Optimising code for apeNEXT

D. Pleiter

DESY

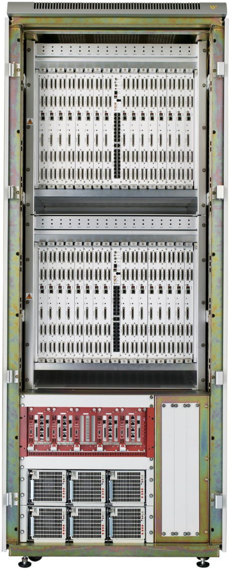
28. November 2006, LAP06, Zeuthen

Agenda

Agenda

Introduction

- Joined European development project
  - DESY
  - Istituto Nationale di Fisica Nucleare (INFN)
  - University Paris-Sud
- Fully custom designed machine
  - Custom hardware
  - Custom software



Agenda

Floating Point Unit (FPU)

- FPU performs one operation $a \times b + c$ per clock cycle, where a, b, c complex numbers
- 64-bit IEEE floating point numbers

Peak performance

→ 8 Flops / cycle = 1.2 GFlops/sec

Memory Hierarchy

- Memory controller
  - Supports 256 MBytes DDR-SDRAM (with ECC)
  - Maximum bandwidth: 2×64 bit word/cycle

Peak performance

➔ 2.4 GBytes/sec or 0.5 Flop/Byte

- Very large register file: 2×256 64-bit registers
- Data pre-fetch queues (no data cache)
- Instruction buffer
  - Burst instruction fetch
  - Different operation modes:
    - Fifo mode: read once, execute once (default)
    - Cache mode: Prefetch instructions for **multiple** execution

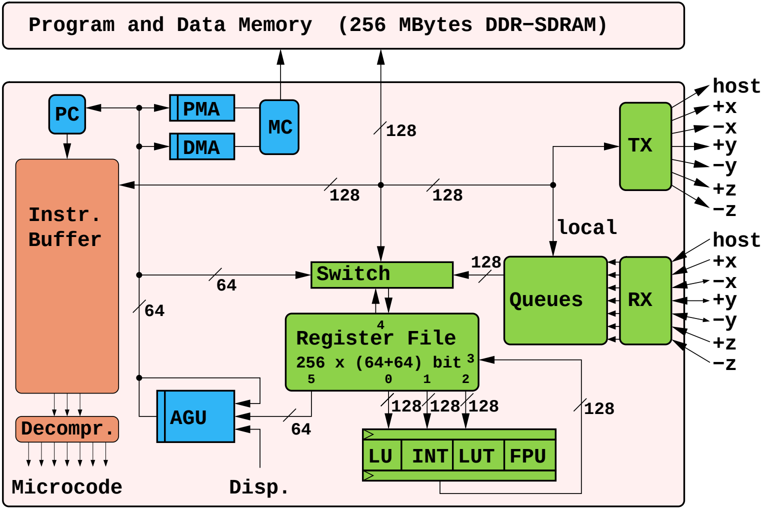
Network

- 7 bi-directional LVDS links: $\pm x$, $\pm y$, $\pm z$, 7th
- Gross bandwidth per link and direction: 8 bit/cycle
- Concurrent send and receive and concurrent transfer along orthogonal directions
- Low latency: $\approx$ 125 ns
- Transmission by frames of 128 bit data + 16 bit CRC

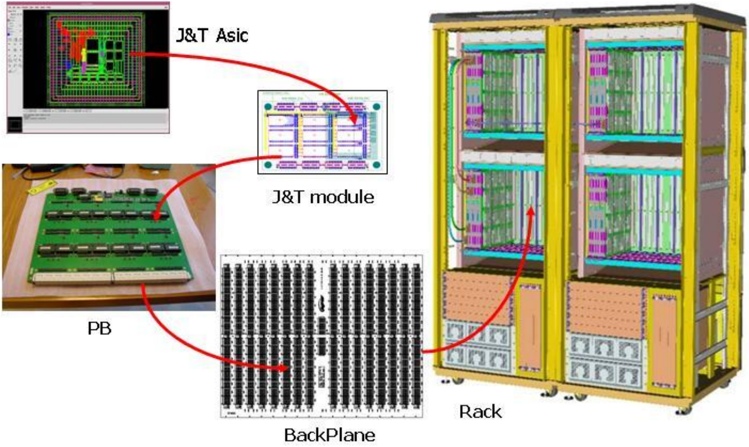
Performance

➔ $<$ 16 Bytes / 18 cycles = (1/16) memory bandwidth

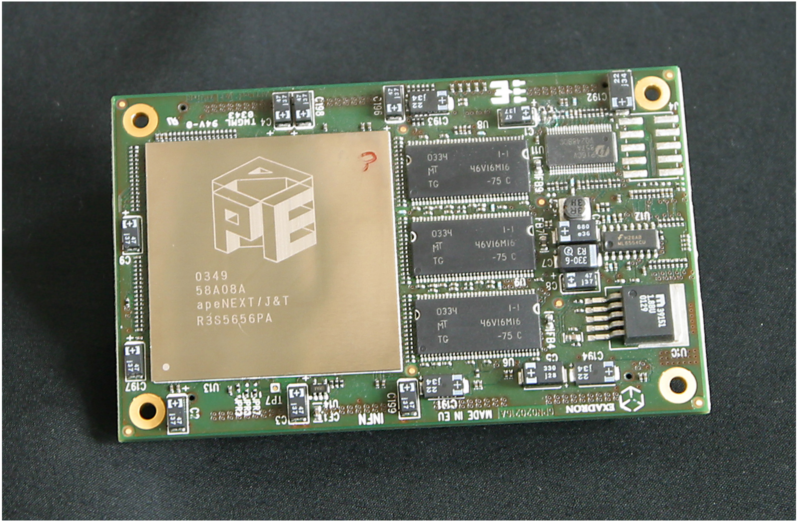
Processor Overview



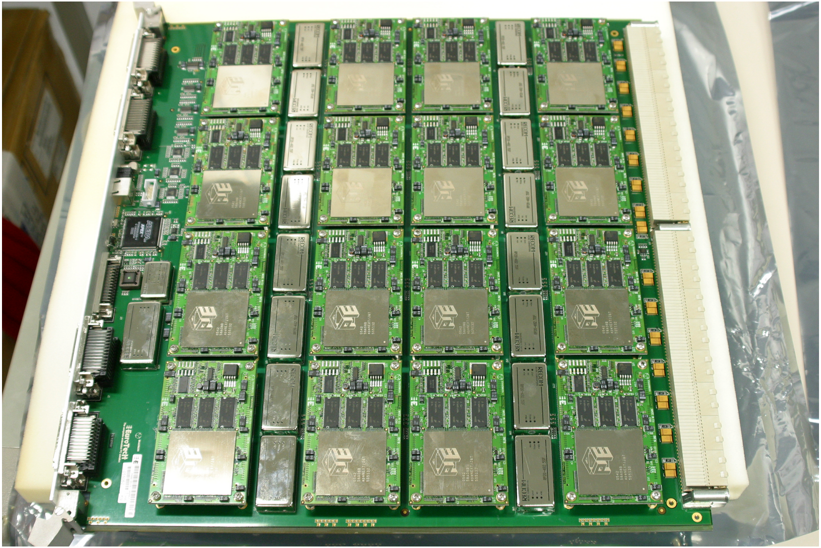
System Overview



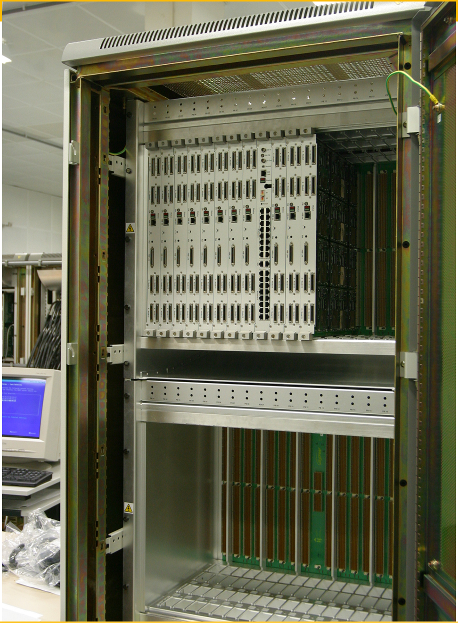
Processing Module



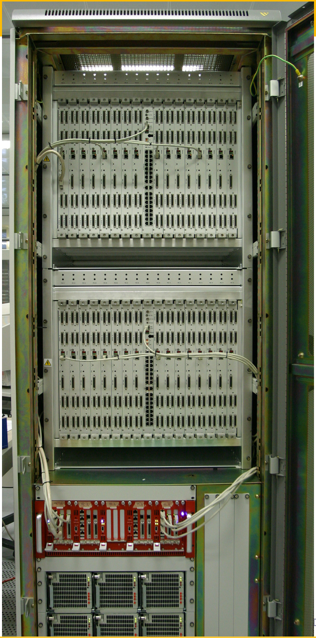
Processing Board



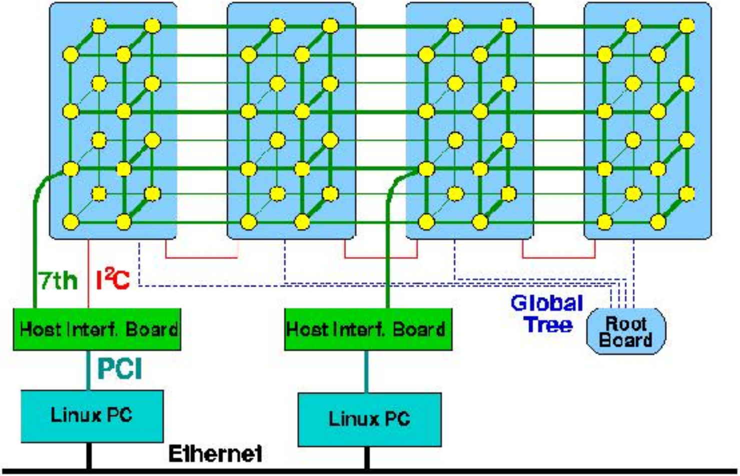
During Installation



Complete Rack

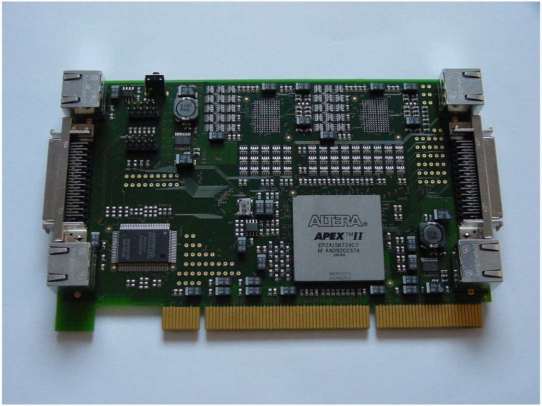


Global Architecture



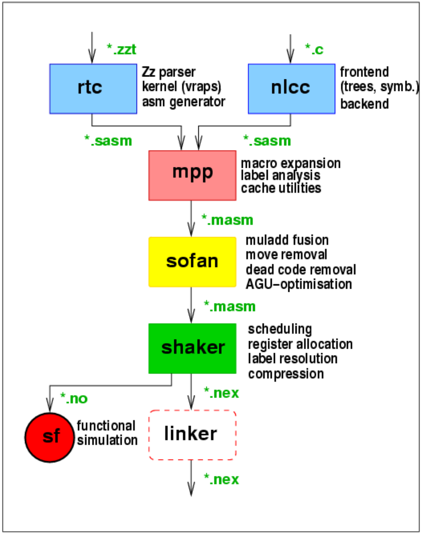
Host Interface Board

- 4 I2C links
- 1 LVDS link
- PCI interface



Agenda

Compiler Chain



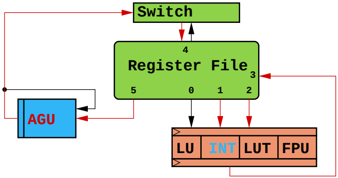
- Compile high level program:
  `rtc myprog.zzt`
- Generate low level assembly:
  `mpp -os7 myprog.sasm`
- Optimise assemble:
  `sofan myprog.masm`
- Generate microcode (VLIW):
  `npsk +a myprog.masm`

Programming Languages

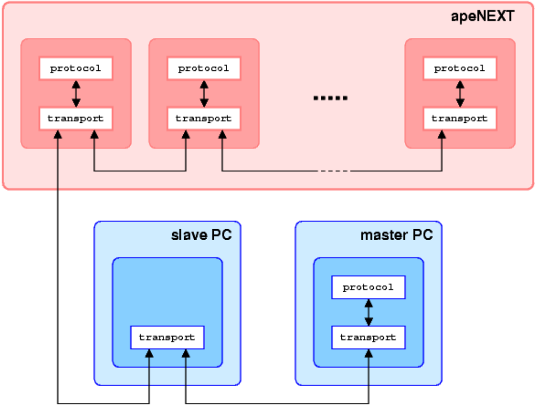
- TAO
  - Procedural programming language (like Fortran, C)
  - Dynamical grammar allows OO-style programming
- C (experimental)
  - Based on freely available lcc + custom implementation of libc
  - Most of ISO C99 standard supported
  - Few APE-specific language extensions
- SASM
  - High level assembler

Assembly Optimiser: SOFAN

- Optimisation operating on **low-level assembly**
- Reconstruction of control- and data-flow graphs
- Optimisation steps:
  - merge multiply-add operations
  - remove dead code
  - eliminate register moves
  - optimise address generation:
  - ...



System Services



- Essential services only: read/write, fopen/fclose, ...
- Parallel I/O transport operations:
  - Global write
  - Slice write
  - Broadcast read
  - Multidata read
- Multi-stage transport ➔ latencies

Agenda

Optimisation Strategies

Optimise for *instruction level parallelism*

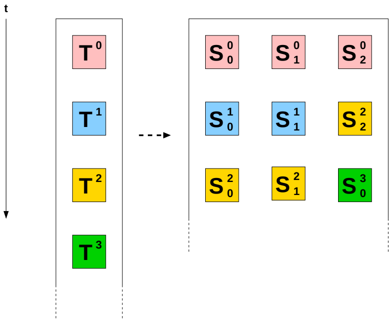
- Task = set of **independent** subtasks
- Hardware devices may allow for concurrent execution, e.g.
  - Arithmetic unit
  - Address generation unit
  - Memory controller
  - Network

Optimise for *pipelining*

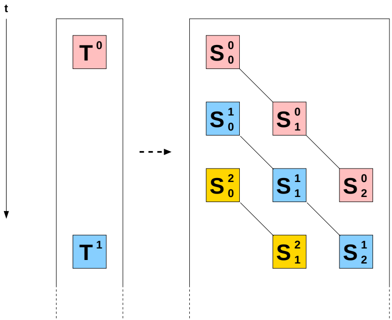
- Task = set of **dependent** subtasks
- Hardware device = pipeline
- Next subtask scheduled before previous completed

Optimisation Strategies: Cartoons

Instruction level parallelism:



Pipelining:



Operation Costs

- Latency λ: t(first result) - t(start first operation)
- Throughput τ (=1/bandwidth): resource occupation time
- Distance δ: minimal distance between two operations

Operation	λ	τ	δ
complex mul-add	10	1	-
fp division	107	11	-
integer add	4	1	-
memory load	20	1	4
memory store	10	1	4
jump	O(10-20)	-	-

Performance Analysis

- Static performance analysis
  - Analysis at compile time
  - Tools:
    - `mem2ncd myprog.mem` → `myprog.ncd`
    - `nperf myprog.mem`
    - (`nprof`)
- Run-time performance analysis
  - Gather run-time information from hardware counters:

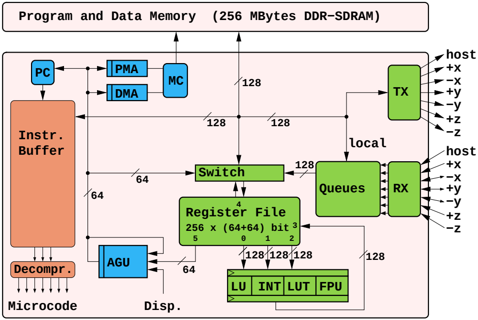
`clock`	clock cycles while not in system mode
`run`	clock cycles while not stalling
`qwait`	clock cycles while waiting for data in queue

  - TAO syntax: `get clockcnt in clockcnt_var`
    - Example:

```
clockcnt runCounter
...
get clockcnt in runCounter
```

Performance Critical Devices

- Arithmetic unit
  - Pipeline filling
- Memory
  - Effective bandwidth
- Register file
  - Input/output port occupation (P4)
- Network
  - Bandwidth



Agenda

Optimise Memory Access

- Memory bus is used for
  - Data load
  - Data store
  - Instruction load
- **Goal:** reduce memory bus occupancy

Possible solutions:

- Optimise data load/store
  - Avoid re-loading data (**data re-use**)
  - Reduce number of memory operations (**burst memory access**)
- Optimise instruction load
  - **Compress microcode** (`npsk -z ...`)
  - **Use instruction cache**

Instruction Load Optimisation

- **Example**:

```
matrix complex su3.[9]
su3 v[512], w[512]

begin cache
do i = 0, 511
  v[i] = w[i]
enddo
end cache
```

- **Results** for size of loop body (static performance analysis):

Compress	Cache	instructions	words
no	no	82	82
no	yes	61	61
yes	no	82	36
yes	yes	61	23

- **Caveat:**
  - Only gain if there is a resource conflict
  - Need time to pre-fetch instructions

Keep Data in Physical Registers

- **virtual registers**: lifetime ends at control flow instruction
- **physical registers**: lifetime ends at subroutine call/end

```
constant N = 512
complex a, x[N], y[N], z[N]



do i = 0, N-1
  z[i] = a * x[i] + y[i]
enddo
```

```
constant N = 512
complex a, x[N], y[N], z[N]
physreg complex ra

ra = a    !! load a only once
do i = 0, N-1
  z[i] = ra * x[i] + y[i]
enddo
```

- Change of loop body size: 90 → 82 instructions
  - Remark: We assume the use of Sofan
- We are still far from optimal:
  - Memory bandwidth: 16 byte per cycle
  - FPU throughput: 1 complex add per cycle

Burst Memory Access

```
constant N = 512
complex x[N], y[N]



do i = 1, N-1
  x[i] = y[i]
enddo
```

```
constant N = 512
constant B = 64
matrix complex bcomplex.[B]
bcomplex x[N/B], y[N/B]

do i = 1, (N/B)-1
  x[i] = y[i]
enddo
```

- Number of memory operations reduced 64$\times$
- Change of loop body size: 69 $\rightarrow$ 192 instructions
  ($192 \ll 64 \cdot 69 = 4416$)

Burst Memory Access (2)

- Alternative implementation:

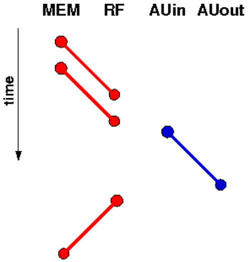
```
constant N = 512
constant B = 64
matrix complex bcomplex.[B]
complex x[N], y[N]
register bcomplex rx, ry

do i = 1, N-1, B
  extract rx from x[i]
  ry = rx
  replace ry into y[i]
enddo
```

- Advantages:
  - Use “natural” objects outside application kernels
  - Possible to use different burst length
- Keyword `register` defines *virtual registers*

Pipeline Optimisation

- **Example:**

```
constant N = 512
complex x[N], y[N], z[N]

begin cache
do i = 0, N-1
  x[i] = y[i] + z[i]
enddo
end cache
```

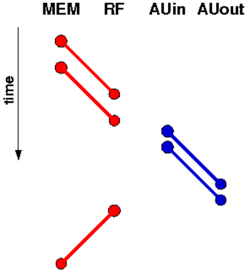


- Size of loop body: 49 instructions

Pipeline Optimisation (2)

- **Optimised Example:**

```
constant N = 512
constant B = 2
matrix complex bcomplex.[B]
bcomplex x[N/B], y[N/B], z[N/B]

begin cache
do i = 0, N/B−1
  x[i] = y[i] + z[i]
enddo
end cache
```



- Size of loop body: 56 instructions ($\ll 2 \cdot 49$)
- Expect optimal choice B $\simeq \lambda_{\text{complex add}}$
- Double gain:
  - Improved filling of FPU pipeline
  - Burst memory access

Latency Hiding by Data Prefetch

- Aim: use time needed to load data for other operations
- Example:

$$\phi^a_{\mu,i} = U^{ab}_i \, \psi^b_{\mu,i}$$

```
/include <qcd>

constant N = 512
su3           u[N]
color_spin  psi[N], phi[N]

begin cache
do i = 0, N-1
  phi[i] = u[i] * psi[i]
enddo
end cache
```

```
...
su3                       u[N]
physreg su3               ru
color_spin                psi[N], phi[N]
physreg color_spin        rpsi
register color_spin       res

rpsi = psi[0]                !! load(0)
ru   = u[0]
begin cache
do i = 0, N-1
  res    = ru * rpsi         !! compute(i)
  rpsi   = psi[i+1]          !! load(i+1)
  ru     = u[i+1]
  phi[i] = res               !! store(i)
enddo
end cache
```

Explicit Use of Prefetch Queues

- Data load via queue:
  - `MTQ`: Load data from memory to queue
  - `QTR`: Load data from queue to register file
- Processor will stall at `QTR` if queue is empty
- TAO instructions:

```
begin prefetch      !! start prefetch section
queue = x           !! load word from memory to queue
rx = queue          !! load word from queue to RF
end prefetch        !! end prefetch section
```

  - **begin/end pre-fetch** enforces local memory access for all implicit load operations and changes scheduling strategy
  - Correct order of pre-fetch/fetch is **user responsibility**!

Latency Hiding: Using the Queue

```
...
begin prefetch
queue = psi[0]              !! prefetch(0)
queue = u[0]

begin cache
do i = 0, N-1
  rpsi   = queue            !! load(i)
  ru     = queue
  queue  = psi[i+1]         !! prefetch(i+1)
  queue  = u[i+1]
  phi[i] = ru * rpsi        !! compute(i)
enddo
end cache

rpsi   = queue              !! load(N)
ru     = queue
end prefetch
```

Latency Hiding: Results

- Results:

	Number of instructions
Original version:	113
With pre-fetch:	89

- Attention to pre-fetch during last loop iteration:
  - Must be **suppressed or flushed**
  - May need **extended array boundaries**

Example: Optimise (Local) Vector Product

- Consider operation: $c = \sum_i |z_i|^2$
- Naive implementation:

```
constant N = 512
complex  c, z[N]

c = complex_0

do i = 0, N-1
  c = c + z[i] * z[i]~
enddo
```

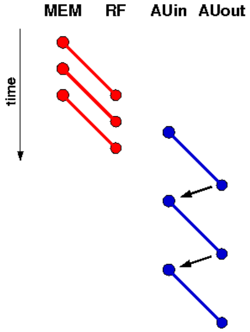
First Attempt

```
constant N = 512
constant B = 16
matrix complex bcomplex.[B]

complex              c, z[N]
register bcomplex    rz
physreg complex      rc

rc = complex_0
begin cache
do i = 0, N-1, B
  extract rz from z[i]
  /for k = 0 to B-1 {
    rc = rc + rz.[k] * rz.[k]~
  }
enddo
end cache
c = rc
```

First Attempt (2)

- Change of loop body size: $93 \to 179$ instructions $= 11$ instructions per array element
- Problem **read-after-write** dependency:

```
extract rz from z[i]
/for k = 0 to B−1 {
  rc = rc + rz.[k] * rz.[k]˜
}
```



- **Solution**: Partial sums

Implementation of Partial Sums

```
...
physreg  bcomplex   rc
register bcomplex   rz

/for i = 0 to B-1 {
  rc.[i] = complex_0
}
begin cache
do i = 0, N-1, B
  extract rz from z[i]
  /for k = 0 to B-1 {
    rc.[k] = rc.[k] + rz.[k] * rz.[k]~
  }
enddo
end cache
c = rc.[0]
/for i = 1 to B-1 {
  c = c + rc.[i]
}
```

Implementation of Partial Sums (2)

- Number of instructions in loop body:

	total	per element
Original version	53	53
Optimisation 1	179	11
Partial sums	59	3

- Optimal number of instructions per element: 1
  - For larger `B` it is possible to get close to optimum

Condition Stack

- Effect of operations may be conditioned
- Example:

```
integer i, j
real s

s = 0.
if (i > j)
  s = 1.
endif
```

```
integer i, j
real s

s = 0.
where (localint(i) > localint(j))
  s = 1.
endwhere
```

- Change of main program size: 102 → 66 instructions
- Performance penalties due to control flow statements:
  - Jump costs O(20) clock cycles
  - **Pipeline break**

Agenda

Relevant Optimisation Tricks

- Optimise data load/store
  - Avoid re-loading data
    - Use physical registers
  - Perform burst memory access
    - TAO: `extract`, `replace`
  - Hide latency by data pre-fetch
- Optimise instruction load
  - Compress microcode
  - Use instruction cache
- Optimise pipeline usage
  - Loop unrolling (`/for`)
- Eliminate instruction dependencies
- Avoid pipeline breaks