332
Advanced Computer Architecture
Chapter 4

Compiler issues: dependence analysis, vectorisation, automatic parallelisation

February 2007

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd and 4th eds),* and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

Background reading

The material for this part of the course is introduced only very briefly in Hennessy and Patterson (section 4.4 pp319). A good textbook which covers it properly is

- Michael Wolfe. **High Performance Compilers for Parallel Computing**. Addison Wesley, 1996.

Much of the presentation is taken from the following research paper:

- U. Banerjee. **Unimodular transformations of double loops**. In Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, CA. Pitman/MIT Press, 1990.

Banerjee's paper gives a simpified account of the theory in the context only of perfect doubly-nested loops with well-known dependences.

Introduction

- **In this segment of the course we consider compilation issues for loops involving arrays:**
- How execution order of a loop is constrained,
- How a compiler can extract dependence information, and
- How this can be used to optimise a program.

Understanding and transforming execution order can help exploit architectural features:

- Pipelined, superscalar and VLIW processors
- Systems which rely heavily on caches
- Processors with special instructions for vectors (SSE, AltiVec)
- Multiprocessors, multicore, and co-processors/accelerators

Restructuring

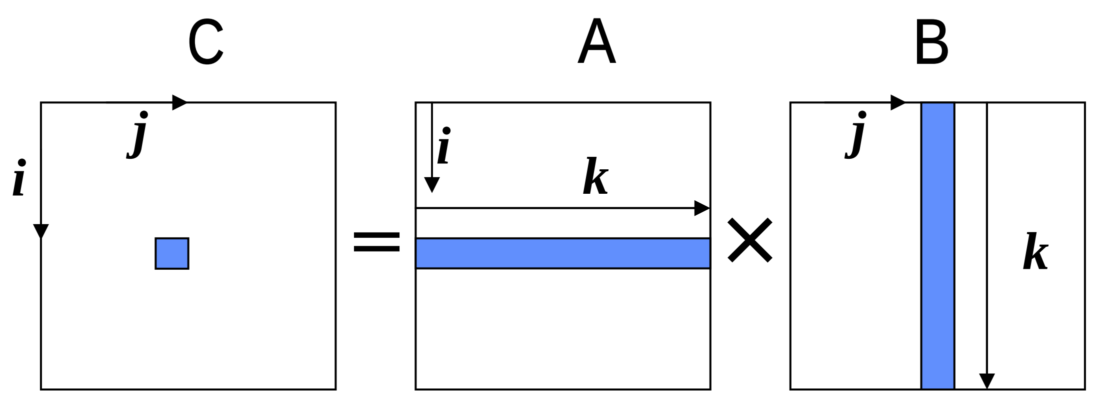
- Here we consider a special kind of optimisation, which is currently performed only by specialist compilers - “restructuring compilers".

- Conventional optimisations must also be performed
- The difference is this:
  - Conventional optimisations reduce the amount of work the computer has to do at run-time
  - Restructuring aims to do the work in an order which suits the target architecture better

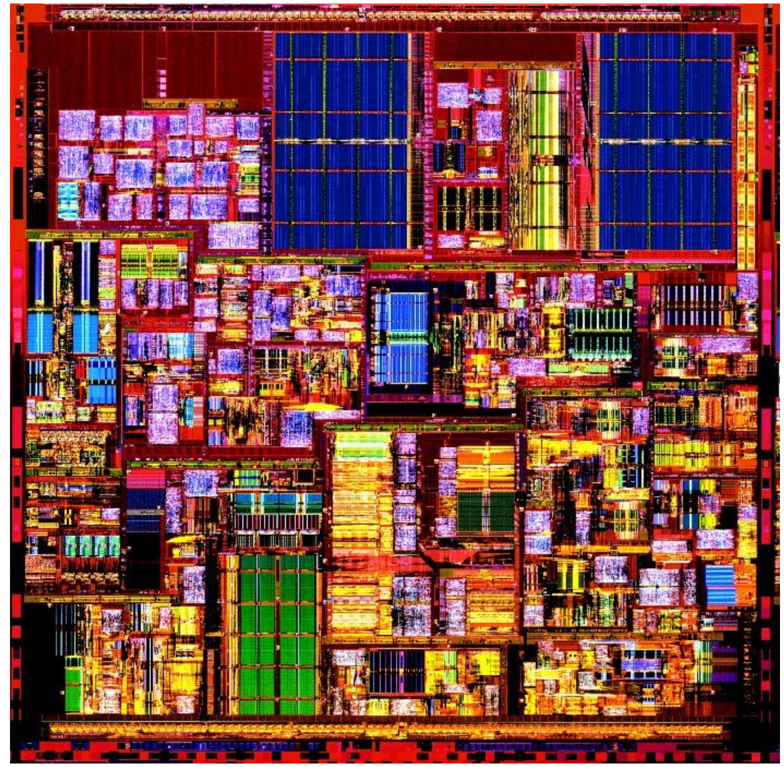
Motivation: an example

```
/*
 * mm: Multiply A by B leaving the
 * result in C.
 * The result matrix is assumed
 * to be initialised to zero.
 */
void mm1(double A[N][N],
         double B[N][N],
         double C[N][N])
{
 int i, j, k;
 for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
   for (k = 0; k < N; k++)
    C[i][j] += A[i][k] * B[k][j];
}
```

- We will begin by looking at double-precision floating point matrix multiply
- We will investigate the performance of various versions in order to determine what transformations a compiler should apply



Outside



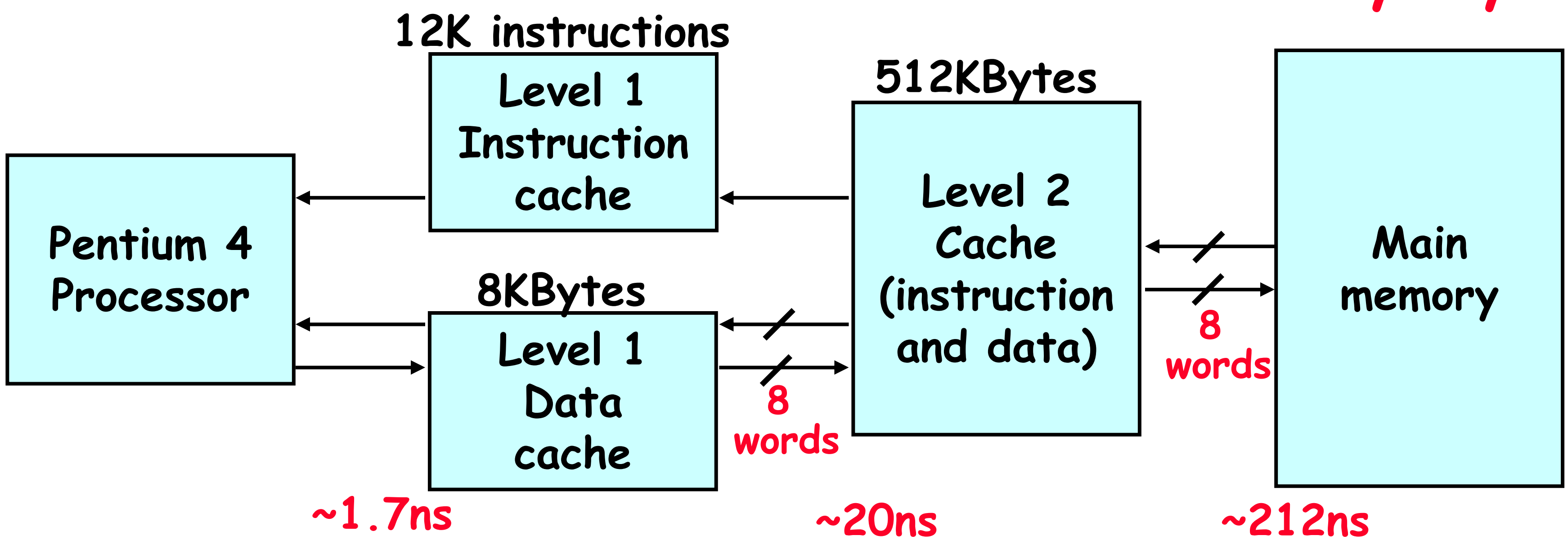
Inside: Pentium 4 processor

Let's experiment with a perfectly ordinary laptop:

- The experiments were performed on a Toshiba Satellite Pro 6100 laptop
- This machine has a 1.6GHz Intel Pentium 4 Mobile processor (we'll look at some other processors shortly)

T21: www.howstuffworks.com, die photo: http://www.sandpile.org/impl/p3.htm

Memory system



- L1 Instruction ("trace") cache: 12K microinstructions
- L1 Data cache: 8 KB, 4-way, 64 bytes/line, non-blocking, dual-ported, write-through, pseudo-LRU
- L2 unified cache: 512 KB, 2-Way, 64 Byte/Line, non-blocking

Suppose we're interested in quite big matrices, N=1088

- **The matrix occupies 1088^2x8 = 9.5MBytes**
- **Each row of the matrix occupies 1088x8 = 8.5KBytes.**

Performance

- For N=1088, the initial version runs in 130 seconds.

- The matrix multiplication takes 1088^3 steps, each involving two floating-point operations, an add and a multiply, i.e. 2.6×10^9 "FLOPs"
- This loop achieves a computation rate of 2600/130=19.8 MFLOPs.

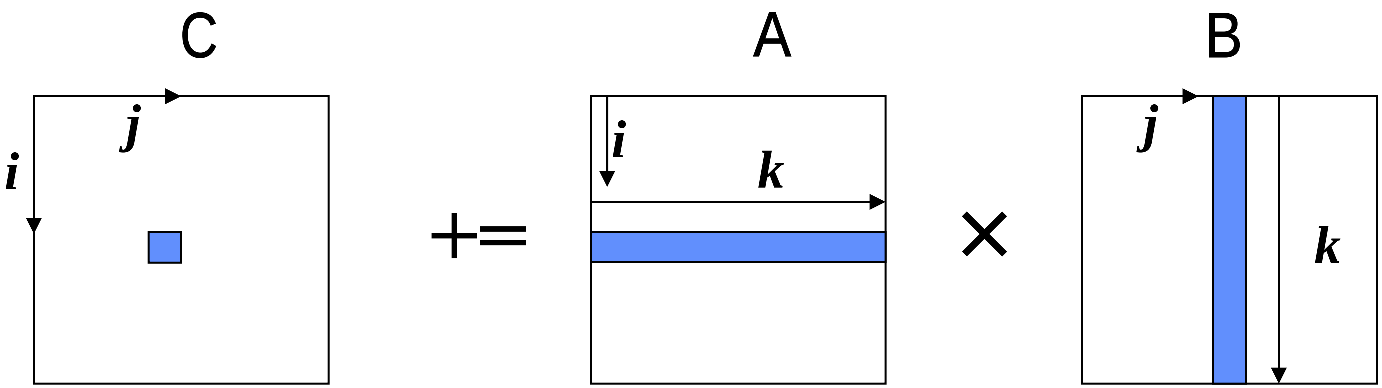
- **That is, one floating-point operation completed every 80 clock cycles (the chip runs at 1.6GHz)**
- **How are we going to get value for money?**

Interchange loops

```
for (i = 0; i < N; i++)
  for (k = 0; k < N; k++)
    for (j = 0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j];
}
```

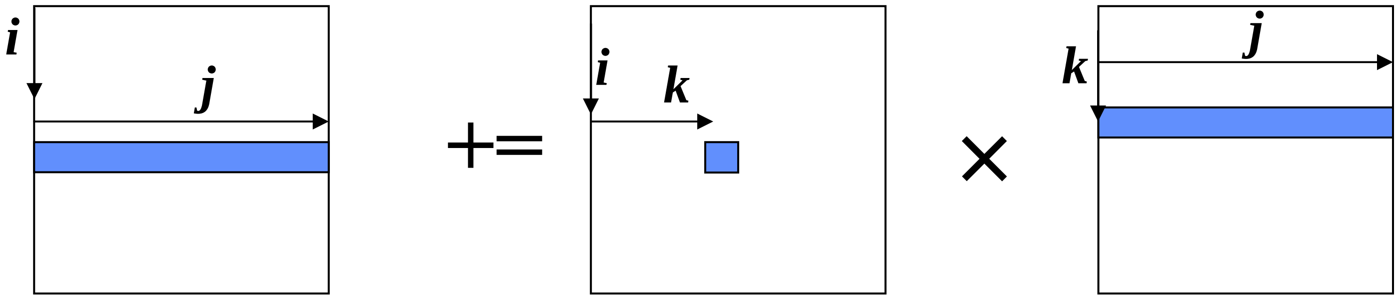
- 9.6 seconds (267 MFLOPS).
- Why is this such a good idea?
- How might a compiler perform this transformation?
- Does it still give the right output?
- **Can we do better still?**

What was going on?




```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  for (k=0; k<N; k++)
   C[i][j]+=A[i][k]*B[k][j]
```

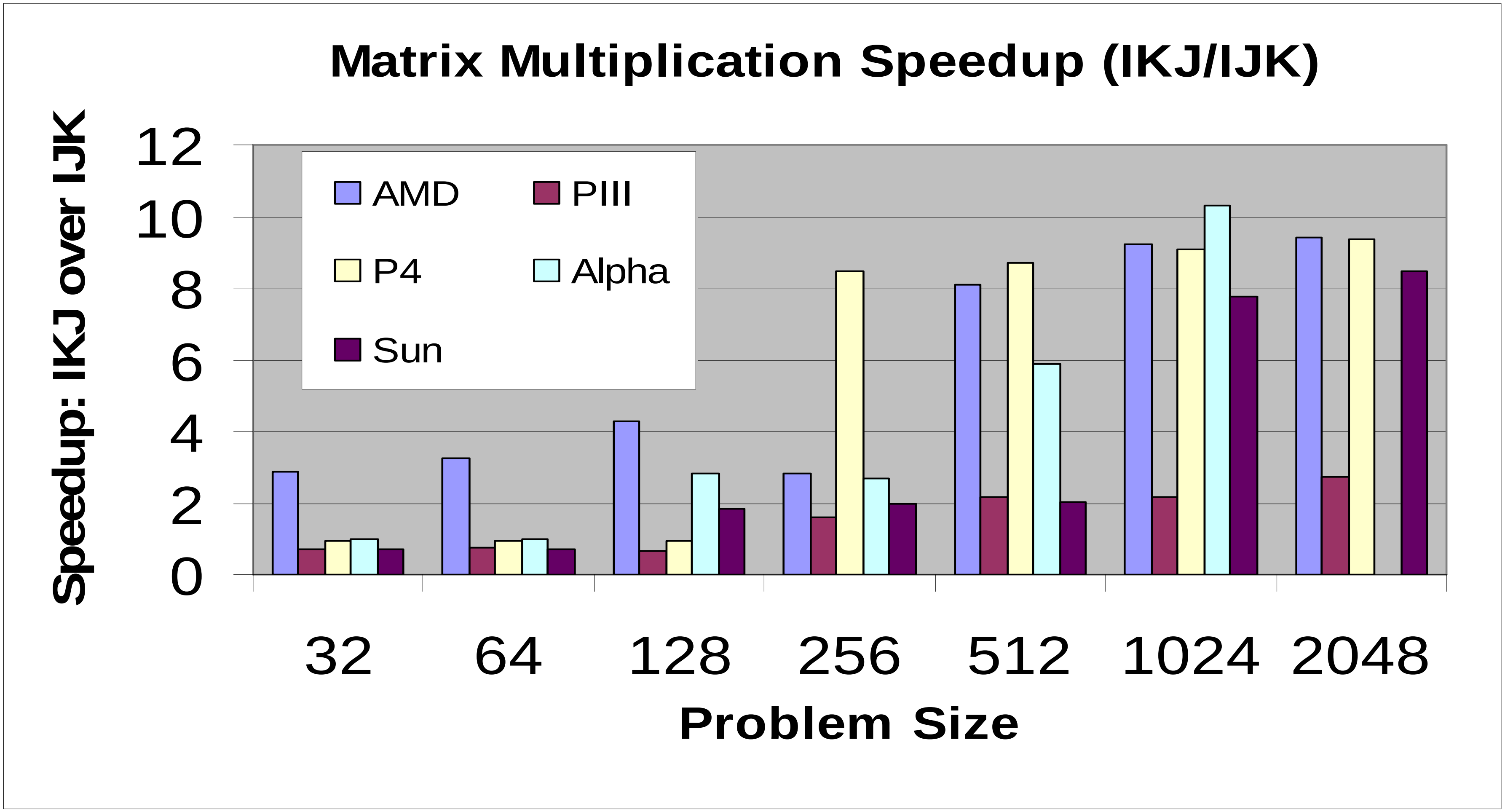
- IJK variant computes each element of result matrix C one at a time, as inner product of row of A and column of B
  - Traverses A in row-major order, B in column-major




```
for (i=0; i<N; i++)
 for (k=0; k<N; k++)
  for (j=0; j<N; j++)
   C[i][j]+=A[i][k]*B[k][j]
```

- IKJ variant accumulates partial inner product into a row of result matrix C, using element of A and row of B
  - Traverses C and B in row-major order

The price of naivety



- Relative speedup of IKJ version over IJK version (per machine, per problem size)
- On large problems, the IKJ variant is 2-10 times faster

Blocking (a.k.a. "tiling")

- Idea: reorder execution of loop nest so data isn't evicted from cache before it's needed again.
- Blocking is a combination of two transformations: "strip mining", followed by interchange; we start with

```
for (i = 0; i < N; i++)
  for (k = 0; k < N; k++){
    r = A[i][k];
    for (j = 0; j < N; j++)
      C[i][j] += r * B[k][j]; }
```

- Strip mine the k and j loops:

```
for (i = 0; i < N; i++)
  for (kk = 0; kk < N; kk += S)
    for (k = kk; k < min(kk+S,N); k++){
      r = A[i][k];
      for (jj = 0; jj < N; jj += S)
        for (j = jj; j < min(jj+S, N); j++)
          C[i][j] += r * B[k][j];
}
```

Blocking/tiling – stripmine then interchange

- Now interchange so blocked loops are outermost:

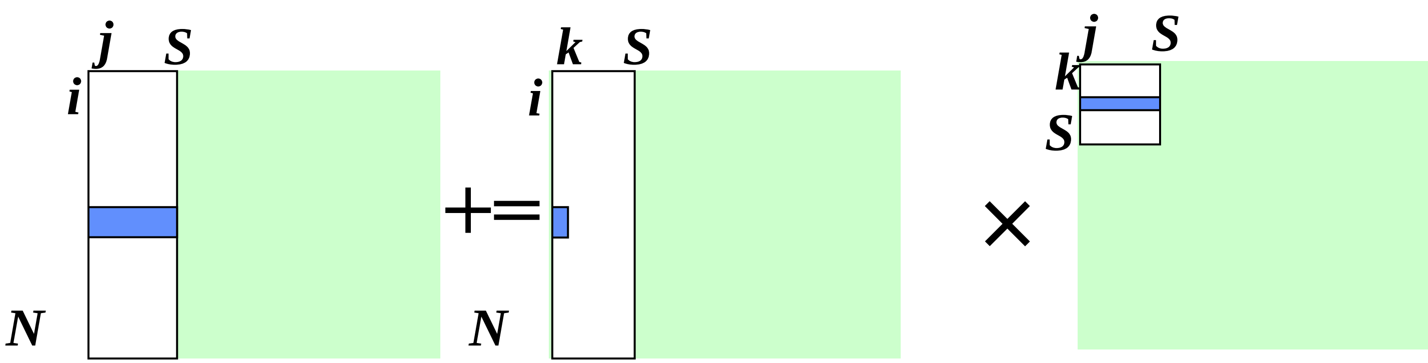
```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++){
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
          C[i][j] += r * B[k][j];
}
```

- The inner i,k,j loops perform a multiplication of a pair of partial matrices.
- S is chosen so that a S x S submatrix of B and a row of length S of C can fit in the cache.
- What is the right value for S?

Blocking/tiling – stripmine then interchange

- Now interchange so blocked loops are outermost:

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++){
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
          C[i][j] += r * B[k][j];
}
```

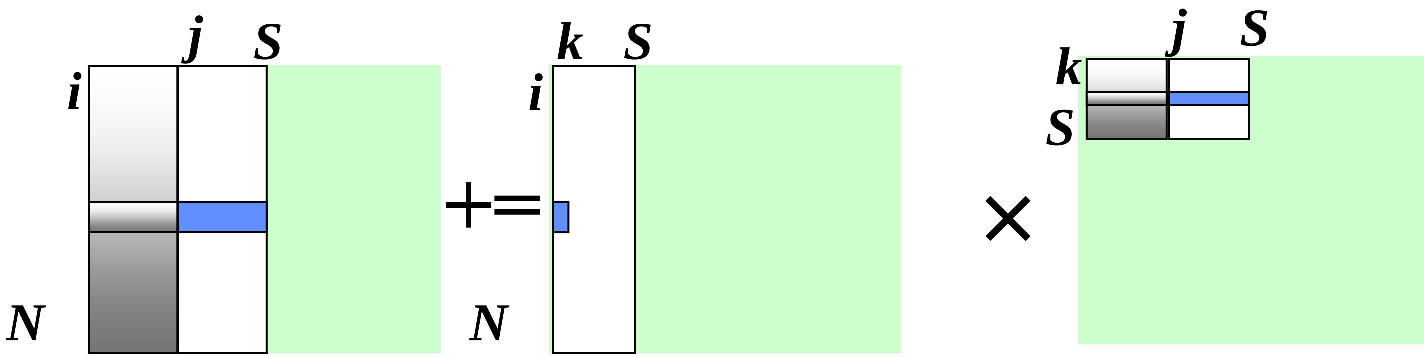


C[0:N][jj:jj+S] += A[0:N][kk:kk+S] * B[kk:kk+S][jj:jj+S]

Blocking/tiling – stripmine then interchange

- Now interchange so blocked loops are outermost:

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++){
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
         C[i][j] += r * B[k][j];
}
```

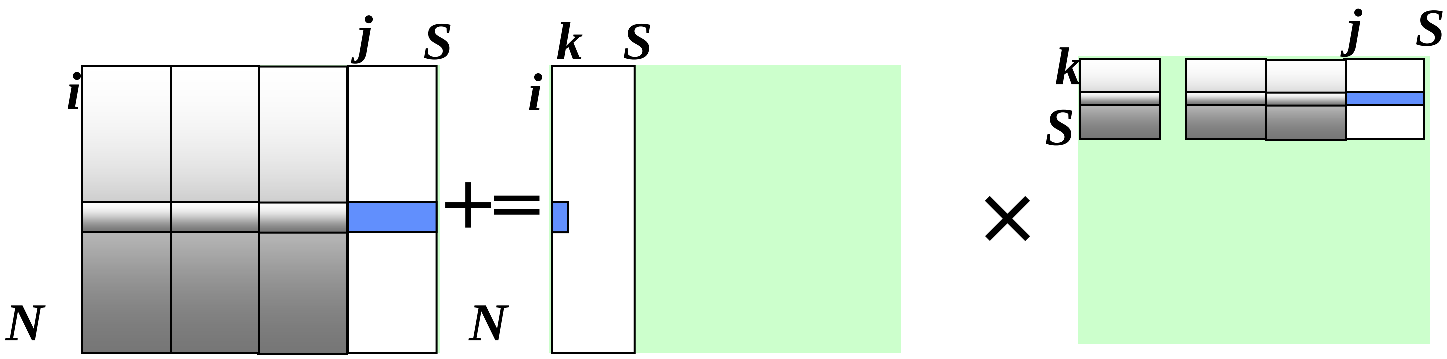


C[0:N][jj:jj+S] += A[0:N][kk:kk+S] * B[kk:kk+S][jj:jj+S] for next jj

Blocking/tiling – stripmine then interchange

- Now interchange so blocked loops are outermost:

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++){
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
         C[i][j] += r * B[k][j];
}
```

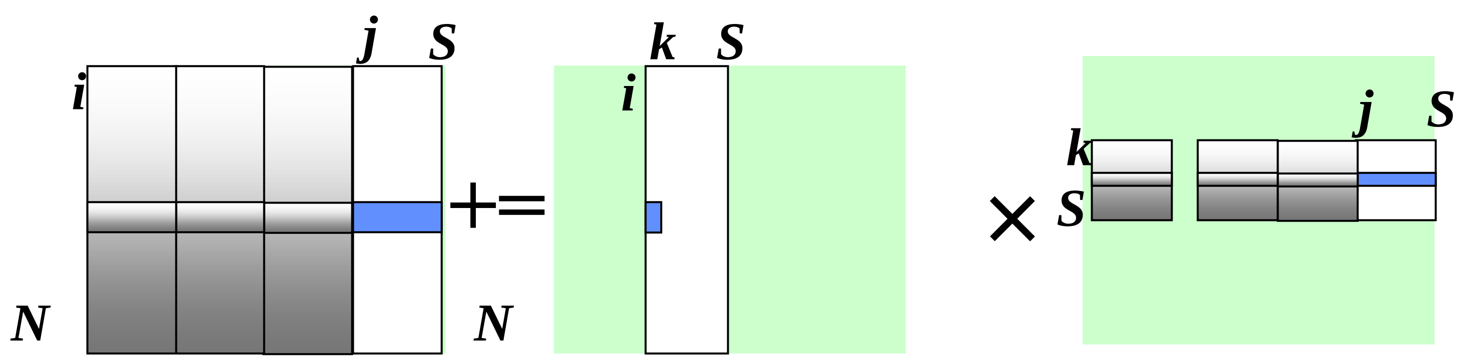


C[0:N][jj:jj+S] += A[0:N][kk:kk+S] * B[kk:kk+S][jj:jj+S] for last jj

Blocking/tiling – stripmine then interchange

- Now interchange so blocked loops are outermost:

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++){
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
         C[i][j] += r * B[k][j];
}
```

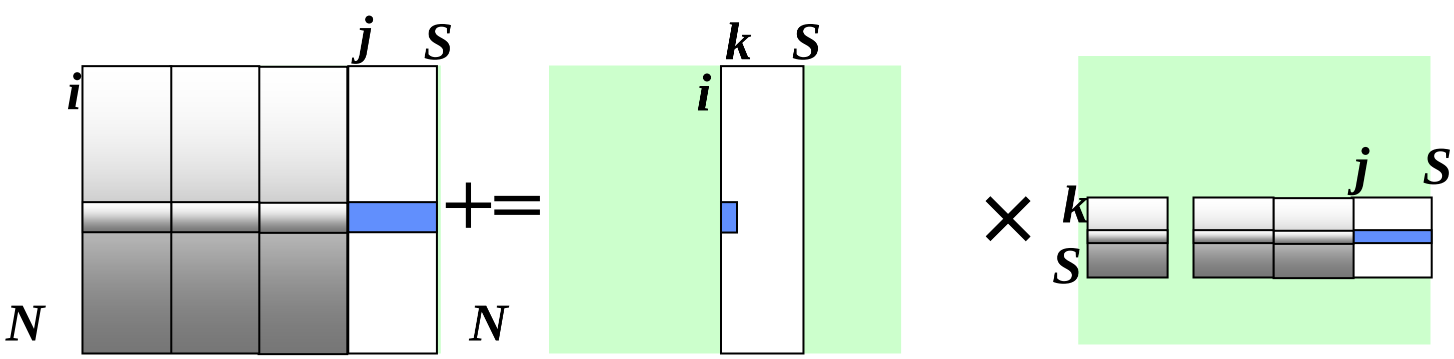


for (jj=0:N:S) C[0:N][jj:jj+S] += A[0:N][kk:kk+S] * B[kk:kk+S][jj:jj+S] for next kk

Blocking/tiling - stripmine then interchange

- Now interchange so blocked loops are outermost:

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++){
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
         C[i][j] += r * B[k][j];
}
```

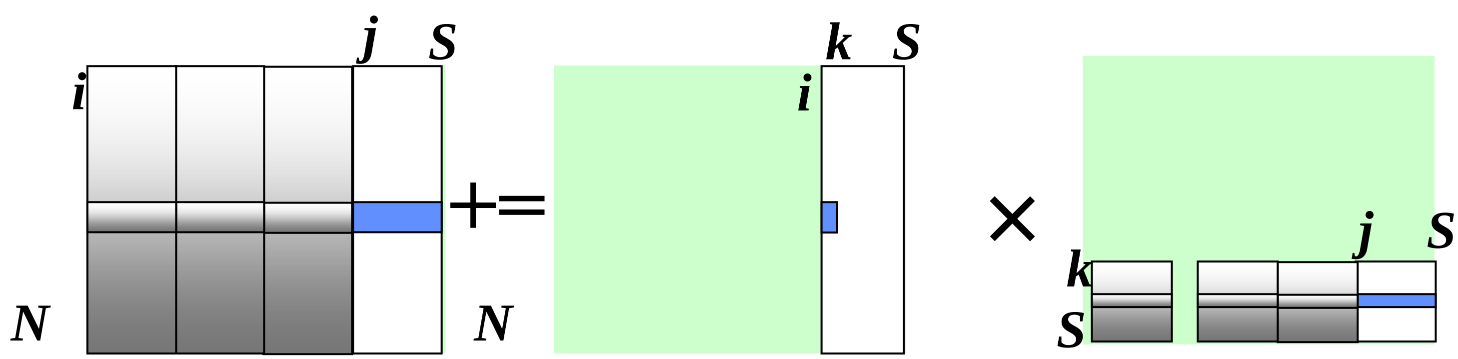


for (jj=0:N:S) C[0:N][jj:jj+S] += A[0:N][kk:kk+S] * B[kk:kk+S][jj:jj+S] for next kk

Blocking/tiling – stripmine then interchange

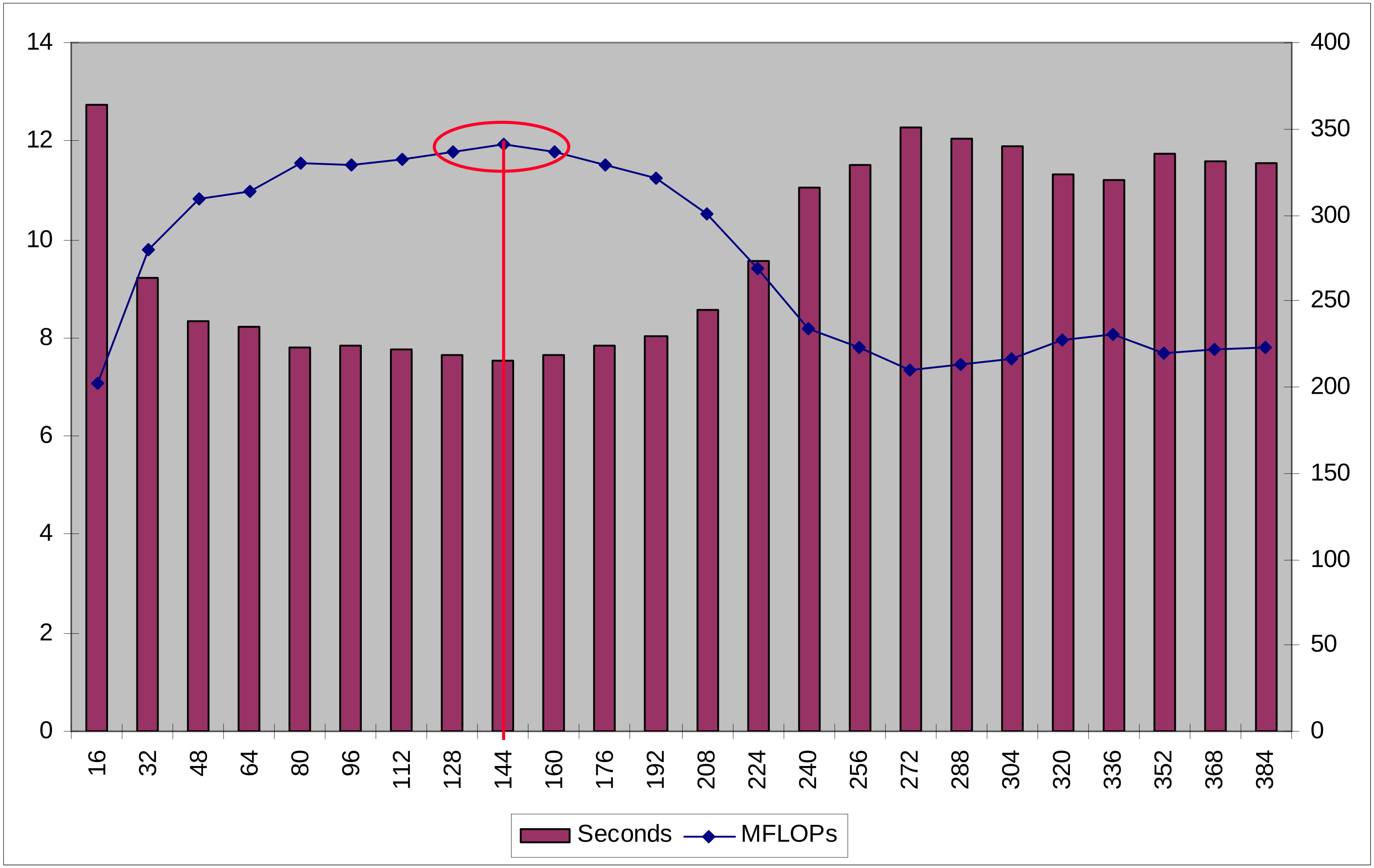
- Now interchange so blocked loops are outermost:

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++){
        r = A[i][k];
        for (j = jj; j < min(jj+S, N); j++)
         C[i][j] += r * B[k][j];
}
```



for (jj=0:N:S) C[0:N][jj:jj+S] += A[0:N][kk:kk+S] * B[kk:kk+S][jj:jj+S] for last kk

Performance of blocked version: 1.6GHz Pentium 4M (N=1088)



Problem size 1088 **1.6GHz Pentium 4 Mobile (gcc3.4.4)**

Optimum blocking factor is 144, where we reach 341 MFLOPs

Performance of blocked version: Thinkpad T60 (N=1003)

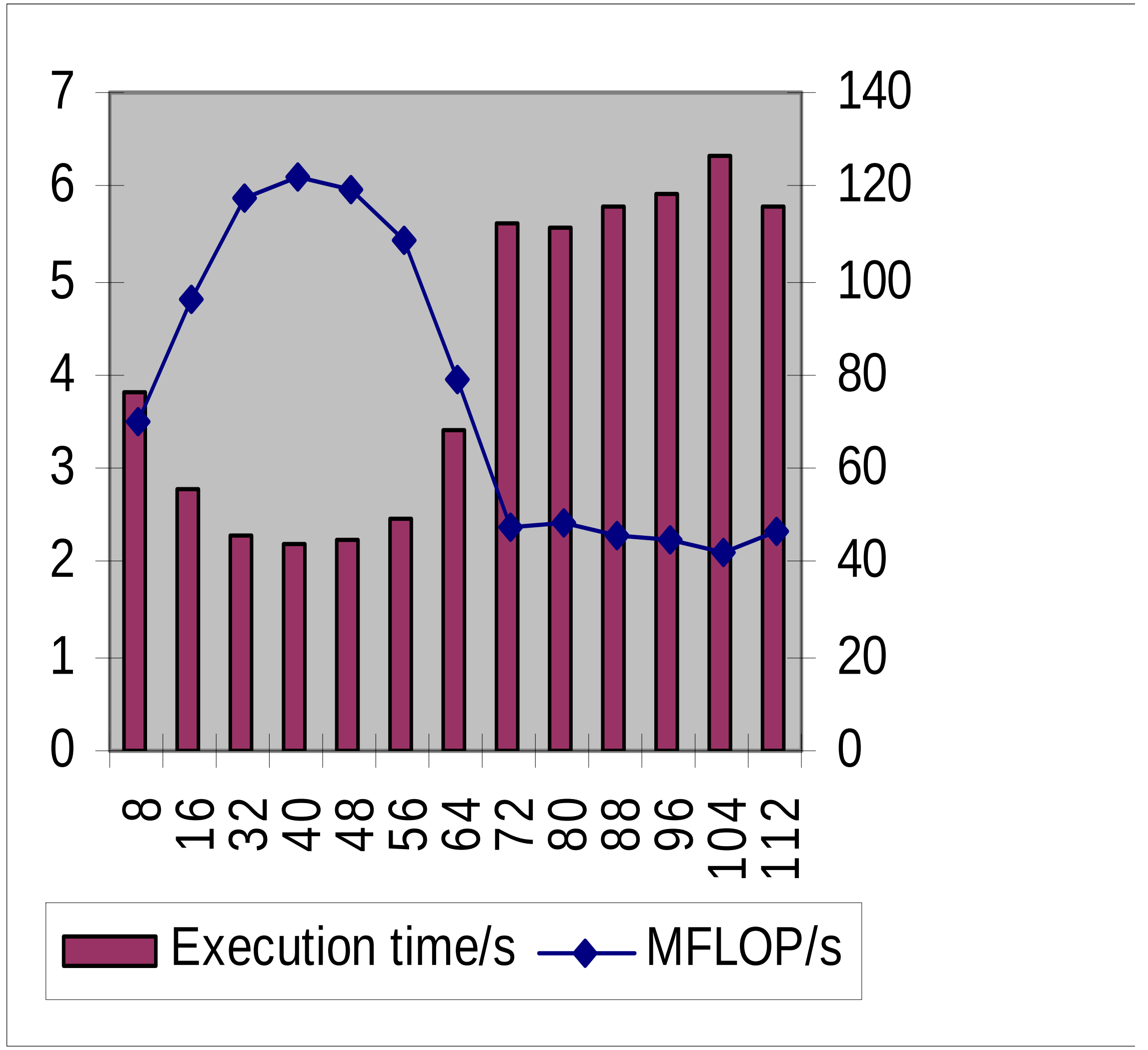
1.8 GHz Intel Core Duo (Lenovo Thinkpad T60) (gcc3.4.4)

Optimum blocking factor is 48, where we reach 866 MFLOPs
Avoiding "min" operator doesn't help.
On battery power, clock rate drops to 987MHz, so only 469 MFLOPS (48 still best). In direct proportion to clock rate reduction.

Performance of blocked version: Pentium 3 (N=512)

Blocking factor	Execution time	MFLOPS
8	3.815	70.4
16	2.784	96.4
32	2.283	117.6
40	2.193	122.4
48	2.253	119.1
56	2.473	108.5
64	3.404	78.9
72	5.608	47.9
80	5.578	48.1
88	5.808	46.2
96	5.928	45.3
104	6.309	42.5
112	5.778	46.5

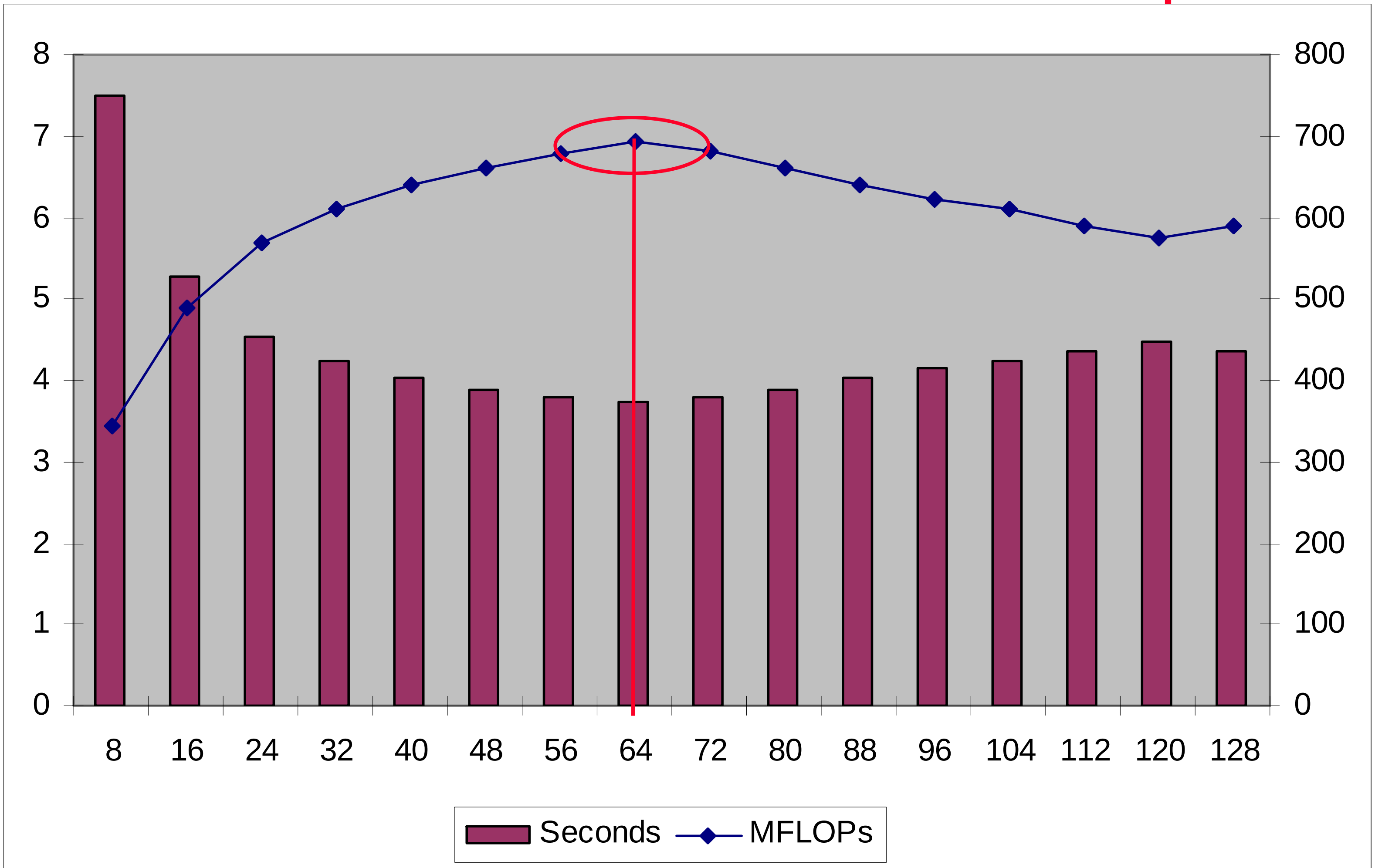
Thinkpad T21 800MHz Pentium III (VS6.0)



The min operators are a performance hit; if we choose a good blocking factor which divides the problem size exactly...

Blocksize 32: 2.013 seconds, 133.4 MFLOP/s

Performance of blocked version: Opteron (N=1088)



Problem size 1088 2.4 GHz AMD Opteron (gcc3.4.3)

Optimum blocking factor is 64, where we reach 692.4 MFLOPs

Since 64 divides 1088 exactly, we can avoid "min" operator, giving 833.6 MFLOPs

Using Intel compiler (-Wl,-melf_i386) this reaches 998 MFLOPs

Using AMD's AMCL library this machine can reach ~4GFLOPS… there is a lot more you can do to improve matrix multiply performance!

Impact….

On Toshiba Satellite Pro 6100 laptop (1.6GHz Pentium 4M):

- Original version: 130 seconds (19.8 MFLOP/s)
- Blocked version: 7.55 seconds (341 MFLOP/s)
  - We started with a "good" optimising compiler!
  - Factor of 17 performance improvement.
  - No reduction in amount of arithmetic performed.

- (Using the Intel library or the ATLAS library does even better)

Dependence

How?

- Define:
  - IN(S): set of memory locns which might be read by some execn of statement S
  - OUT(S): set of memory locns which might be written by some execn of statement S
- Reordering is constrained by dependences;
- There are four types:
  - Data ("true") dependence: S1 δ S2
    - OUT(S1) $\cap$ IN(S2)

    ("S1 must write something before S2 can read it")
  - Anti dependence: S1 $\bar{\delta}$ S2
    - IN(S1) $\cap$ OUT(S2)

    ("S1 must read something before S2 overwrites it")
  - Output dependence: S1 δ^{o} S2
    - OUT(S1) $\cap$ OUT(S2)

    ("If S1 and S2 might both write to a location, S2 must write after S1")
  - Control dependence: S1 δ^{c} S2

Dependence

How?

- Define:
  - IN(S): set of memory locns which might be read by some execn of statement S
  - OUT(S): set of memory locns which might be written by some execn of statement S
- Reordering is constrained by dependences;
- There are four types:
  - Data ("true") dependence: S1 δ S2
    - OUT(S1) $\cap$ IN(S2)

    ("S1 must write something before S2 can read it")
  - Anti dependence: S1 $\bar{\delta}$ S2
    - IN(S1) $\cap$ OUT(S2)

    ("S1 must read something before S2 overwrites it")
  - Output dependence: S1 δ^o S2
    - OUT(S1) $\cap$ OUT(S2)

    ("If S1 and S2 might both write to a location, S2 must write after S1")
  - Control dependence: S1 δ^c S2
- These are static analogues of the dynamic RAW, WAR, WAW and control hazards which have to be considered in processor architecture

Loop-carried dependences

- Consider:

```
S1 :    A[0] := 0
        for I = 1 to 8
S2 :      A[I] := A[I-1] + B[I]
```

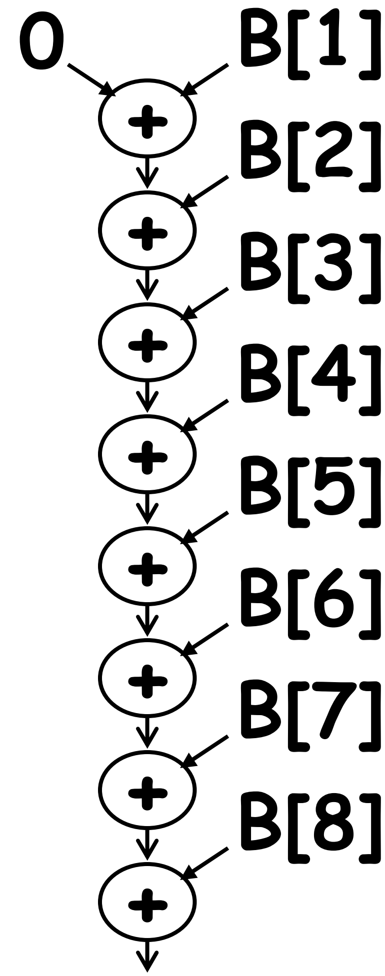
- What does this loop do?

B:		1	1	1	1	1	1	1	1
A:	0								

Loop-carried dependences

- Consider:

```
S1 :      A[0] := 0
          for I = 1 to 8
S2 :        A[I] := A[I-1] + B[I]
```



- What does this loop do?

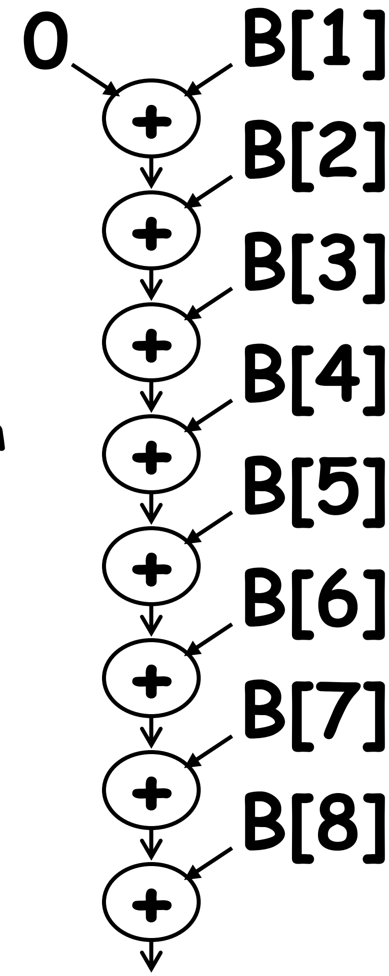
B:		1	1	1	1	1	1	1	1
A:	0	1	2						

- In this case, there is a data dependence
  - This is a loop-carried dependence - the dependence spans a loop iteration
  - This loop is inherently sequential

Loop-carried dependences

- Consider:

```
S1 :    A[0] := 0
        for I = 1 to 8
S2 :      A[I] := A[I-1] + B[I]
```



- Loop carried:

Dependences cross, from one iteration to next

$S2^1$:	A[1] := A[0] + B[1]
$S2^2$:	A[2] := A[1] + B[2]
$S2^3$:	A[3] := A[2] + B[3]
$S2^4$:	A[4] := A[3] + B[4]
$S2^5$:	A[5] := A[4] + B[5]
$S2^6$:	A[6] := A[5] + B[6]
$S2^7$:	A[7] := A[6] + B[7]
$S2^8$:	A[8] := A[7] + B[8]

What is a loop-carried dependence?

- Consider two iterations I^1 and I^2
- A dependence occurs between two statements S_p and S_q (not necessarily distinct), when an assignment in S_p^{I1} refers to the same location as a use in S_q^{I2}
- In the example,

```
S1 : A[0] := 0
     for I = 1 to 5
S2 :   A[I] := A[I-1] + B[I]
```

- The assignment is "**A[I_1] :=** ..."
- The use is "... := **A[I_2-1]** ..."
- These refer to the same location when $I^1 = I^2-1$
- Thus $I^1 < I^2$, ie the assignment is in an earlier iteration

- Notation: $S_2 \, \delta_< \, S_2$

Definition: The dependence equation

- A dependence occurs
  - between two statements S_p and S_q (not necessarily distinct),
  - when there exists a pair of loop iterations I^1 and I^2,
  - such that a memory reference in S_p in I^1 may refer to the same location as a memory reference in S_q in I^2.
- This might occur if S_p and S_q refer to some common array *A*
- Suppose S_p refers to $A[\phi_p(I)]$

($\phi_p(I)$ is some subscript expression involving I)

- Suppose S_q refers to $A[\phi_q(I)]$
- A dependence of some kind occurs between S_p and S_q if there exists a solution to the equation

$$\phi_p(I^1) = \phi_q(I^2)$$

- for integer values of I^1 and I^2 lying within the loop bounds

Types of dependence

- If a solution to the dependence equation exists, a dependence of some kind occurs
- The dependence type depends on what solutions exist
- The solutions consist of a set of pairs (I^1, I^2)
- We would appear to have a *data* dependence if

  $A[\phi_p(I)] \in OUT(S_p)$

  and

  $A[\phi_q(I)] \in IN(S_q)$

- But we only really have a data dependence if the assignments *precede* the uses, ie
  - $S_p \; \delta_< \; S_q$
  - if, for each solution pair (I^1, I^2), $I^1 < I^2$

Dependence versus anti-dependence

- If the *uses* precede the *assignments*, we actually have an *anti*-dependence, ie

$$S_p \ \overline{\delta}_< \ S_q$$

if, for each solution pair (I^1, I^2), $I^1 > I^2$

- If there are some solution pairs (I^1, I^2) with $I^1 < I^2$ and some with $I^1 > I^2$, we write

$$S_p \ \delta_* \ S_q$$

- If, for all solution pairs (I^1, I^2), $I^1 = I^2$, there are dependences *within* an iteration of the loop, but there are no loop-carried dependences:

$$S_p \ \delta_= \ S_q$$

Dependence distance

In many common examples, the set of solution pairs is characterised easily:

- Definition: **dependence distance**
  - If, for all solution pairs ($\mathbf{I}^1$, I^2),

    $\mathbf{I}^1 = I^2 - k$

    then the dependence distance is k

- For example in the loop we considered earlier,

$$
\begin{array}{ll}
S_1: & \texttt{A[0] := 0} \\
 & \texttt{for I = 1 to 5} \\
S_2: & \quad \texttt{A[I] := A[I-1] + B[I]}
\end{array}
$$

  We find that $S_2 \ \delta_< \ S_2$ with dependence distance 1.

- *((of course there are many cases where the difference is not constant and so the dependence cannot be summarised this way)).*

Reuse distance

- When optimising for cache performance, it is sometimes useful to consider the re-use relationship,
  - $IN(S_1) \cap IN(S_2)$
- Here there is no dependence - it doesn't matter which read occurs first
- Nonetheless, cache performance can be improved by minimising the *reuse distance*
- The reuse distance is calculated essentially the same way
- Eg

```
        for I = 5 to 100
    S1:  B[I] := A[I] * 2
    S2:  C[I] := A[I-5] * 10
```

- Here we have a loop-carried reuse with distance 5

Nested loops

- Up to now we have looked at single loops
- Now let's generalise to loop "nests"
- We begin by considering a very common dependence pattern, called the "wavefront":

$$\texttt{for } I_1 \texttt{ = 0 to 3 do}$$
$$\quad \texttt{for } I_2 \texttt{ = 0 to 3 do}$$
$$S: \quad \texttt{A[}I_1\texttt{,}I_2\texttt{] := A[}I_1 - 1\texttt{,}I_2\texttt{] + A[}I_1\texttt{,}I_2 - 1\texttt{]}$$

- Dependence structure?

System of dependence equations

- **Consider the dependence equations for this loop nest:**

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S :    A[I1,I2] := A[I1 − 1,I2] + A[I1,I2 − 1]
```

$$\texttt{for } I_1 \texttt{ = 0 to 3 do}$$
$$\texttt{for } I_2 \texttt{ = 0 to 3 do}$$
$$S: \quad \texttt{A[}I_1\texttt{,}I_2\texttt{] := A[}I_1 - 1\texttt{,}I_2\texttt{] + A[}I_1\texttt{,}I_2 - 1\texttt{]}$$

- **There are two potential dependences arising from the three references to A, so two systems of dependence equations to solve:**

1. Between $\texttt{A[}I_1^1\texttt{,}I_2^1\texttt{]}$ and $\texttt{A[}I_1^2 - 1\texttt{,}I_2^2\texttt{]}$:

$$\begin{cases} I_1^1 = I_1^2 - 1 \\ I_2^1 = I_2^2 \end{cases}$$

2. Between $\texttt{A[}I_1^1\texttt{,}I_2^1\texttt{]}$ and $\texttt{A[}I_1^2\texttt{,}I_2^2 - 1\texttt{]}$:

$$\begin{cases} I_1^1 = I_1^2 \\ I_2^1 = I_2^2 - 1 \end{cases}$$

Iteration space graph

- The same loop:

$$\texttt{for } I_1 \texttt{ = 0 to 3 do}$$
$$\quad\texttt{for } I_2 \texttt{ = 0 to 3 do}$$
$$S: \quad \texttt{A}[I_1, I_2] \texttt{ := A}[I_1 - 1, I_2] \texttt{ + A}[I_1, I_2 - 1]$$

- For humans the easy way to understand this loop nest is to draw the *iteration space graph* showing the iteration-to-iteration dependences:

$$
\begin{array}{ccccccc}
S^{00} & \xrightarrow[\delta]{} & S^{01} & \xrightarrow[\delta]{} & S^{02} & \xrightarrow[\delta]{} & S^{03} \\
\delta\downarrow & & \delta\downarrow & & \delta\downarrow & & \delta\downarrow \\
S^{10} & \xrightarrow[\delta]{} & S^{11} & \xrightarrow[\delta]{} & S^{12} & \xrightarrow[\delta]{} & S^{13} \\
\delta\downarrow & & \delta\downarrow & & \delta\downarrow & & \delta\downarrow \\
S^{20} & \xrightarrow[\delta]{} & S^{21} & \xrightarrow[\delta]{} & S^{22} & \xrightarrow[\delta]{} & S^{23} \\
\delta\downarrow & & \delta\downarrow & & \delta\downarrow & & \delta\downarrow \\
S^{30} & \xrightarrow[\delta]{} & S^{31} & \xrightarrow[\delta]{} & S^{32} & \xrightarrow[\delta]{} & S^{33}
\end{array}
$$

- The diagram shows an arrow for each solution of each dependence equation. Is there any parallelism?

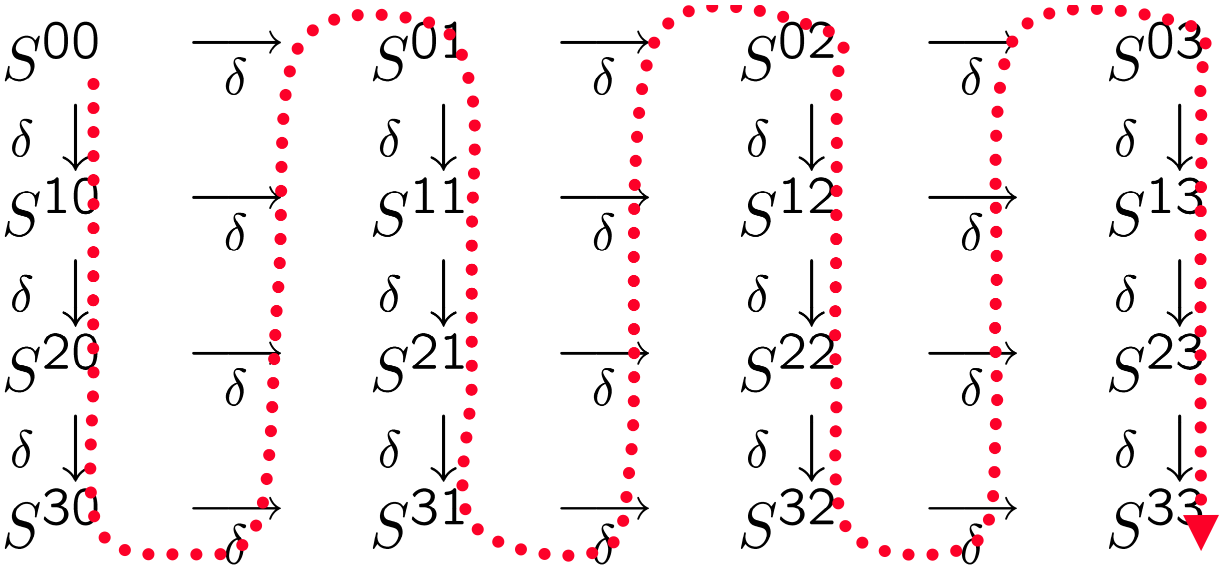
$$
\begin{array}{ccccccc}
S^{00} & \xrightarrow[\delta]{} & S^{01} & \xrightarrow[\delta]{} & S^{02} & \xrightarrow[\delta]{} & S^{03} \\
\delta \downarrow & & \delta \downarrow & & \delta \downarrow & & \delta \downarrow \\
S^{10} & \xrightarrow[\delta]{} & S^{11} & \xrightarrow[\delta]{} & S^{12} & \xrightarrow[\delta]{} & S^{13} \\
\delta \downarrow & & \delta \downarrow & & \delta \downarrow & & \delta \downarrow \\
S^{20} & \xrightarrow[\delta]{} & S^{21} & \xrightarrow[\delta]{} & S^{22} & \xrightarrow[\delta]{} & S^{23} \\
\delta \downarrow & & \delta \downarrow & & \delta \downarrow & & \delta \downarrow \\
S^{30} & \xrightarrow[\delta]{} & S^{31} & \xrightarrow[\delta]{} & S^{32} & \xrightarrow[\delta]{} & S^{33}
\end{array}
$$

for I_1 = 0 to 3 do

  for I_2 = 0 to 3 do

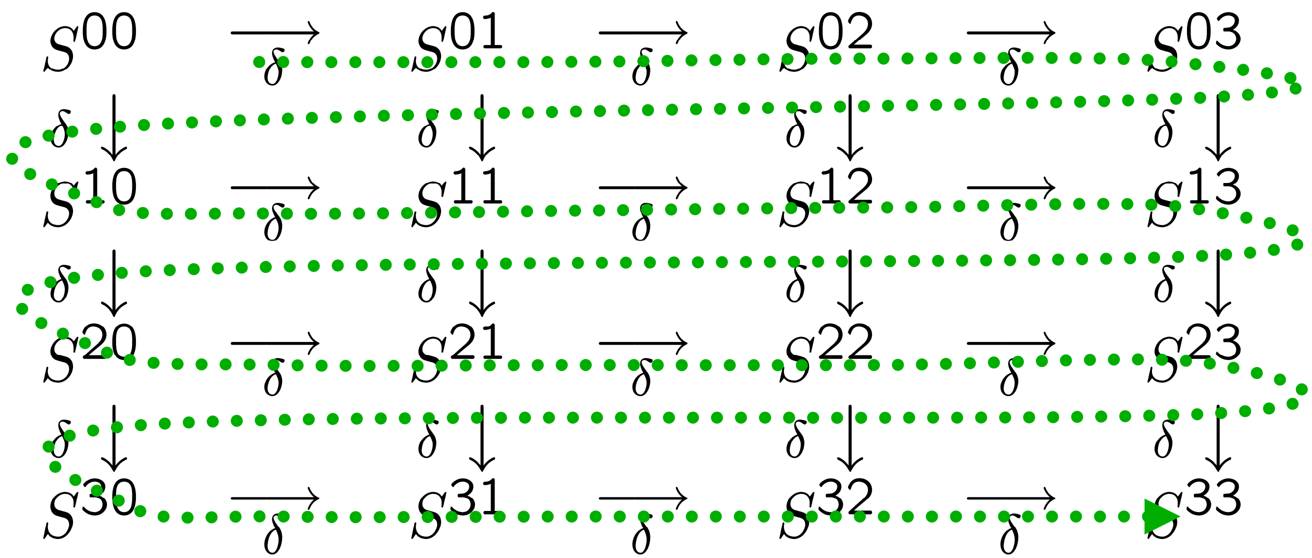
S: A[I_1,I_2] := A[$I_1 - 1$,I_2] + A[I_1,$I_2 - 1$]

- **The inner loop is not vectorisable since there is a dependence chain linking successive iterations.**
  - **(to use a vector instruction, need to be able to operate on each element of the vector in parallel)**
- **Similarly, the outer loop is not parallel**
- **This loop is *interchangeable*: the top-to-bottom, left-to-right execution order is also valid since all dependence constraints (as shown by the arrows) are still satisfied.**
- **Interchanging the loop does not improve vectorisability or parallelisability**




```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S :   A[I1,I2] := A[I1-1,I2] + A[I1,I2-1]
```

- The inner loop is not vectorisable since there is a dependence chain linking successive iterations.
  - (to use a vector instruction, need to be able to operate on each element of the vector in parallel)
- Similarly, the outer loop is not parallel
- This loop is *interchangeable*: the top-to-bottom, left-to-right execution order is also valid since all dependence constraints (as shown by the arrows) are still satisfied.
- Interchanging the loop does not improve vectorisability or parallelisability



for I_1 = 0 to 3 do

  for I_2 = 0 to 3 do

S: A[I_1,I_2] := A[$I_1 - 1$,I_2] + A[I_1,$I_2 - 1$]

- The inner loop is not vectorisable since there is a dependence chain linking successive iterations.
  - (to use a vector instruction, need to be able to operate on each element of the vector in parallel)
- Similarly, the outer loop is not parallel
- This loop is *interchangeable*: the top-to-bottom, left-to-right execution order is also valid since all dependence constraints (as shown by the arrows) are still satisfied.
- Interchanging the loop does not improve vectorisability or parallelisability

Interchange: counter-example

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S :    A[I1,I2] := A[I1+1,I2-1] + B[I1,I2]
```

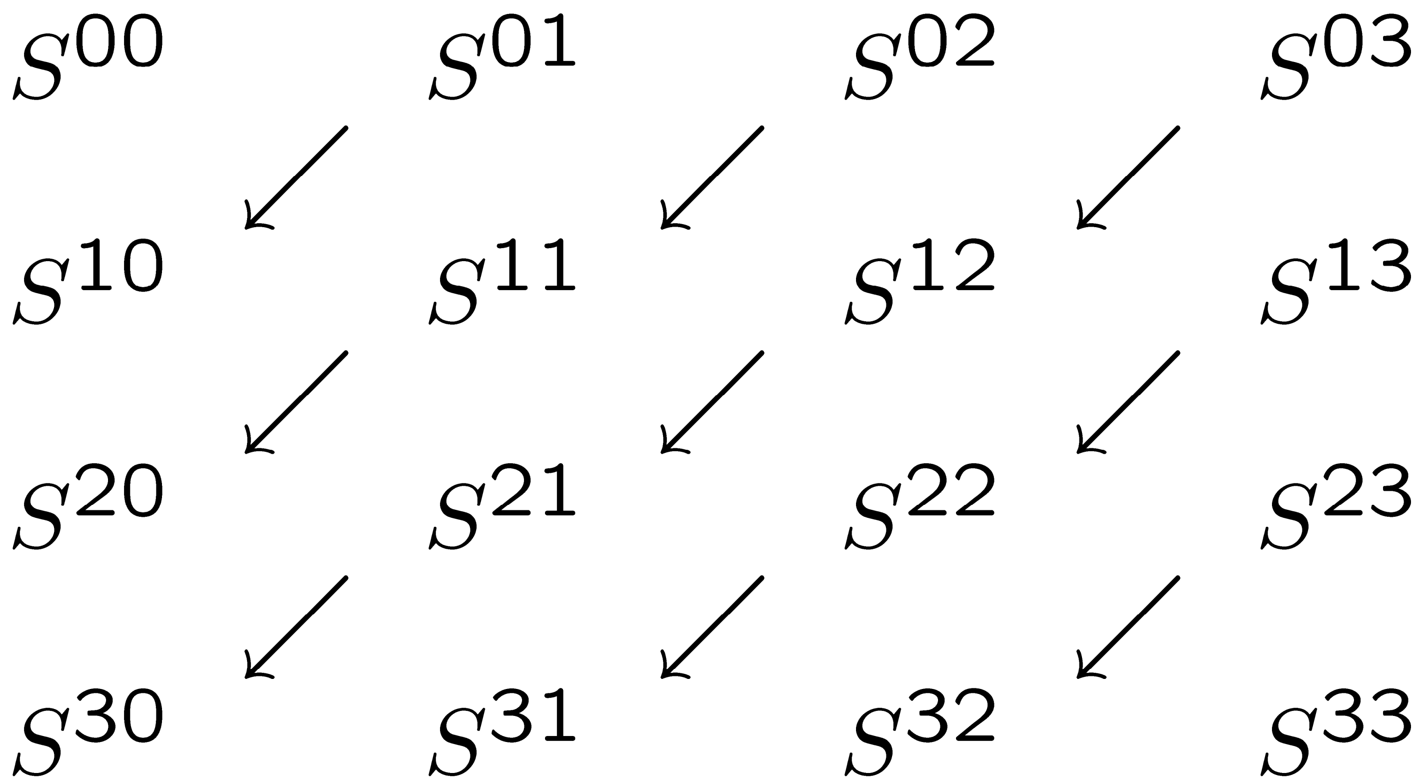
$$\texttt{for } I_1 = 0 \texttt{ to } 3 \texttt{ do}$$
$$\quad \texttt{for } I_2 = 0 \texttt{ to } 3 \texttt{ do}$$
$$S: \quad \texttt{A}[I_1, I_2] := \texttt{A}[I_1 + 1, I_2 - 1] + \texttt{B}[I_1, I_2]$$

Interchange: counter-example

for I_1 = 0 to 3 do

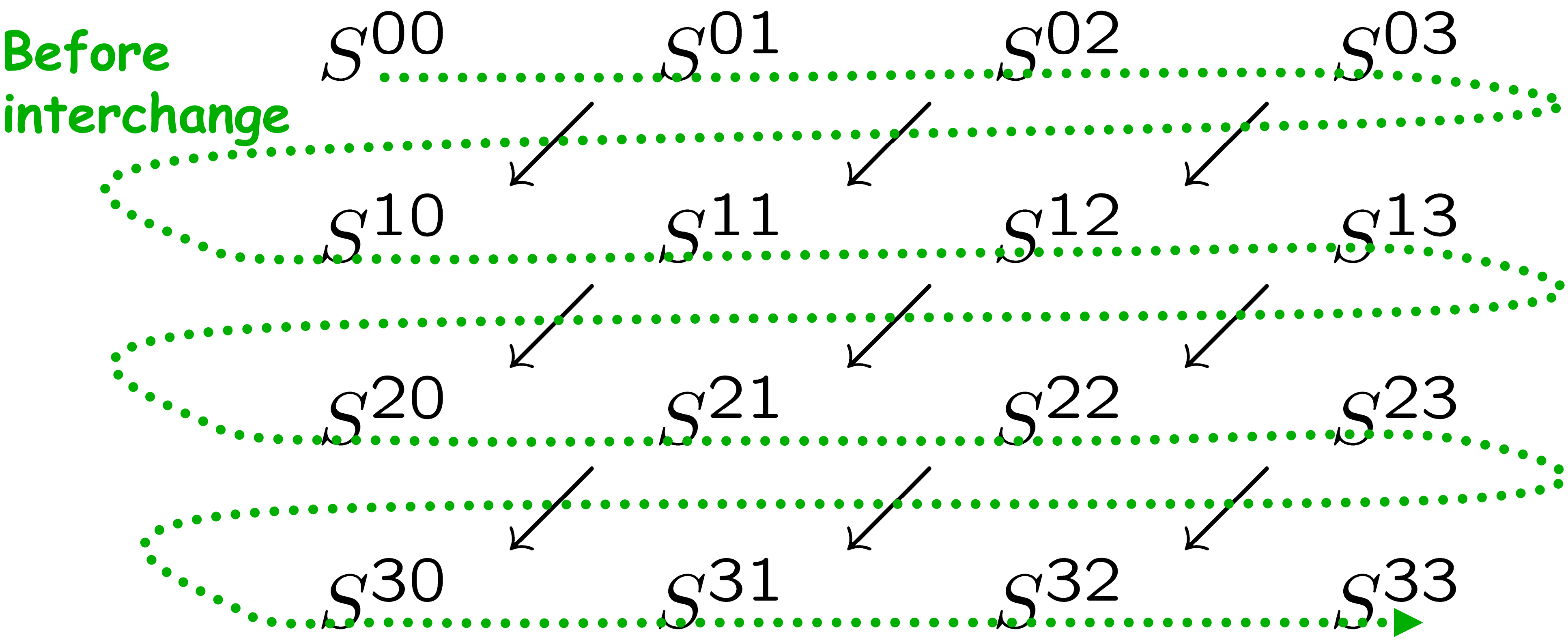
  for I_2 = 0 to 3 do

$S:$ $A[I_1, I_2] := A[I_1 + 1, I_2 - 1] + B[I_1, I_2]$



Interchange: counter-example

```
for I1 = 0 to 3 do
  for I2 = 0 to 3 do
S :   A[I1,I2] := A[I1+1,I2-1] + B[I1,I2]
```



Interchange: counter-example

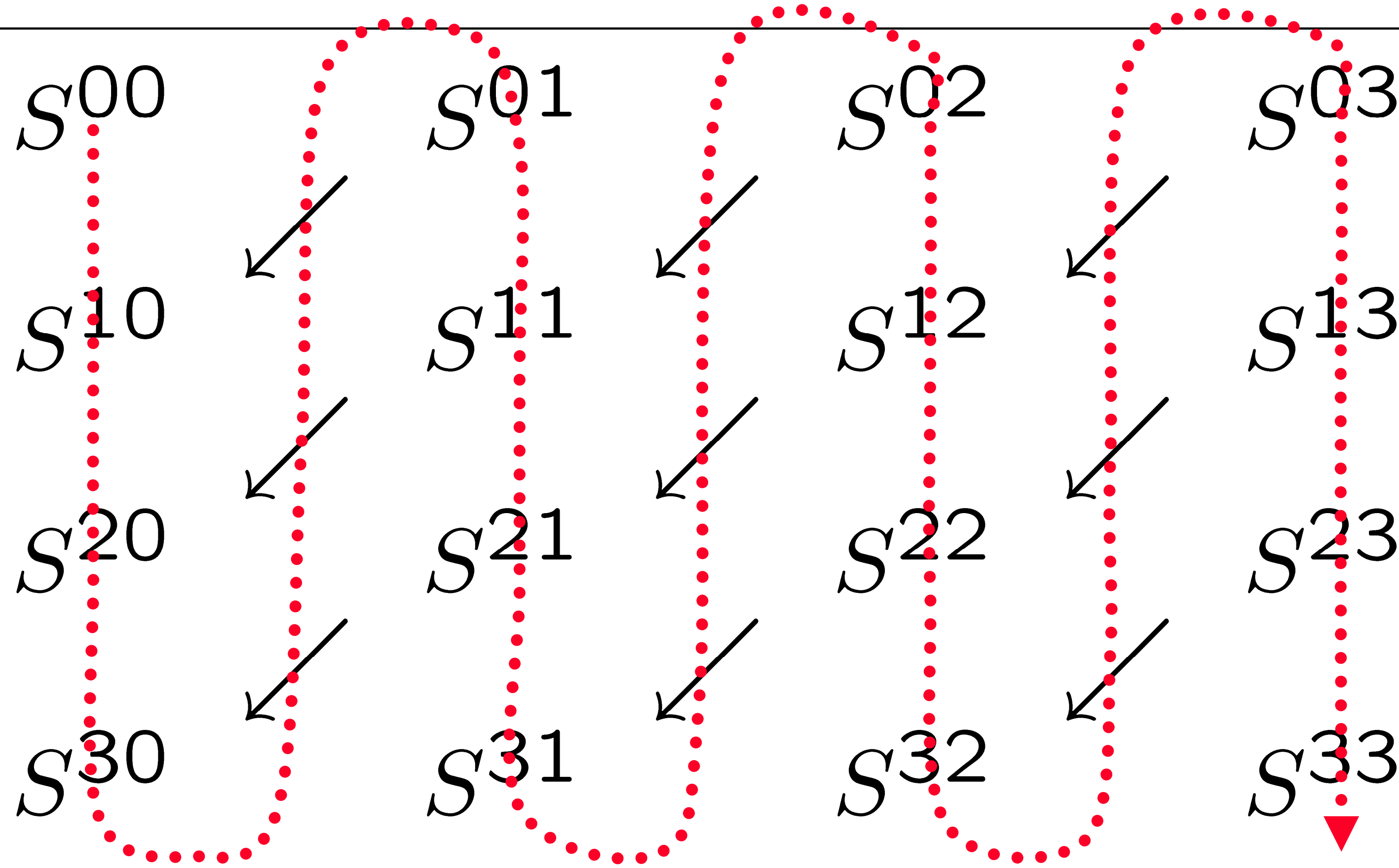
for I_1 = 0 to 3 do

  for I_2 = 0 to 3 do

S :   A[I_1,I_2] := A[I_1+1,I_2-1] + B[I_1,I_2]

After interchange:

New traversal order crosses dependence arrows backwards



Interchange: condition

- A loop is *interchangeable* if all dependence constraints (as shown by the arrows) are still satisfied by the top-to-bottom, left-to-right execution order
- How can you tell whether a loop can be interchanged?
- Look at it's dependence direction vectors:
  - Is there a dependence direction vector with the form (<,>) ?
  - ie there is a dependence distance vector (k_1,k_2) with $k_1>0$ and $k_2<0$?
  - If so, interchange would be ***invalid***
  - Because the arrows would be traversed backwards
  - All other dependence directions are OK.

Skewing

- Consider this variation on the wavefront loop:

```
for k1 := 0 to 3 do
  for k2 := k1 to k1+3 do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

- The inner loop's control variable runs from k_1 to k_1+3.
- The iteration space of this loop has 4^2 iterations just like the original loop.
- If we draw the iteration space with each iteration S^{K_1,K_2} at coordinate position (K_1,K_2), it is skewed to form a lozenge shape:

S^{00}	S^{01}	S^{02}	S^{03}			
	S^{11}	S^{12}	S^{13}	S^{14}		
		S^{22}	S^{23}	S^{24}	S^{25}	
			S^{33}	S^{34}	S^{35}	S^{36}

This loop performs the same computation as the original.

Skewing preserves semantics

- To see that this loop performs the same computation, lets work out its dependence structure.
- First label each iteration with the element of A to which it assigns:

S^{00} A_{00}	S^{01} A_{01}	S^{02} A_{02}	S^{03} A_{03}			
	S^{11} A_{10}	S^{12} A_{11}	S^{13} A_{12}	S^{14} A_{13}		
		S^{22} A_{20}	S^{23} A_{21}	S^{24} A_{22}	S^{25} A_{23}	
			S^{33} A_{30}	S^{34} A_{31}	S^{35} A_{32}	S^{36} A_{33}

- The loop body is

  $A[k_1,k_2-k_1] := A[k_1-1,k_2-k_1]+A[k_1,k_2-k_1-1]$

- E.g. iteration S_{23} does:

  $A[2,1] := A[1,1]+A[2,0]$

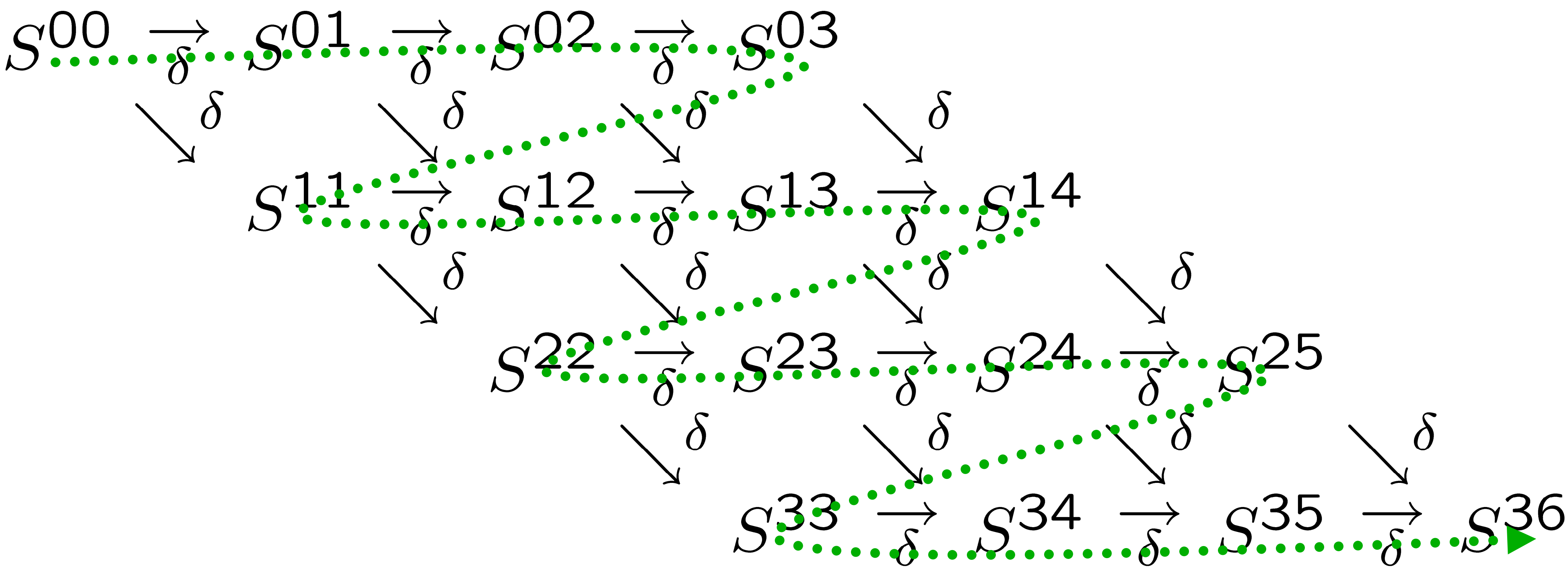
- **Thus the dependence structure of the skewed loop is shown by marking the iteration space with all the dependences:**

$$
\begin{array}{ccccccccccccc}
S^{00} & \underset{\delta}{\longrightarrow} & S^{01} & \underset{\delta}{\longrightarrow} & S^{02} & \underset{\delta}{\longrightarrow} & S^{03} & & & & & & \\
 & \searrow^{\delta} & & \searrow^{\delta} & & \searrow^{\delta} & & \searrow^{\delta} & & & & & \\
 & & S^{11} & \underset{\delta}{\longrightarrow} & S^{12} & \underset{\delta}{\longrightarrow} & S^{13} & \underset{\delta}{\longrightarrow} & S^{14} & & & & \\
 & & & \searrow^{\delta} & & \searrow^{\delta} & & \searrow^{\delta} & & \searrow^{\delta} & & & \\
 & & & & S^{22} & \underset{\delta}{\longrightarrow} & S^{23} & \underset{\delta}{\longrightarrow} & S^{24} & \underset{\delta}{\longrightarrow} & S^{25} & & \\
 & & & & & \searrow^{\delta} & & \searrow^{\delta} & & \searrow^{\delta} & & \searrow^{\delta} & \\
 & & & & & & S^{33} & \underset{\delta}{\longrightarrow} & S^{34} & \underset{\delta}{\longrightarrow} & S^{35} & \underset{\delta}{\longrightarrow} & S^{36}
\end{array}
$$

- **Can this loop nest be vectorised?**
- **Can this loop nest be interchanged?**

Skewing changes effect of interchange

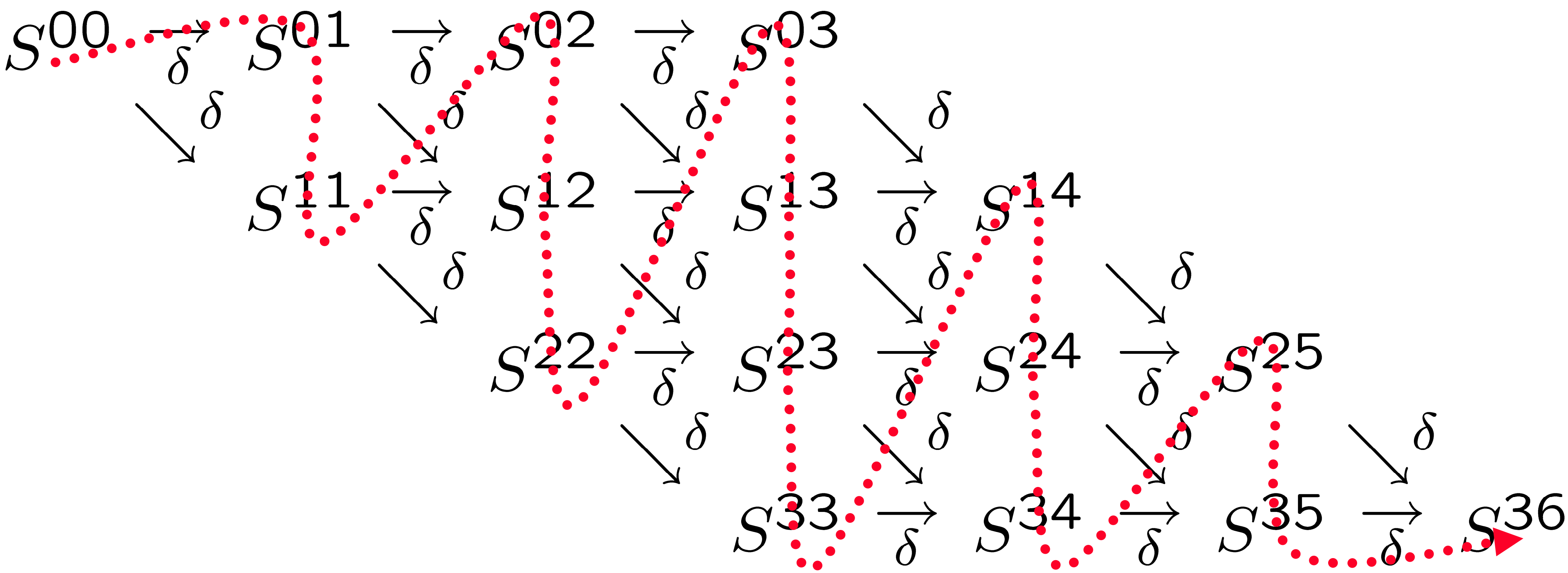
- Thus the dependence structure of the skewed loop is shown by marking the iteration space with all the dependences:



- Original execution order

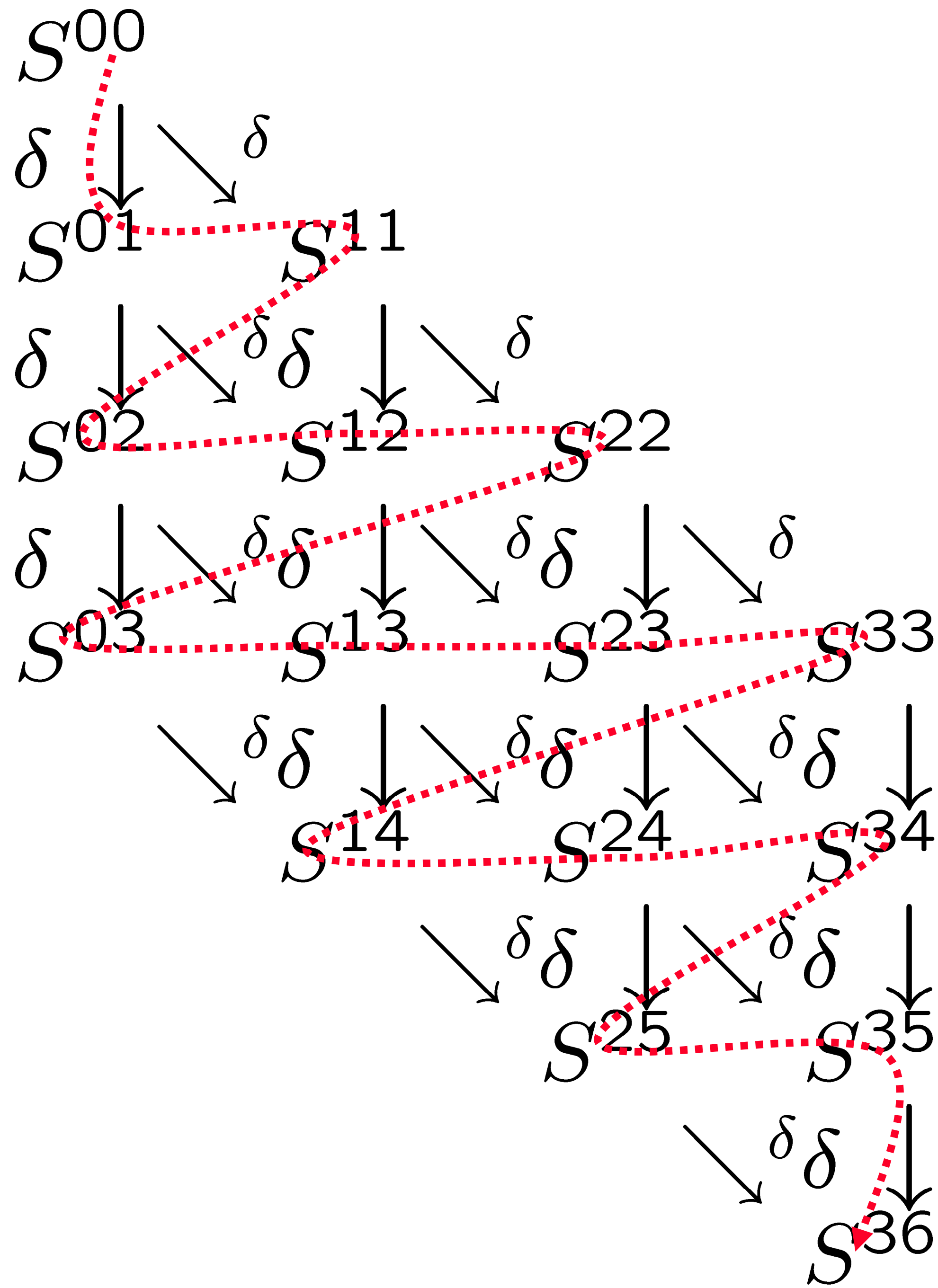
Interchange after skewing

- **Thus the dependence structure of the skewed loop is shown by marking the iteration space with all the dependences:**

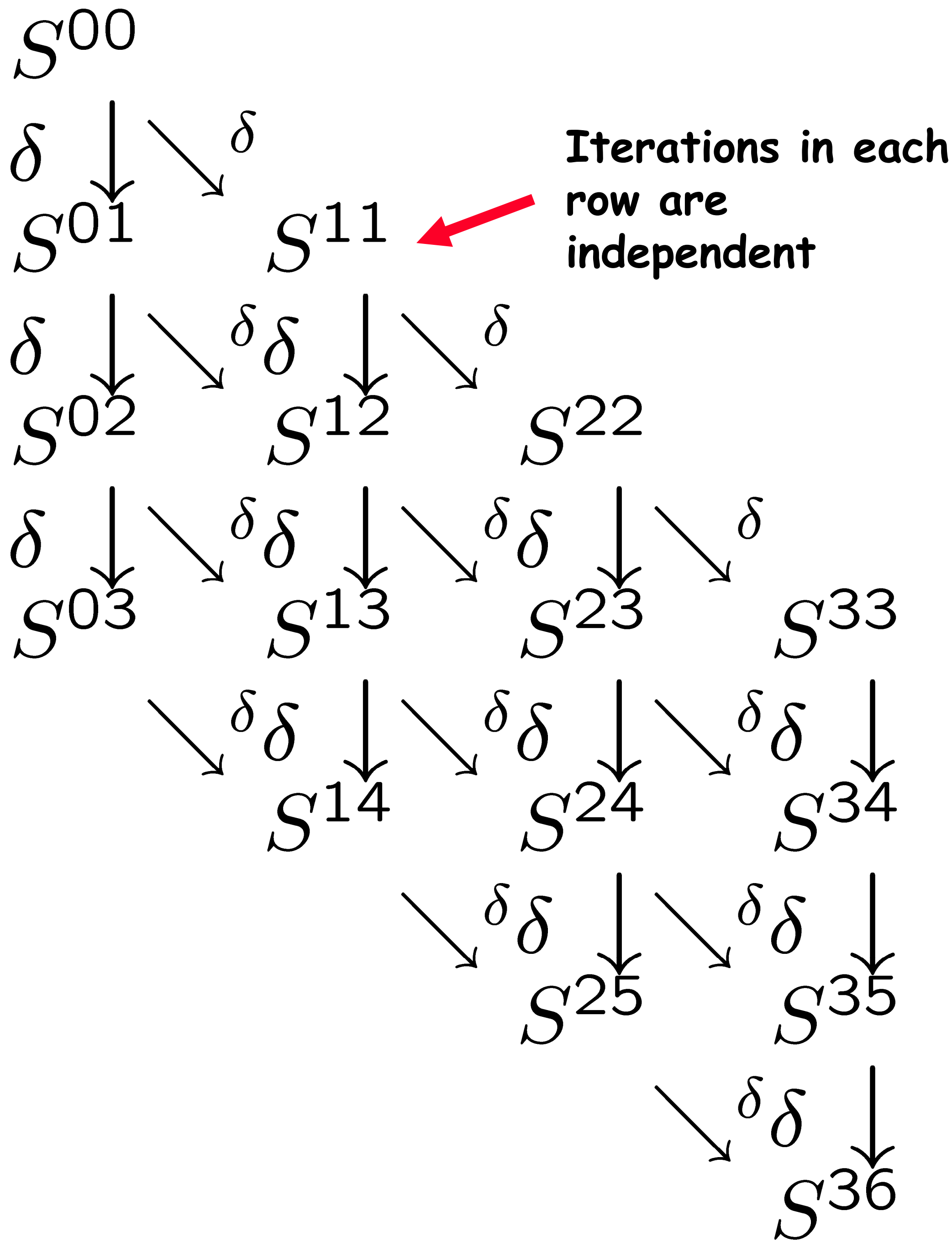


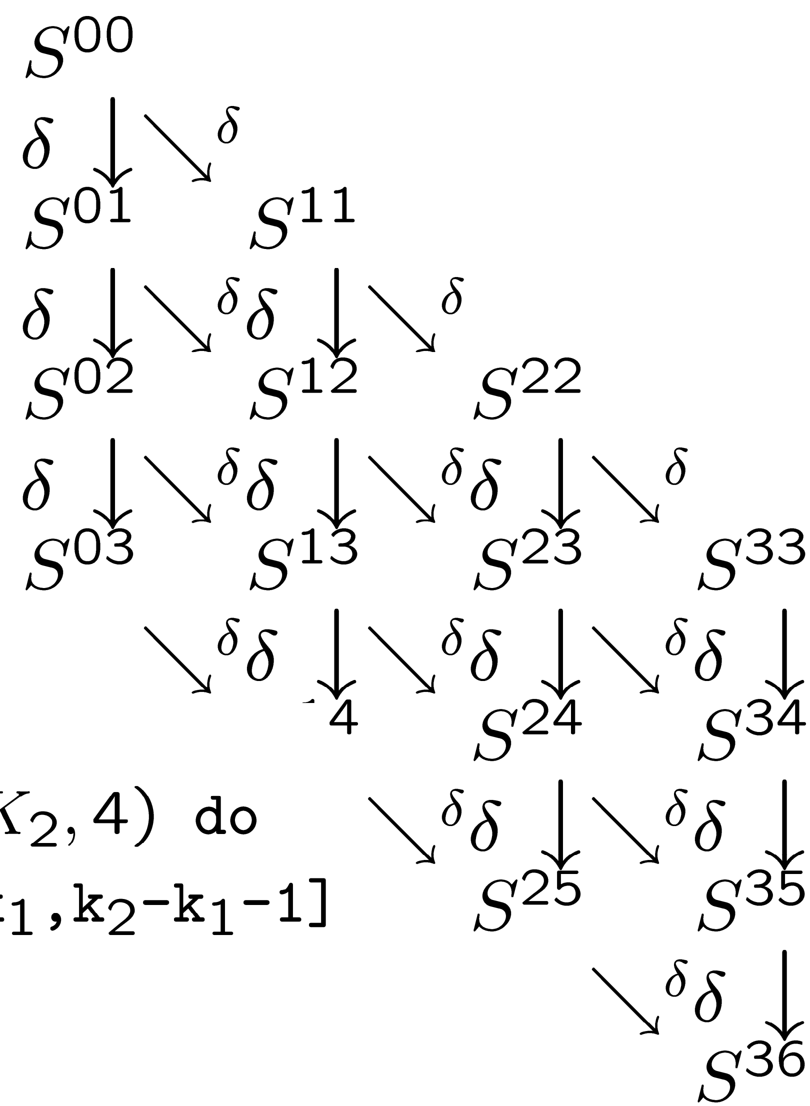
- **Transposed execution order**

- You can think of loop interchange as changing the way the iteration space is traversed
- Alternatively, you can think of it as a change to the way the runtime code instances are mapped onto the iteration space
- Traversal is always lexicographic – ie left-to-right, top-down



- The inner loop is now vectorisable, since it has no loop-carried dependence
- The skewed iteration space has N rows and 2N-1 columns, but still only N^2 actual statement instances.





- **The loop bounds are now a little complicated:**

$\texttt{for } k_2 \texttt{ := } 0 \texttt{ to } 8 \texttt{ do}$

$\quad \texttt{for } k_1 \texttt{ := } \max(0, K_2 - 3) \texttt{ to } \min(K_2, 4) \texttt{ do}$

$S : \texttt{A[}k_1\texttt{,}k_2\texttt{-}k_1\texttt{] := A[}k_1\texttt{-1,}k_2\texttt{-}k_1\texttt{]+A[}k_1\texttt{,}k_2\texttt{-}k_1\texttt{-1]}$

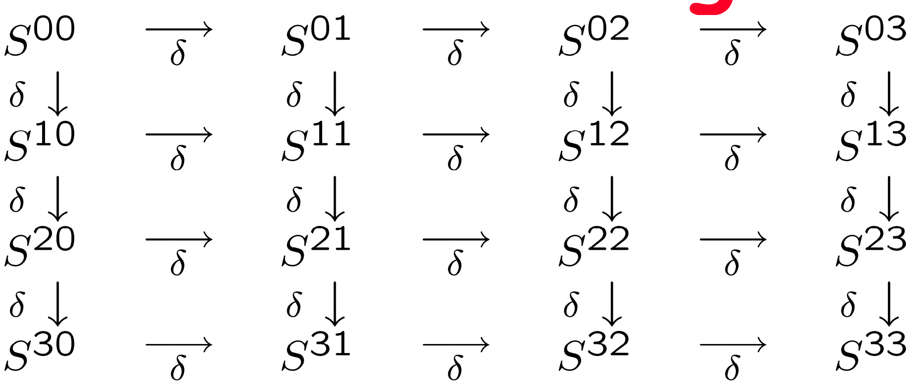
- **For loop bounds N_1 and N_2:**

$\texttt{for } k_2 \texttt{ := } 0 \texttt{ to } 2N_2 - 2 \texttt{ do}$

$\quad \texttt{for } k_1 \texttt{ := } \max(0, K_2 - N_2 + 2) \texttt{ to } \min(K_2, N_1) \texttt{ do}$

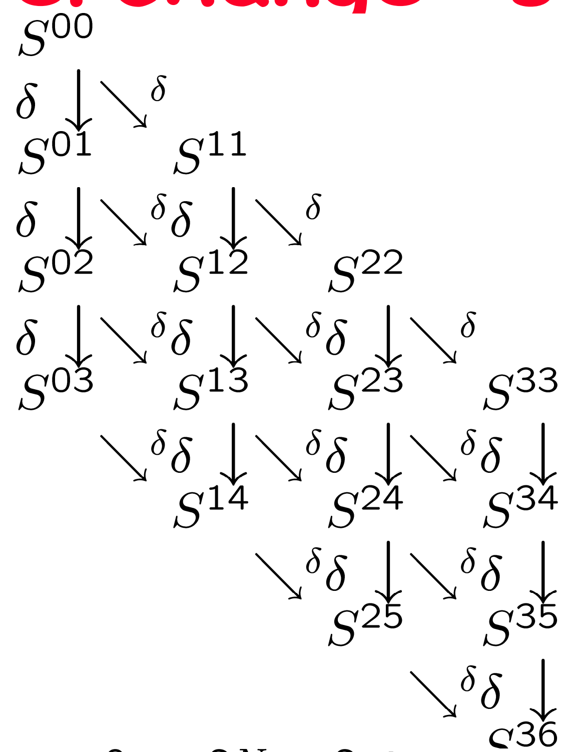
$S : \texttt{A[}k_1\texttt{,}k_2\texttt{-}k_1\texttt{] := A[}k_1\texttt{-1,}k_2\texttt{-}k_1\texttt{]+A[}k_1\texttt{,}k_2\texttt{-}k_1\texttt{-1]}$

Skewing and interchange: summary



for I_1 = 0 to 3 do
  for I_2 = 0 to 3 do
S : A[I_1,I_2] := A[$I_1 - 1$,I_2] + A[I_1,$I_2 - 1$]

- Original loop interchangeable but not vectorisable.
- We skewed inner loop by outer loop by factor 1.
- Still not vectorisable, but interchangeable.
- Interchanged, skewed loop *is* vectorisable.
- Bounds of new loop not simple!



for k_2 := 0 to $2N_2 - 2$ do
  for k_1 := $\max(0, K_2 - N_2 + 2)$ to $\min(K_2, N_1)$ do
S : A[k_1,k_2-k_1] := A[k_1-1,k_2-k_1]+A[k_1,k_2-k_1-1]

- Is skewing ever invalid?
- Does skewing affect interchangeability?
- Does skewing affect dependence distances?
- Can you predict value of skewing?

Summary: dependence

- **Dependence equation for single loop:**
  - Suppose S_p refers to $A[\phi_p(I)]$
  - Suppose S_q refers to $A[\phi_q(I)]$
  - A dependence of some kind occurs between S_p and S_q if there exists a solution to the equation

    $$\phi_p(I^1) = \phi_q(I^2)$$

  - for integer values of I^1 and I^2 lying within the loop bounds
- **For doubly-nested loops** over multidimensional arrays, generalise to system of simultaneous dependence equations for two iterations, (I_1^1, I_2^1) and (I_1^2, I_2^2)
- **Iteration space graph**, lexicographic schedule of execution
- Arrows in graph show solutions to dependence equation
- **Dependence distance vectors** characterise families of congruent arrows

Summary: transformations

- **A loop can be executed in parallel** if it has no loop-carried dependence
- **A loop nest can be interchanged** if the transposed dependence distance vectors are lexicographically forward
- **Strip-mining** is always valid
- **Tiling** = strip-mining + interchange
- Skewing is always valid
- **Skewing can expose parallelism** by aligning parallel iterations with one of the loops
- Skewing can make interchange (and therefore tiling) valid

Matrix representation of loop transformations

- To skew the inner loop by the outer loop by factor 1 we adjust the loop bounds, and replace I_1 by K_1, and I_2 by K_2-K_1. That is,

  $$(K_1,K_2) = (I_1,I_2) \,.\, U$$

- where U is a 2 x 2 matrix

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

- That is,

  $$(K_1,K_2) = (I_1,I_2) \,.\, U = (I_1,I_2+I_1)$$

- The inverse gets us back again:

  $$(I_1,I_2) = (K_1,K_2) \,.\, U^{-1} = (K_1,K_2-K_1)$$

- Matrix U maps each statement instance $S^{I_1 I_2}$ to its position in the new iteration space, $S^{K_1 K_2}$:
- Original iteration space:

I_1 \ I_2:	0	1	2	3
0	S^{00}	S^{01}	S^{02}	S^{03}
1	S^{10}	S^{11}	S^{12}	S^{13}
2	S^{20}	S^{21}	S^{22}	S^{23}
3	S^{30}	S^{31}	S^{32}	S^{33}

- Transformed iteration space:

K_1 \ K_2:	0	1	2	3	4	5	6
0	S^{00}	S^{01}	S^{02}	S^{03}			
1		S^{11}	S^{12}	S^{13}	S^{14}		
2			S^{22}	S^{23}	S^{24}	S^{25}	
3				S^{33}	S^{34}	S^{35}	S^{36}

The dependences are subject to the same transformation.

Using matrices to reason about dependence

Recall that:

- There is a dependence between two iterations (I_1^1, I_2^1) and (I_1^2, I_2^2) if there is a memory location which is assigned to in iteration (I_1^1, I_2^1), and read in iteration (I_1^2, I_2^2).

  ((unless there is an intervening assignment))

- If (I_1^1, I_2^1) precedes (I_1^2, I_2^2) it is a *data*-dependence.
- If (I_1^2, I_2^2) precedes (I_1^1, I_2^1) it is a *anti*-dependence.
- If the location is assigned to in both iterations, it is an *output*-dependence.

- The dependence distance vector (D_1, D_2) is $(I_1^2 - I_1^1, I_2^2 - I_2^1)$.

Transforming dependence vectors

- Iterations $(I_1^1, I_2^1) \cdot U$ and $(I_1^2, I_2^2) \cdot U$ will also read and write the same location.
- The transformation U is ***valid*** iff

  $(I_1^1, I_2^1) \cdot U$ precedes $(I_1^2, I_2^2) \cdot U$

  whenever there is a dependence between

  (I_1^1, I_2^1) and (I_1^2, I_2^2).
- In the transformed loop the dependence distance vector is also transformed, to

  $(D_1, D_2) \cdot U$

Definition: Lexicographic ordering:

(I_1^1, I_2^1) precedes (I_1^2, I_2^2)

If $I_1^1 < I_1^2$, or $I_1^1 = I_1^2$ and $I_2^1 < I_2^2$

("Lexicographic" is dictionary order – both "baz" and "can" precede "cat")

Example: loop given earlier

Before transformation we had two dependences:

1. Distance: (1,0), direction: (<,.)
2. Distance: (0,1), direction: (.,<)

- After transformation by matrix

$$U = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

- (i.e. skewing of inner loop by outer) we get:

1. Distance: (1,1), direction: (<,<)
2. Distance: (0,1), direction: (.,<)

- We can also represent loop interchange by a matrix transformation.
- After transforming the skewed loop by matrix

$$V = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- (i.e. loop interchange) we get:

1. Distance: (1,1), direction: (<,<)
2. Distance: (1,0), direction: (<,.)

- The transformed iteration space is the transpose of the skewed iteration space:

$$\begin{array}{llll} S^{00} & & & \\ S^{10} & S^{11} & & \\ S^{20} & S^{21} & S^{22} & \\ S^{30} & S^{31} & S^{32} & S^{33} \\ & S^{41} & S^{42} & S^{43} \\ & & S^{52} & S^{53} \\ & & & S^{63} \end{array}$$

Summary

- (I_1,I_2) . U maps each statement instance (I_1,I_2) to its new position (K_1,K_2) in the transformed loop's execution sequence
- (D_1,D_2) . U gives new dependence distance vector, giving test for validity
- Captures skewing, interchange and reversal
- Compose transformations by matrix multiplication

  $U_1 \; . \; U_2$
- Resulting loop's bounds may be a little tricky
  - Efficient algorithms exist [Banerjee90] to maximise parallelism by skewing and loop interchanging
  - Efficient algorithms exist to optimise cache performance by finding the combination of blocking, block size, interchange and skewing which leads to the best reuse [Wolf91]

References

- Hennessy and Patterson: Section 4.4 (pp.319)
- Background: "conventional" compiler techniques
  - A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques and Tools. Addison Wesley, 1986.
  - Andrew Appel and Jens Palsberg, Modern Compiler Implementation. Cambridge University Press, 2002.
  - Cooper and Torczon, Engineering a Compiler. Morgan Kaufmann 2004.
  - Morgan, Building an Optimizing Compiler
- Textbooks covering restructuring compilers
  - Michael Wolfe. High Performance Compilers for Parallel Computing. Addison Wesley, 1996.
  - Steven Muchnick, Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.
  - Ken Kennedy and Randy Allen, Optimizing Compilers for Modern Architectures. Morgan Kaufmann, 2001.

READ THIS ONE

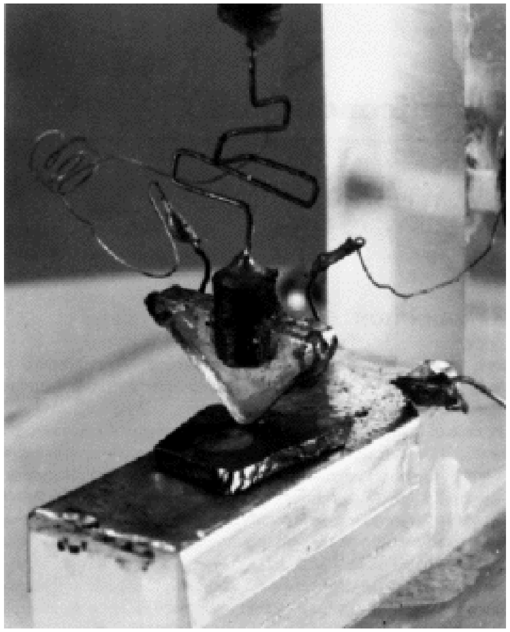
- Research papers:
  - D. F. Bacon and S. L. Graham and O. J. Sharp, "Compiler Transformations for High-Performance Computing". ACM Computing Surveys V26 N4 Dec 1994 http://doi.acm.org/10.1145/197405.197406
  - U. Banerjee. Unimodular transformations of double loops. In Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, CA. Pitman/MIT Press, 1990.
  - M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, volume 26, pages 30-44, Toronto, Ontario, Canada, June 1991.

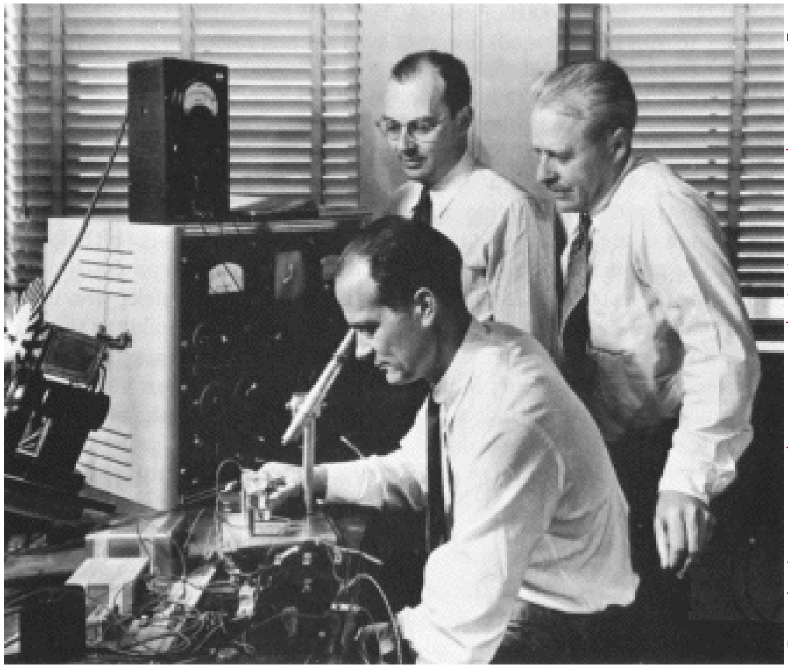
Additional material for background

A little history…early days at Bell Labs

- 1940: Russell Ohl develops PN junction (accidentally…)
- 1945: Shockley's lab established
- 1947: Bardeen and Brattain create point-contact transistor with two PN junctions, gain=18
- 1951: Shockley develops junction transistor which can be manufactured in quantity
- 1952: British radar expert GWA Dummer forecasts "solid block [with] layers of insulating, conducting and amplifying materials
- 1954: first transistor radio. Also Texas Instruments makes first silicon transistor (price $2.50)

First point-contact transistor invented at Bell Labs.
(Source: Bell Labs.)





The three inventors of the transistor: William Shockley, (seated), John Bardeen (left) and Walter Brattain (right) in 1948; the three inventors shared the Nobel prize in 1956.

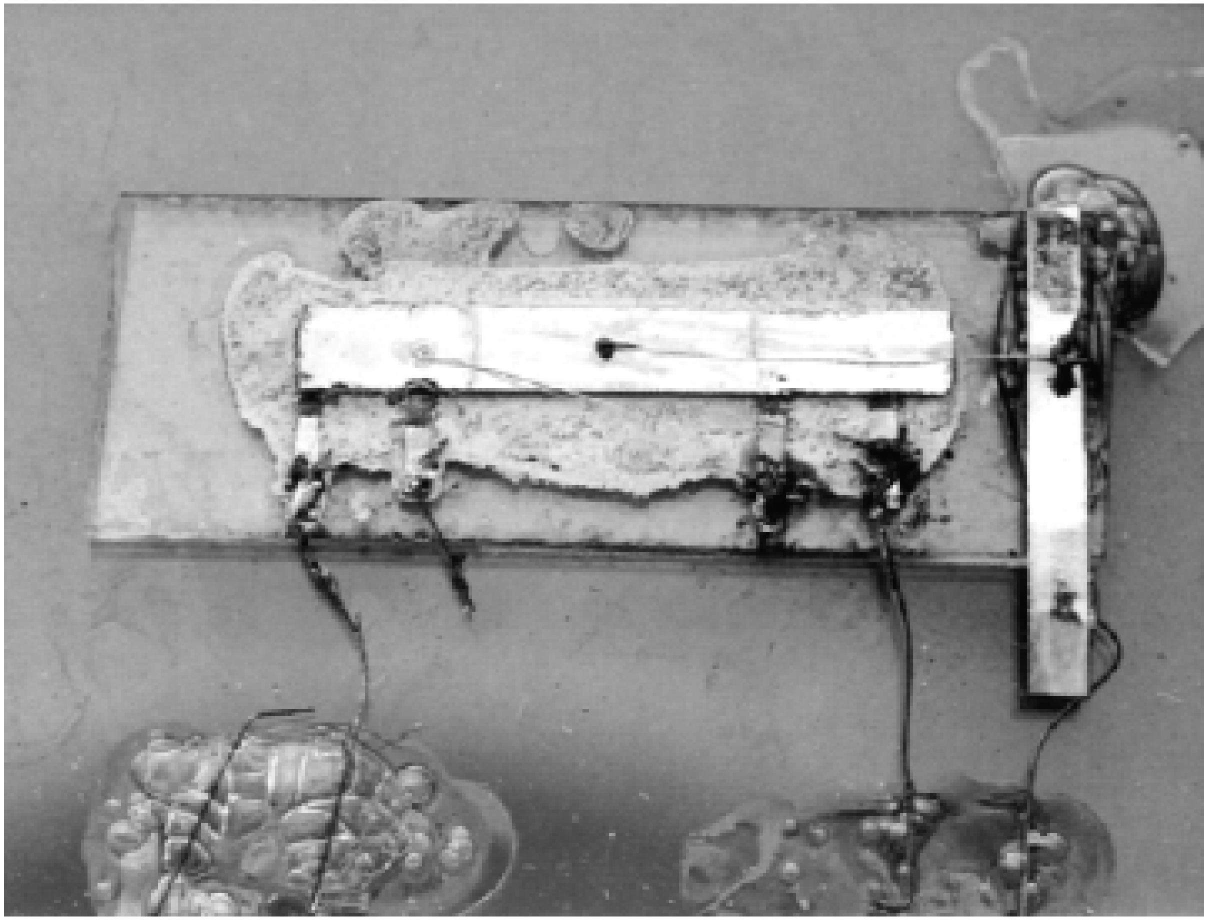
(Source: Bell Labs.)

This background section is not covered in the lectures

Source: http://6371.lcs.mit.edu/Fall96/lectures/L1/P005.html ; See also http://www.maxmon.com/1952ad.htm

Pre-historic integrated circuits

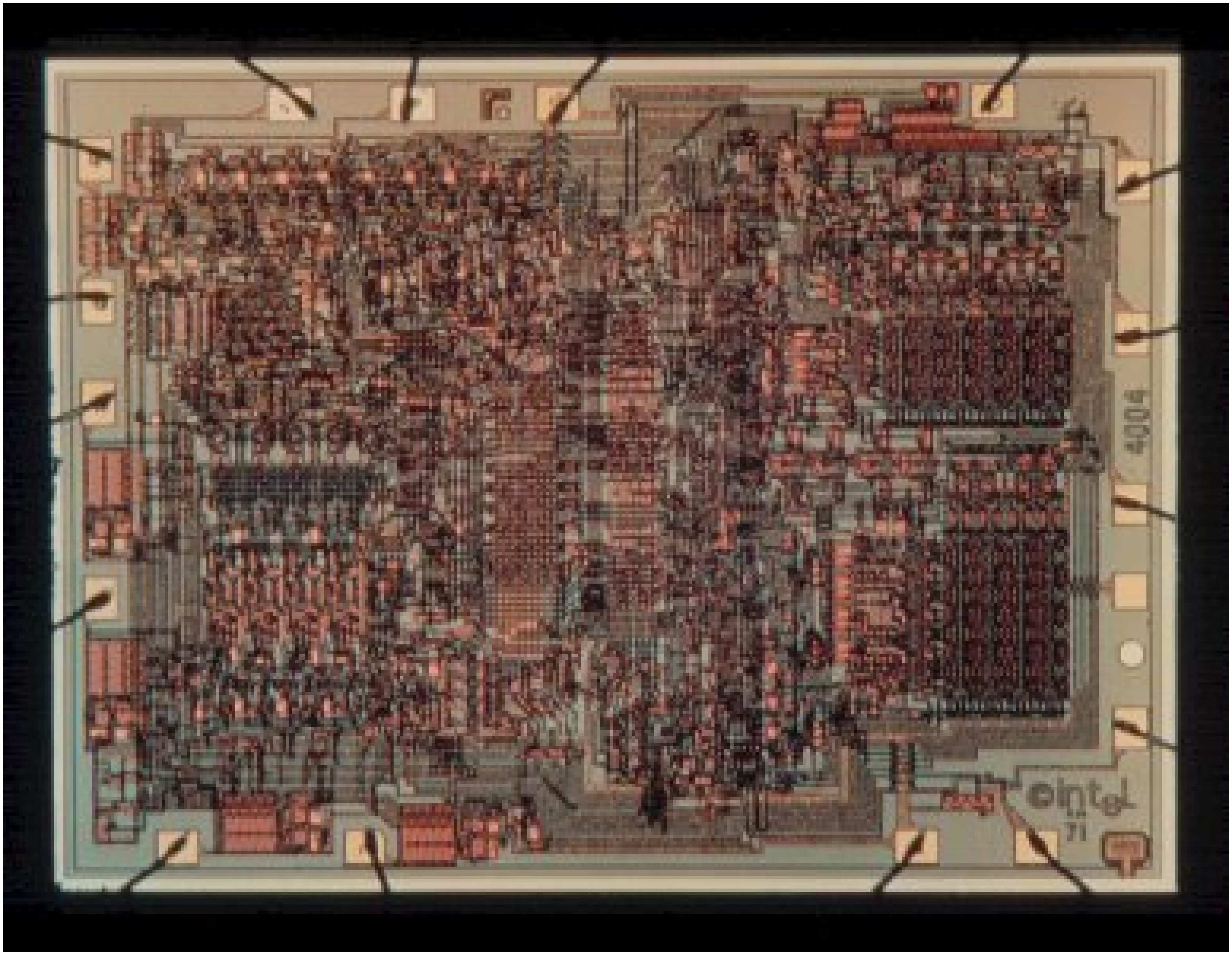
- 1958: The first monolithic integrated circuit, about the size of a finger tip, developed at Texas Instruments by Jack Kilby. The IC was a chip of a single Germanium crystal containing one transistor, one capacitor, and one resistor (Source: Texas Instruments)



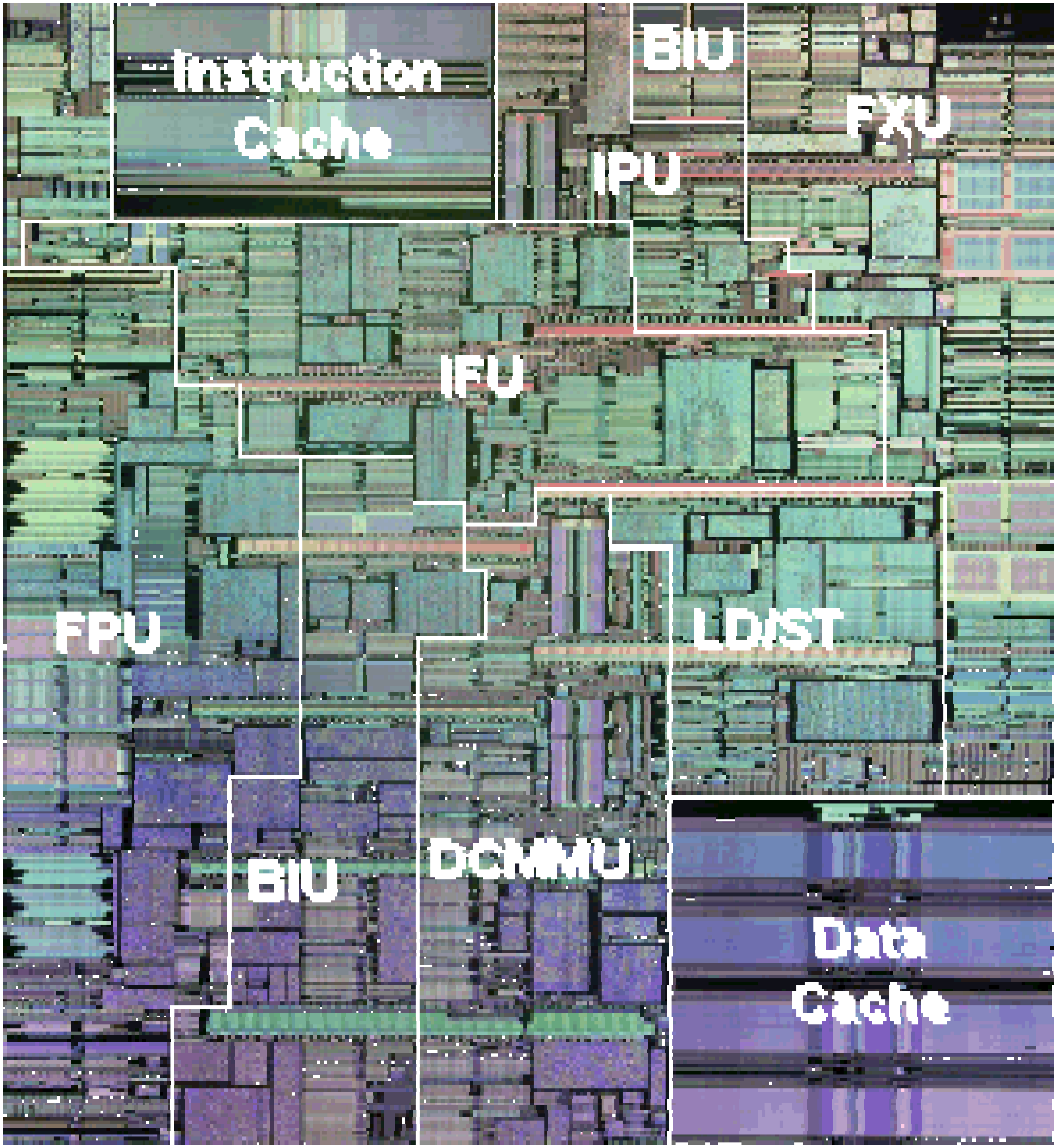
Source: http://kasap3.usask.ca/server/kasap/photo1.html

- 1970: Intel starts selling a 1K bit RAM

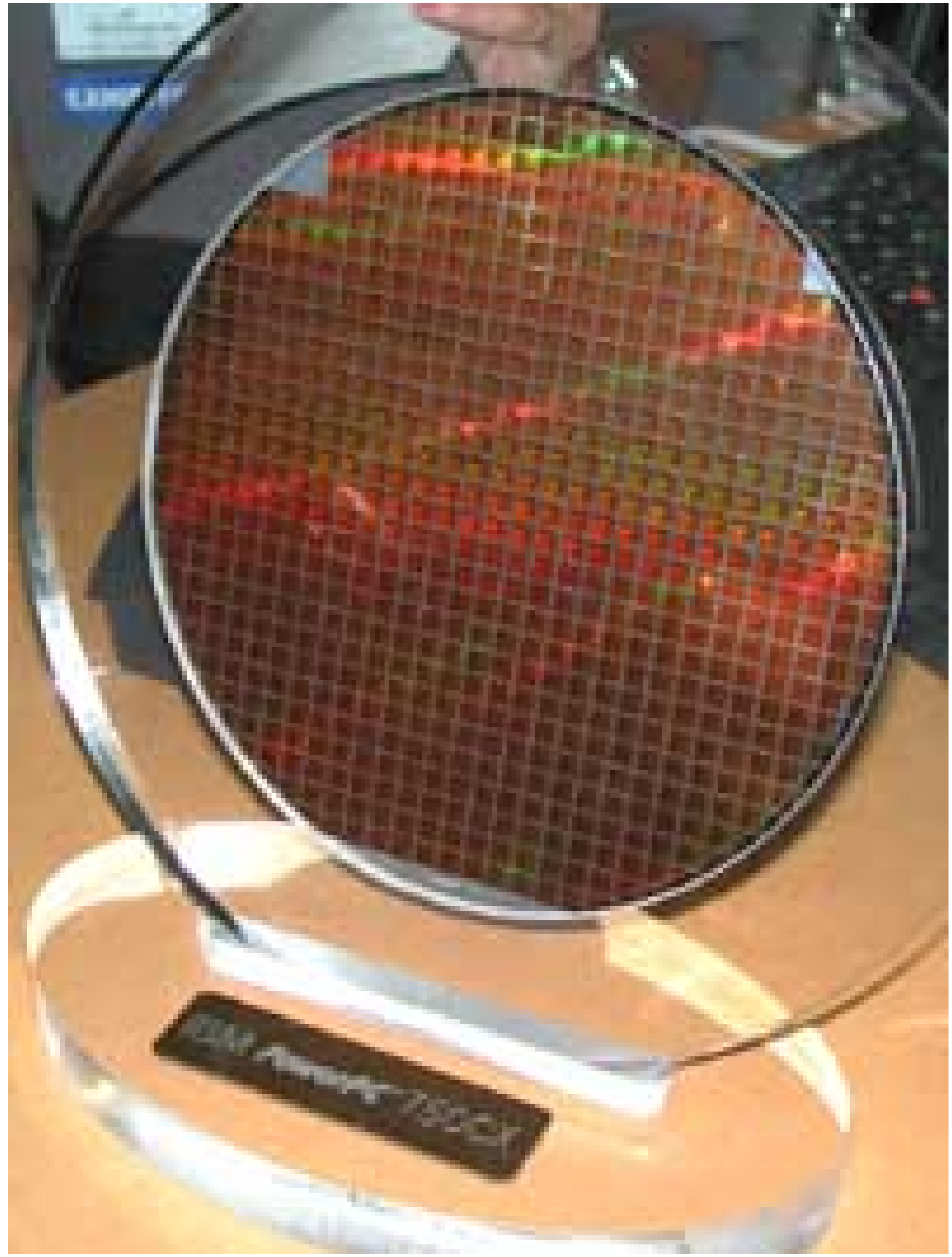
- 1971: Intel introduces first microprocessor, the 4004
  - 4-bit buses
  - Clock rate 108 KHz
  - 2300 transistors
  - 10μm process

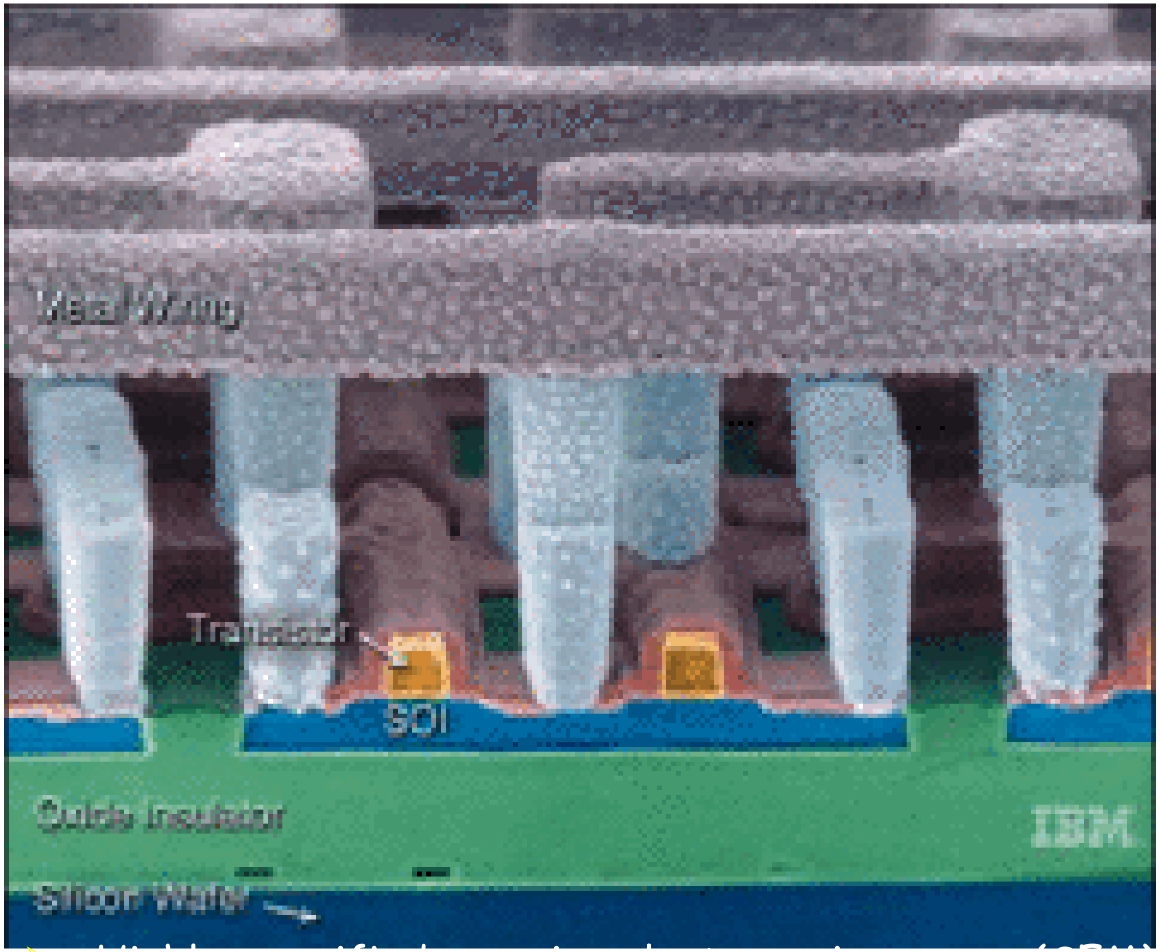


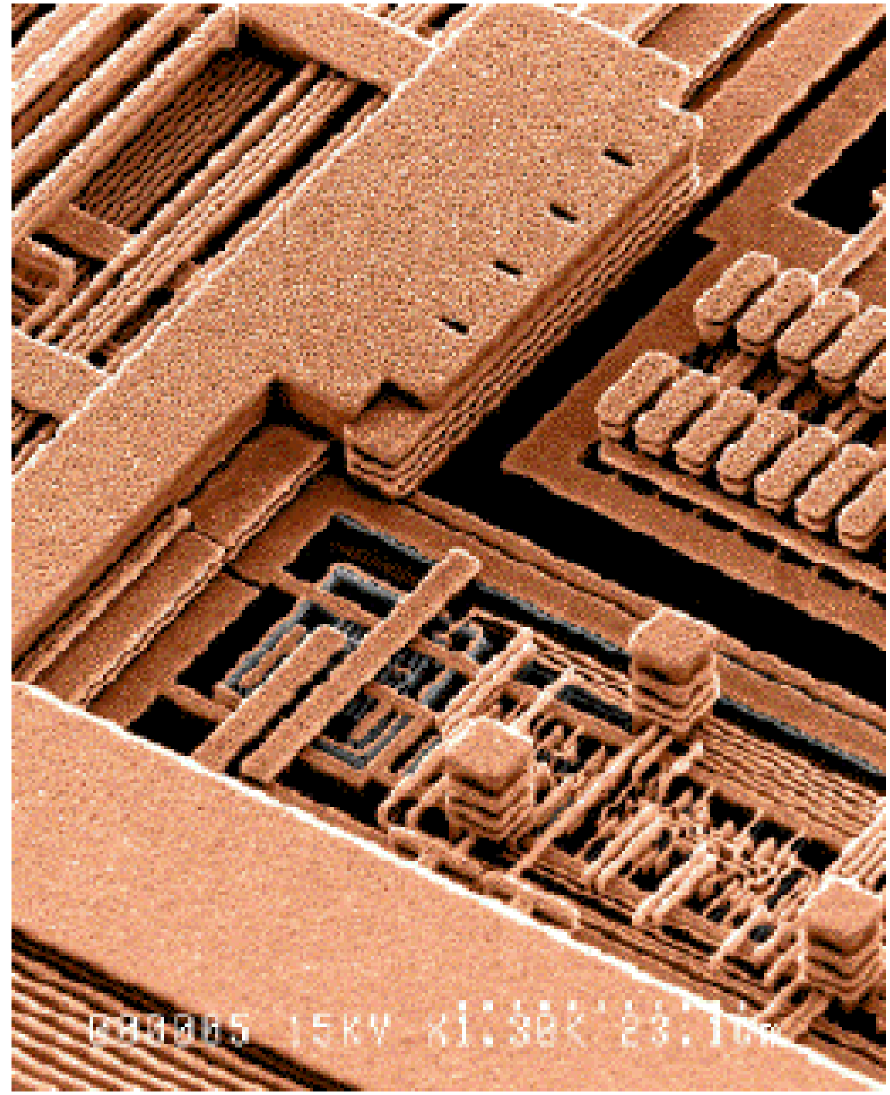
- IBM Power3 microprocessor
- 15M transistors
- 0.18μm copper/SOI process
- About 270mm²



- Chips are made from slices of a single-crystal silicon ingot
- Each slice is about 30cm in diameter, and 250-600 microns thick
- Transistors and wiring are constructed by photolithography
- Essentially a printing/etching process
- With lines ca. 0.18μm wide







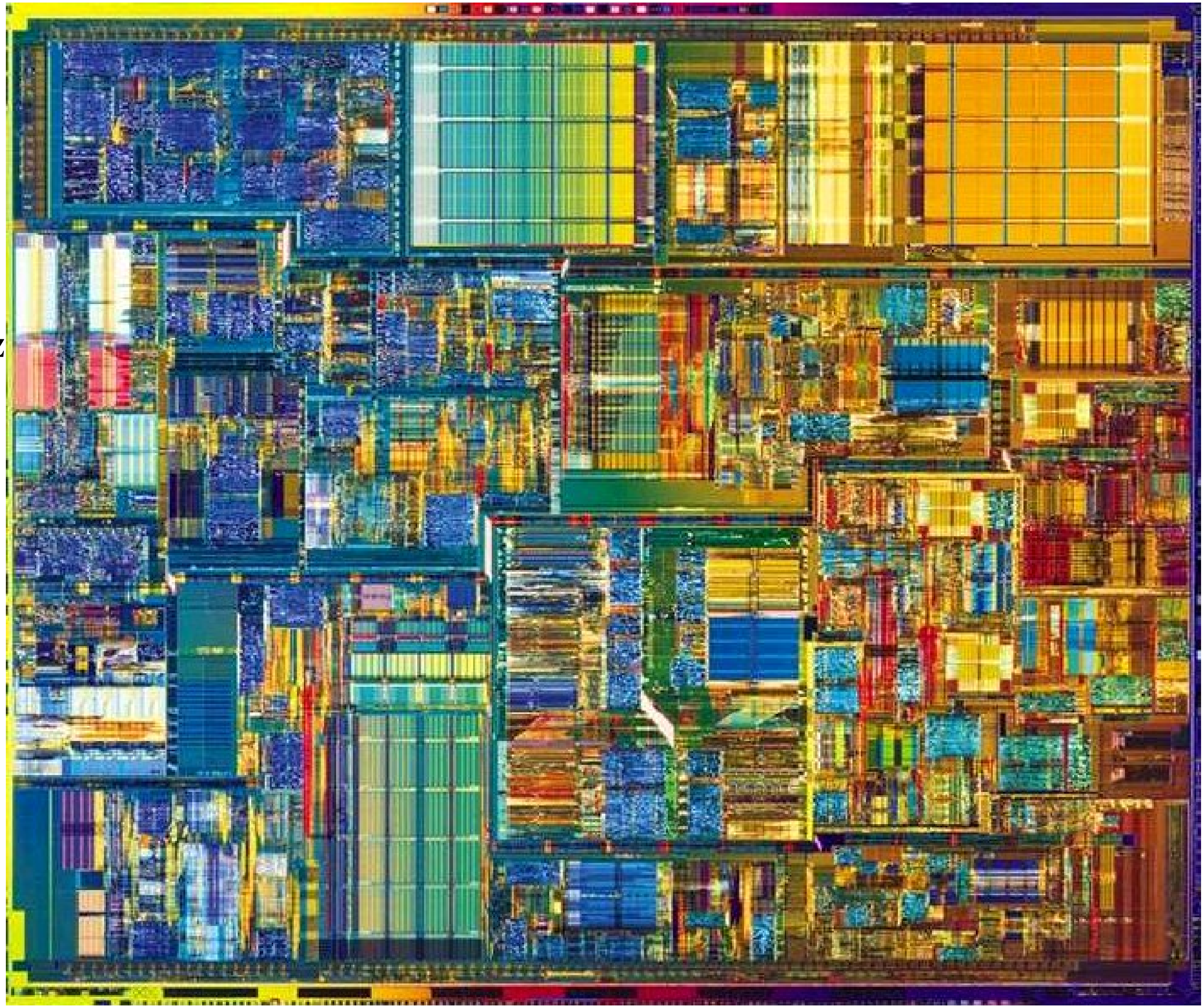
- Highly magnified scanning electron microscope (SEM) view of IBM's six-level copper interconnect technology in an integrated circuit chip. The aluminum in transistor interconnections in a silicon chip has been replaced by copper that has a higher conductivity (by nearly 40%) and also a better ability to carry higher current densities without electromigration. Lower copper interconnect resistance means higher speeds and lower RC constants (Photograph courtesy of IBM Corporation, 1997.)

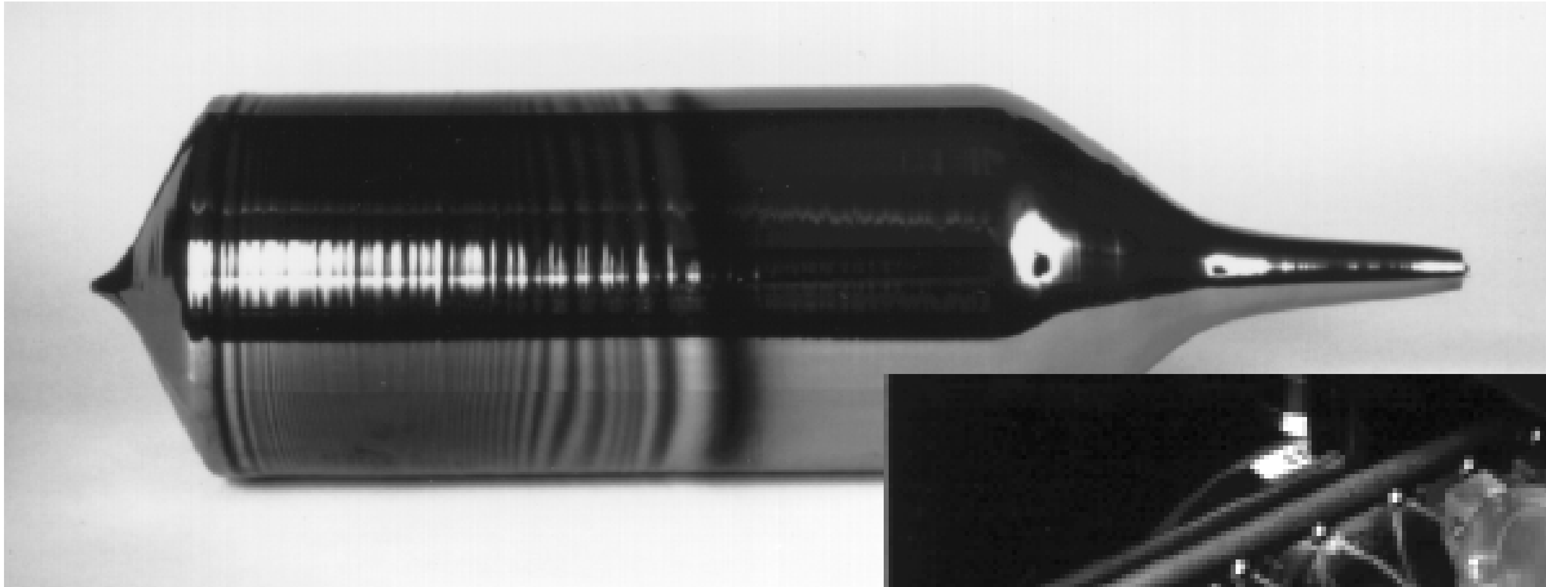
Intel Pentium 4

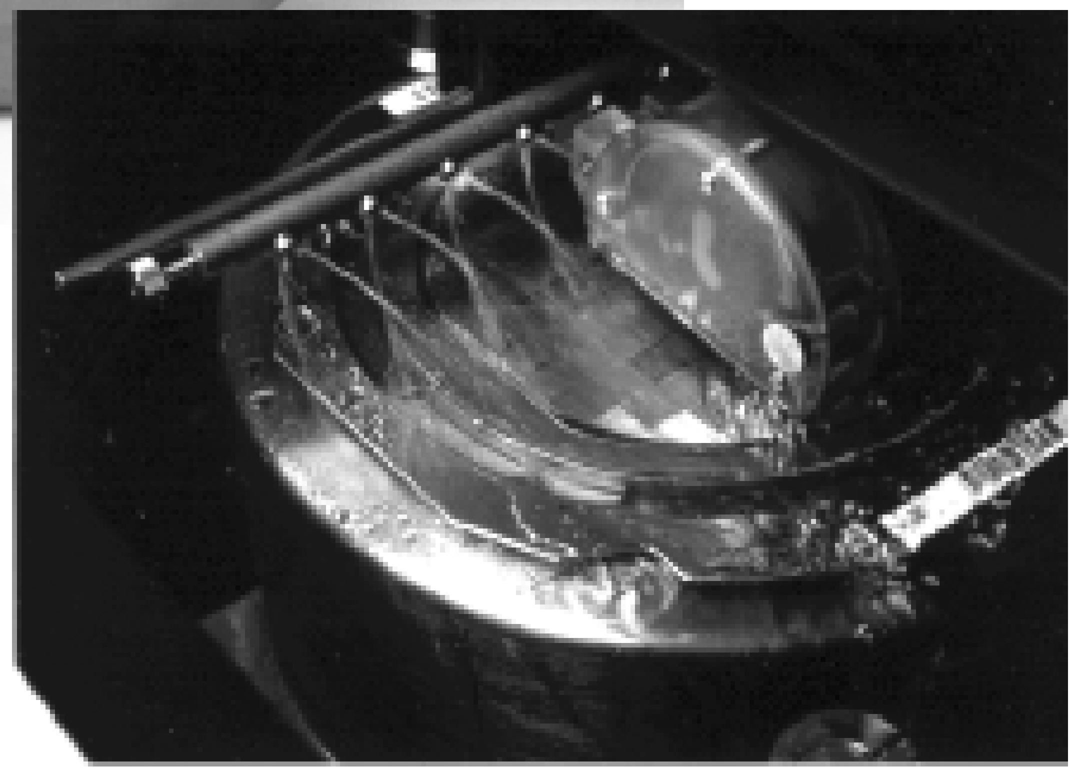
- **42 M transistors**
- **0.13mm copper/SOI process**
- **Clock speeds: 2200, 2000MHz**
- **Die size 146 square mm**
- **Power consumption 55.1W (2200), 52.4W (2000)**
- **Price ($ per chip, in 1,000-chip units, Jan 2002):**

  US$562 (2200)

  US$364 (2000)



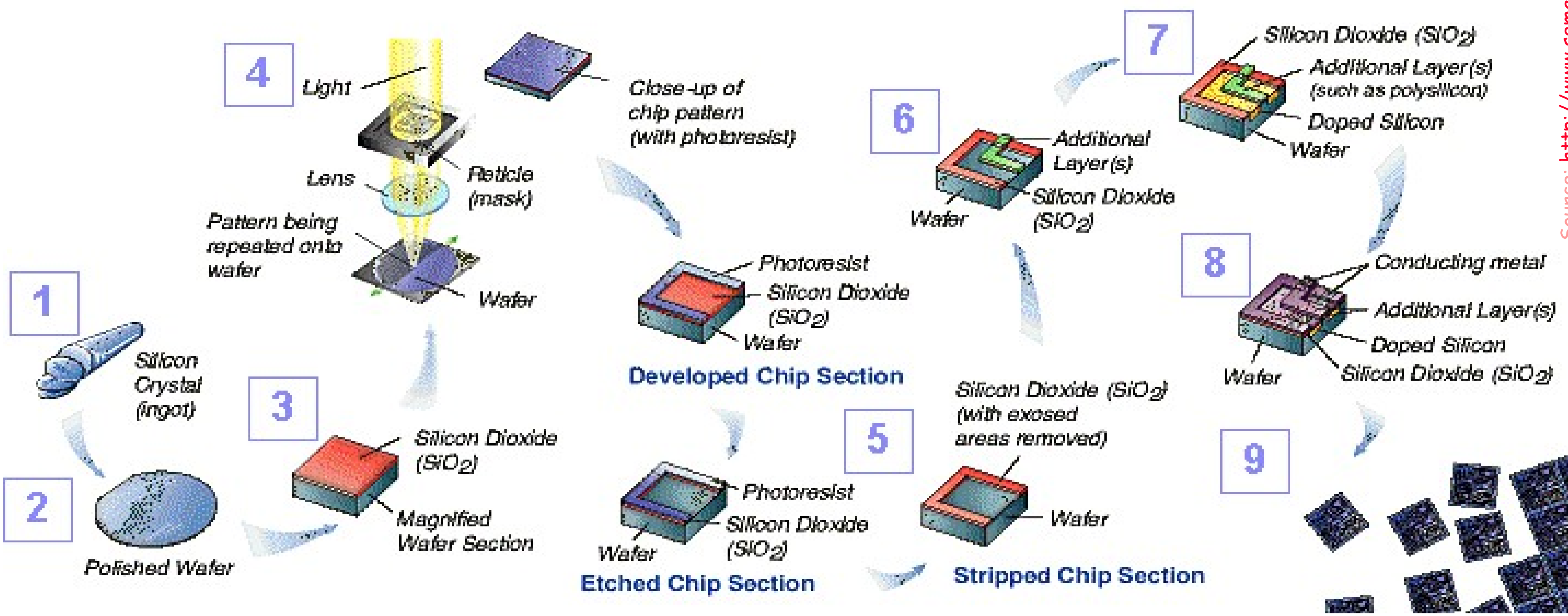




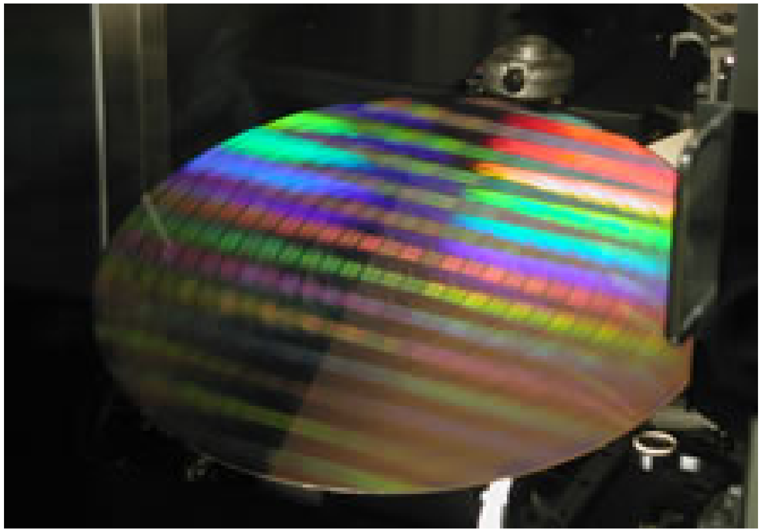
- A single crystal of silicon, a silicon ingot, grown by the Czochralski technique. The diameter of this ingot is 6 inches (Courtesy of Texas Instruments). State of the art fabs now use 300mm wafers

Integrated circuit fabrication is a printing process

1. Grow pure silicon crystal
2. Slice into wafers and polish
3. Grow surface layer of silicon dioxide (ie glass), either using high-temperature oxygen or chemical vapour deposition
4. Coat surface with photoresist layer, then use mask to selectively expose photoresist to ultraviolet light
5. Etch away silicon dioxide regions not covered by hardened photoresist
6. Further photolithography steps build up additional layers, such as polysilicon
7. Exposed silicon is doped with small quantities of chemicals which alter its semiconductor behaviour to create transistors
8. Further photolithography steps build layers of metal for wiring
9. Die are tested, diced, tested and packaged



Source: http://www.sematech.org/public/news/mfgproc/mfgproc.htm



Close up of the wafer as it spins during a testing procedure

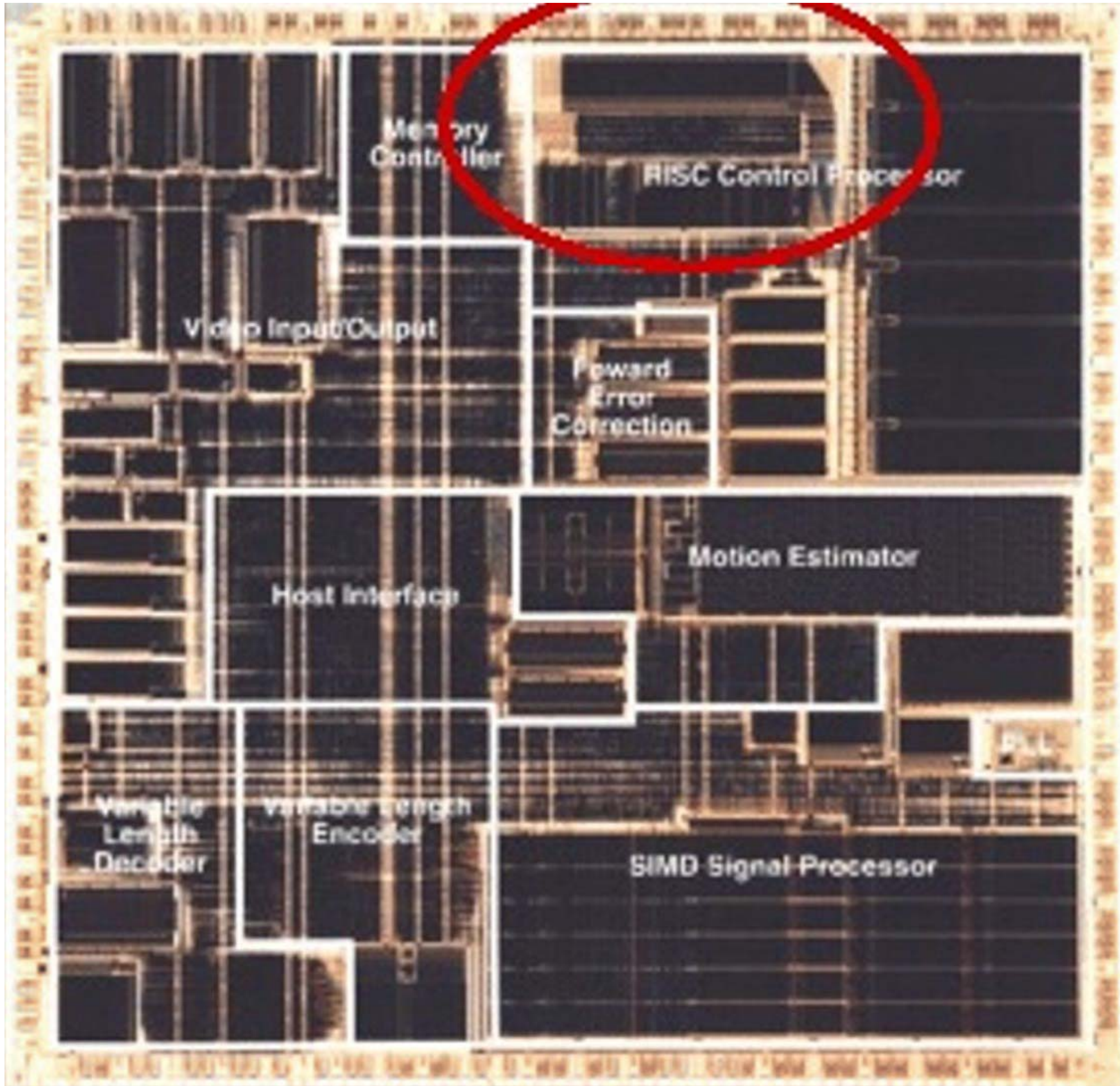
Checking wafers processing in a vertical diffusion furnace

Intel technicians monitor wafers in an automated wet etch tool. The process cleans the wafers of any excess process chemicals or contamination.

Source: http://www.intel.com/pressroom/archive/photos/manufacturing_photos.htm

The future

- More transistors
- Higher clock rates
- Lower power
- System-on-a-chip
- Field-programmable gate arrays
- "Compiling to silicon"
- Optical interconnect
- Quantum computing?



AVP-III Video Codec from Lucent Technologies

Source: http://6371.lcs.mit.edu/Fall96/lectures/L1/P005.html

Intel x86/Pentium Family

CPU	Year	Data Bus	Max. Mem.	Transistors	Clock MHz	Av. MIPS	Level-1 Caches
8086	1978	16	1MB	29K	5-10	0.8	
80286	1982	16	16MB	134K	8-12	2.7	
80386	1985	32	4GB	275K	16-33	6	
80486	1989	32	4GB	1.2M	25-100	20	8Kb
Pentium	1993	64	4GB	3.1M	60-233	100	8K Instr + 8K Data
Pentium Pro	1995	64	64GB	5.5M +15.5M	150-200	440	8K + 8K + Level2
Pentium II	1997	64	64GB	7M	266-450	466-	16K+16K + L2
Pentium III	1999	64	64GB	8.2M	500-1000	1000-	16K+16K + L2
Pentium IV	2001	64	64GB	42M	1300-2000		8K + L2

On-line manuals: http://x86.ddj.com/intel.doc/386manuals.htm

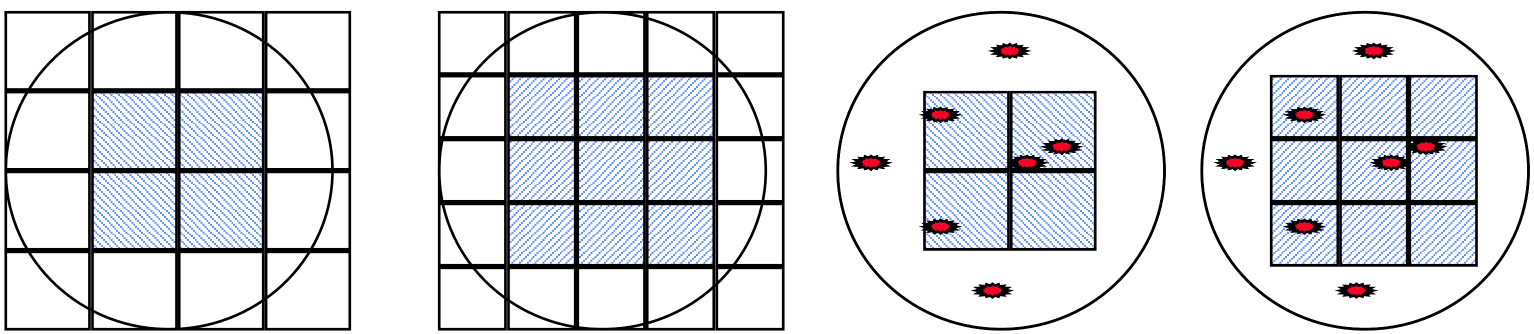
On-line details: http://www.sandpile.org/ia32/index.htm

Integrated Circuits Costs

$$\text{IC cost} = \frac{\text{Die cost} + \text{Testing cost} + \text{Packaging cost}}{\text{Final test yield}}$$

$$\text{Die cost} = \frac{\text{Wafer cost}}{\text{Dies per Wafer} \times \text{Die yield}}$$

$$\text{Dies per wafer} = \frac{\pi\,(\text{Wafer\_diam}/2)^2}{\text{Die\_Area}} - \frac{\pi \times \text{Wafer\_diam}}{\sqrt{2 \cdot \text{Die\_Area}}} - \text{Test\_Die}$$



$$\text{Die Yield} = \text{Wafer\_yield} \times \left\{ 1 - \left(\frac{\text{Defect\_Density} \times \text{Die\_area}}{\alpha} \right)^{-\alpha} \right\}$$

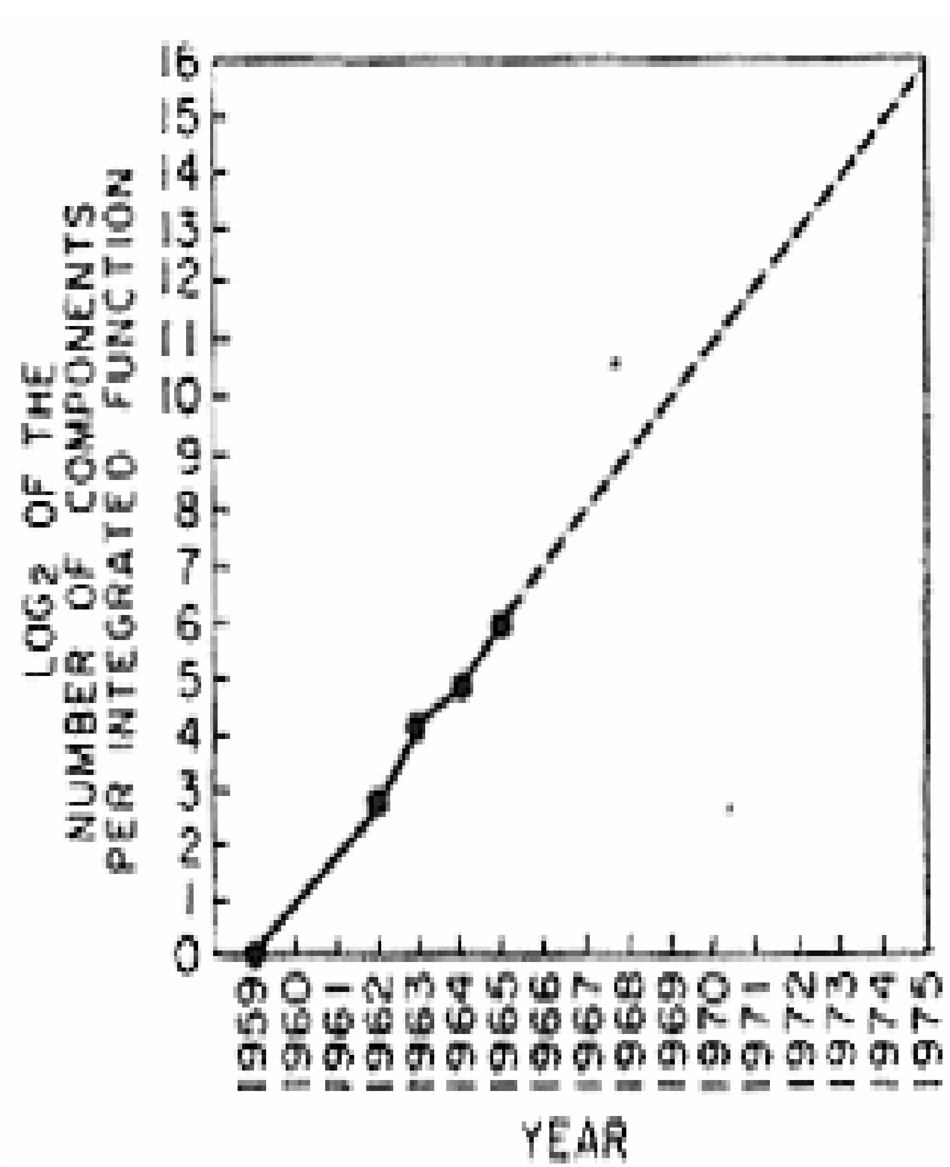
Die Cost goes roughly with die area^4

Real World Examples

Chip	Metal layers	Line width	Wafer cost	Defect /cm²	Area mm²	Dies/ wafer	Yield	Die Cost
386DX	2	0.90	$900	1.0	43	360	71%	$4
486DX2	3	0.80	$1200	1.0	81	181	54%	$12
PowerPC 601	4	0.80	$1700	1.3	121	115	28%	$53
HP PA 7100	3	0.80	$1300	1.0	196	66	27%	$73
DEC Alpha	3	0.70	$1500	1.2	234	53	19%	$149
SuperSPARC	3	0.70	$1700	1.6	256	48	13%	$272
Pentium	3	0.80	$1500	1.5	296	40	9%	$417

- **From "Estimating IC Manufacturing Costs," by Linley Gwennap, *Microprocessor Report*, August 2, 1993, p. 15**

Moore's "Law"



Graph extracted from Moore's 1965 article

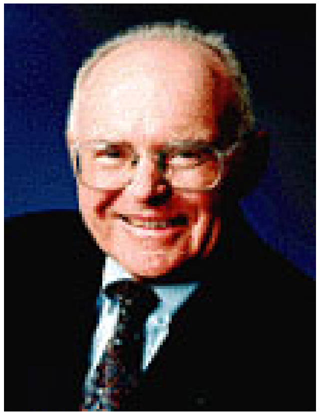
Cramming more components onto integrated circuits
By Gordon E. Moore

Electronics, Volume 38, Number 8, April 19, 1965
(See http://www.intel.com/research/silicon/mooreslaw.htm)

"With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip"

Gordon Moore left Fairchild to found Intel in 1968 with Robert Noyce and Andy Grove,

Technology Trends: Microprocessor Capacity



CMOS improvements:
- Die size: 2X every 3 yrs
- Line width: halve / 7 yrs