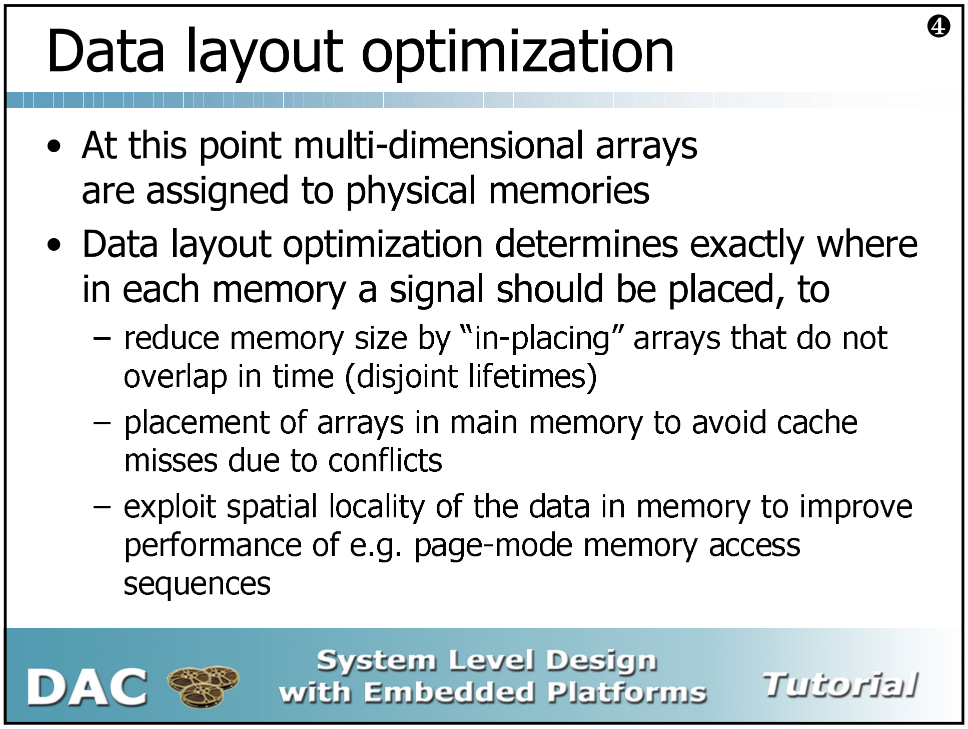
```
for (y=0; y<M+3; ++y) {
  for (x=0; x<N+2; ++x) { /* first in_pixel initialized */
    if (x==0 && y>=1 && y<=M-2)
      for (k=0; k<1; ++k)
        in_pixels[(x+k)%3][y%1] = image_in[x+k][y];
    /* copy rest of in_pixel's in row */
    if (x>=0 && x<=N-2 && y>=1 && y<=M-2)
      in_pixels[(x+1)%3][y%1] = image_in[x+1][y];
    if (x>=1 && x<=N-1-1 && y>=1 && y<=M-2) {
      gauss_x_tmp=0;
      for (k=-1; k<=1; ++k)
        gauss_x_tmp += in_pixels[(x+k)%3][y%1]*Gauss[Abs(k)];
      gauss_x_lines[x][y%3]= foo(gauss_x_tmp);
    } else
      if (x<N && y<M) gauss_x_lines[x][y%3] = 0;
```

DAC System Level Design with Embedded Platforms Tutorial



Data reuse & memory hierarchy ❸

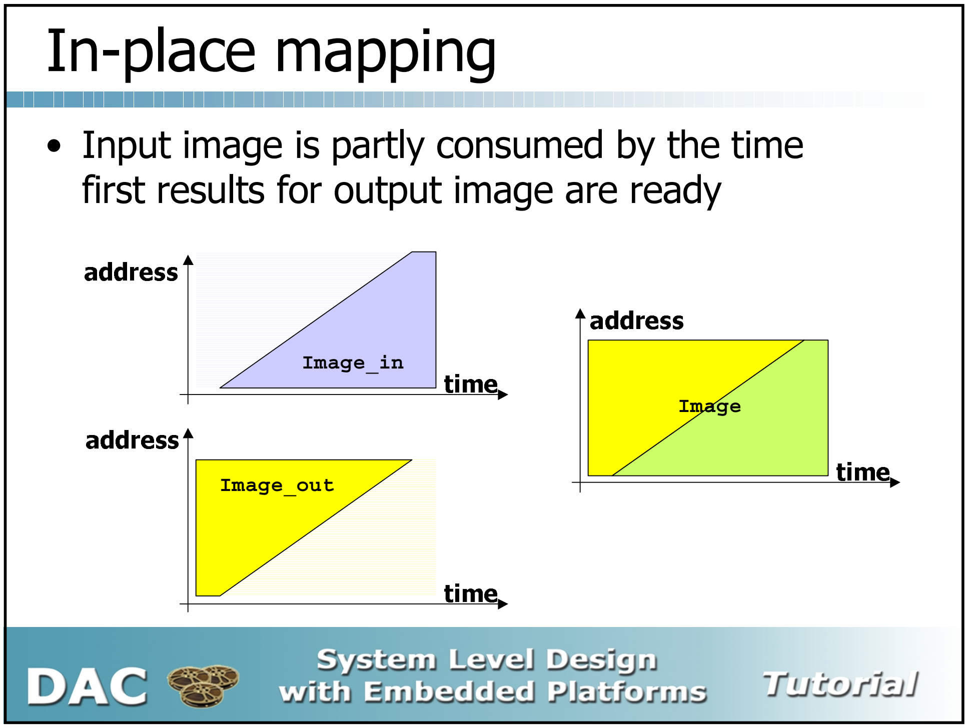
DAC System Level Design with Embedded Platforms Tutorial



Data layout optimization ❹

- At this point multi-dimensional arrays are assigned to physical memories
- Data layout optimization determines exactly where in each memory a signal should be placed, to
  - reduce memory size by "in-placing" arrays that do not overlap in time (disjoint lifetimes)
  - placement of arrays in main memory to avoid cache misses due to conflicts
  - exploit spatial locality of the data in memory to improve performance of e.g. page-mode memory access sequences

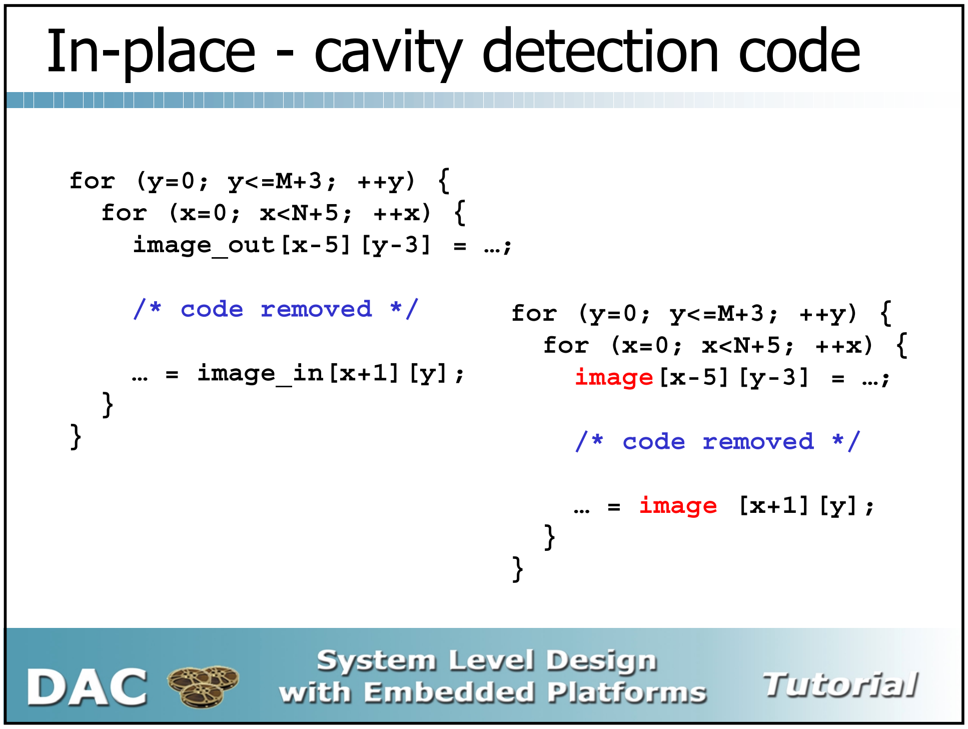
DAC System Level Design with Embedded Platforms Tutorial



In-place mapping

- Input image is partly consumed by the time first results for output image are ready

DAC System Level Design with Embedded Platforms Tutorial

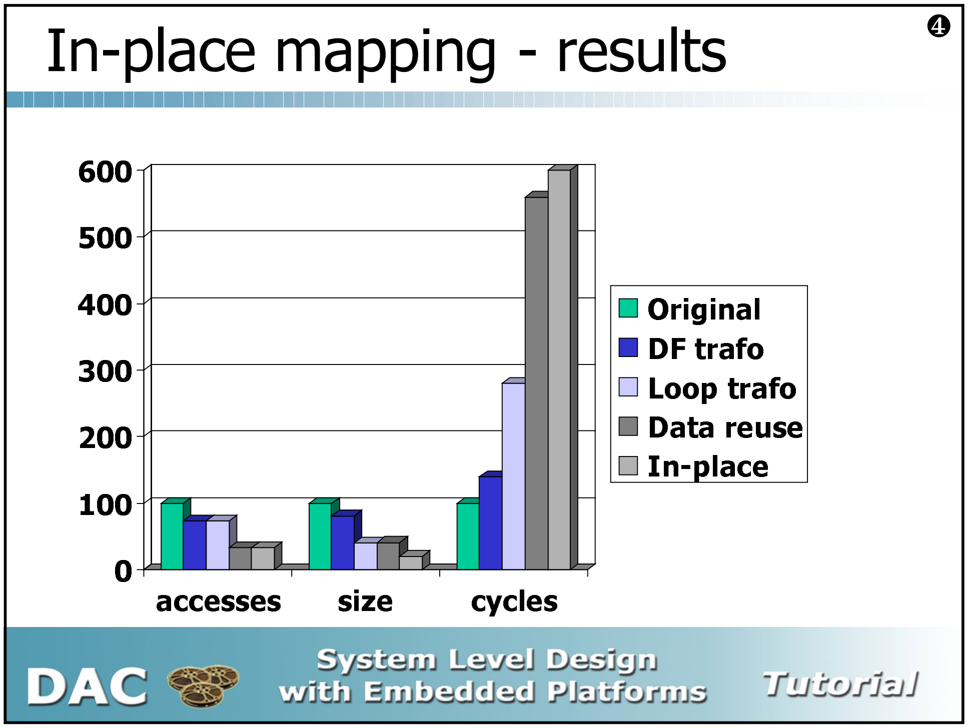


In-place - cavity detection code

```
for (y=0; y<=M+3; ++y) {
  for (x=0; x<N+5; ++x) {
    image_out[x-5][y-3] = …;

    /* code removed */

    … = image_in[x+1][y];
  }
}
```

```
for (y=0; y<=M+3; ++y) {
  for (x=0; x<N+5; ++x) {
    image[x-5][y-3] = …;

    /* code removed */

    … = image [x+1][y];
  }
}
```

DAC System Level Design with Embedded Platforms Tutorial



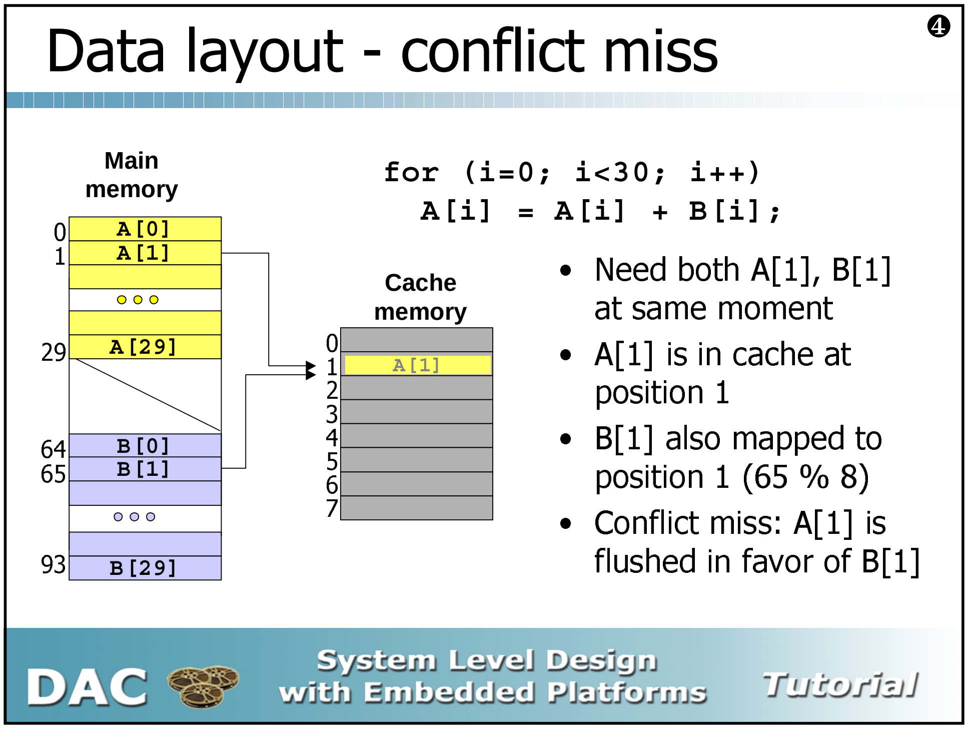
In-place mapping - results

❹

DAC

System Level Design
with Embedded Platforms

Tutorial



Data layout - conflict miss

❹

```
for (i=0; i<30; i++)
  A[i] = A[i] + B[i];
```

- Need both A[1], B[1] at same moment
- A[1] is in cache at position 1
- B[1] also mapped to position 1 (65 % 8)
- Conflict miss: A[1] is flushed in favor of B[1]

DAC

System Level Design
with Embedded Platforms

Tutorial

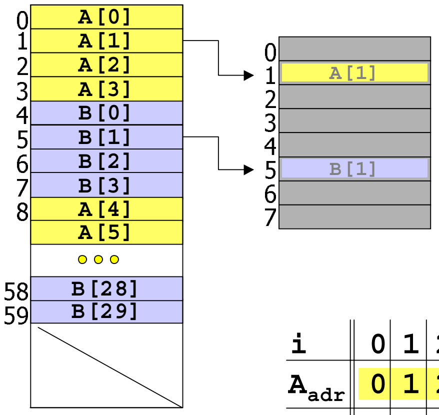
Cache misses vs. associativity

❹

- Traditional compilers use single contiguous data organization
  - Without taking cache parameters into account
  - May (or may not) give rise to conflicts
- Associative caches reduce cache conflict misses but are expensive
- Instead of using associative caches
  - Organize data in memory in such a way that conflicts are avoided

Data layout - avoid conflict miss

❹



```
for (i=0; i<30; i++)
  A[i] = A[i] + B[i];
```

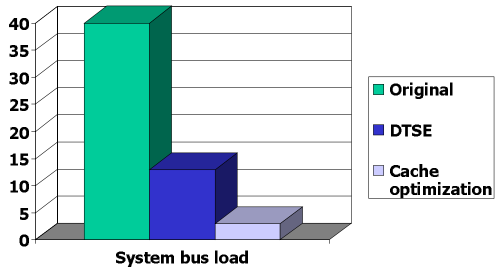
```
 for (i=0; i<30; i++)
   mem[2*i-i%4]=
     mem[2*i-i%4] +
       mem[2*i-i%4+4];
```

i	0	1	2	3	4	5	6	7	8	9
A_{adr}	0	1	2	3	8	9	10	11	16	17
B_{adr}	4	5	6	7	12	13	14	15	18	19

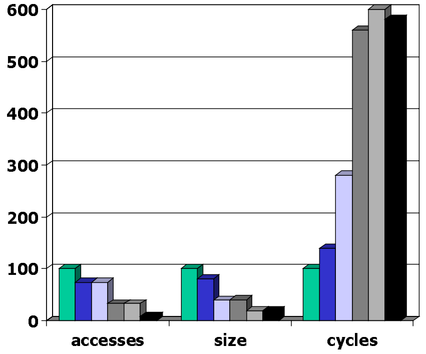
Cache optimization - results

❹

- Total size of all signals (except image_in[][] and image_out[][]) now less than 8 KB
- All signals are locked in TM1000 cache



Cavity detection summary

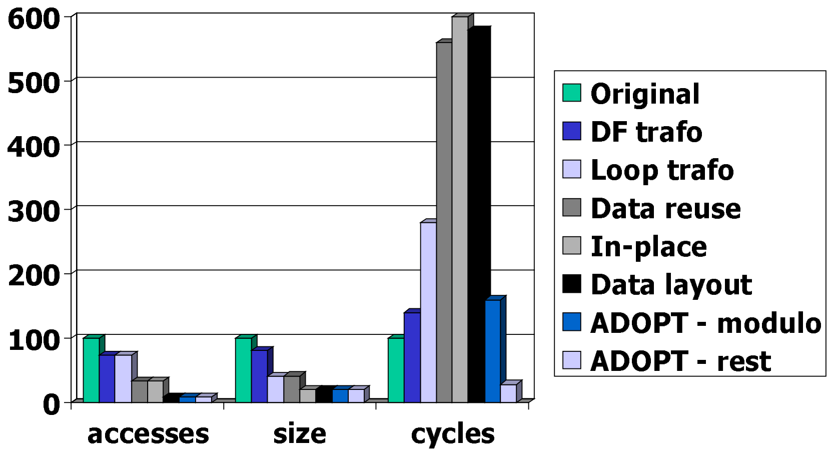


- Local accesses reduced by factor 3
- Memory size reduced by factor 5
- Power reduced by factor 5
- System bus load reduced by factor 12
- Performance worsened by factor 6

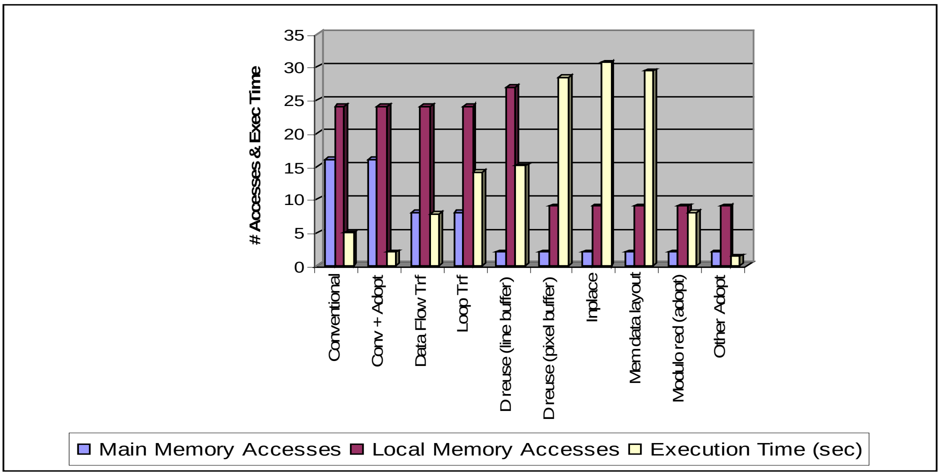
The last step

- Increased execution time introduced by DTSE from
  - Complicated address arithmetic
  - Additional complex control flow (loop & conditionals)
- Multimedia platform not adapted to address calculations
- Additional transformations needed to
  - Simplify control flow
  - Simplify address arithmetic: common sub-expression elimination, modulo expansion, …
  - Match remaining expressions on target machine

Address optimization - result

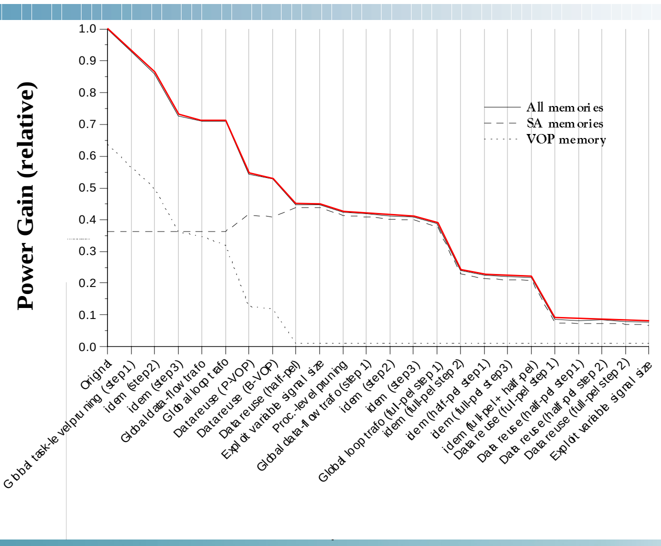


Cavity detection on Pentium-MMX



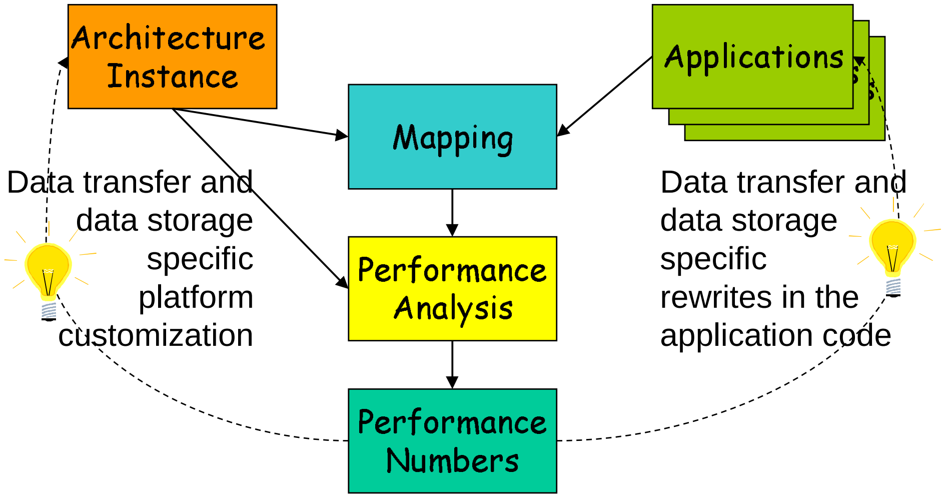
DAC System Level Design with Embedded Platforms Tutorial

MPEG-4 motion estimation on MMX



DAC System Level Design with Embedded Platforms Tutorial

The Y-chart revisited



Architecture Instance

Applications

Mapping

Performance Analysis

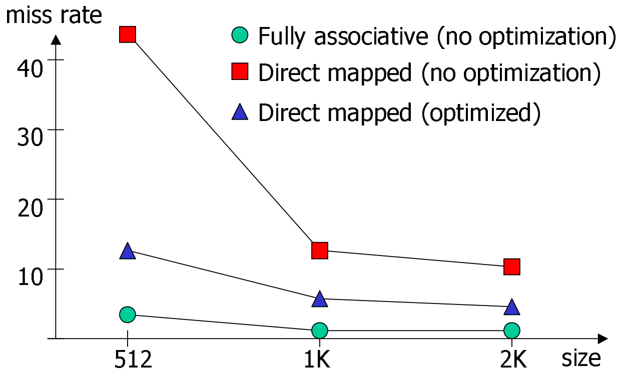
Performance Numbers

Data transfer and data storage specific platform customization

Data transfer and data storage specific rewrites in the application code

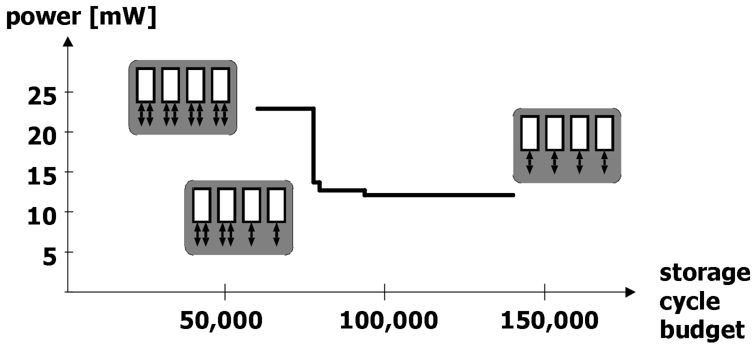
Fixing platform parameters

- Configurable cache size: trade-off versus miss rate



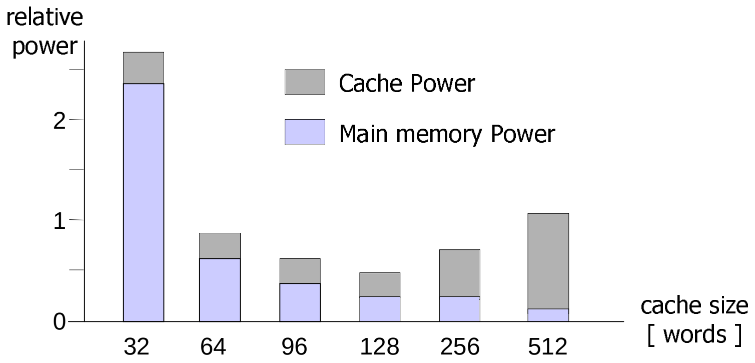
Fixing platform parameters

- Assume configurable on-chip memory hierarchy
  - Trade-off power versus cycle-budget



Fixing platform parameters

- Cache size versus power

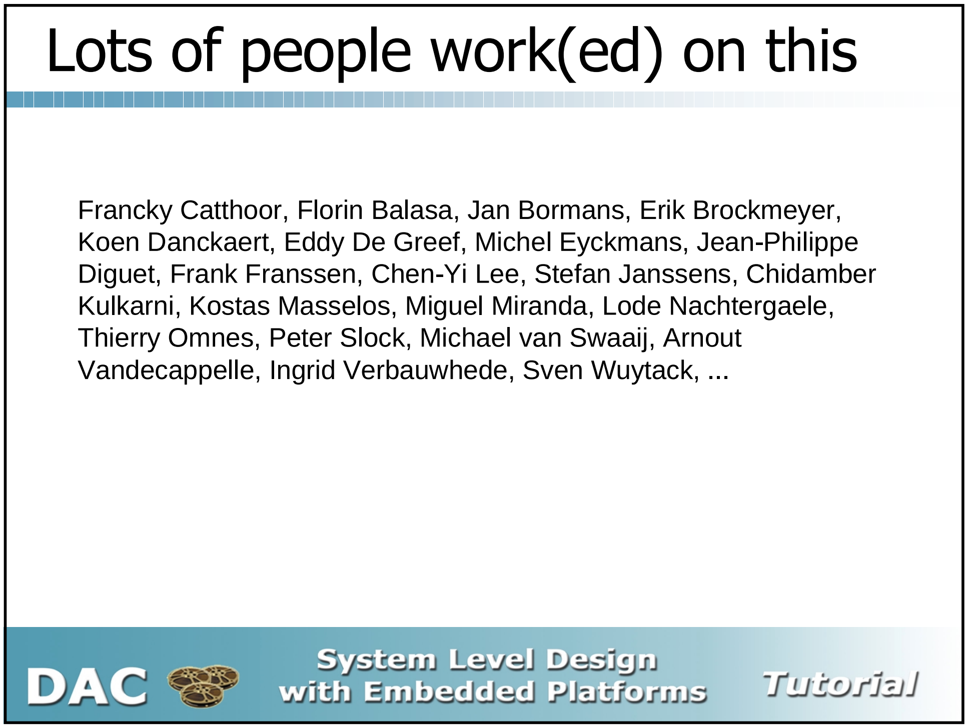


Conclusion

- In multi-media applications exploring data transfer and storage issues should be done at system level
- DTSE is a methodology for *Data Transfer and Storage Exploration* based on manual and/or tool-assisted *code rewriting*
  - Platform independent high-level transformations
  - Platform dependent transformations exploit platform characteristics (optimal use of cache, …)
  - Substantial reduction in power and memory size demonstrated on MPEG-4, OFDM, H.263, ADSL, ...

More information

http://www.imec.be/acropolis/
http://www.imec.be/atomium/



Lots of people work(ed) on this

Francky Catthoor, Florin Balasa, Jan Bormans, Erik Brockmeyer, Koen Danckaert, Eddy De Greef, Michel Eyckmans, Jean-Philippe Diguet, Frank Franssen, Chen-Yi Lee, Stefan Janssens, Chidamber Kulkarni, Kostas Masselos, Miguel Miranda, Lode Nachtergaele, Thierry Omnes, Peter Slock, Michael van Swaaij, Arnout Vandecappelle, Ingrid Verbauwhede, Sven Wuytack, ...