Register Allocation in Kernel Generators

Matteo Frigo

Cilk Arts

July 9, 2007

CILK ARTS

Summary

- Poor register allocation $\Longrightarrow$ poor kernel performance.
- Kernel generators must do register allocation one way or the other.
- Register allocation can be factored into two subproblems:
  - Scheduling.
  - Register allocation of straight-line code.
- Ordinary compilers can register-allocate straight-line code.
- Compilers cannot schedule properly. Kernel generators are responsible for the schedule.
- FFTW uses a fixed "cache oblivious" schedule. Although independent of the processor, this schedule seems to be hard to beat.
- Other problems may require more sophistication.

Impact of inefficient register allocation

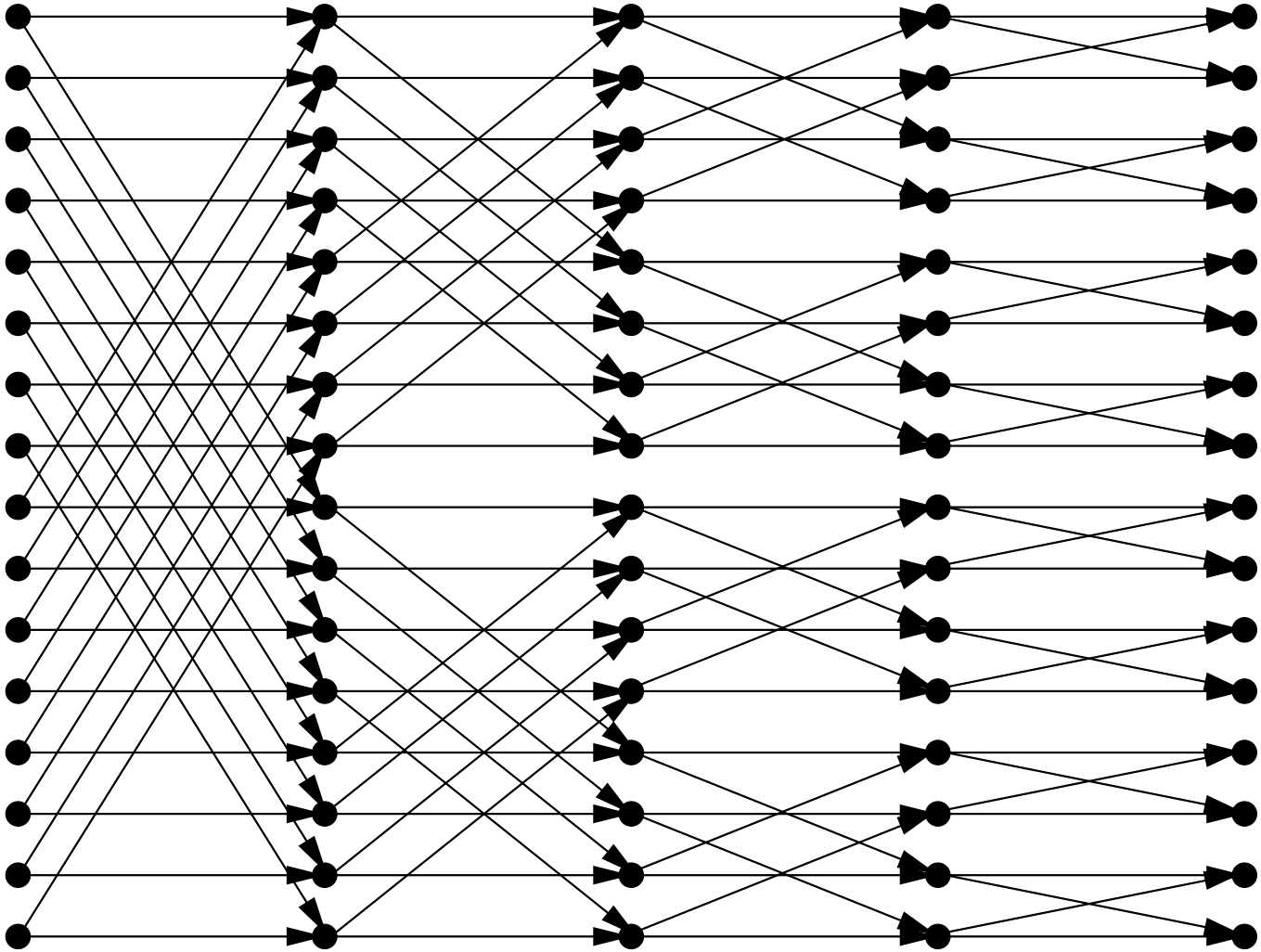
32-point complex FFT in FFTW, PowerPC 7447

add/sub	fma	load	store	code size	cycles
C source:					
236	136	64	64	$\approx$ 600 lines	
Output of `gcc-3.4 -O2`:					
236	136	484	285	5620 bytes	$\approx$ 1550
Output of `gcc-3.4 -O2 -fno-schedule-insns`:					
236	136	134	125	2868 bytes	$\approx$ 640

- Twice as many instructions (all register spills).
- 2.5x slowdown.

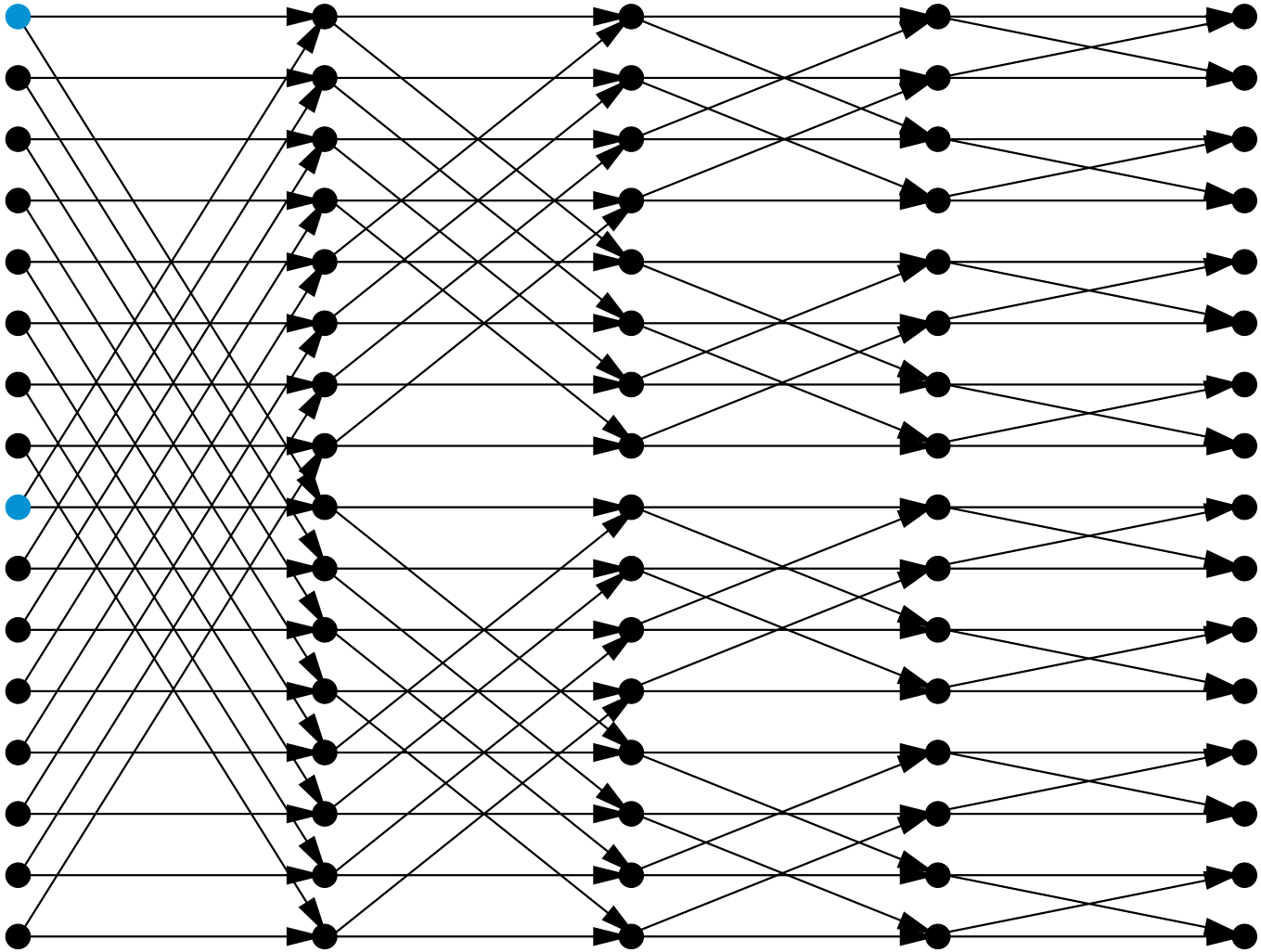
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



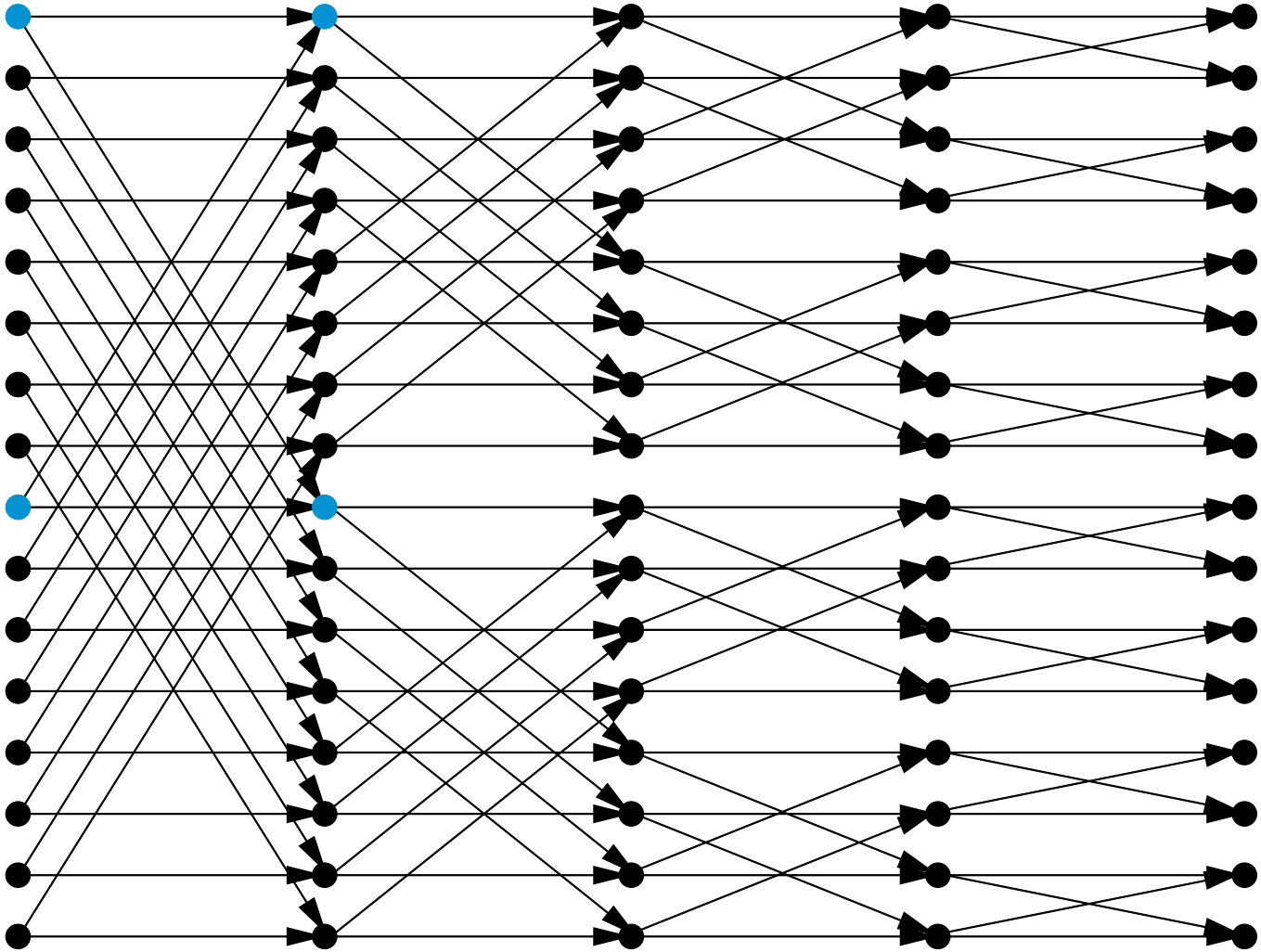
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



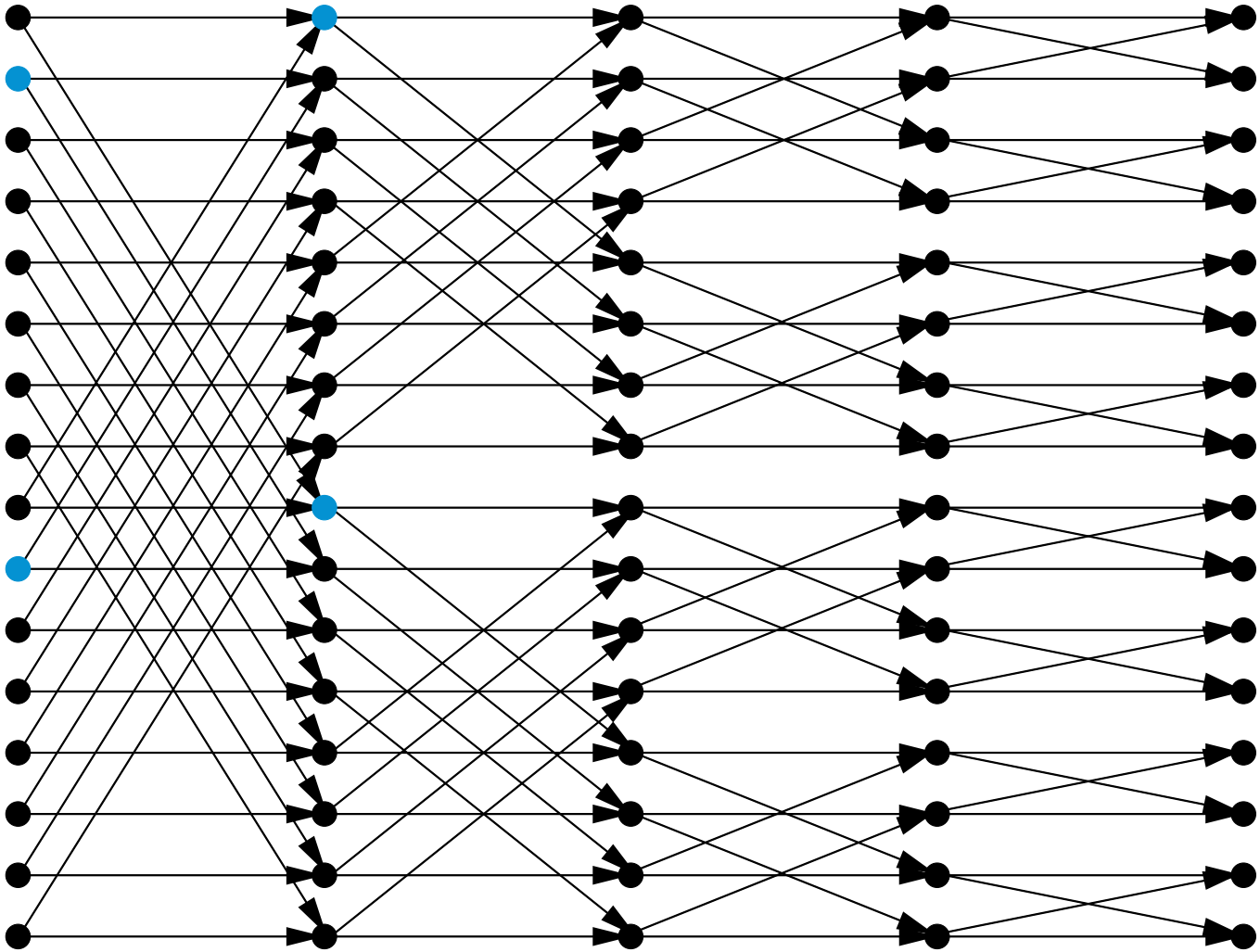
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



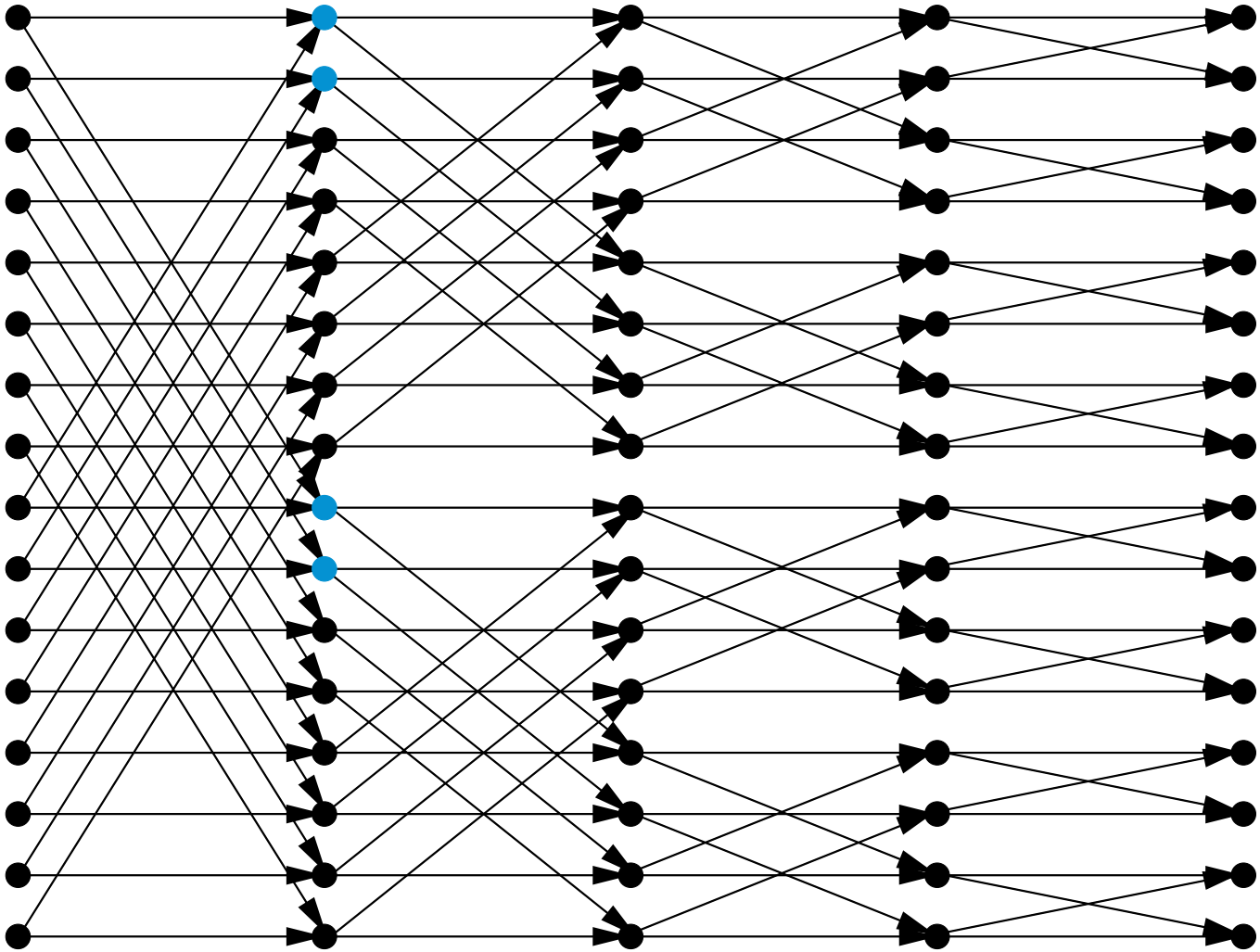
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



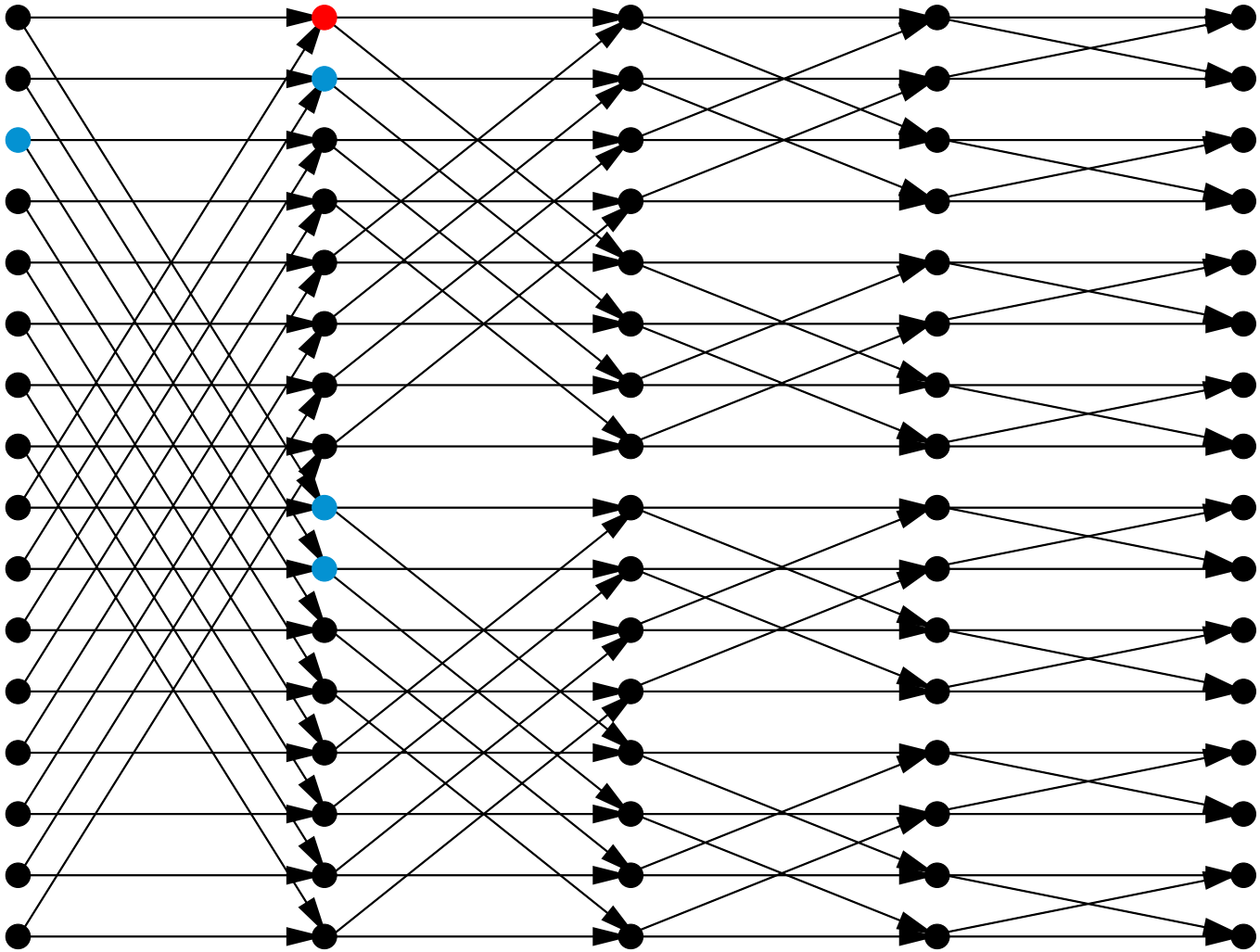
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



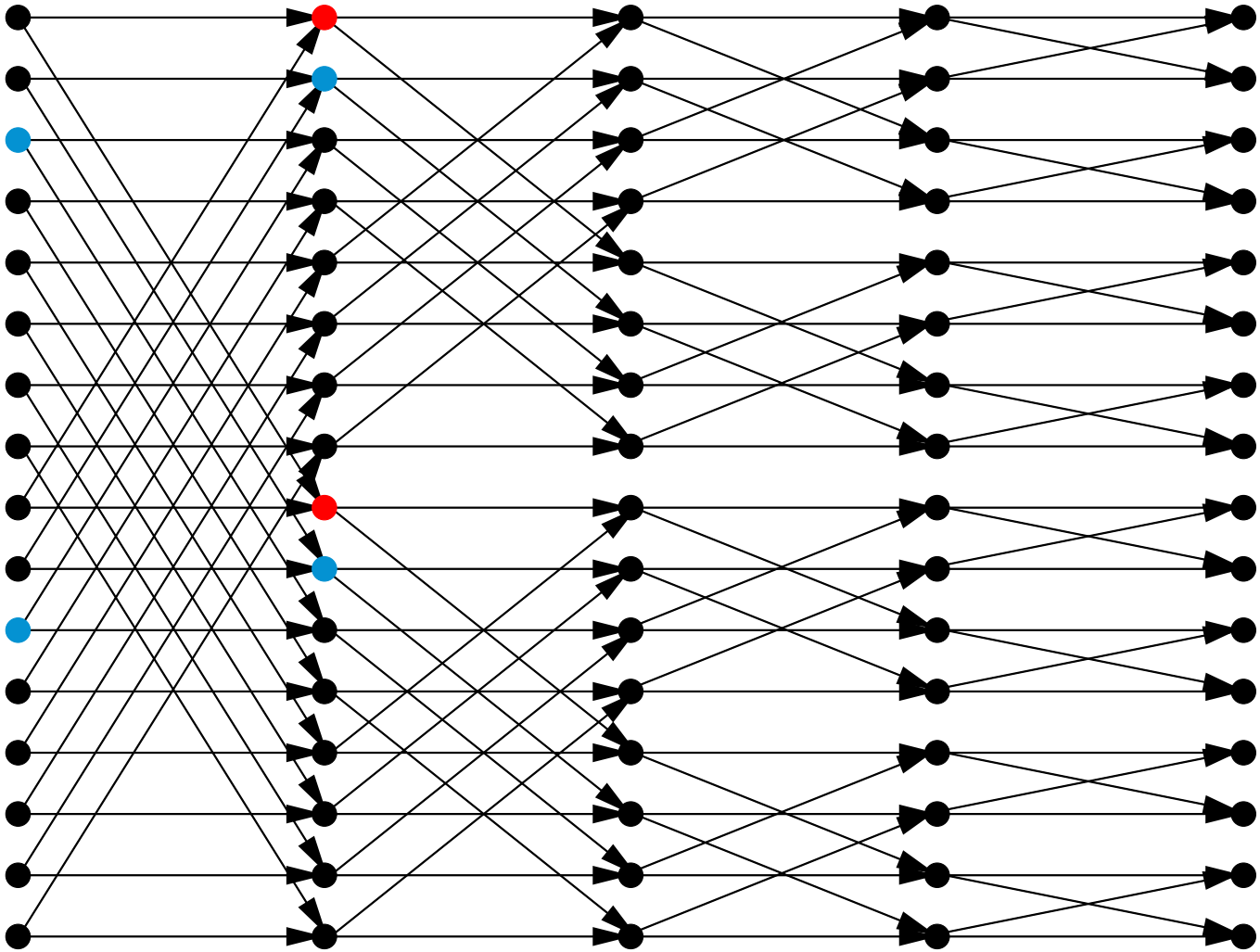
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



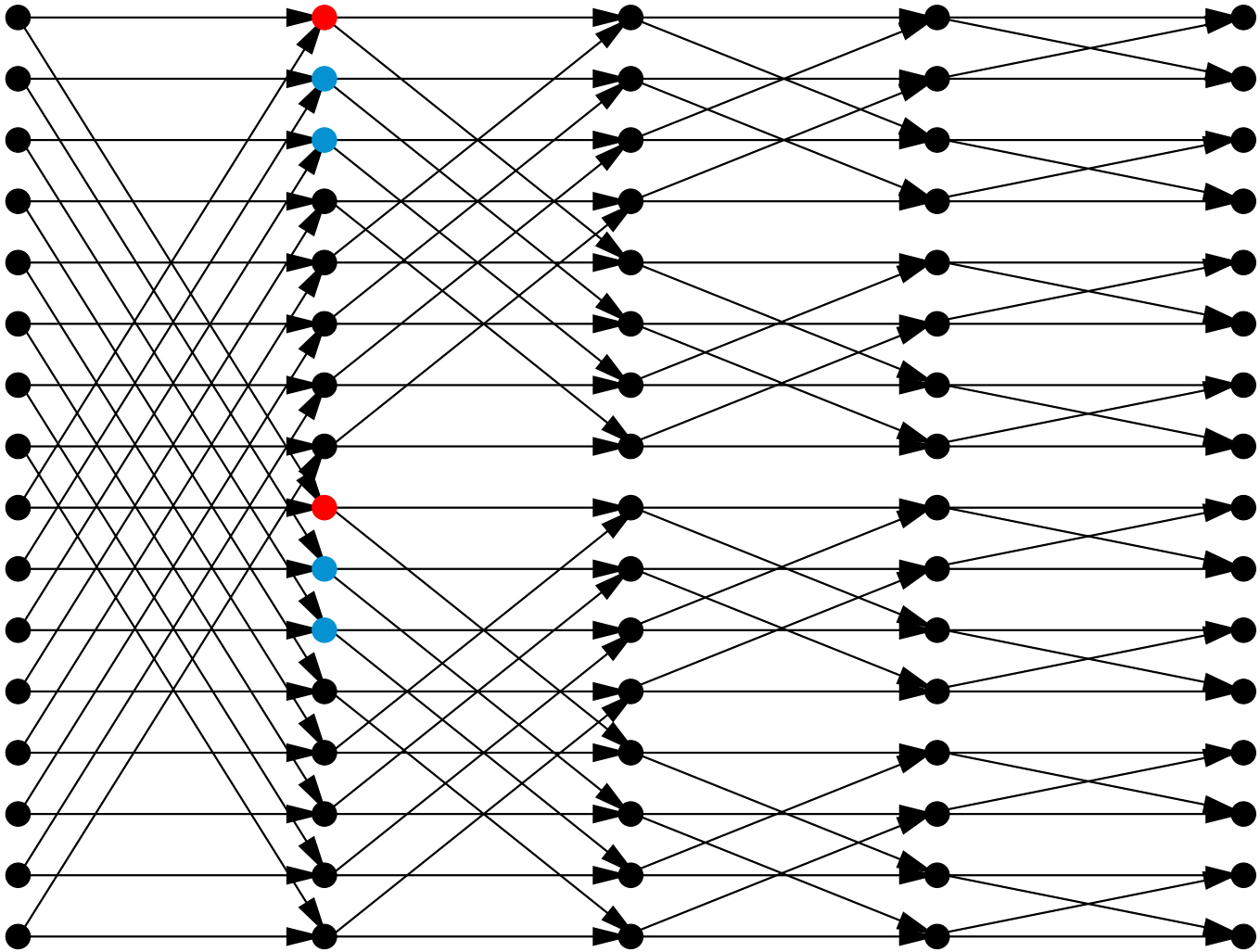
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.

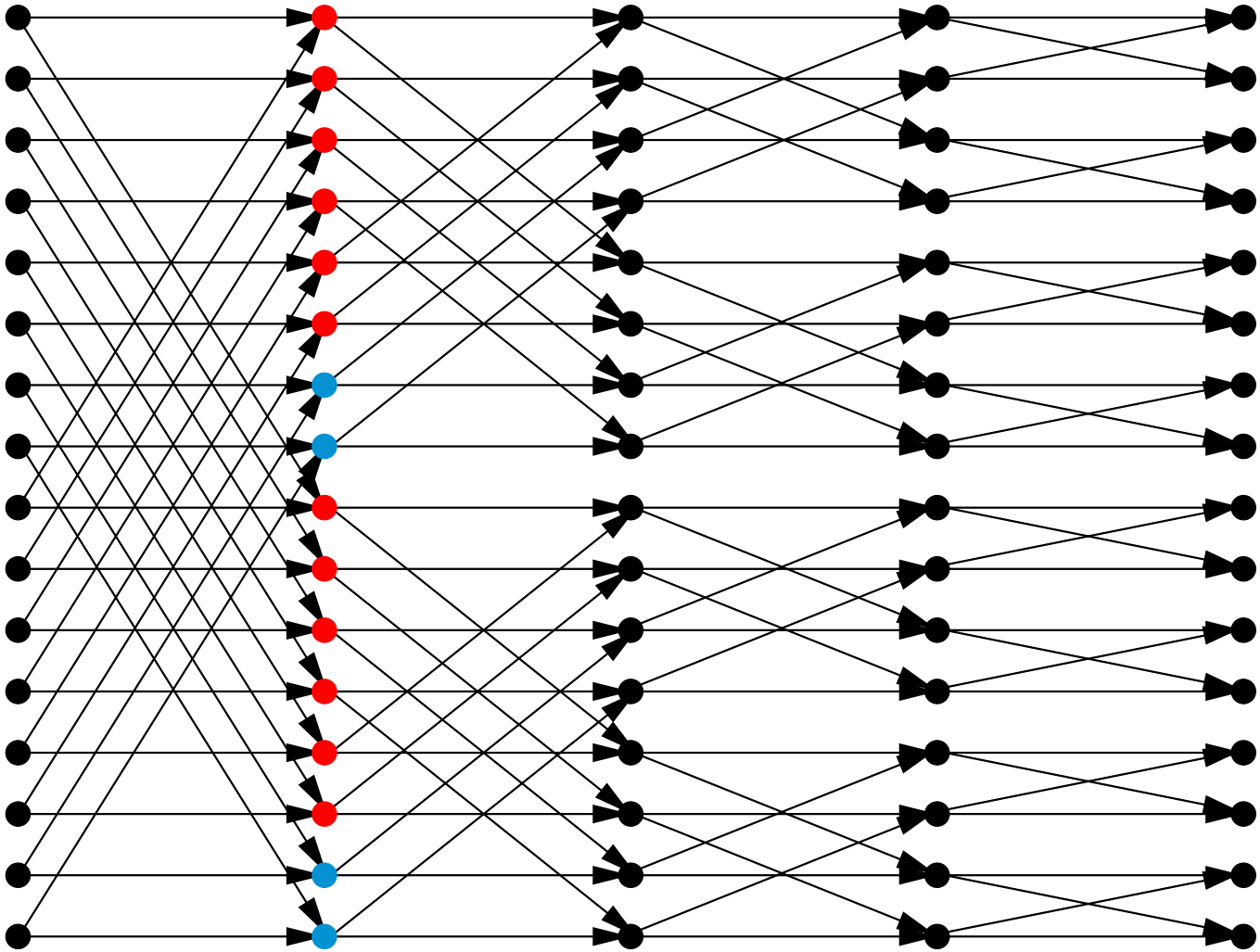


Register allocation in `gcc -O2`

keep going for a while...

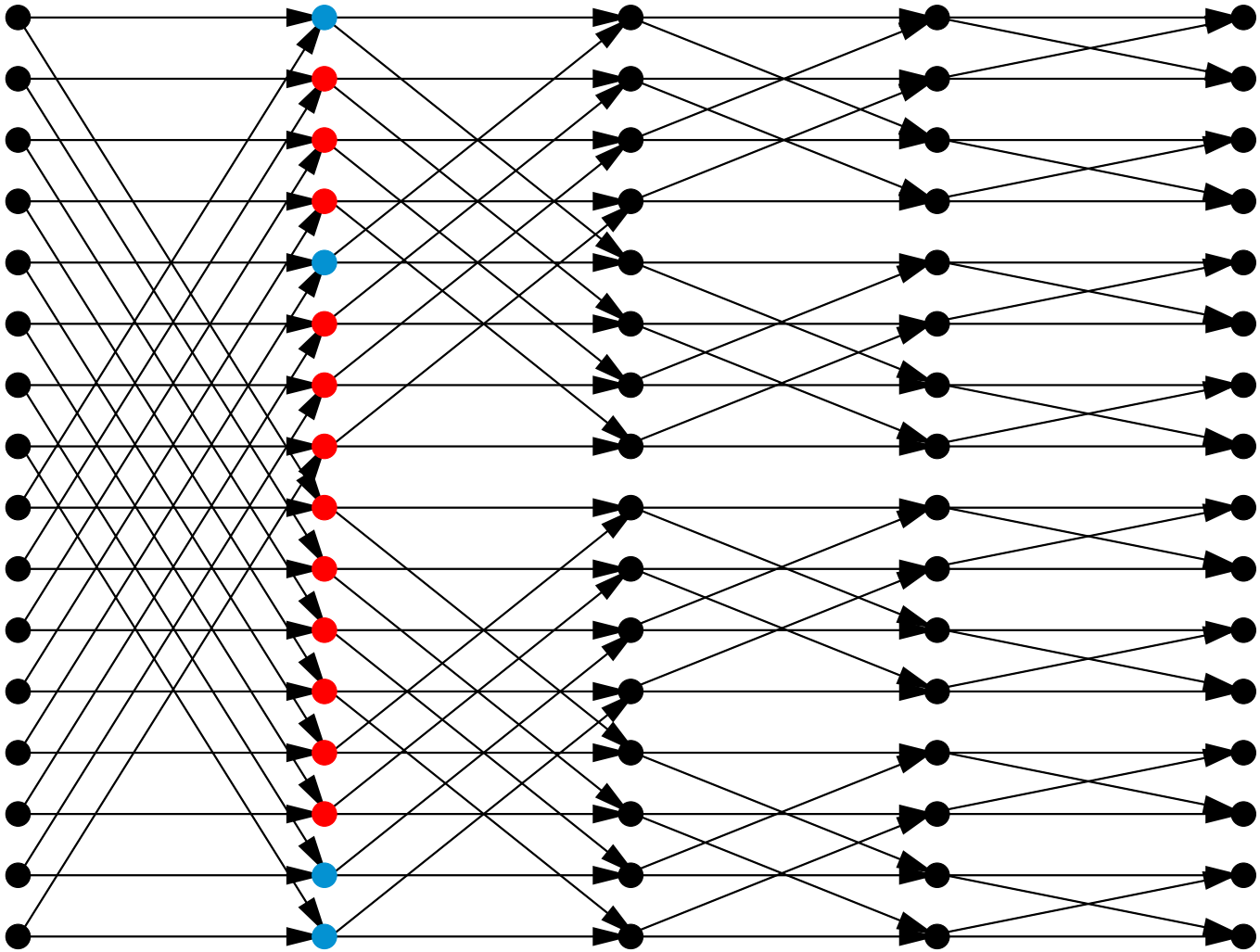
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



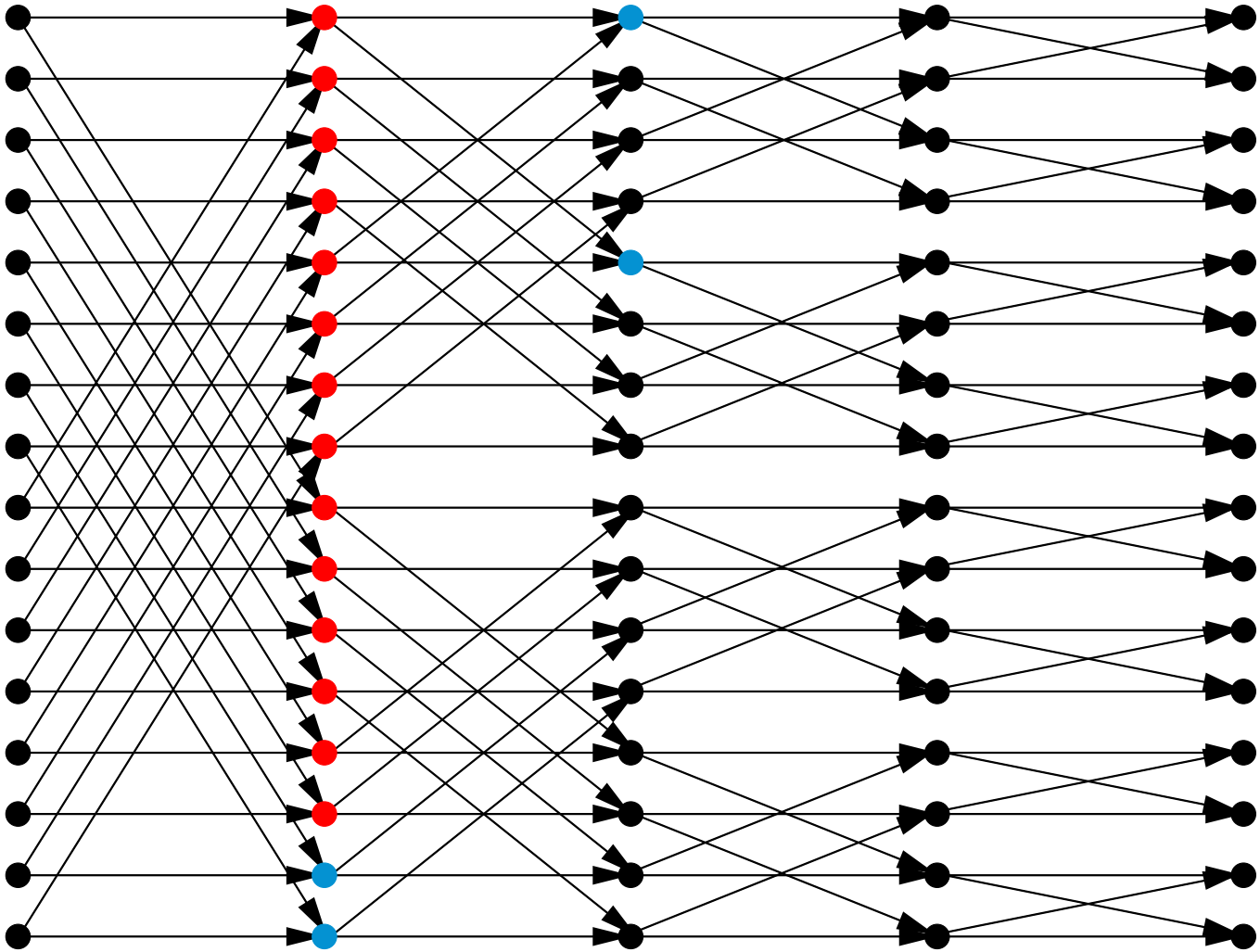
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.



Why the `gcc -O2` strategy cannot work

Theorem

If

- *you compute the FFT level by level; and*
- $n \gg$ *number of registers*

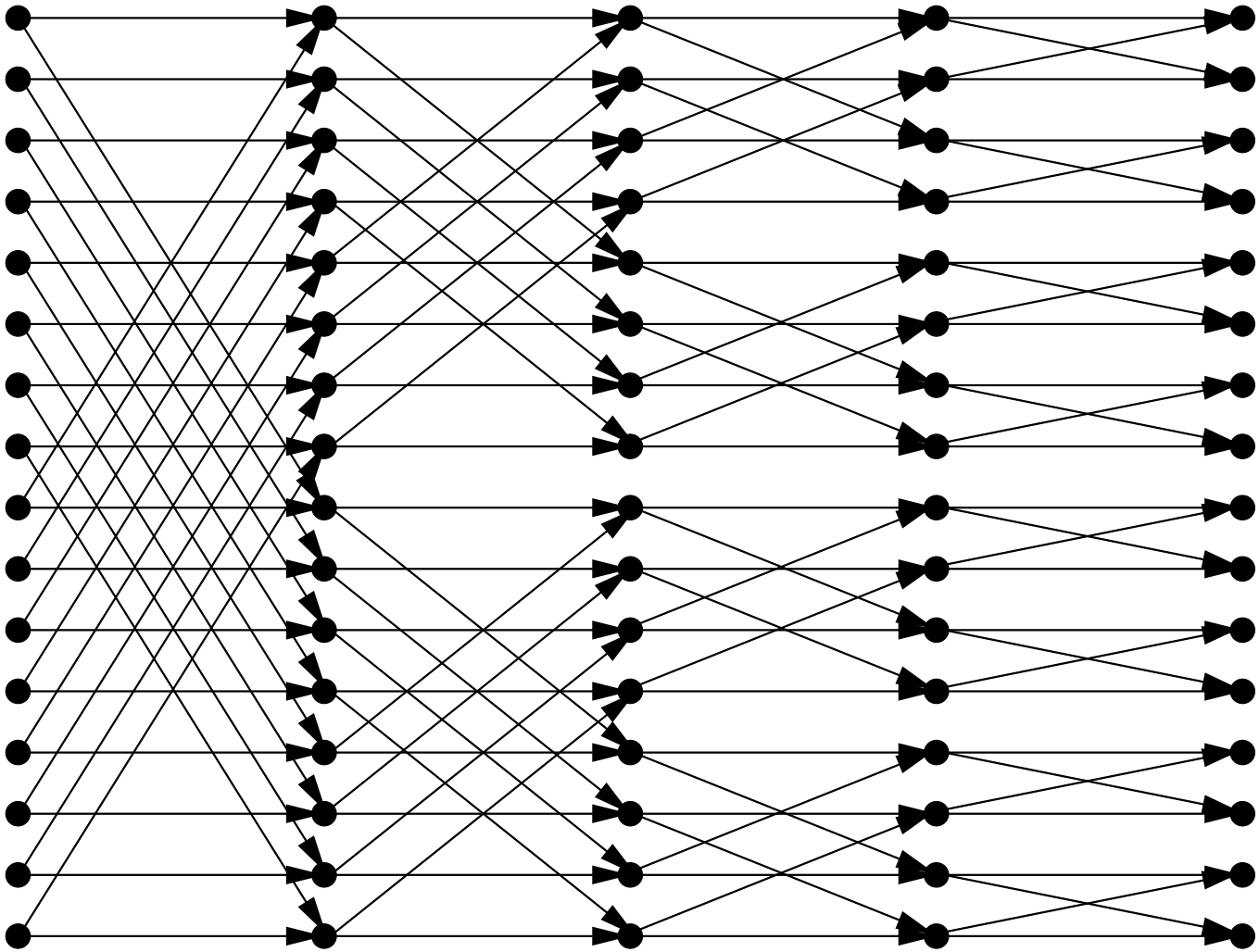
then

- *any register allocation policy incurs* $\Theta(n \log n)$ *register spills.*

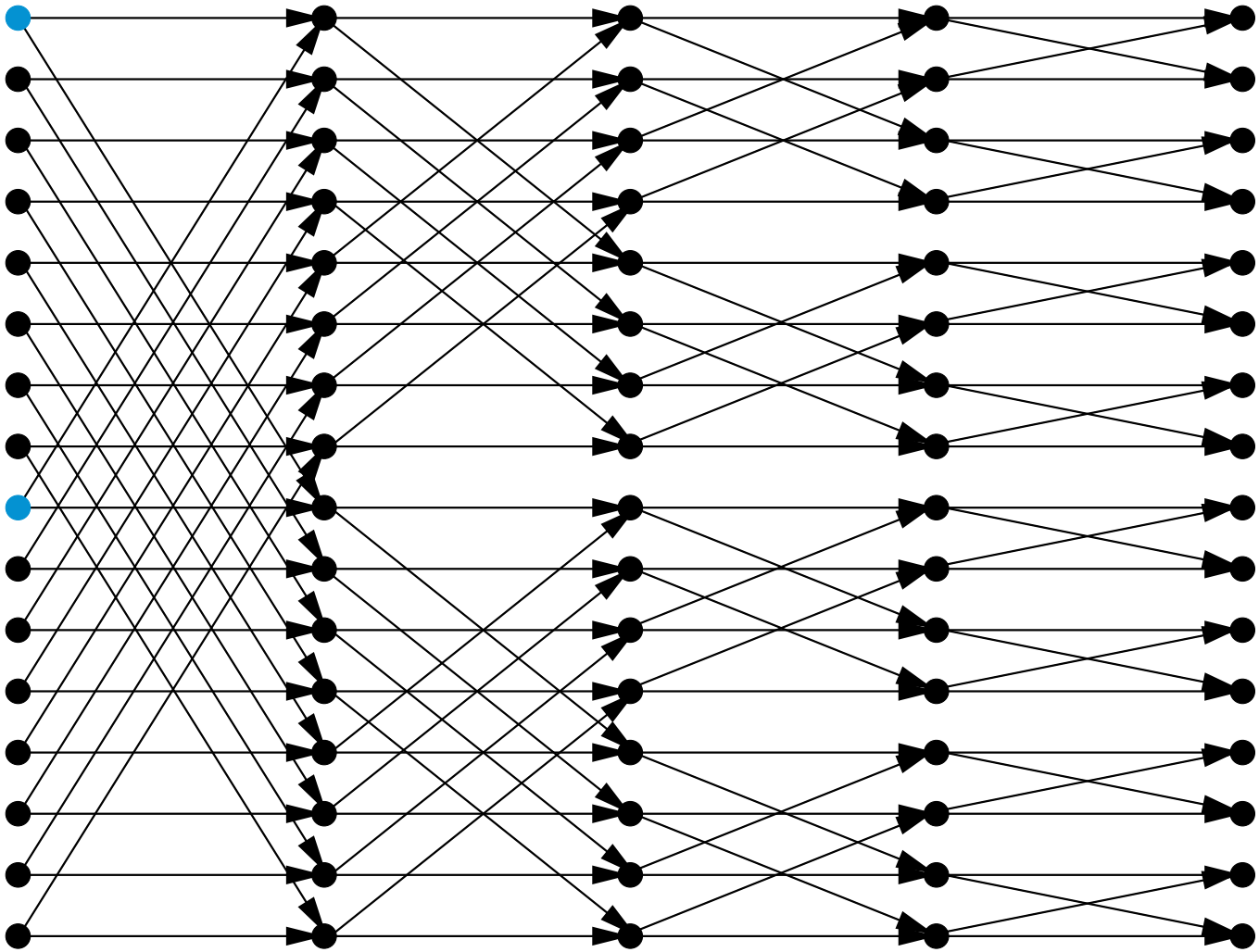
Corollary

$O(1)$ *spills/flop no matter how many registers the machine has.*

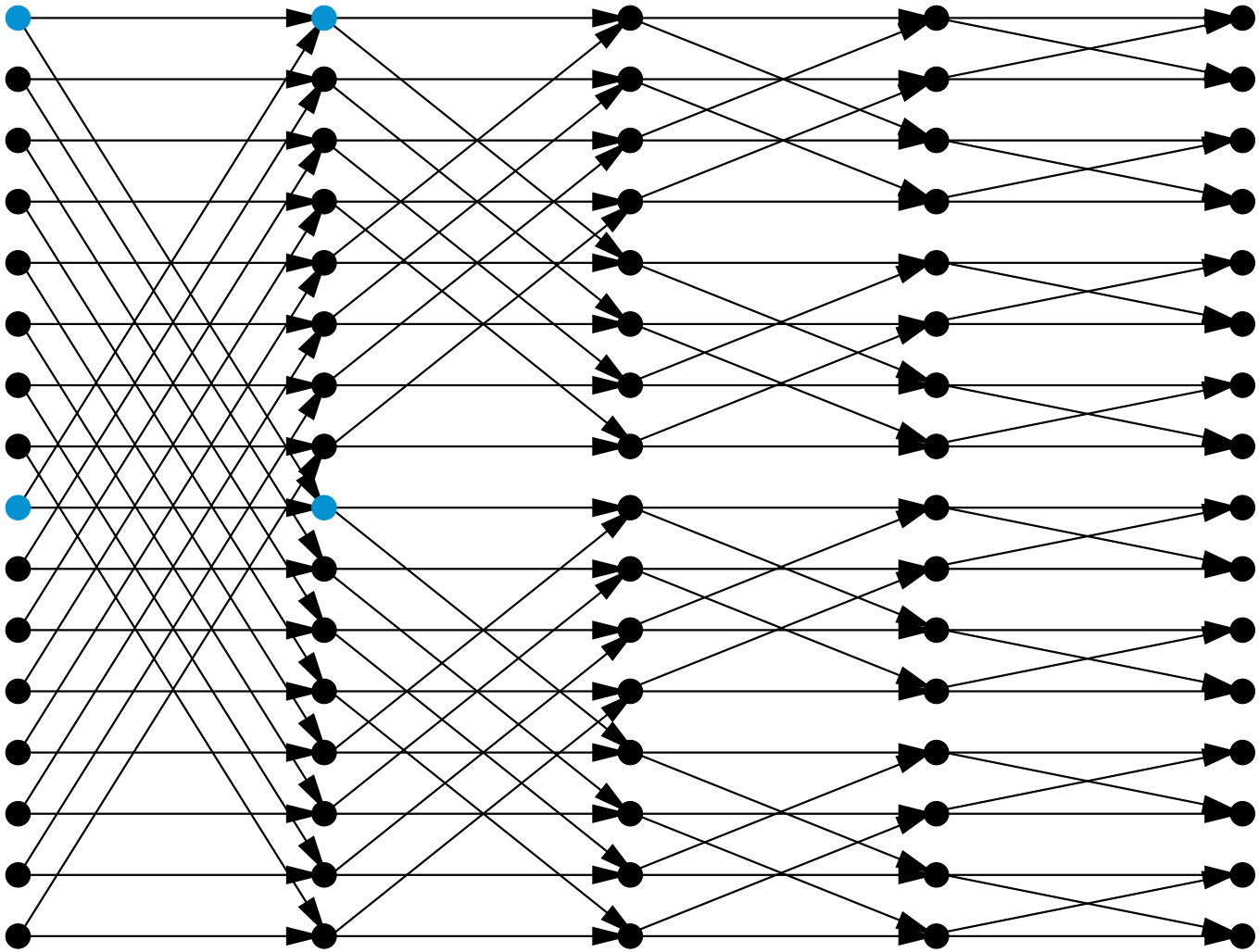
Better strategy: blocking



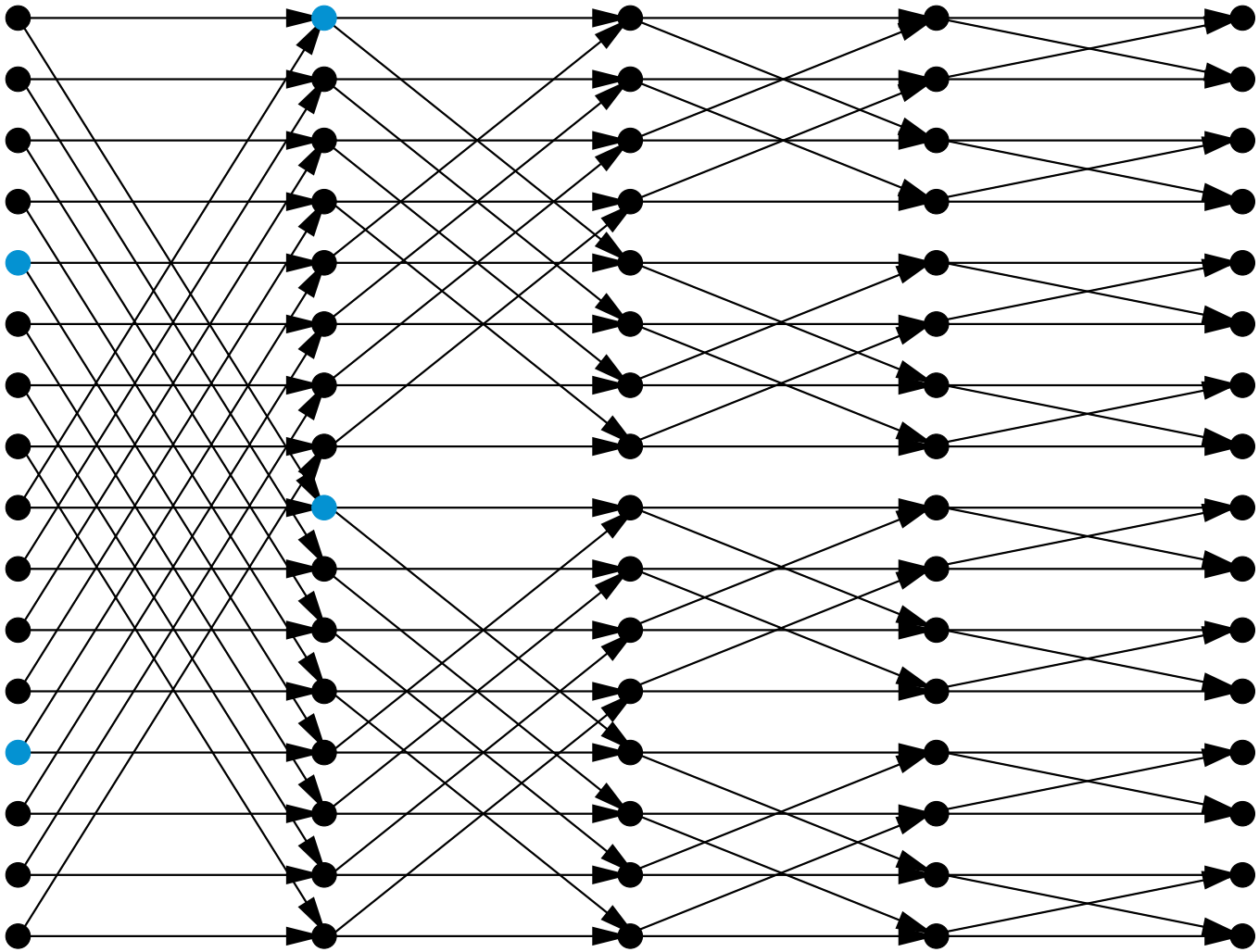
Better strategy: blocking



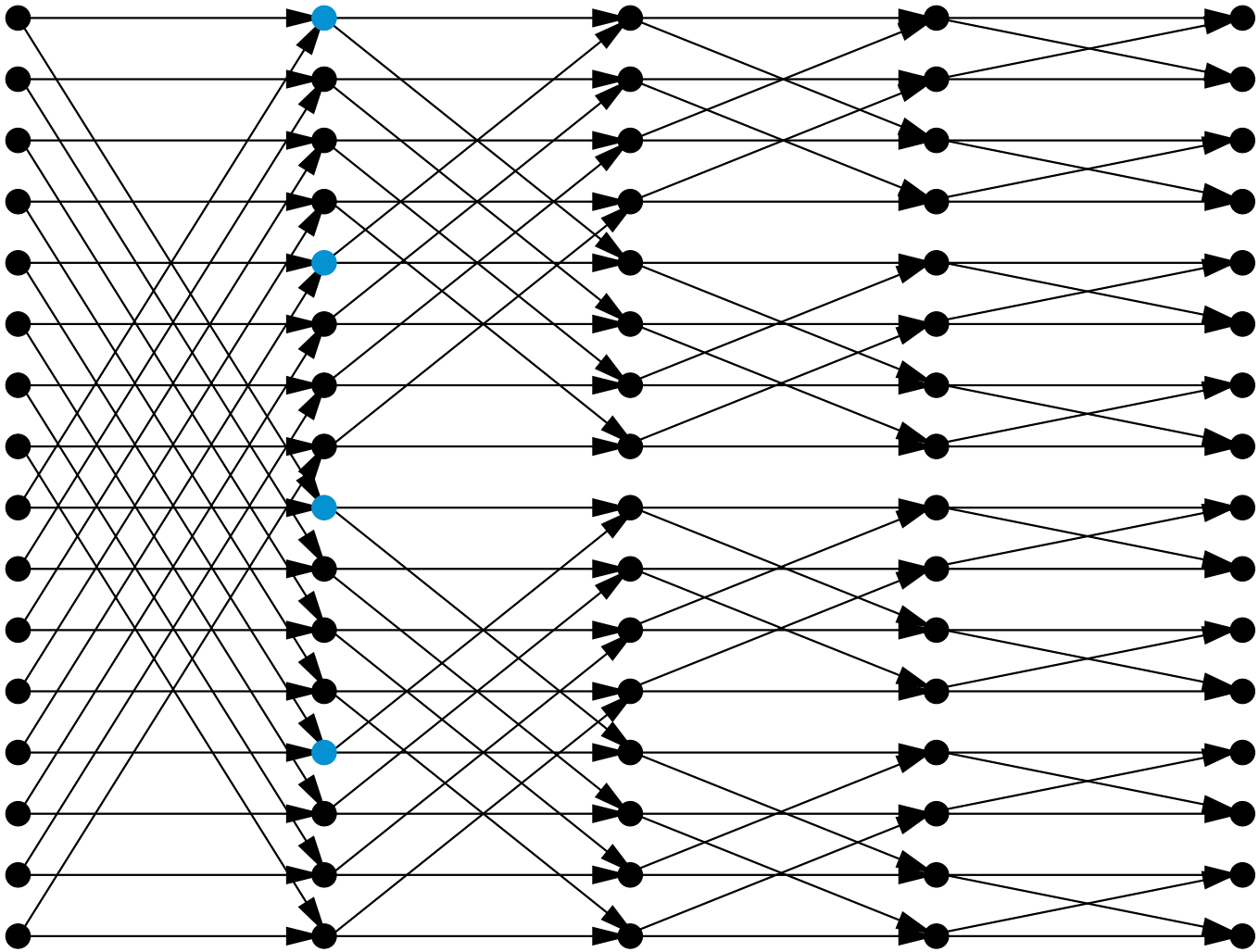
Better strategy: blocking



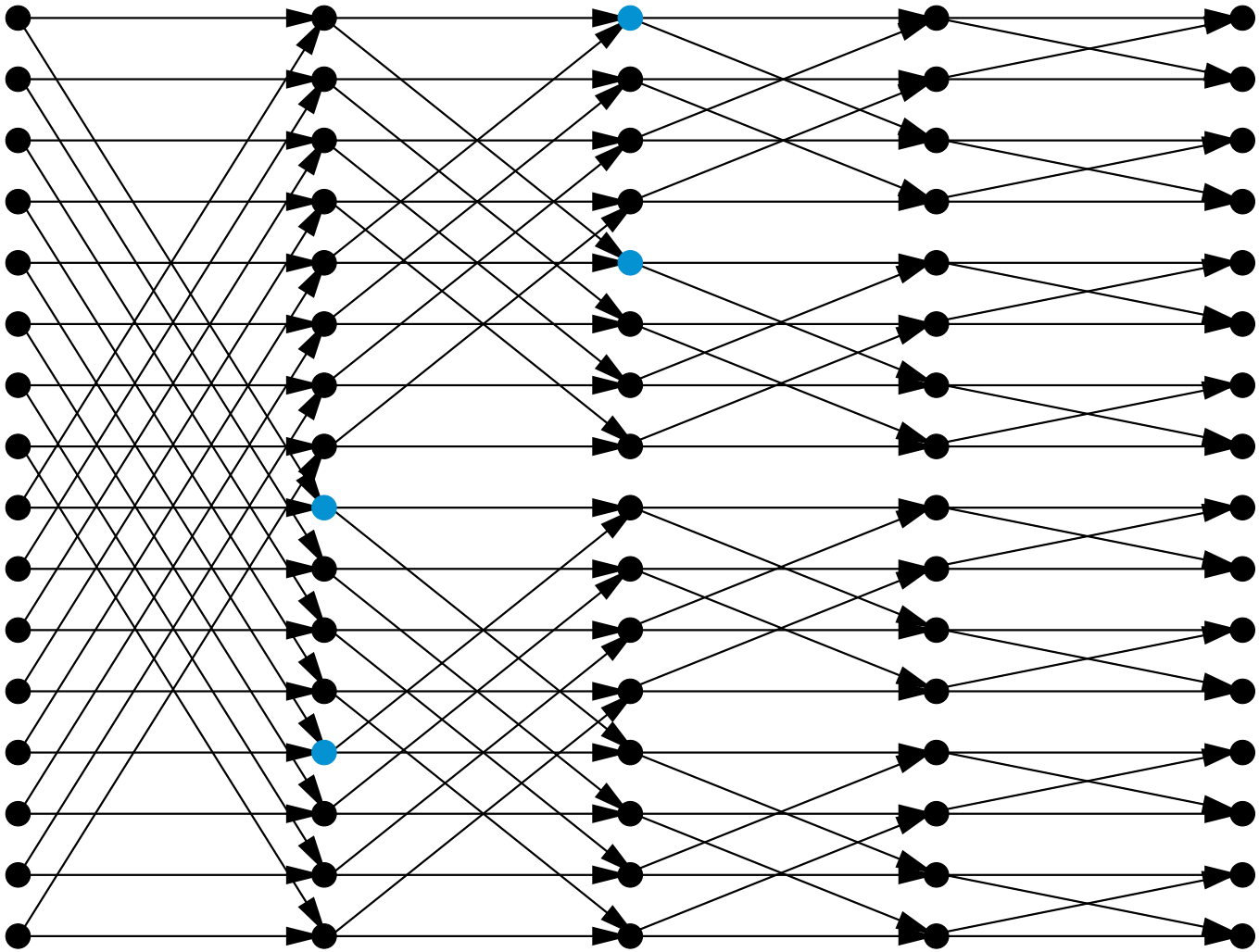
Better strategy: blocking



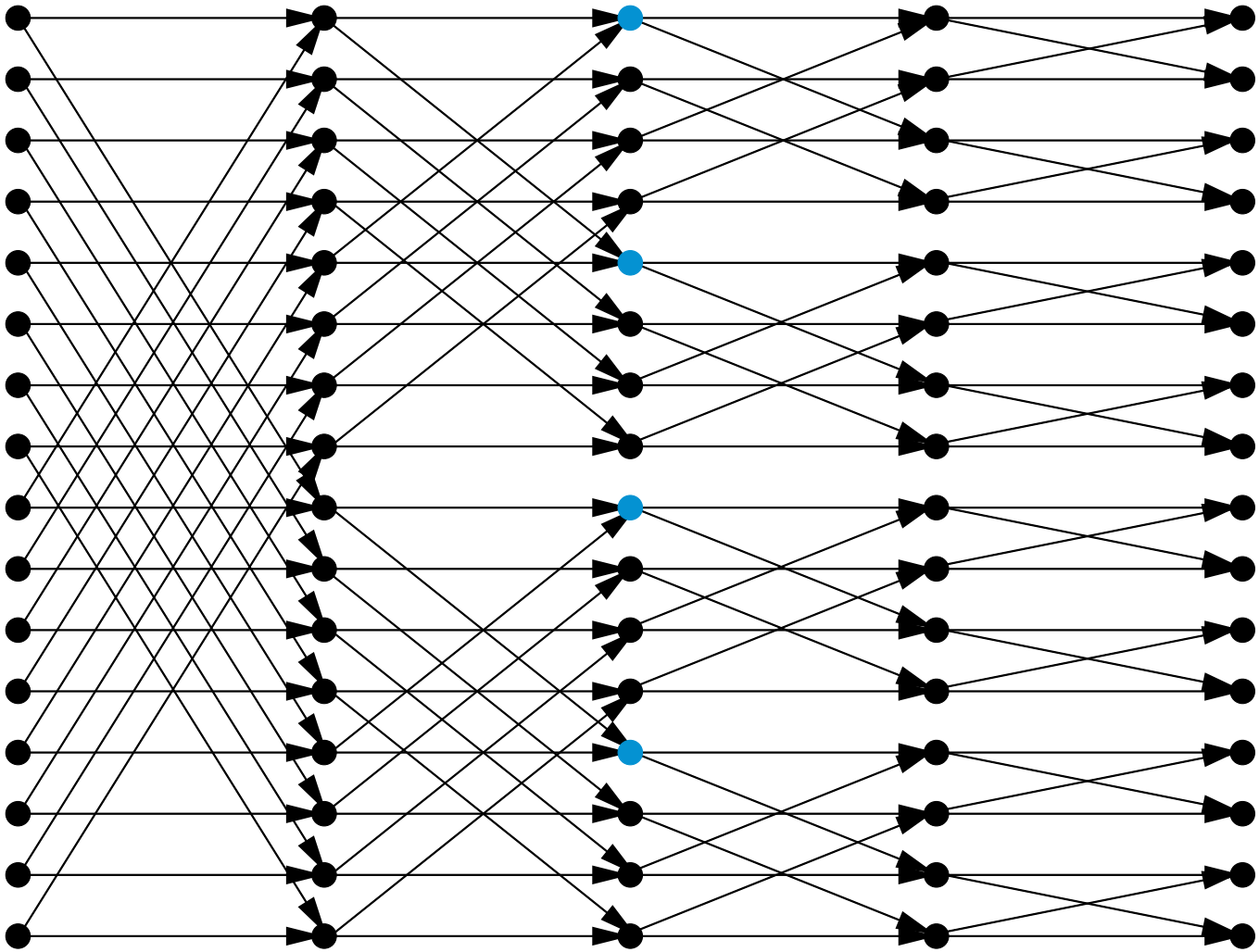
Better strategy: blocking



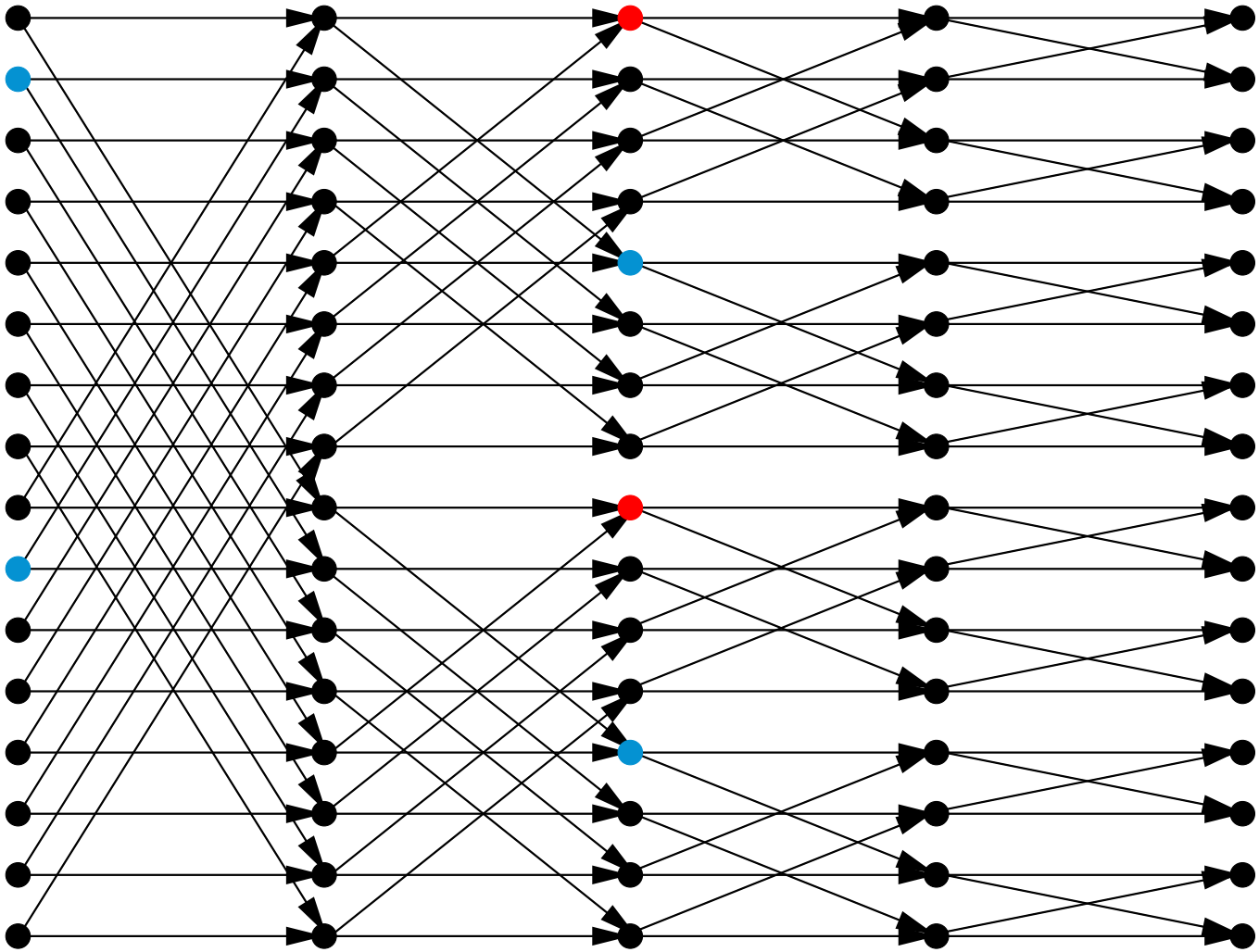
Better strategy: blocking



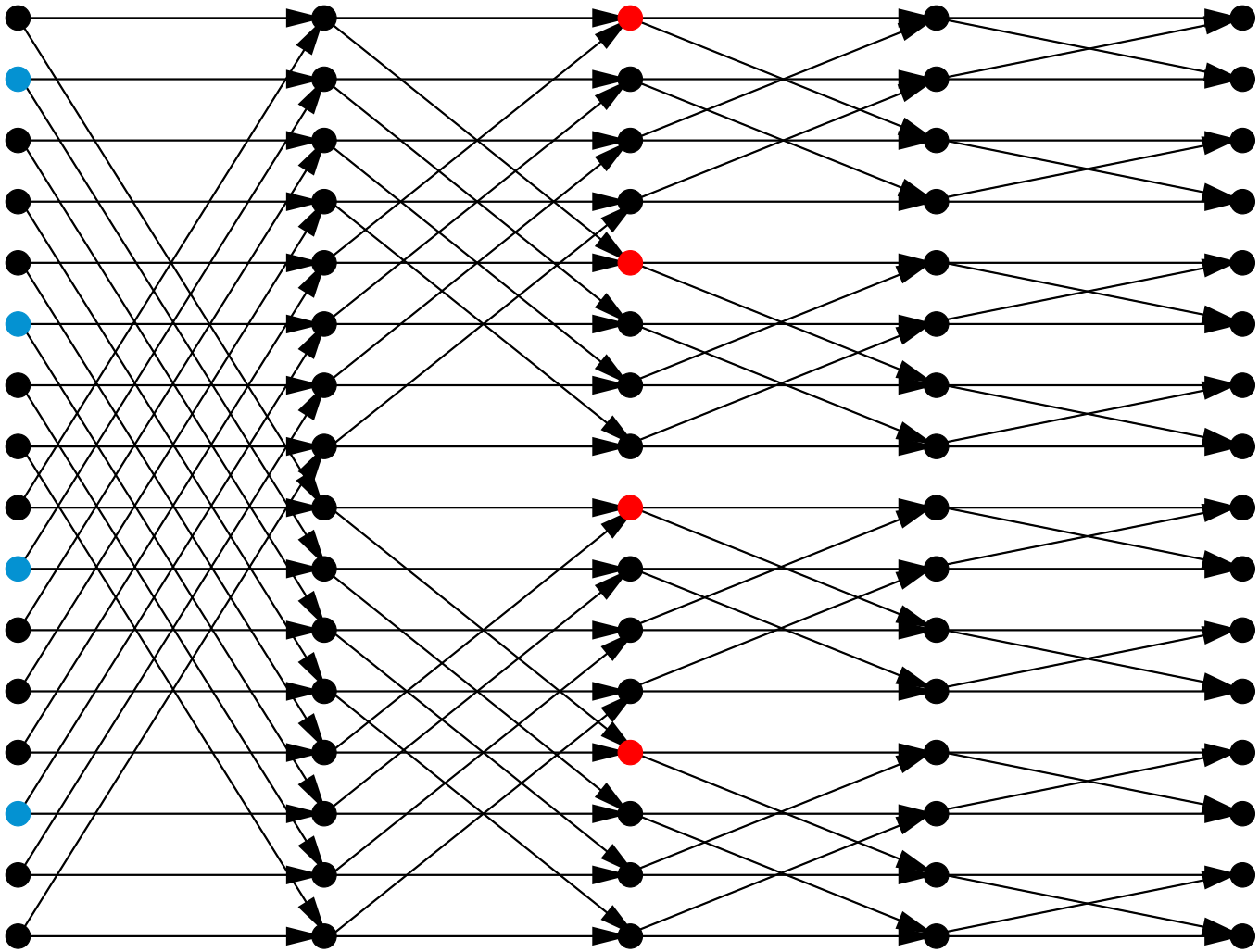
Better strategy: blocking



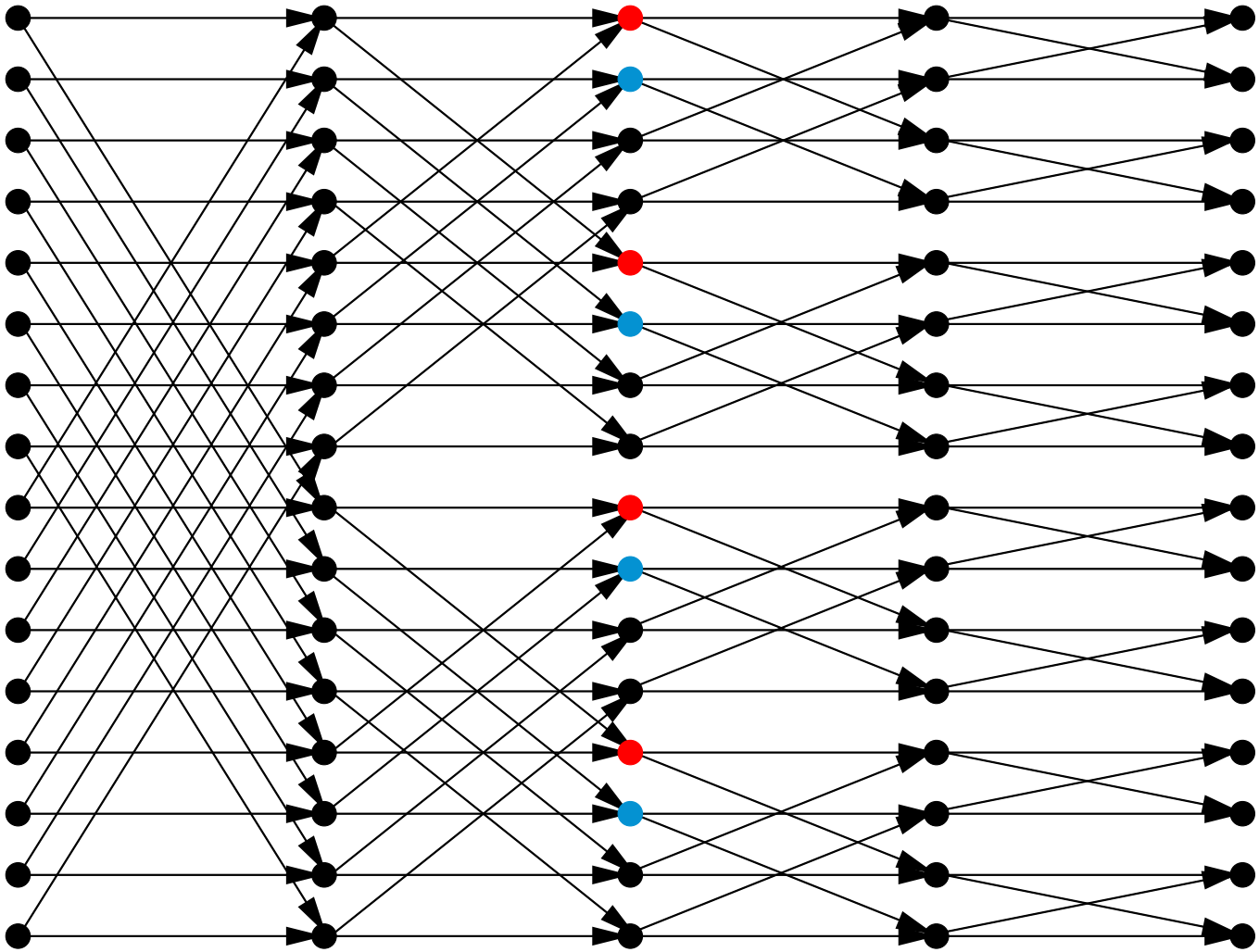
Better strategy: blocking



Better strategy: blocking



Better strategy: blocking



Analysis of the blocking schedule

Theorem (Upper bound)

With R registers,

- *a schedule exists such that*
- *a register allocation exists such that*
- *the execution incurs $O(n \log n / \log R)$ register spills.*

Proof.

Block for R registers. Loading R inputs allows you to compute $\log R$ levels without spilling, i.e., $\log R$ flops per spill. □

Theorem (Lower bound, Hong and Kung '81)

Any execution of the butterfly graph with R registers incurs $\Omega(n \log n / \log R)$ register spills.

Complexity of register allocation

Theorem (Motwani et al., 1995)

Given dag, find schedule of the dag and register assignment that minimizes the number of register spills: **NP-hard**.

Theorem (Belady 1966)

Given a schedule of the dag, find register assignment that minimizes the number of register spills: $\approx$ **linear time**.

Corollary

- *You are responsible for the schedule.*
- *You don't have worry about the register assignment. The compiler can do it.*

Belady's algorithm

- Traverse the schedule in execution order.
- If an instruction needs a value not in a register, obtain a register and load the value.
- If you need to evict a value from a register, evict the one used furthest in the future.

Register allocation in FFTW

- The FFTW "codelet" generator produces C.
- The generator schedules the C code.
  - The scheduling algorithm is FFT-specific.
  - This is scheduling for register allocation, not "instruction scheduling" for pipelining purposes.
- We **assume** (i.e., hope) that the C compiler implements the optimal register allocation for the given schedule.

Ordinary compilers handle straight-line code well

Cycle counts on Pentium III, circa 2002

	FFT(8)	FFT(16)	FFT(32)	FFT(64)
Belady	150	350	838	2055
gcc-2.95 -O2	165	372	913	2254
gcc-2.95 -O	151	354	845	2091
gcc-3.2 -O2	152	390	892	2236
gcc-3.2 -O	148	356	910	2278
icc-6.0 -O3	166	397	946	2291

Cycle counts on PowerPC 7400, circa 2002

	FFT(8)	FFT(16)	FFT(32)	FFT(64)
Belady	112	272	688	1648
`gcc-2.95 -O2`	112	368	1168	2896
`gcc-2.95 -O2 -fno-schedule-insns`	112	320	784	1840
`gcc-3.1 -O2`	112	432	1312	3120
`gcc-3.1 -O2 -fno-schedule-insns`	112	288	768	1712

Number of spills on PowerPC 7400

	FFT(8)	FFT(16)	FFT(32)	FFT(64)
Loads:				
Belady	5	21	75	146
`gcc-3.1`	6	26	107	251
Stores:				
Belady	5	21	64	133
`gcc-3.1`	6	23	73	155

(`gcc-3.1 -mcpu=750 -O2 -fno-schedule-insns`)

How does FFTW produce the schedule?

Blocking:

Could generate a different program for each R. (But we don't.)

Cache oblivious:

It turns out that a universal schedule works well for all R.

Cache oblivious FFT

Cooley-Tukey with $p = q = \sqrt{n}$ [Vitter and Shriver]

If $n > 1$:

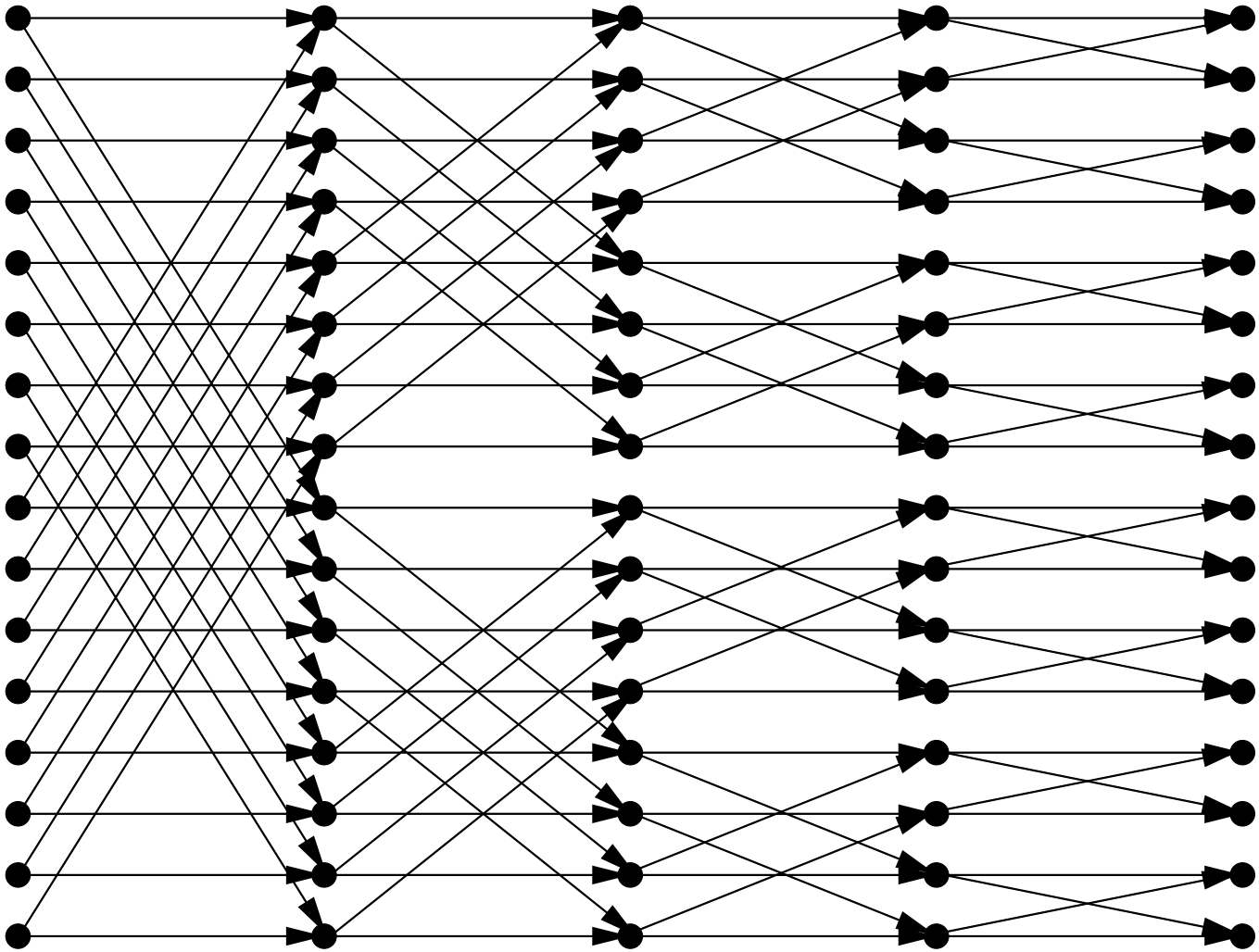
1. Recursively compute $\sqrt{n}$ FFTs of size $\sqrt{n}$.
2. Multiply $O(n)$ elements by the twiddle factors.
3. Recursively compute $\sqrt{n}$ FFTs of size $\sqrt{n}$.

Analysis:

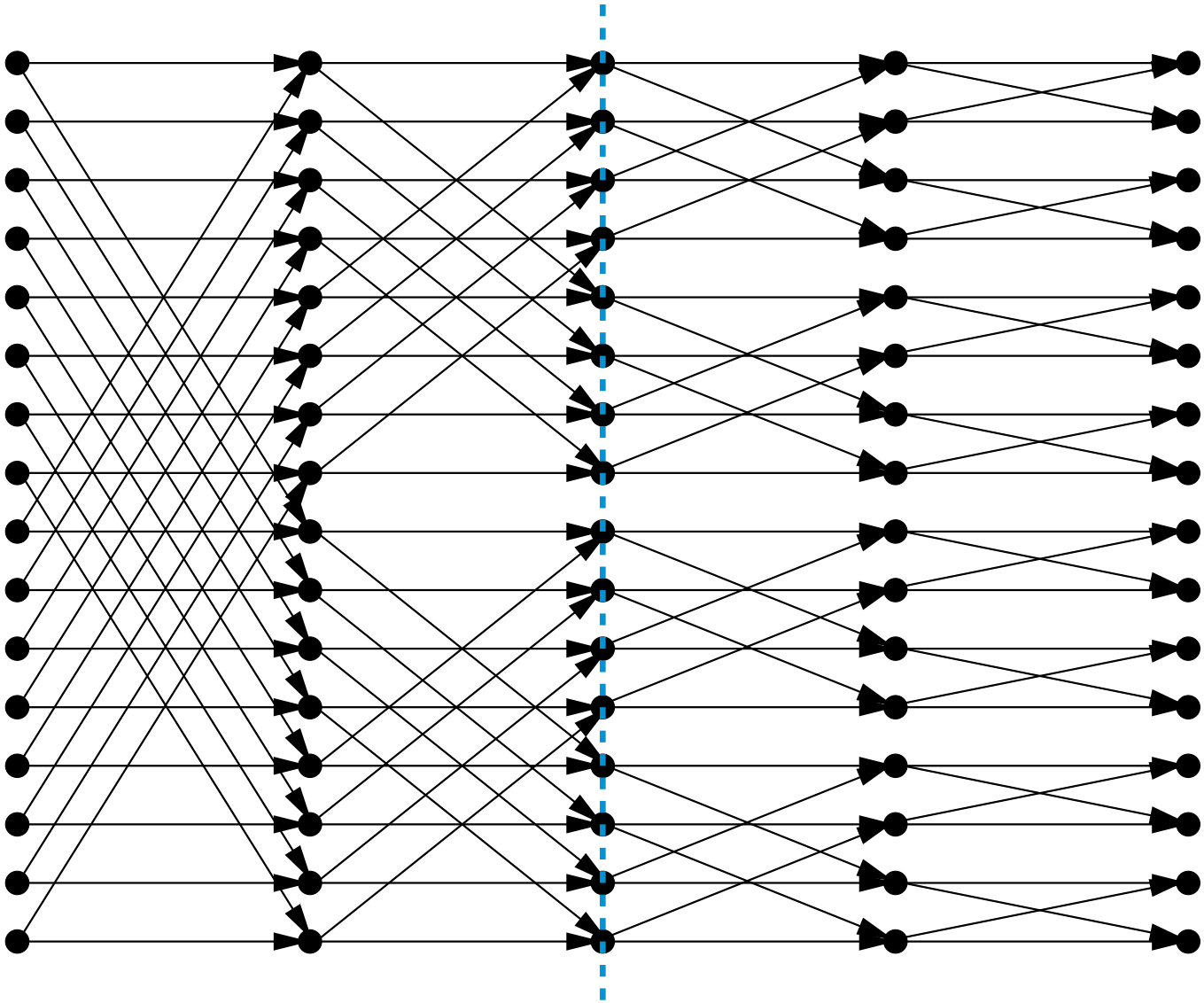
$$
\begin{aligned}
R &= \#\text{ of registers.} \\
S(n) &= \#\text{ of spills in optimal register allocation.} \\
S(n) &\leq \begin{cases} O(n) & \text{if } n \leq \Theta(R)\ ; \\ 2\sqrt{n}S(\sqrt{n}) + O(n) & \text{if } n > \Theta(R)\ . \end{cases} \\
S(n) &\leq O(n \log n / \log R)\ . \qquad \textbf{Optimal.}
\end{aligned}
$$

CILK ARTS

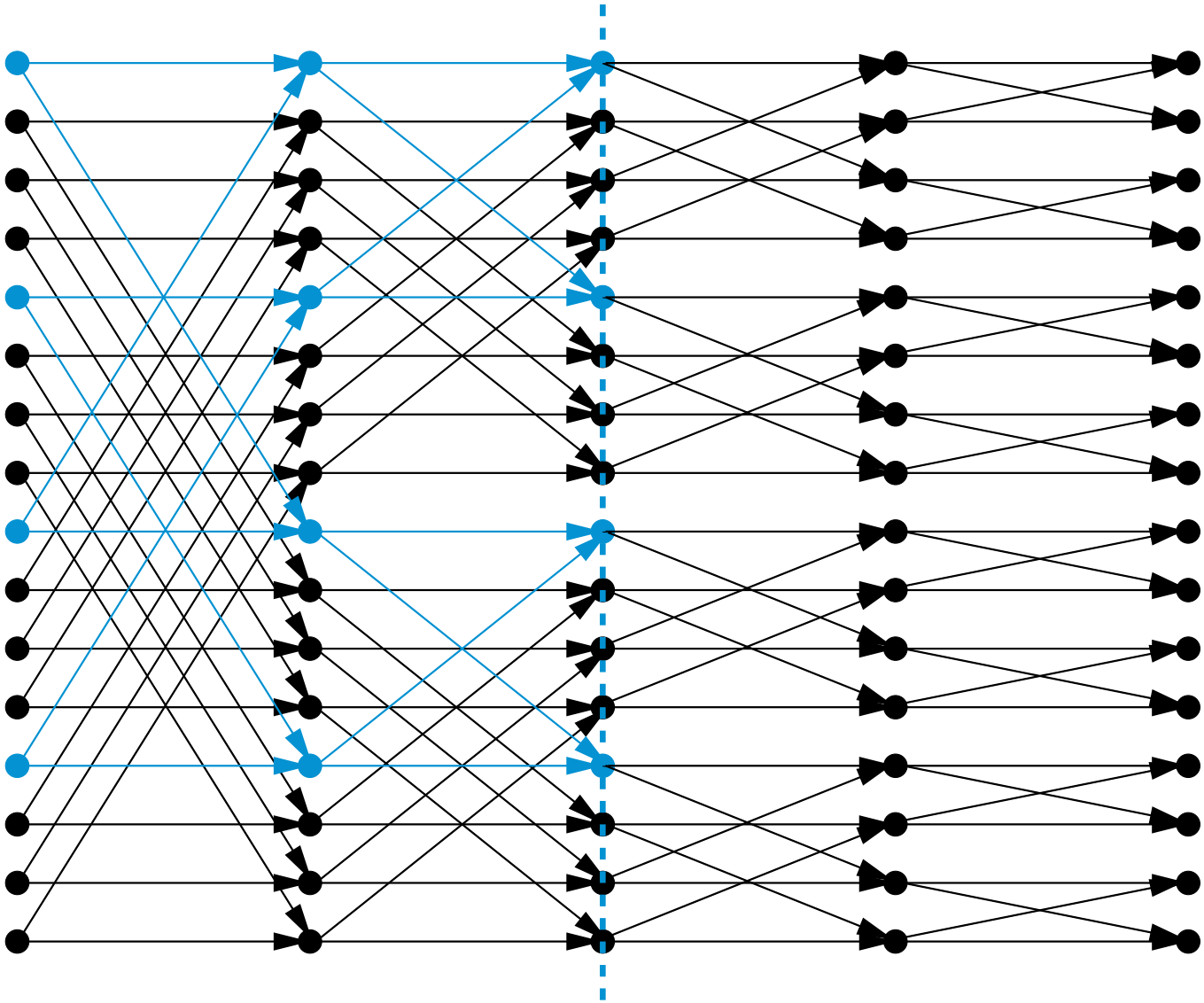
Cache oblivious schedule



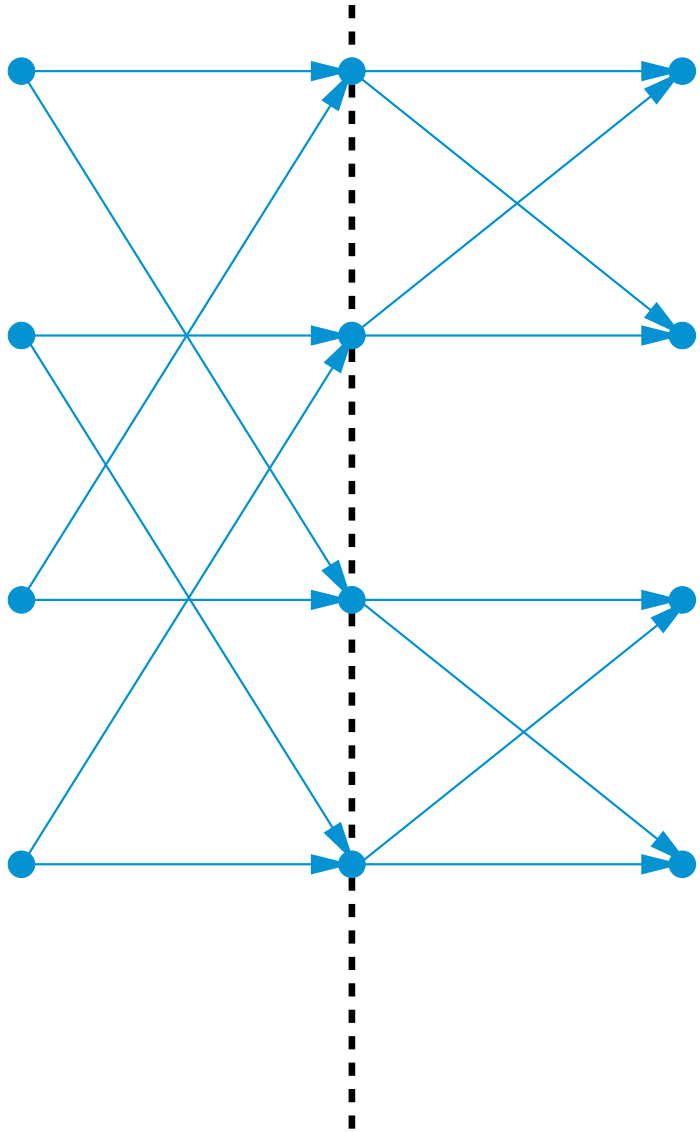
Cache oblivious schedule



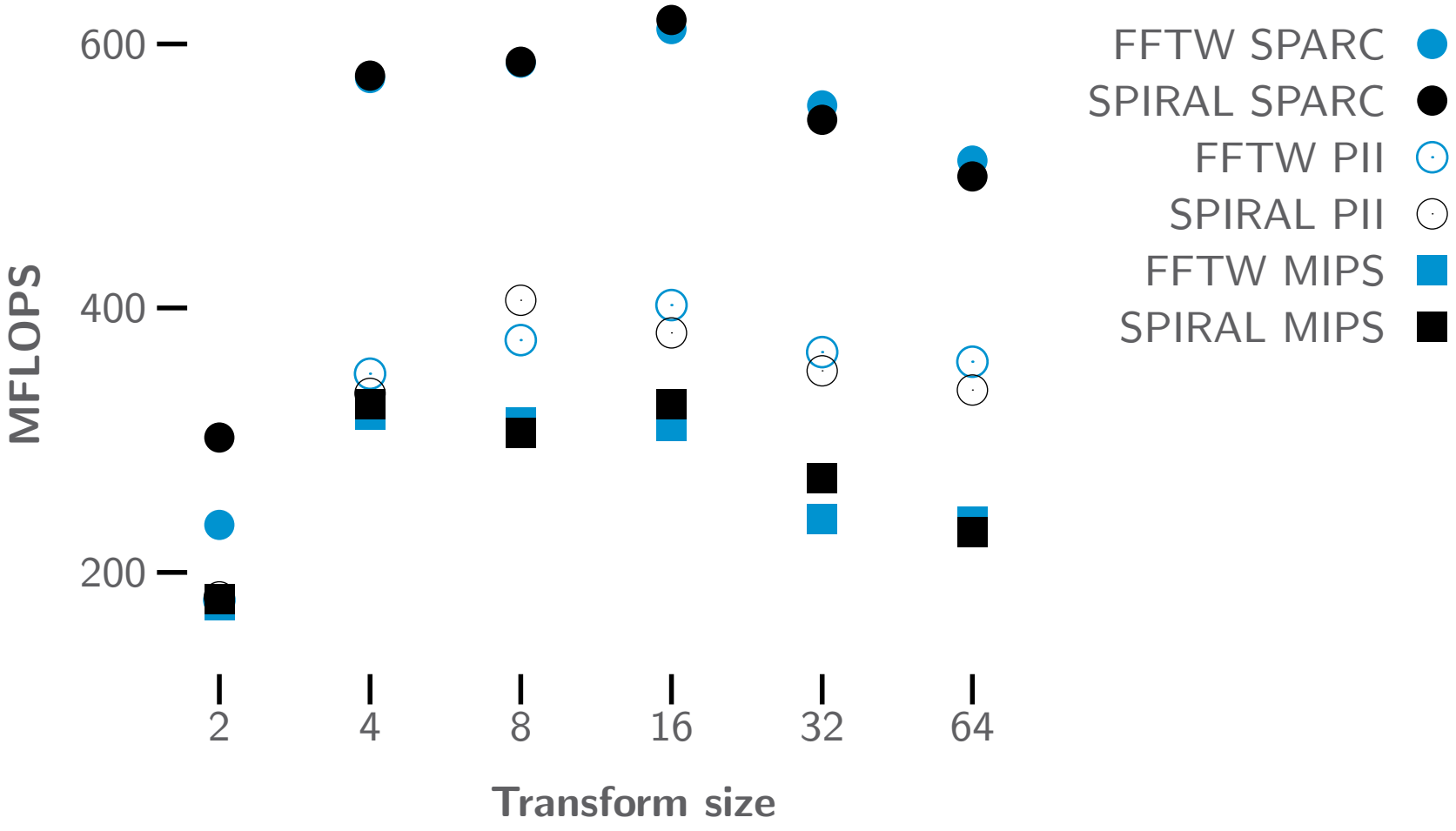
Cache oblivious schedule



Cache oblivious schedule



Machine-specific code does not seem to help



[Xiong et al., PLDI 2001]

Does this technique apply to other problems?

Anecdotal evidence

- FFT, RFFT, DCT, FFT+SIMD:
  - Butterfly-like graphs, $O(n \log n)$ time, $O(n \log n / \log R)$ register spills.
  - Cache oblivious works.
- 1D stencils, 1D convolutions, Gauss-Seidel, probably LCS-style "1D" dynamic programming:
  - $O(n^2)$ time, $O(n^2/R)$ register spills.
  - Cache oblivious works.
- 2D stencils, GEMM/BLAS3, simple "2D" dynamic programming:
  - $O(n^3)$ time, $O(n^3/\sqrt{R})$ register spills.
  - Cache oblivious alone not sufficient. Other effects become significant.

Matrix multiplication kernels

Machine	% peak performance cache oblivious	% peak performance iterative
Power5	58	98
UltraSPARC IIIi	53	98
Itanium II	93	94

[Yotov et al., 2007]

What are these “other effects”?

- Asymptotic theory applied to small n and R.
- Asymmetry of loads and stores.
- Belady does not account for the latency of spills.
- Cache oblivious does not account for the pipeline latency.

Asymmetry of loads and stores

Power5:

- 2 fma/cycle.
- 2 L1 loads/cycle, in parallel with FPU.
- 1 L1 store/cycle, consumes one FPU cycle.

Impact on $n \times k$ by $k \times n$ matrix multiplication kernel:

- $2nk + n^2$ loads, n^2 stores.
- If cost of load is 1, cost of store is γ, then optimal aspect ratio of the kernel is nonsquare:

$$k/n = 1 + \gamma \ .$$

- Must modify the cache oblivious algorithm to account for γ.
- Still cache oblivious, but not γ-oblivious.

Belady and loads/stores

Theorem (Belady 1966)

Given a schedule of the dag, find register assignment that minimizes the number of **loads**: $\approx$ *linear time.*

Theorem (Farach and Liberatore 1997)

Given a schedule of the dag, find register assignment that minimizes the number of **stores**: *NP-hard.*

Theorem (Farach and Liberatore 1997)

Heuristic for the number of stores that is within a small constant factor of optimal: $\approx$ *linear time. Works in practice.*

Latency of reloads

Power5:

- FP load latency: 5 cycles.
- Must schedule 10 flops before using the loaded value.

Problem:

- Belady knows nothing about load latencies.

Belady with lookahead:

- At time t, schedule spills/reloads for instruction at time $t +$ load latency.
- Current compilers don't seem to do it.
- Optimal?

FPU latency

Power5:

- FPU latency: 6 cycles.
- 12 independent flops in flight to keep FPU busy.

Problem:

- Cache oblivious schedule ignores latencies.

Possible solutions:

- Do nothing, hope that out-of-order execution will save you.
- Attack the problem using [Blelloch and Gibbons, 2005].

Register allocation with latencies

Theorem (Blelloch and Gibbons, SPAA 2005)

Given:

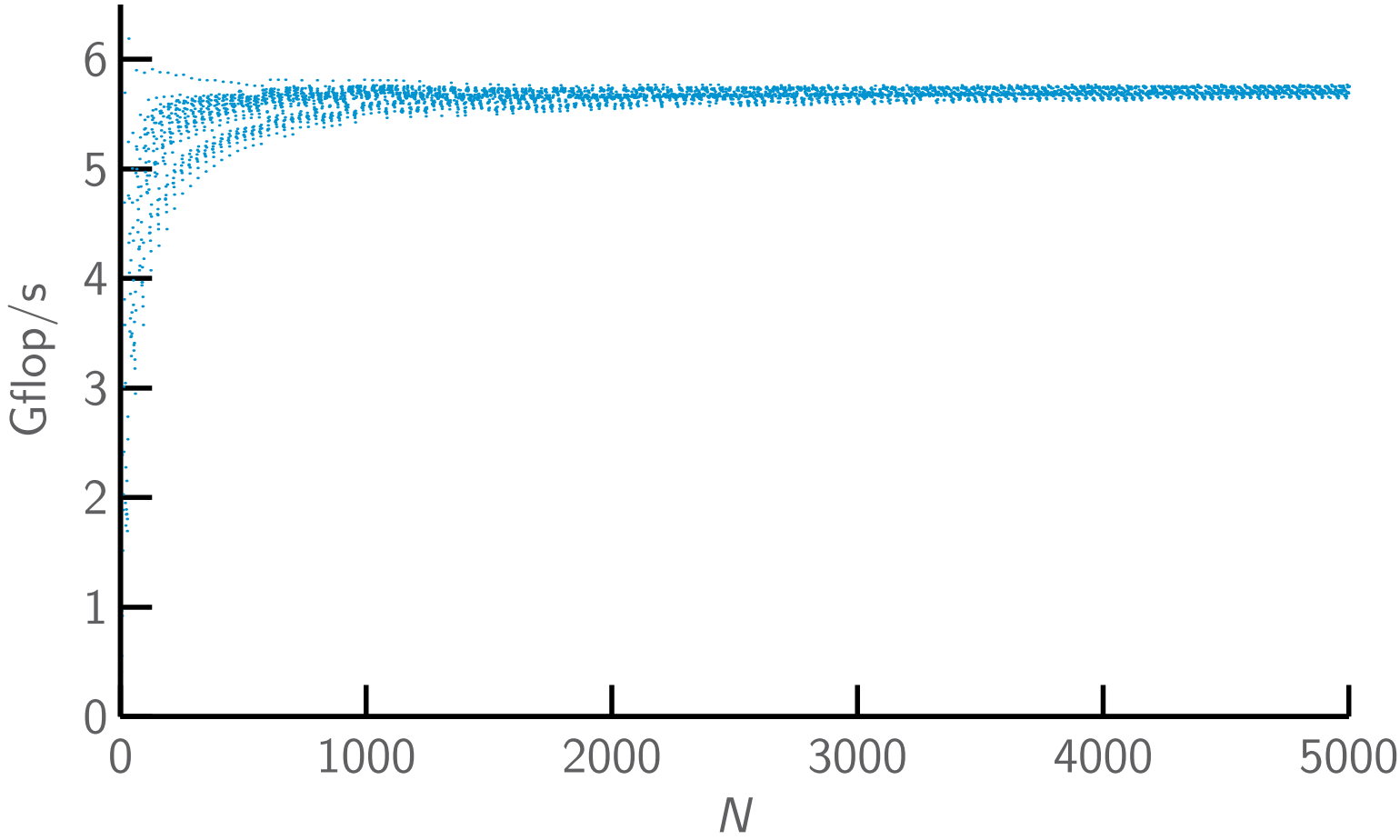
- *A machine with R registers;*
- *A dag of critical path T_∞;*
- *A schedule of the dag that incurs Q_1 spills with Belady.*

Then

- *A schedule of the dag exists that incurs Q_1 spills* **with at most** *LT_∞* **stalls** *on a machine with $R + LT_\infty$ registers and maximum latency L.*
- *The schedule is easy to compute.*
- *Exact result, not asymptotic.*
- *Optimal?*

Cache oblivious DGEMM with all tricks

Power5 (peak 6.6 Gflop/s).
$(N, N) \times (N, N) \rightarrow (N, N) \quad \forall N \in \{1, \ldots, 5000\}$.



CILK ARTS

Conclusions

This page may contain forward-looking statements that are based on management's expectations, estimates, projections and assumptions.

- When they work, as in FFTW, universal cache oblivious kernels are attractive.
- If the "other effects" become significant, then the cache-oblivious approach is much less attractive.
- Belady/lookahead and [Blelloch and Gibbons] are kernel-independent techniques.
- Perhaps an autotuner can be structured as
  - Universal kernel-specific schedule, followed by
  - Sophisticated kernel-independent register allocator parametrized by the latencies.
- Such an autotuner would reduce the search space w.r.t. current systems.