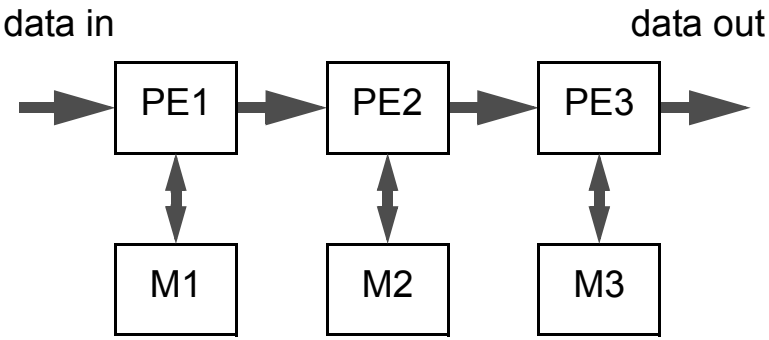
Code Transformations at System Level

Motivation

- Multimedia applications
  - MPEG 4 – > 2 GOP/s, > 6 GB/s, > 10 MB storage
- Embedded systems, SoC, NoC
  - Memory is a power bottleneck
    - MPEG-2 decodes – logic 27%, i/o 24%, clock 24%, on-chip memory 25%
    - CPU + caches – logic 25%, i/o 3%, clock 26%, on-chip memory 46%
  - Memory is a speed bottleneck [Patterson]
    - Moore's Law – μProc 60%/year
    - DRAM – 7%/year
    - Processor-memory speed gap – 50%/year!
- High Performance Computing
  - CPU constrained vs Memory constrained



What can be done?

- Reduce overheads!
  - do necessary calculations only
    - do not duplicate calculations vs. do not store unnecessary data
    - data must have the right size (bitwidth, array size)
  - select the right data layout
    - bitwidth can be reduced by coding
    - number of switchings on a bus can be reduced by coding
  - exploit memory hierarchy
    - local copies may significantly reduce data transfer
  - distribute load
    - systems are designed to stand the peak performance
- But do not forget trade-offs...

Data layout optimization

- Input/output data range given
  - what about intermediate data?
- Integer vs. fixed point vs. floating point
  - integer
    - + simple operations
    - - no fractions
  - fixed point
    - + simple add/subtract operations
    - - normalization needed for multiplication/division
  - floating point
    - + flexible range
    - - normalization needed

Data layout optimization
Software implementations

- Example: 16-bit vs. 32-bit integers?
  - depends on the CPU, bus & memory architectures
  - depends on the compiler (optimizations)
  - 16-bit
    - + less space in the main memory
    - + faster loading from the main memory into cache
    - - unpacking/packing may be needed (compiler & CPU depending)
    - - relative redundancy when accessing a random word
  - 32-bit
    - + data matches the CPU word ["native" for most of the modern CPUs]
    - - waste of bandwidth (bus load!) for small data ranges
  - The same applies 32 vs. 64-bit integers

Data layout optimization
Hardware implementations

- Example #1: 16-bit integer vs. 16-bit fixed point vs. 16-bit floating point
  - integer – 1+15 bits ~~ -32000 to +32000, precision 1
  - fixed point – 1+5+10 bits ~~ -32 to +32, precision ~0.001 (~0.03%)
    - normalization == shifting
  - floating point – 1+5+10 bits ~~ -64000 to +64000, precision ~0.1%
    - normalization == analysis + shifting

- Example #2: 2's complement vs. integer with sign
  - 2's complement – +1 == 00000001, 0 == 00000000, -1 == 11111111
  - integer + sign – +1 == 00000001, 0 == 00000000/10000000, -1 == 10000001
  - number of switchings when data swings around zero!!!! [e.g., DSP]

Optimizing the number of calculations

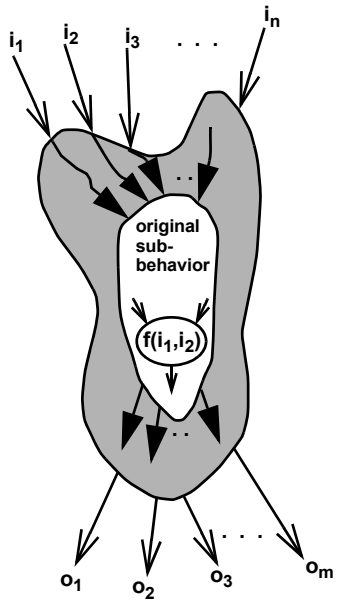
- Input space adaptive design
  - Wang, et al @ Princeton [DAC'2001]
- The main idea – avoid unnecessary calculations
  - input data is analyzed for boundary cases
  - boundary cases have simplified calculations (vs. the full algorithm)
  - "boundary case" – it may happen in the most of the time
    - ... and the full algorithm is actually calculating nothing
- Trade-offs
  - extra hardware is introduced
  - implementation is faster
  - implementation consumes less power

Input space adaptive design – Trade-offs

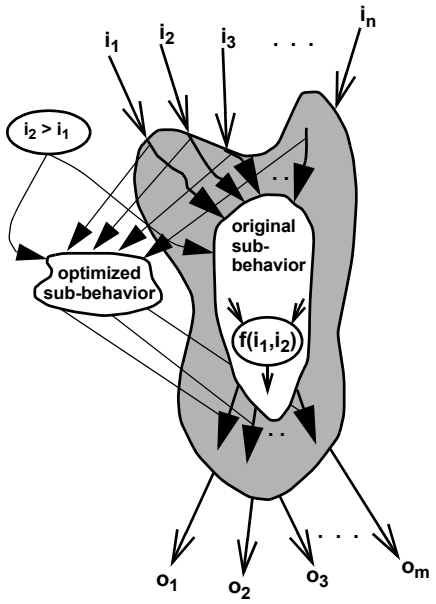
- Extra hardware is introduced
  - the algorithm is essentially duplicated (or triplicated or ...)
  - extra units for analysis are needed
- Implementation is faster
  - less operations require less clock steps
  - operations themselves can be faster
- Implementation consumes less power
  - a smaller number of units are involved in calculations
  - the clock frequency can be slowed
  - supply voltage can be lowered

Input space adaptive design – The Case

- **Discrete wavelet transformation**
  - four input words – A, B, C, D
  - in 95.2% cases: A = B = C = D
  - four additions, four subtractions and four shifts are replaced with four <u>assignments</u>
- **Results**
  - area: +17.5%
  - speed: 52.1% (cycles)
  - power: 58.9%
    V_{dd} scaled: 26.6%



original behavior



optimized behavior

Input space adaptive design

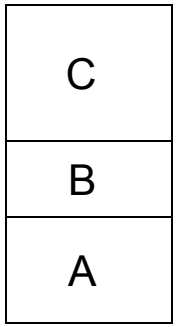
Few examples

- Multiplication with zero
  - result is zero –> nothing to calculate
  - two comparators, few blocking gates, plus some extras to the controller
  - multiplication with one, etc.
  - it must be cost efficient – the extra HW consumes power too!
- Multiplier size
  - $A_{32}*B_{32} = (A_{H16}*2^{16}+A_{L16})*(B_{H16}*2^{16}+B_{L16}) = ...$
  - $... = A_{H16}*B_{H16}*2^{32}+A_{H16}*B_{L16}*2^{16}+A_{L16}*B_{H16}*2^{16}+A_{L16}*B_{L16}$
  - $A < 2^{16}$ & $B < 2^{16}$ –> $A_{H16}=0$ & $B_{H16}=0$
- Multiplication with a constant –> shift-adds
  - 6*x = 4*x+2*x = (x<<2) + (x<<1)
  - - 1.75*x = 0.25*x - 2*x = (x>>2) - (x<<1)

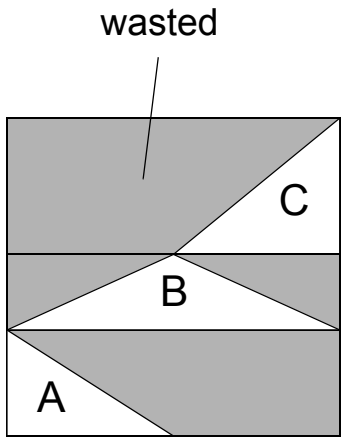
Memory architecture optimization

Life time of array elements

B = f(A);
C = g(B);



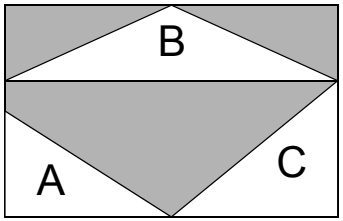
typical allocation



life times

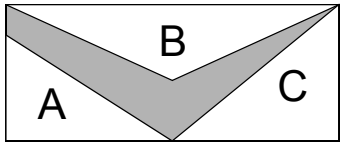
a[i]==mem[i]
b[i]==mem[base_b+i]
c[i]==mem[base_c+i]

Improved allocations



a smarter allocation

a[i]==mem[i]
b[i]==mem[base_b+i]
c[i]==mem[i]



the smallest memory waste

a[i]==mem[i]
b[i]==mem[size-i]
c[i]==mem[i]

Memory architecture optimization

Code transformations to reduce/balance data transfers

- ATOMIUM + ACROPOLIS
  - M. Miranda, A. Vandecapelle, E. Brockmeyer, P. Slock, F. Catthoor, et al @ IMEC
- Code transformations to reduce/balance data transfers
- Data copies to exploit memory hierarchy
  - Remove redundant transfer, increase locality
  - Explicit data reuse: add explicit data copies
  - Reduce capacity misses: advanced life time analysis
  - Reduce conflict misses: change data layout

Code transformations

- **Loop transformations**
  - **improve regularity of accesses**
  - **improve locality: production –> consumption**
  - **expected influence – reduce temporary and background 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];
}
```

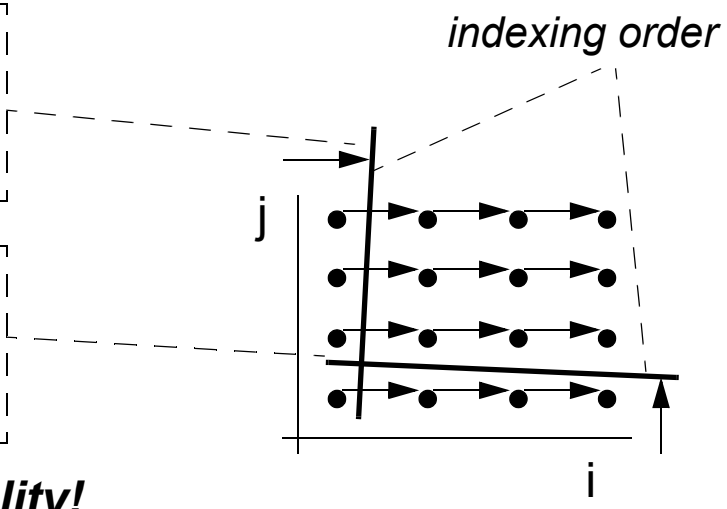
storage size 1

Polyhedral model - polytope placement

Life time analysis of array elements

```
for (i=1; i<=N; i++)
  for (j=N; j>=1; j--)
    A[i][j]=foo(A[i-1][j],...);
```

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



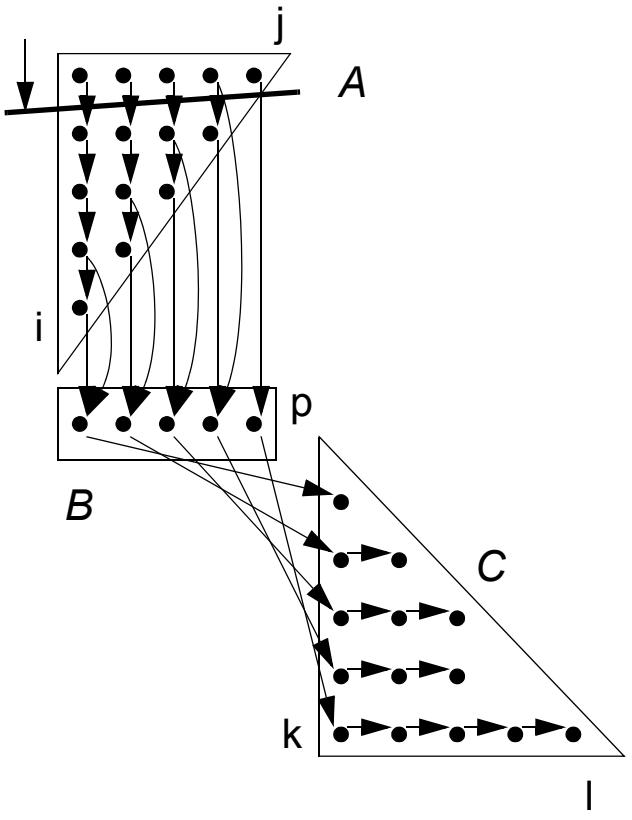
The indexing order must preserve causality!

Storage requirements – # of dependency lines

- **select the right indexing order**
- **merge the loops**
- **can be used for "loop pipelining" – possible index range expansion**

Polytope placement – initial code

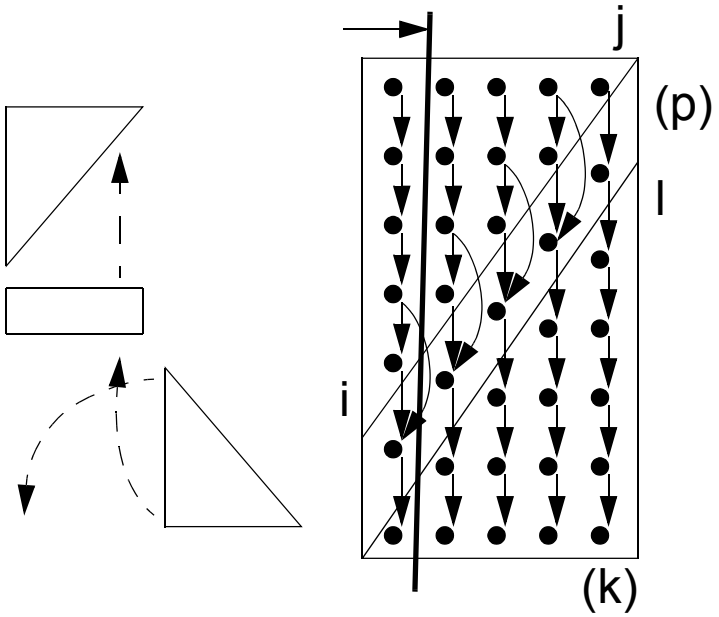
```
/* A */
for (i=1; i<=N; i++)
  for (j=1; j>=N-i+1; j++)
    a[i][j]=in[i][j]+a[i-1][j];
/* B */
for (p=1; p<=N; p++)
  b[p][1]=f(a[N-p+1][p],a[N-p][p]);
/* C */
for (k=1; i<=N; i++)
  for (l=1; l>=k; l++)
    b[k][l+1]=g(b[k][l]);
```



- **storage size – 2*N-1**

Polytope placement – reordering + loop merging

```
for (j=1; j<=N; j++) {
  for (i=1; i>=N-j+1; i++)
    a[i][j]=in[i][j]+a[i-1][j];
  b[j][1]=f(a[N-j+1][j],a[N-j][j]);
  for (l=1; l>=j; l++)
    b[j][l+1]=g(b[j][l]);
}
```



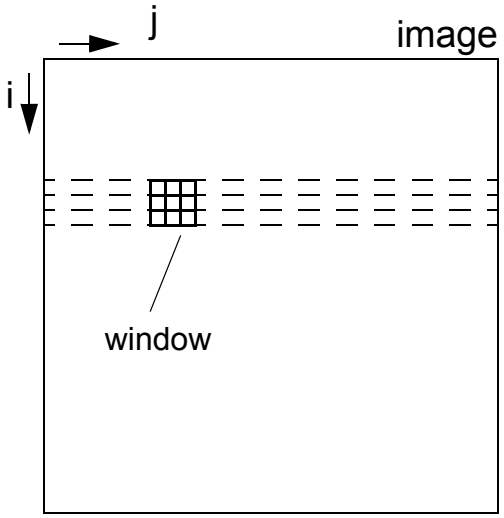
- **storage size – 2**

Exploiting memory hierarchy

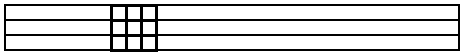
- Inter-array in-place mapping reduces the size of combined memories
  - source array is consumed while processing and can be replaced with the result
- Introducing local copies reduces accesses to the (main) memory
  - cost of the local memory should be smaller than the cost of reduced accesses
- Proper base addresses reduce cache penalties
  - direct mapped caches –
    the same block may be used by two different arrays accessed in the same loop!
- All optimizations
  - ~60% of reduce in memory accesses
  - ~80% of reduce in total memory size
  - ~75% of reduce in cycle count

Introducing local copies

Image processing
blurring
edge detection
etc.



no buffering –
9 accesses per pixel



3 line buffers
(in cache) –
3 accesses per pixel
in the buffer memory

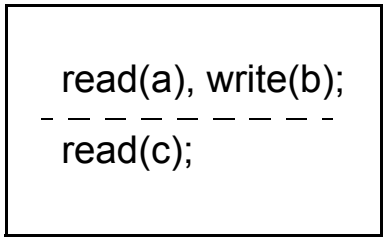
9 pixels in register file –
1 access per pixel
in the main memory

Extra copies should be analyzed for efficiency – time & power

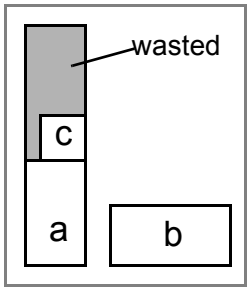
Memory architecture optimization

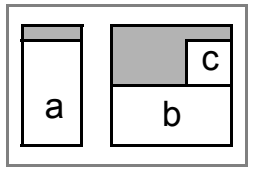
Packing data into memories (#1)

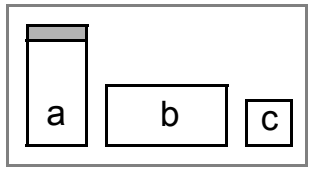
- Packing arrays into memories
- Memory allocation and binding task
  - memory access distribution in time
    - simultaneous accesses
  - mapping data arrays into physical memories
    - single-port vs. multi-port memories
    - unused memory area



access sequence







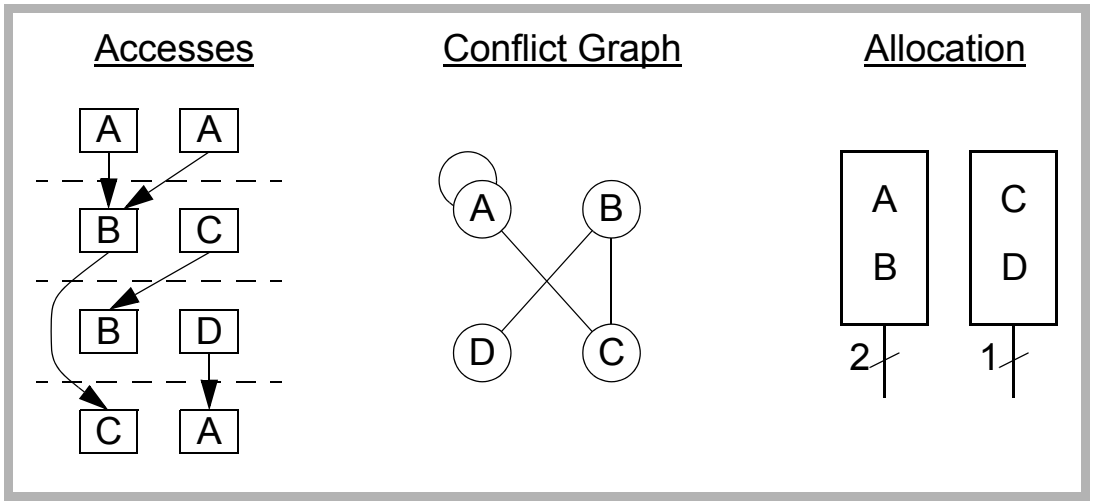
legal mappings

Packing data into memories (#2)

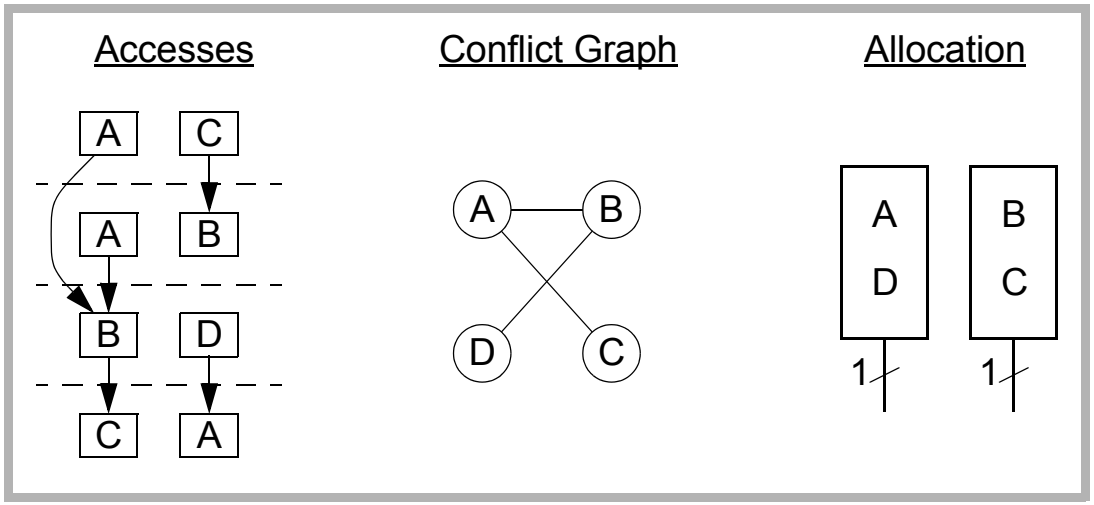
Storage operation scheduling

- Usually scheduled together with other operations
- Affects memory architecture
  - number of memories
  - single-port versus multi-port memories
- Memory operations can be scheduled before others to explore possible memory architectures
  - puts additional constraints for ordinary scheduling
  - distributes access more uniformly – reduces required bus bandwidth
  - reduces the number of possible combinations

Storage operation scheduling



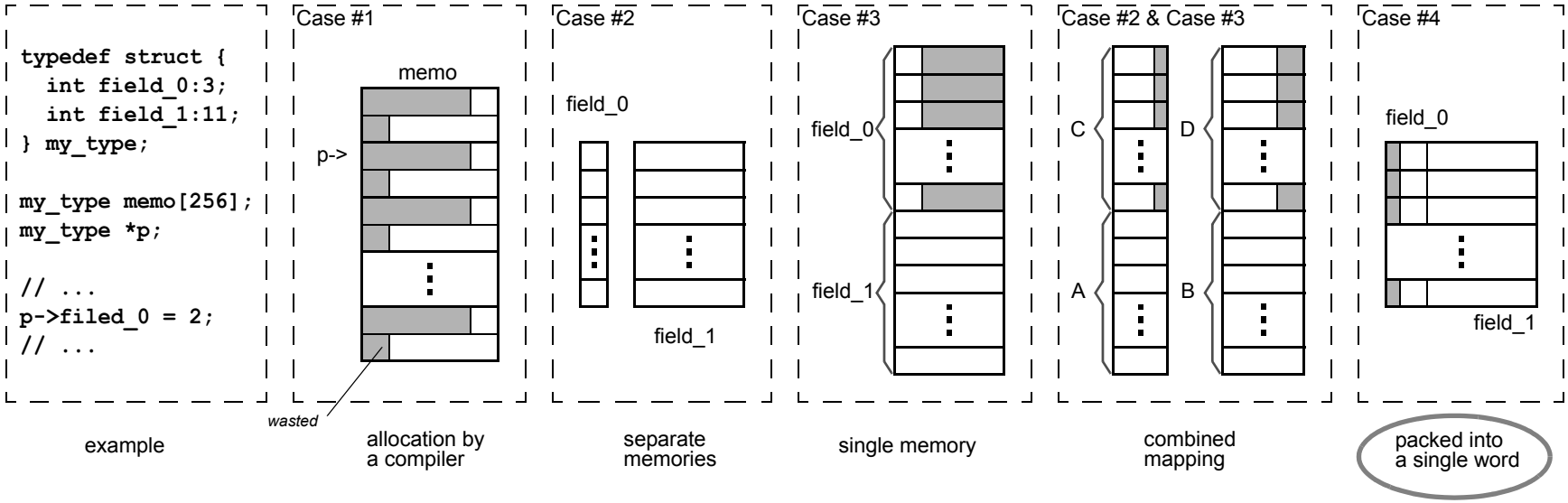
Initial



After scheduling of memory accesses

Packing data into memories (#3)

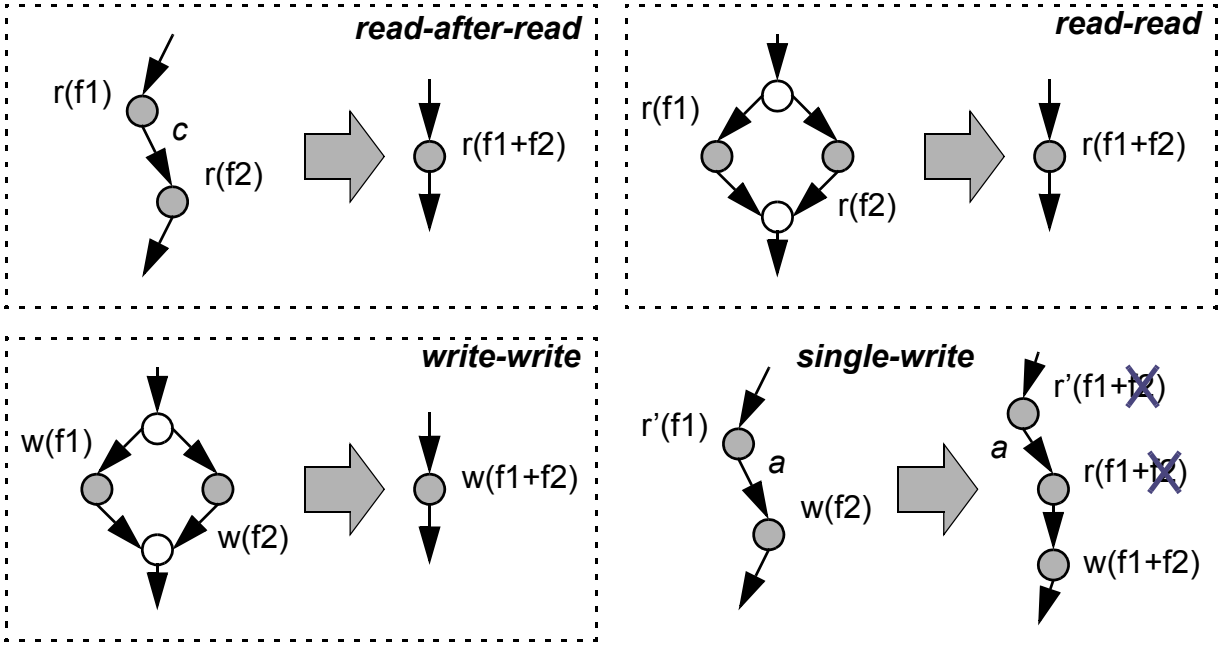
Packing data words (data fields) into memory words



- **How to select candidates for packing?**

Packing data fields into memory words

- Data access duplications may occur
  - a word (field) is read but discarded
  - extra reads may be needed to avoid data corruption when writing



Memory architecture optimization

Address code optimization

- ADOPT – Processor architecture independent ADDress code OPTimization
  - M. Miranda, F. Catthoor, et al @ IMEC
- An array access ...
  - base address
  - dimensions
    - multidimensional arrays vs. physical memories
  - repeated access patterns
    - e.g., windows in image processing
- ... may require a large number of calculations
  - duplications can be removed
  - operations can be simplified

Code hoisting – loop-invariant code motion

- Address calculations that are repeated at every iteration
- Not handled by compilers - it's out of basic block optimization

```
for (k=-4; k<3; k++)
  for (i=-4; i<3; i++)
    Ad[32*(8+k)+16+i]
```

(3+, 1*) x 49 = 245

```
272+32*k+i
```

```
for (k=-4; k<3; k++) {
  tmp_k = 272 + 32*k;
  for (i=-4; i<3; i++)
    Ad[tmp_k+i]
}
```

(1+, 1*) x 7 +

+ (1+) x 49 = 70

Modulo operations

```
for (i=0; i<=20; i++)
  addr = i % 3;
```

```
ptr = -1;
for (i=0; i<=20; i++) {
  if (ptr>=2) ptr-=2;
  else ptr++;
  addr = ptr;
}
```

Counter base address generation units

- Customized hardware
  - special addressing units are cheaper
  - data-path itself gets simpler
  - essentially distributed controllers

Scheduling of high performance applications

- Critical path – sequence of operations with the longest finishing time
  - the longest path in the (weighted) graph
  - Critical path in clock steps
  - Critical path inside clock period (in time units)
  - Topological paths vs. false paths
  - Latency ~ clock_period * critical_path_in_clock_steps

Loop folding and pipelining

- Loop folding
  - fixed number of cycles
  - no need to calculate conditions in loop header
    - iterations of loop body can overlap
- Loop pipelining
  - executing some iterations of loop body in parallel (without data dependencies)

Scheduling of high performance applications

Speculative execution

- Out of order execution
- Execution of operations in a conditional branch can be started before the branch itself starts
- Extra resources required
- Speeding up the algorithm
  - guaranteed speed-up
  - overall speed-up (statistical)
- Scheduling outside basic blocks
  - “flattening” hierarchy
  - increase in the optimization complexity

Conclusion

- Efficient implementation is not only the hardware optimization
- Efficient implementation is
  - selecting the right algorithm
  - selecting the right data types
  - selecting the right architecture
  - making the right modifications
  - and optimizing...
  - right == the most suitable
- Think first, implement later!